Mälardalen University Licentiate Thesis
No.59

Symmetric Cardinality Constraints

Waldemar Kocjan

December 2005

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 91-85485-03-9
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This thesis contains previously published papers concerning different aspects of symmetric cardinality constraints. A symmetric cardinality constraint generalizes cardinality constraints like constraint of difference and global cardinality constraint. The algorithms formulating the basis for consistency checking and filtering of a symmetric cardinality constraint can be successfully used to model and solve several other problems like cardinality matrix problems, Latin square problems and rostering problems.

This thesis begins with an introduction to constraint satisfaction and describe several problems, mainly dynamic scheduling problems, which can be modeled using this framework. The two following papers introduce symmetric cardinality constraint and its weighted version. They also provide methods for computing their consistency and algorithms for filtering their domains. Finally, the last paper describes methods for maintaining consistency of a symmetric cardinality constraint with costs, in the context of dynamic constraint satisfaction.

Preface

Constraint Programming is a programming paradigm which has been very successful in modeling and solving many combinatorial and optimization problems. This success has gradually led to attention from industry and commerce and growing popularity among different research communities.

One of the reasons for the success of Constraint Programming is its eclecticism. Any method from any computational field can be incorporated into this framework as long as it brings satisfying results in solving user defined problems. During the years Constraint Programming assimilated methods from optimization, operation research, graph theory and many others. Many other methods are on the way to be included among Constraint Programming tools.

This thesis deals with a special class of constraints called cardinality constraints. It has been shown that this class of constraints can be successfully used to model and solve large industrial problems like personnel planning and rostering problems as well as more theoretical problems like Latin square problem or n-queens problem or simple recreational problems like the sudoku-puzzle.

The work on this thesis was partially supported by Mälardalen University, Swedish Institute of Computer Science and IRECO.

I would like to thank my supervisors Björn Lisper of Mälardalen University and Per Kreuger of the Swedish Institute of Computer Science for their engagement and support. I would also like to express my gratitude to people who encouraged me to continue this work despite occurring difficulties.

Waldemar Kocjan
Västerås, August, 2005

Contents

Chapter 1

Introduction

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solve it." [4]. However controversial this statement might be it is also true that Constraint Programming provides a modeling tool, which enable programmers to concentrate at the intrinsic nature of the problem they are dealing with rather than drawing their attention to low level algorithms and data structures.

Informally, Constraint Programming is defined as a programming paradigm in which a set of constraints or criteria which a solution must meet is defined rather than a set of steps to obtain such solution. Constraint Programming has been thought of as a programming tool which is to help to solve constraint satisfaction problems (CSP), defined precisely as a problem of finding out which values can be assigned to problem variables to satisfy constraints imposed on them.

The study of constraint satisfaction problems and methods for solving them started during the 70's with studies of scene labeling in vision research. Subsequently, other problems, like the N-queens problem and the graph coloring problem has been described as constraint satisfaction problems. During the end of the 80's and beginning of 90's CSP found its application in the area of temporal and spatial reasoning as well as in modeling scheduling problems. Since then many researchers has recognized constraint satisfaction as a crucial part of planning in Artificial Intelligence.

With the time it also became evident that many of the problems from Operation Research (OR) and Optimization can be formulated as constraint sat-

isfaction problems. The CSP framework was shown to be successful in assigning values to each problem variable occurring in any feasible solution to a defined problem. Exchange between OR and CSP is mutual, many methods known from OR has been successfully applied to numerous constraint satisfaction problems.

The generality of CSP lead to the development of new computational methods and the formulation of a new programming paradigm known as Constraint Programming (CP). Constraint Programming can not be described by referring to any specific programming language. In fact, developers of CP tend to encapsulate solving methods in already existing languages rather than defining new ones. Examples of such successful embeddings are numerous, it includes Prolog (ECLiPSe [2], SICStus [10]), Oz (Mozart [6]), C++ (ILOG Solver [3]) and most recently Java (CHOCO [1], Koalog [5]).

1.1 Basics of Constraint Satisfaction

A constraint satisfaction problem is a problem of assigning certain values to variables in a way, which satisfies conditions, or constraints, imposed on its solution.

For example, a graph coloring problem, which is a problem of assigning different colors to the nodes of a graph in a way that no two adjacent nodes have the same color, can be modeled as a constraint satisfaction problem. In such a case, each node of a graph is *a variable* of the CSP and each color is *a value*, which can be taken by a variable. In 3-coloring problem, each node can be colored by one of the tree colors, let say, red, green and blue, so in a CSP a value assigned to a variable would be chosen from one of the colors: red, green or blue. The set of values which can be assigned to a variable, in this case the set containing all tree colors, is called the *domain* of the variable.

The conditions, which must be satisfied by the solution to the problem are that no two adjacent nodes has the same color. So, if n_1 and n_2 are two adjacent nodes in the graph and x_1 and x_2 are two variables of our CSP representing respective nodes, then we can state that $x_1 \neq x_2$, which is *a constraint* imposed on a solution to our problem. A similar constraint is defined for each pair of variables representing a pair of adjacent nodes.

A constraint satisfaction problem is solved by problem reduction and search. Problem reduction is a way of transforming a problem to equivalent but an easier to solve problem by reducing domains of variables and constraint in the problem. During the problem reduction values, which can not be assigned to

a variable are removed from its domain and constraints imposed on a problem are tightened. During the search to each subsequent variable is assigned a value satisfying the defined constraints.

These two steps for searching for solution are often interleaved. For example, the graph coloring problem can be solved by such a technique. At first we assign a color value to a variable. Lets say we assign color red to variable x_1 representing node n_1. Then, for each variable representing a node adjacent to n_1 we remove color red from the domain of such variable. At the next step we assign a value to x_2 and again reduce domains of the adjacent variables. We repeat each step until we find a solution or until we have tried all possible assignments and failed.

In the graph coloring problem we are interested in assigning only one value to a variable, which corresponds to assigning a color to a node of a graph. However, in certain problems we would like to assign *a set* of values rather than an individual value. For example, in a social golfer problem, we are interested in assigning a set of four players to each group. This type of problems are called *set constraint satisfaction problems* or set-CSP. The only difference between CSP and set-CSP is that in set-CSP to a variable is assigned a subset of values from its domain.

The problems described above are well defined and demand only one-time solutions. The variables, their domains and the set of constraints are fixed and known in advance. However, in many situations, not all parameters of a problem are known in advance or they can vary over the time. A typical example of such problem is a scheduling problem, that is a problem consisting of assigning start and duration times to a set of activities which must be performed on a set of machines. This kind of problems may change their parameters due to different, most often impossible to foresee events like delays, machine failures etc. When any of such disruption occurs, the original solution to a problem is most often no longer valid.

This kind of problems are modeled as *dynamic constraint satisfaction problems* (DCSP). A dynamic constraint satisfaction problem is a sequence of static constraint satisfaction problems, where each CSP is a result of changes in the preceding one. A difference between two CSPs is expressed as a set of added and deleted constraints. For example, deleting an activity from a scheduling problem is expressed by deleting all constraints containing this activity like constraints defining its start times and duration or constraint stating precedence relations with other activities, that is constraints stating that the activity must be performed before or after some other activities. Adding an activity to a scheduling problem results in turn in adding this type of constraints to the

problem.

Similarly, if we would have defined described earlier graph coloring problem as a dynamic problem, we would model it as a DCSP. In such case adding an edge between nodes n_i and n_j in the graph, which are represented by variables x_i and x_j in a CSP, would result in adding a new constraint stating that $x_i \neq x_j$. Adding a new node with a set of edges results in adding a set of constraints similar to the previous one stating that added node must not have the same color like the nodes adjacent to it. Removing an edge or a node from the graph results in removing corresponding constraints from the CSP.

There exists several other models in constraint satisfaction for handling problem with varying or not complete parameters. Paper A of this thesis survey most of the techniques and models for dynamic constraint solving.

Given this introduction into constraint satisfaction we can continue with the description of cardinality constraints and later show how this type of constraints can be used in practice.

1.2 Cardinality Constraints

Cardinality constraints is a class of constraints which explicitly restricts the number of values assigned to a variable or/and the number of occurrence of a value in a solution. The most basic constraint of this class is the constraint of difference [7], which assure that to each problem variable are assigned different values, that is each value is assigned to at most one variable and that each variable is assigned exactly to one value. The constraint of difference is one of the most extensively used constraints and many of the constraint satisfaction problems like the N-queens problem, the sudoku or Latin squares are modeled using this constraint.

Later, this class of constraints was extended by the global cardinality constraint [8]. If the constraint of difference assures that each value is assigned to at most one variable, the global cardinality constraint restrict the number of occurrence of a value in a solution to be in interval $[min\_occurrence..max\_occurrence]$. Another constraint in this class is the symmetric cardinality constraint, which is the subject of Paper B included in this thesis. In contrast to the constraints mentioned earlier, the symmetric cardinality constraint is defined in a framework of the set constraint satisfaction. In set constraint satisfaction each problem variable is assigned not just a single value but a set of values. The symmetric cardinality constraint assures that the cardinality of a set assigned to a problem variable is in an interval $[min\_cardinality..max\_car-$

$dinality]$ while simultaneously each value occurs in the union of sets in an interval $[min\_occurrence \ldots max\_occurrence]$.

The symmetric cardinality constraint might be viewed as an extension of the previously mentioned constraints. For example, a constraint of difference can be modeled as a symmetric cardinality constraint where the cardinality of sets assigned to variables is set to be exactly 1 and the occurrence of each value is restricted to the interval $[0, 1]$. Similarly, a global cardinality constraint can be modeled as a symmetric cardinality constraint by restricting the cardinality of sets assigned to problem variables to 1.

The symmetric cardinality constraint may also be viewed as a *(0,1)-CardinalityMatrix* constraint, that is a constraint which places zeros and ones in a $m \times n$ matrix in such a way that the number of ones in each row and each column is restricted to some interval. In this case each row of a matrix is a variable of a symmetric cardinality constraint and each column is a value of the constraint. Restriction on the number of ones in each row and column of the matrix is then the restriction on the cardinality of a set assigned to each variable and the number of occurrences of each value in such sets. Each value occurring in a set assigned to a variable corresponds with a one in the respective cell of a matrix. For example, if a value 6, corresponding with 6'th column of a matrix occurs in a set assigned to variable x_1, corresponding with the first row of the matrix, then a one is placed in a cell in first row and sixth column.

The *(0,1)-CardinalityMatrix* constraint has been formalized by J.-Ch. Régin and C. Gomes [9], who also pointed out its similarity to the symmetric cardinality constraint. The *(0,1)-CardinalityMatrix* constraint uses the same algorithm for verifying consistency and filtering as the one introduced by us in Paper B.

The most general cardinality constraint is the cardinality matrix constraint [9]. Here, the symbols placed in a matrix are not restricted to 0 and 1. In this constraint the placement of each symbol is solved by solving *(0,1)-CardinalityMatrix* problem. This basically means that in order to solve one cardinality matrix problem involving ten different symbols we have to solve ten *(0,1)-CardinalityMatrix* problems, one for each symbol. As shown above, it corresponds to solving the same number of symmetric cardinality constraints.

Some cardinality constraints, like the constraint of difference and the global cardinality constraint, have been extended by attaching cost to every assignment of a value to a variable. The extension of the symmetric cardinality constraint with a cost function is the subject of Paper C in this thesis.

All the cardinality constraints described above deal with so called static problems, that is problems which have fixed, known in advance inputs and which do not vary with the time. This is seldom a case for real life, indus-

trial problems. Problems like scheduling jobs on machines, assigning workers to shifts or tasks or building rostering schedules are often subjects of additional constraints emerging during the execution of a initial schedule. Machine failure, absence of a worker or temporary work overload leads often to the necessity of modifying the original solution to cope with the new problem setup.

Basically, there exist two different approaches in dealing with dynamic problems. One is proactive, which means that it tries to take into consideration all possible disturbances which can emerge during execution of a plan or a schedule and produces a solution to the problem that is as robust as possible. The other, reactive approach, tries to deal with situations as they appear.

In constraint satisfaction, the dynamic problems are dealt with usually by recomputing the solution from scratch every time the original problem is modified. This is however very often too expensive with regard to computational time.

Its not surprising then, that computing methods for dynamic problems has recently gained a lot of attention within Constraint Programming community. The research in this area has recently made a substantial progress by the rapid development of explanation based methods for dealing with dynamic problems. Many of the constraints has been extended to handle the dynamic aspect using explanation. For example, the methods for maintaining consistency of values with constraint for dynamic constraint of difference and global cardinality constraint has been developed. No method however has been defined for maintaining consistency of cardinality constraints with costs. The consistency for dynamic cardinality constraints with costs is a subject of Paper D in this thesis.

1.3 Cardinality Constraints in Action

In this section we show how cardinality constraints can be used in practice to solve some problems. We start with an example of the sudoku puzzle.

The sudoku puzzle is popular placement puzzle. The aim of the puzzle is to place numerals 1 to 9 in a 9×9 grid consisting of nine 3×3 subgrids, called regions. Some of the cells in a grid already contains numbers. The remaining cells must be filled in such a way that each numeral appear exactly one time in each row, column and region. A sudoku has exactly one solution. An example of the sudoku puzzle is given in Figure 1.1.

By the definition of the puzzle it is clear that it can be solved by imposing one constraint of difference at each row, column and region of the grid. In such

	3	4		7				
		2	1	9	5			
1		8						6
8				6				3
	2		8		3		9	
7				2				
	6						8	
			4		9			5
				8		1	7	9

Figure 1.1: Example of sudoku puzzle

a formulation each cell of a grid is a variable which can take a value 1 to 9, except cells which already has been filled. For example, at the beginning it is known that the cell in row one and column four $V_{1,4}$ can take one of the following values $D(V_{1,4}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. We call $D(V_{1,4})$ the domain of $V_{1,4}$. The constraint of difference will remove the numbers already placed in row one from this domain, thus $D(V_{1,4})$ will be reduced to the following numbers $\{1, 2, 5, 6, 8, 9\}$. The constraint of difference imposed on column four will reduce $D(V_{1,4})$ to $\{2, 5, 6, 9\}$. Subsequent constraints of difference will assure that 2 can not be assigned to $V_{1,4}$. Finally, the constraint of difference imposed on the region including $V_{1,4}$ will reduce $D(V_{1,4})$ to 6.

The same result can be achieved with more general cardinality constraints like a symmetric cardinality constraint. When using a symmetric cardinality constraint we restrict the cardinality of the set assigned to each variable and the number of occurrence of each value to be exactly one.

An more effective way to model sudoku is to create one cardinality constraint for each numeral 1 to 9 and one constraint for each region. In such case the constraints imposed on numerals will remove the respective numerals from the domains of variables, where such a numeral may not be placed. For

example, the cardinality constraint imposed on the numeral 1 will remove this symbol from the domains of the variable representing cells in rows 2, 3, 9 and in columns 1, 4, 7. Then cardinality constraints imposed on each region will assure proper placement of each numeral.

Now, assume that each column of the puzzle represents a time slot and each row represent a member of a factory staff. Assume also that each of numbers 1..9 corresponds to an activity which needs to be performed by the staff. Moreover assume that each activity must be performed at each time slot and that each worker must perform each defined activity. Let also some of the activities be assigned to specific workers in advance. Solving this kind of scheduling problem is equivalent with solving the sudoku puzzle.

Indeed, the structural similarity of sudoku with some scheduling and rostering problems is striking and since cardinality constraints can be used to solve sudoku they certainly might be helpful in solving similar, more or less complex real life problems. Of course, such problems do not always have the structure of sudoku. For example, often this kind of problems do not restrict one worker to performing at most one activity of a kind. Even the number of workers needed to perform an activity is seldom restricted to one. In such cases using the constraint of difference to solve such problem will not work and a global cardinality constraint must be applied to each row and column. However, one symmetric cardinality constraint for each activity will easily tackle to problem.

More complex problems, where each person can be assigned to more than one activity or a responsibility area during one time slot and where one activity or role may require more than one person to be performed can be handled neither by a constraint of difference nor by a global cardinality constraint. In such cases it is necessary to model the problem with symmetric cardinality constraints. This kind of problem often appears in creating multi-skilled teams as well as in managing human resources in health care, shops and other similar areas.

In many personnel scheduling problems, assigning a worker to a task is attributed by a specific cost. Often, it is desirable to minimize the overall cost of such assignment or ensure that such costs do not exceed a specific bound. This kind of problem can be modeled with a symmetric cardinality constraints extended with a cost feature, which is introduced in Paper C.

1.4 Thesis Outline and Contributions

This thesis contains revised versions of four, previously published articles.

Paper A W. Kocjan, "Constraint Satisfaction and Dynamic Scheduling". This is the revised and extended version of the paper W. Kocjan, "Dynamic Scheduling. State-of-the-art report" published as SICS Technical Report T2002:28, 2002. First of all, the paper has been limited to constraint satisfaction for dynamic problems. Moreover, it was extended with a general description of global constraints and a description of methods proposed for handling dynamic global constraints.

Paper B W. Kocjan, P. Kreuger, "Filtering Methods for Symmetric Cardinality Constraints", published in "CPAIOR 2004. Conference Proceedings", Lecture Notes in Computer Science 3011, Springer 2004, pages 200-208.

Paper C W. Kocjan, P. Kreuger, B. Lisper, "Symmetric Cardinality Constraint with Costs", previously published as MRTC Report, Mälardalen University, 2004

Paper D W. Kocjan, P. Kreuger, B. Lisper, "Maintaining Consistency of Dynamic Cardinality Constraint with Costs". This is extended and revised version of the paper published as MRTC Report, Mälardalen University, 2005.

Paper A contains an introduction to constraint satisfaction and an introduction to dynamic constraint satisfaction. It describes models and methods for solving both static and dynamic constraint satisfaction problems. Dynamic constraint solving has been applied mainly to scheduling problems thus the description of dynamic constraint satisfaction is exemplified mainly by scheduling problems. A few attempts have been made to formulate individual global constraints in the framework of dynamic constraint satisfaction. One of those, using explanations to maintain consistency of the constraint of difference and the flow constraint are of special importance and are discussed in the paper.

Paper B introduces the symmetric cardinality constraint and describes methods for checking consistency and filtering of the introduced constraint. As other cardinality constraint the symmetric cardinality constraint explores underlying flow network structure of the problem. However the introduced constraint generalizes previously known cardinality constraints and extends them into set constraint satisfaction. In contrast to previously defined cardinality constraints, the symmetric cardinality constraint relies on a feasible flow computation instead of a maximum flow and includes the entire residual graph obtained by computing a feasible flow in the filtering phase.

Paper C extends the symmetric cardinality constraint with a cost function by attaching a cost to an occurrence of each value in a set assigned to a problem variable. The consistency of the constraint as well as the filtering method relies on computing a minimum cost flow in the underlying flow graph. The paper describes also the usage of a shortest path algorithm as a filtering method of the constraint.

Finally, Paper D describes methods for maintaining consistency of cardinality constraints with costs in the framework of dynamic constraint satisfaction. To our knowledge it is the first attempt to use sensitivity analysis for a standard minimum cost flow for maintaining consistency of a global constraint. Moreover, the paper exploits a version of a dynamic shortest path algorithm as a filtering method. The described methods are evaluated on randomly generated benchmarks.

Author Contributions

Paper A is entirely authored by W. Kocjan. Paper B. is written in cooperation between W. Kocjan and P. Kreuger. The idea of the constraint, theorems and proofs, methods etc. comes from W. Kocjan. P. Kreuger contributed with supervision during writing the paper and feedback and discussions about the methods and form of the paper. The contribution of P. Kreuger were especially important while developing the filtering methods for the constraint.

W. Kocjan is the main author of Paper C. P. Kreuger and B. Lisper contributed to the paper by their continuous supervision and feedback on the described methods as well as the form of the paper.

W, Kocjan is the main author of Paper D. P. Kreuger and B. Lisper contributed to the paper by supervision and feedback on the methods and the form of the paper.

Bibliography

[1] CHOCO http://www.choco.sourceforge.net

[2] ECLIPSE http://www.icparc.ic.ac.uk/eclipse/

[3] ILOG Solver http://ilog.com/products/solver/

[4] E. Freuder. In Pursuit of the Holy Grail In *Constraints*, 1997

[5] Koalog http://www.koalog.com/php/jcs.php

[6] Mozart http://www.mozart-oz.org

[7] J.-Ch. Régin. A filtering algorithm for constraints of difference in csps. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[8] J.-Ch. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 209–215, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[9] J.-Ch. Régin and C. Gomes. The cardnality matrix constraint. In *Internatonal Conference on Principle and Practice of Constraint Programming, Proceedings*, LNCS, pages 572–587, 2004.

[10] SICStus Prolog http://www.sics.se/isl/sicstuswww/site/index.html

Chapter 2

Paper A: Constraint Satisfaction and Dynamic Problems

Waldemar Kocjan

Part of this paper has been included in Waldemar Kocjan, “Dynamic scheduling. State-of-the-art report.” SICS Technical Report T2002:28, 2002.

Abstract

Most of the research literature in the field of constraint satisfaction on static problems, i.e. problems where all input data is known and do not vary over the time. However, many real life problems modeled with constraints, are very seldom static. For example, a solution to a scheduling problem is valid only as long as no outer disruption to an executed schedule occurs. The outer disruptions are often unexpected and difficult to take into consideration while computing solution to a problem.

In this paper we describe how constraint satisfaction is used to tackle problems whose input is uncertain or changes over the time.

2.1 Introduction

During the last years constraint satisfaction has been very successful in solving many complex industrial problems. Especially solving problems like planning and scheduling using this framework has drawn a lot of attention from industry and commerce.

The research literature problems modeled as constraint satisfaction problems are often seen as problems with a perfect, well defined data. For example, in planning involving human resources it is assumed that every worker will be accessible during well defined time periods and that no planned activity will be delayed or canceled. However in the real life it is seldom the case. These kind of plans often become disrupted by unexpected events like delays, absence of a worker etc.

Constraint satisfaction like any other method need to deal with this kind of disruptions. This report gives an overview of the state of the art in dynamic constraint solving.

2.2 Motivation

2.2.1 A Toy Example

The essence of a dynamic problem can be illustrated by a scheduling problem, i.e. problem of assigning start times to a set of activities. Figure 2.1 (after [57]) gives a toy example of such problem. In the figure tree activities A,B and C are scheduled. They utilize three identical resources. The activities in the schedule are subject to temporal constraints relating their start and end times. Those constraints are satisfied in the current schedule.

The second part of the figure shows a schedule which is rearranged as a response to a reduction of the number of available resources. The modified schedule is also consistent with respect to all temporal constraints.

2.2.2 Commercial Applications

There is a strong commercial need for techniques which tackle problems with uncertain or varying input. This is especially obvious in industrial problems whose solution is dependent on different kind of resources like planning and scheduling. For example, in manufacturing industry, transport and logistics systems which can effectively deal with dynamic problems can not only increase effectiveness but even save a lot of money. System which can effec-

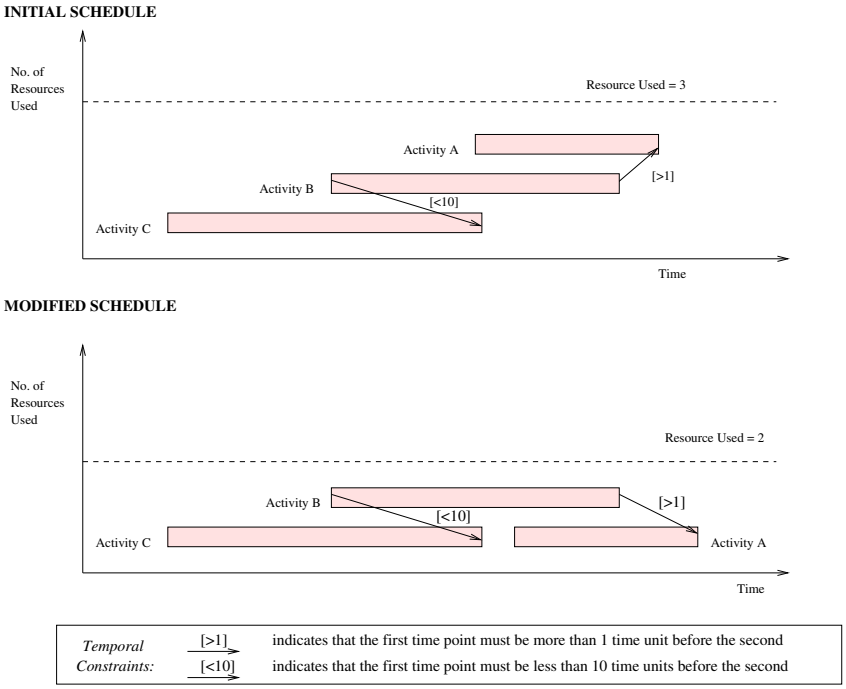

Figure 2.1: Example of a schedule modified to reduce resources

tively manage human resource allocation in e.g. hospitals could dramatically improve results of health care.

Algorithms for dynamic problems should be able to manage any disruption caused by changes in environment. The types of disruption is dependent on the specific application area. E.g., in scheduling for manufacturing industry and in most problems which involves allocating resources to activities we can classify dynamic changes into three major groups:

Activity Changes

Request for new or extended activities can result in resource contention and inconsistency of a schedule. In the long term scheduling introducing new activities can aim at improving the schedule efficiency and degree of resource utilization (e.g. leasing out some resources leads). In the short term scheduling

activities are introduced as they arise (e.g. emergency service). Changes in activity duration and increased level of resource usage can occur.

Resource Changes

Primary reduction of resources (e.g. machine failure) can disrupt a schedule. Resource changes may be also requested to reduce the cost of a schedule (e.g. machine utilization problems). Shorter term resource changes are usually connected with resource failure.

Temporal Changes

The most frequent form of temporal change is a contraction of schedule horizon. Long term temporal changes (e.g. changing a timetable in public transport for regularity) and short time changes (e.g downstream effect of delayed aircraft or train) may also cause schedule inconsistency.

2.3 Related Work

Constraint satisfaction in uncertain and dynamic environments is subject of recently published survey [63]. This paper presents a general description of the problem and short overview of methods for solving such problem without in–depth description of any technique.

Scheduling under uncertainty was also surveyed by Davenport and Beck in [15]. Their paper gives a short classification and a brief description of different off–line and on–line scheduling techniques. Many of the methods described in the survey were used in scheduling real–life problems. This paper includes also, but is not limited to dynamic constraint satisfaction methods.

An introductory survey of on-line scheduling was also given in [52]. This survey concentrates on theoretical problems of on-line job–shop scheduling and does not cover large scale, complex, optimization problems.

2.4 Structure of this paper

In Section 2.5 a short introduction to constraint satisfaction and dynamic models originating from it is given. The section briefly presents methods for solving those models.

Section 2.7 describes some dynamic problems, mainly from the area of scheduling, which often arise in industry. The way of formalizing those problems, using models described in Section 2.5, is presented.

The idea of cooperative solvers is explained in Section 2.9. The chapter exploits mainly so called probing methods originating in operative aircraft planning.

Finally, Section 2.10 concludes this report and gives directions for future work.

2.5 Constraint Satisfaction Models

This section presents the constraint satisfaction problem and dynamic constraint satisfaction models originating from it. The first section describes some basic concepts of CSP and presents methods to solve it. The following sections describe how the idea of CSP is extended to handle dynamic problems involving permanent (Dynamic CSP) and temporary changes (Recurrent Dynamic CSP) in environment. In most of the examples we use scheduling problems to illustrate described models.

2.5.1 Constraint Satisfaction Problem

Constraint satisfaction problems are problems of assigning values to the variables restricted by certain conditions. For example, a graph coloring problem can be represented as a CSP. In that case the nodes of a graph are variables of the CSP and the colors which can be assigned to a node represent values in the domain of every variable. An arc between two nodes represents the constraint that those two nodes may not take the same color value.

Basic definitions

Before we give a formal definition of CSP we need to introduce some basic concepts. The definitions below follow those given in [59] if not stated otherwise.

Definition 2.5.1. *The **domain of a variable** is a set of all possible values that can be assigned to the variable.*

The domain of variable x is denoted D_x.

Definition 2.5.2. *A **label** is a variable–value pair that represents the assignment of the value to the variable.*

In this paper we use $< x, v >$ to denote the label assigning value v to the variable x, where $v \in D_x$.

Definition 2.5.3. *A **compound label** is the simultaneous assignment of values to the set of variables.*

A compound label is denoted as $(< x_1, v_1 >, < x_2, v_2 >, \ldots, < x_n, v_n >)$ and represents assigning $v_1, v_2, \ldots, v_n$ to $x_1, x_2, \ldots, x_n$ respectively.

Definition 2.5.4. *A **constraint** on the set of variables is a set of compound labels on the variables.*

A constraint on the set of variables is generally a restriction on the values which they can simultaneously take. Conceptually, a constraint can be seen as a set that contains all the legal compound labels for the variables of a constraint. A constraint on the set of variables S is denoted C_S.

Definition 2.5.5. *If the variables of the compound label $\mathcal{X}$ are the same as those of the compound labels in a constraint $\mathcal{C}$, then $\mathcal{X}$ **satisfies** $\mathcal{C}$ if and only if $\mathcal{X}$ is an element of C.*

Given these basic definitions we can give here formal definition of a Constraint Satisfaction Problem.

Definition 2.5.6. *A **Constraint Satisfaction Problem** is a triple (Z, D, C) where Z is a finite set of variables $\{x_1, x_2, \ldots, x_n\}$, D is a set of functions which maps every variable in Z to a finite set of objects of arbitrary type and C is a finite set of constraints on the variables in Z.*

As mentioned above the task in CSP is to assign a value to each variable such that all the constraints are simultaneously satisfied.

Definition 2.5.7. *A **solution tuple** of a CSP(Z, D, C) is a compound label $\{< x_1, v_1 >, \ldots, < x_n, v_n >\}$ for all variables $\{v_1 \in D, \ldots, v_n \in D_{x_n}\}$ which satisfies all the constraints $c \in C$:*

2.5.2 Representation

Any given CSP may be represented as a *constraint hypergraph*.

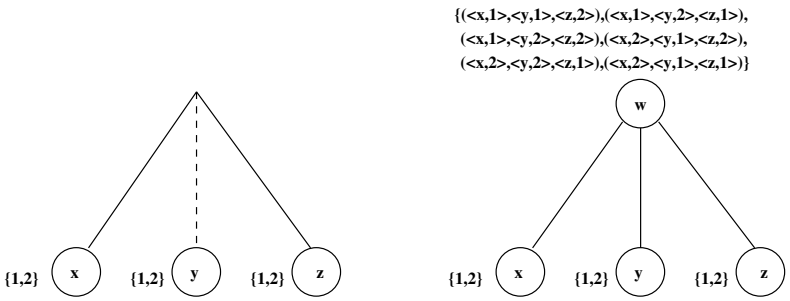

Figure 2.2: Transformation of 3-constraint problem into a binary problem by introducing a new variable w with the specified domain. Labels of x, y and z are projections of the value of w.

Definition 2.5.8. *A **hypergraph** is a tuple $(\mathcal{V}, \mathcal{E})$,where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of hyperedges, each of them connecting a subset of nodes.*

Definition 2.5.9. *The **constraint hypergraph** of a CSP $\mathcal{P} = (Z, D, C)$ is a hypergraph in which each node represents a variable in Z and each hyperedge represents a constraint in C.*

The hypergraph of CSP $\mathcal{P}$ is denoted $\mathcal{H}(P)$.

Since every CSP with k–ary constraint can be transformed into a binary problem, i.e., a CSP with only unary and binary constraints [6, 8, 56], every constraint hypergraph can be reduced to *constraint graph*. An example of such transformations is given in Figure 2.2.

Definition 2.5.10. *The **constraint graph** of a CSP(Z, D, C) is an undirected graph in which each node represents a variable in Z, and for every pair of distinct nodes whose corresponding variables are involved in any k–constraint in C there is an edge between them. The constraint graph of a CSP $\mathcal{P}$ is also called a **primal graph** of $\mathcal{P}$.*

In this paper $G(\mathcal{P})$ will be used to denote the constraint graph of CSP$(\mathcal{P})$.

In some literature originating in Truth Maintenance Systems, e.g. [17], a Constraint Satisfaction Problem is referred to as a Constraint Network. It is said that a binary Constraint Network may be associated with a constraint graph.

Solving methods

Solving constraint satisfaction problem is often carried out by a combination of problem reduction and search. In the literature, problem reduction is often referred to as *achieving consistency*.

Consistency

Consistency algorithms were first introduced for binary constraints problems. Binary CSPs are associated with graphs, where the nodes represent variables and arcs binary constraints. The concept of consistency is related to this representation.

Definition 2.5.11. *A CSP is **node consistent** if and only if for all variables, all values in its domain satisfy the unary constraints on that variable.*

Achieving node consistency is trivial. All that is needed is to determine if each element in each domain satisfies a unary constraint [41, 42, 37].

Definition 2.5.12. *An arc (x, y) in the constraint graph of a CSP(Z, D, C) is **arc–consistent** (AC) if and only if for every value a in the domain of x which satisfies the constraint on x, there exists a value in the domain of y which is compatible with $< x, a >$.*

There exists several methods for achieving arc–consistency. The most naive approaches, *AC–1* and *AC–2*, initiate a queue of all edges and checks for each element in the queue if for each element in domain of variable x there exists a value in domain of variable y, where x and y are subject to the binary constraint C_{xy}, such that C_{xy} holds [37, 64].

More effective algorithms *AC-3*[37], *AC-4* [39] and up to *AC-7* [7] have been developed. All of them check for consistency for all values of $x, y \in C_{x,y}$. They differ on the level of specialization and implementation.

Since arc consistency algorithms referred above check for consistency for every value in domain of one variable with the values of another variable in the constraint, they are usually very costly with respect to run time. A more effective method of propagation, so called *arc B–consistency*, was developed for numerical CSP.

A numerical CSP is a subset of CSPs which has numerical variables connected with domains, which are sets of numerical values and which are subject to numerical constraints. In case of arc B–consistency the consistency check is applied only on the bounds of variable domain. The formal definition of the

arc B–consistency was first given in [34]. However, the method was used in practise long before its formal definition was given.

Definition 2.5.13. *Let $\mathcal{P}$ be a numerical CSP, X a variable of $\mathcal{P}$, $D_X = [a, b]$. D_X is **arc B–consistent** if and only if for all constraints $C(X_1, \ldots, X_k)$ over X holds that:*
1. $\exists v_1, \ldots, v_k \in D_1 \times \ldots D_k : C(a, v_1, \ldots, v_k)$ is satisfied
2. $\exists v_1, \ldots, v_k \in D_1 \times \ldots D_k : C(b, v_1, \ldots, v_k)$ is satisfied
An NCSP is arc B–consistent if all the domains are arc B–consistent.

More informally arc B–consistency is a form of arc consistency restricted to the bounds of the domain.

Backtrack Search

Solving CSP propagation methods are usually combined with search. The basic algorithm of searching for a solution tuple is *simple backtracking*, the method widely used in problem solving. In CSP context, the basic operation is to pick one variable at the time, and consider one value for it at the time, making sure that the newly picked label is compatible with all the labels picked so far. Assigning a value to a variable is called *labeling*. If the current variable with the picked value violates some constraints (consistency check), then the alternative value, if available, is chosen. If all the variables are labeled, then the problem is solved. At any stage, if for some variable there is no value which can be assigned to it without violating any constraints, the label which was last picked is revised and an alternative value, if available, is picked. This search is carried until a solution is found or all the combination of labels have been tested and have failed, which indicates that the problem has no solution.

For more detailed description of backtrack search see e.g. [59].

Global constraints

One weakness of consistency based methods is that all the primitive constraints are examined in isolation from each other. In many cases knowledge about other constraints can dramatically improve domain pruning. Consider the `all-different(`$\{V_1, \ldots, V_n\}$`)` constraint [48], which holds whenever each of the variables V_1 to V_n in its argument takes a different value. Consider X, Y, Z with domains $D_X = D_Y = D_Z = \{1, 2\}$ to be arguments to `alldifferent`. It is obvious, that there is no such assignment to X, Y and Z, which satisfies such conditions, so the consistency check should return *false*.

Nevertheless, if the constraint is propagated using arc–consistency techniques, Figure 2.3, then, for every pair of variables $(X, Y), (X, Z), (Y, Z)$, for every value in the domain of one variable there exists some value in domain of the other such that inequality holds. To discover inconsistency for constraints

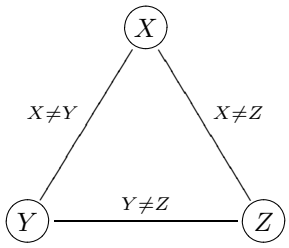

Figure 2.3: The `alldifferent` constraint

of `alldifferent` type it is necessary to take into consideration information about all variables which are subject to such a constraint and their domains. For this purpose a maximal bipartite matching algorithm which matches variables against all values was developed [35, 48]. E.g. the maximal bipartite matching algorithm for the `alldifferent` example will determine that the most 2 variables can be assigned legal values, so the constraint is unsatisfiable, Figure 2.4.

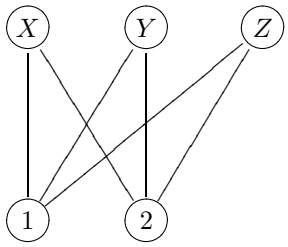

Figure 2.4: Matching variables against values

A specialized group of constraints, so called global constraints, which uses

various algorithms to achieve higher level of consistency, was developed. Many of global constraints, e.g. `serialize, cumulative, diffn` [13, 14, 1, 3] etc., were developed to solve complex scheduling and geometrical problems [4, 32].

2.5.3 Dynamic CSP

The Constraint Satisfaction model presented in the previous section applies to the problems, i.e. problems which require a one time solution of a system representing all the available information.

An extension of CSP which deals with uncertainty and dynamic changes to a problem was presented as Dynamic Constraint Satisfaction Problem (DCSP) by Dechter and Dechter in [17]. The concept of DCSP was introduced in context of Truth Maintenance Systems [19], but can be easily adopted to Constraint Satisfaction Problem. On the relation between Truth Maintenance System and CSP see [38, 18].

Definition 2.5.14. *A **Dynamic Constraint Satisfaction Problem** is a sequence of CSPs, where every CSP is a result of changes in the preceding one. A difference between two consecutive CSPs is expressed by the set of constraints added C_{add} and constraints deleted from the problem. C_{del}.*

As mentioned above a constraint satisfaction problem is a static problem. A dynamic constraint satisfaction problem is modelled as a sequence of static problems, which suggest that the changes in the scheduling environment are more or less permanent.

Most of the methods based on DCSP, solve each new problem separately as an ordinary, static CSP, using the original problem to e.g. measure the discrepancy between the new and the original solution (see [31]).

Nevertheless, solving every new CSP, created as a result of an alternation to the original problem, from scratch has two important drawbacks: inefficiency, and possible instability of the new solution in face of new changes to the problem. Several methods were proposed to remedy these drawbacks. The following three groups of algorithms can be distinguished:

- incremental search methods, [29];
- local repair methods, e.g. [40, 62];
- constraint recording methods, which records any kind of constraint which can be deduced in the framework of a CSP and reuse this information when solving any new CSP. [29, 62, 61].

This tree classes of algorithms for dynamic CSPs are described below.

2.5.4 Incremental search methods

The main idea of incremental search methods for a sequence of CSPs is to preserve the execution context of a problem and use it while solving subsequent CSPs.

The incremental search for dynamic CSPs was first presented in [60] and developed later in [45, 11, 12, 43]. These approaches achieve incrementality using backtracking search. The incremental search based on backtracking works as follows. For any given DCSP $\mathcal{P} =< \Theta_0, \ldots, \Theta_n >$, first Θ_0 is solved as an ordinary, static CSP and the solution α_0 is obtained. Then, for any given Θ_i, if the solution α_{i-1} is violated by any added constraint, the program backtracks to the latest decision point. Another solution is then generated and tested until all of the constraints in Θ_i are satisfied or the search fails.

The method has some severe deficiencies pointed out in [29]. One is connected with the fact that the backtracking point selected when the constraints are not satisfied, has probably nothing to do with the cause of constraint violation. This results in a lot of redundant computation, which could be avoided by choosing more accurate backtracking point.

To remedy this problem the authors of [29] propose a method based on re–execution, where the search for a solution to a new problem is guided by informations gathered during solving previous problems. The gathered information consists of the computation path leading to a solution. The specific computation path used to guide the search is called an oracle and is denoted $O(\Theta_i)$. In the framework presented in [29] the oracle $O(\Theta_i)$ is associated with the set of constraints of CSP Θ_i. These oracles will be used during re–execution to achieve incrementality.

In a case of addition of constraints, the solution to the new CSP $\Theta_i \in \mathcal{P}$ can be found by following the oracle $O(\Theta_{i-1})$ until a failure, due to the added constraint, occurs or a solution is found. If a failure occurs, execution goes back to the recent choice point and proceeds from there in the standard way, i.e. without any oracle.

Since the node where backtracking occurs is associated with the first choice point where the constraints are not satisfiable, this point will be a best possible backtracking point (for a proof see [29]).

In the case of deleting some constraints an oracle which achieves most pruning without losing any solution is chosen. The basic scheme for deletions proceeds with oracle O in the same way like for additions. Note that the addi-

tion of constraints must be now refined to pick up the best oracle. While the last oracle was the best when only additions were concerned it is no longer true when additions are allowed.

The re–execution approach presented in [29] outperforms incremental search based on the backtrack search. However, it should be noted that the same pruning can be achieved without using an oracle, simply by the replacing the re–execution scheme by a backjumping mechanism (for more information on backjumping see [59]).

Local repair methods

The second class of algorithms which deals with inconsistencies of a solution caused by dynamic changes in a problem is based on the local repair methods. The basic idea behind this approach can be described as follows. Any solution which became infeasible because of addition of constraints, may be gradually impoved until the new solution is acceptable. Deletion does not influence feasibility of a solution, however it might influence its optimality.

An example of a method using local repair was presented by Verfaillie and Schiex in [62] and was exemplified by a scheduling problem. The main idea behind this algorithm can be intuitively described in following way. Variables which violate some constraint may be removed from the current solution. All removed variables can be then gradually re–entered into the solution. Re–entering a variable t into a solution is performed even if t can be assigned a value in such way that all the variables incompatible with such assignment can be removed and gradually re–entered, without modifying the value of the t.

More formally, given a CSP, let A be a consistent assignment of a subset Z of variables. Let v be a variable which does not belong to Z. The variable v can be assigned, i.e. a consistent assignment $Z \cup \{v\}$ can be obtained, if and only if there exists a value val such that val can be assigned to v, then all the assignments
$< v', val' >$ which are incompatible with assignment $< v, val >$ can be unassigned and reassigned one after another without modifying the assignment of v.

An evaluation of the algorithm was conducted on randomly generated CSPs [62]. The local changes procedure was used together with the simple heuristics for choosing variables and values for assignment. When choosing a variable for assignment the variables with smallest domains are chosen first.

The evaluation shows that the algorithm performs best on the least and most constrained problems. Nevertheless, the algorithm is inefficient on intermedi-

ate strongly constrained problems.

Another heuristic repair method called **min–conflicts** was presented by Minton et al. in [40]. Given a DCSP $\mathcal{P} =< \Theta_1, \ldots, \Theta_n >$, let α_i be a solution to the Θ_i and Θ_{i+1} a CSP which differs by the set of constraints added or removed from the problem. To find a solution to Θ_{i+1} the method starts with the initial, complete but inconsistent assignment, which is this case is a solution to the proceding problem. It is said that two variables are in *conflict* if their values violate a constraint. At each choice point during the search, the heuristic chooses an arbitrary variable which is currently in conflict and, if possible, reassigns its value, until a solution is found. The system thus searches the space of possible assignments, favouring assignments with fewer total conflicts.

The described method is usally used with the hill climbing search, presented in Algorithm 1. The min–conflics heuristics is then applied at every search step of the hill climbing algorithm. The variable $conflicts$ is a measure of a number of conflicts.

Algorithm 1 Hill climbing

```
Require: a problem
  current ← Initial state (problem)
  loop
    next ← a highest value successor of current
    if conflicts[next] > conflicts [current] then
      return current
    else
      current←next
    end if
  end loop
```

Hill climbing procedure combined with the minimum conflicts heuristics was successful in handling dynamical changes in scheduling observations of the Hubble telescope and another problems [40]. Nevertheless, several factors limit the applicability of the method. The most important factor is that all the conflicts are assumed to be independent. For highly structured problems minimum conflicts heuristics performs poorly. Moreover, since this method ignores all fine structure in the problem, there exists possibility of pathological configurations occuring during the search procedure. This manifests itself for example in “cycles” where the same variable is repaired again and again without any progress towards finding a solution.

Constraint recording methods

A third group of algorithms which deal with dynamic DCSPs includes the methods which record earlier solutions (information) and reuse them to solve subsequent problems. Some of the methods mentioned before, like incremental search presented in [29], depends on recorded information.

An example of an algorithm which uses information recorded during the search is *Dynamic Backtracking(ddbt) for DCSPs*. This method was introduced by Verfaille and Schiex in [61]. The algorithm is extension of *dynamic backtracking(dbt)* presented by Ginsberg in [26]. The dynamic backtracking algorithm (dbt) is presented first.

The dynamic backtracking algorithm, Algorithm 2, is based on the idea of recording information about earlier search phases. At each step of the search, for every variable v and each value val, which has been eliminated from the current domain of v, sets of previously assigned variables responsible for this elimination are recorded. These sets are called *eliminating explanations* and require $O(n^2d)$ space to be recorded, where n is a number of variables and d is the maximum domain size. The *conflict sets* of a variable is the union of eliminating explanations and all its eliminated values. If a domain of a variable v is empty and v' is a last variable recorded in the conflict set of variable v then the algorithm backtracks to v', unassigns it but does not assign any new variable. Then, all the eliminating explanations which contain variable v' must be removed and their values need to be returned to their corresponding current domains. Moreover, new eliminating explanations are created for the value val' previously assigned to v'.

The procedure `backward-checking(`Z, v`)` (Algorithm 3 checks the current value of v against the current values of the variables of Z and returns the first failing constraint, if any, or $success$, if none. Procedure `create-eliminating-explanations(`v, val, Z`)` records the set Z of variables as an explanation for the elimination of the value val from the current domain of the variable v. For all the variables in Z, `remove-eliminating-explanations(`v, Z`)` removes the eliminating explanations which contain v and returns the corresponding values in their corresponding current domains.

The Dynamic Backtrack algorithm can be easily extended to take into account constraints in eliminating explanations. This requires adjusting the `dbt-value` and `dbt-bt-values` procedures. These procedures are modified as in Algorithm 4.

The eliminating explanations in Algorithm 4 are composed of two parts: the set of previously assigned variables (the *variable eliminating explanation*)

Algorithm 2 Dynamic Bactracking

Procedure: `dbt` ($\mathcal{P}$)
  Require: CSP $\mathcal{P} = (Z, D, C)$
  return `dbt-variables` ($\emptyset, Z$)
. . . . . .
Procedure: `dbt-variables` (Z_1, Z_2)
  Require: Z_1 is a set of assigned variables
  Require: Z_1 is a set of unassigned variables
  if($Z_2 = \emptyset$)**then**
    return *success*
  else
    choose $v \in Z_2$
    $d = D(v)$
    if `dbt-variable` (Z_1, v, d) $= failure$ **then**
      return `dbt-bt-variable` (Z_1, Z_2, v)
    else
      return `dbt-variables` ($Z_1 \cup \{v\}, Z_2 - \{v\}$)
    endif
  endif

Algorithm 3 Variable and value choosing procedures

Procedure: `dbt-variable` (Z_1, v, d)
  if $d = \emptyset$ **then**
    return *failure*
  else
    $val = d_x \in d$
    if `dbt-value` $(Z_1, v, val) = success$ **then**
      return *success*
    else
      return `dbt-variable` $(Z_1, v, d - \{val\})$
    endif
  endif

Procedure: `dbt-value` (Z_1, v, val)
  `assign-variable` (v, val)
  $c =$`backward-checking` (Z_1, v)
  if $c = success$ **then**
    return *success*
  else
    $Z_3 =$ set of variables of c
    `unassign-variable` (v)
    `create-eliminating-explanation` $(v, val, Z_3 - \{v\})$
    return $failure$
  endif

Procedure: `dbt-bt-variable` (Z_1, Z_2, v)
  $Z_3 =$ conflict set of v
  if $Z_3 = \emptyset$ **then**
    return $failure$
  else
    $v' =$ last variable of Z_3 in Z_1
    $val' =$ current value of v
    $Z_4 =$ set of variables following v' in Z_1
    `unsign-variable` (v')
    `create-eliminating-explanations` $(v', val', Z_3 - \{v'\})$
    `remove-eliminating-explanations` $(v', Z_4 \cup Z_2)$
    `dbt-variables` $(Z_1 - \{v'\}, V_2 \cup \{v'\})$
  endif

and the set of constraints (the *constraint eliminating explanation*). The procedure `create-eliminating-explanation` (v, val, Z, C) records the set Z of variables and the set C of constraints as an explanation for the elimination of the value val from the current domain of variable v.

Algorithm 4 Modified Dynamic Backtracking

Procedure: `dbt-value` (Z_1, v, val)
  `assign-variable` (v, val)
  $c =$`backward-checking` (Z_1, v)
  if $c = success$ **then**
    return $success$
  else
    $Z_3 =$ set of variables of c
    `unassign-variable` (v)
    `create-eliminating-explanation` $(v, val, V_3 - \{v\}, \{c\})$
    return $failure$
  endif

Procedure: `dbt-bt-variable` $(Z_1, Z_2 v)$
  Z_3 is a union of all the *variable eliminating explanations* of v
  C is a union of all the *constraint eliminating explanations* of v
  if $Z_3 = \emptyset$ **then**
    return C
  else
    v' is last variable of Z_3 in Z_1
    val' is its current value
    Z_4 is the set of variables following v' in Z_1
    `unassign-variable` (v')
    `create-eliminating-explanation` $(v', val', Z_3 - \{v'\}, C)$
    `remove-eliminating-explanations` $(v', Z_4 \cup Z_2)$
    `dbt-variables` $(Z_1 - \{v'\}, Z_2 \cup \{v'\})$
  endif

Solving CSPs using the Dynamic Backtracking algorithm produces the following:

- when the CSP is *consistent*:
  - a *solution*, i.e. a complete and consist assignment;

  - a set of *eliminating explanations* related to possible variable assignments.

- when the CSP is *inconsistent*:

  - a partial consistent assignment, which can be empty;
  - an *inconsistency explanation*
  - a set of *eliminating explanations* related to some of the possible variable assignments.

These results can be recorded and reused when solving a new CSP which differs from the previous one by the sets of added and deleted constraints (see [61]). It can often be expected that most of the recorded explanation remain valid for the new CSP. This explanation recording imported from a previous CSP should allow significant pruning of the search space of the new CSP. The new algorithm with necessary adjustments for Dynamic CSPs is presented as Algorithm 5.

Algorithm 5 Dynamic Backtracking for DCSPs

Require: Z_1 is a set of assigned variables, result of the previous search
Require: Z_2 is a set of assigned variables, result of the same search
Require: C_{add} is a set of added constraints
Require: C_{del} is a set of deleted constraints
  $Z = Z_1 \cup Z_2$
  $Z_3 =$ `remove-assignments(`Z_1, C_{add}`)`
  `remove-variable-eliminating-explanations(`Z_3, Z`)`
  `remove-constraint-eliminating-explanations(`C_{del}, Z`)`
  return `dbt-variables(`$Z_1 - Z_3, Z_2 \cup Z_3$`)`

The three procedures `remove-assignments`,`remove-variable-eliminating-explanations` and `remove-constraint-elimi--nating-explanations` aim at deriving a consistent point for the new search, the procedure `dbt-variables`.

For each constraint $c \in C_{add}$, let $Z(c)$ be a set of its variables. If the assignment of the variables of Z_1 violates c, the procedure `remove-assign--ments(`Z_1, C_{add}`)` unassigns the last variable of $Z(c) \in Z_1$ and creates the corresponding eliminating explanation. The procedure returns the set Z_3 of the variables which have been unassigned.

Experimental evaluations of the algorithm show that Dynamic Backtracking for DCSPs run on the randomly generated benchmarks for dynamic problems with small and intermediate changes clearly outperforms other algorithms based on recording previous assignments. However, this method does not perform well on problems which include major configuration changes. In such situation explanations based on previous CSP become completely incompatible with the new problem.

2.5.5 Explanations for Global Constraints

As mentioned in the previous section, an explanation is a constraint justifying propagation events generated by a solver, such as value removal, bound update and contradiction. More formally [23]

Definition 2.5.15. *An **explanation** of an inference $\mathcal{X}$ consist of a subset of original constraints ($\mathcal{C}' \subset \mathcal{C}$) and a set of instantiation constraints, i.e. choices made during the search $d_1, \ldots, d_k$ such that: $\mathcal{C}' \wedge d_1 \wedge \ldots \wedge d_n \Rightarrow \mathcal{X}$.*

Generating explanation for binary constraints is fairly simple. Given a pair of variables x and y, a value $a \in D_x$ is removed from domain of x if and only if all supporting values for a in domain of y regarding a constraint C_{xy} has been removed. Existence of such support is computed by any of the arc consistency algorithm.

Dealing with global constraint is somewhat more complicated. Generally, global constraints use special purpose algorithms to establish consistency and to filter domains of the constrained variables. Thus generating precise explanations is dependent on the used algorithms.

To illustrate difficulties of generating precise explanations for global constraints consider constraint of difference [48]. This constraint ensure that to each constraint variable is assigned a different value. The consistency of the constraint is computed by computing maximum cardinality bipartite matching in a bipartite graph $G(V_1 \cup V_2, E)$, where V_1 is a set of nodes representing the constraint variables and V_2 is a set of nodes representing union of values in domains of all variables. Moreover, E is a set of arcs between nodes in V_1 and V_2. There exists an arc between $x_i \in V_1$ and $v_j \in V_2$ if value represented by node v_j appears in domain of a variable represented by node x_i.

Computing maximum cardinality matching in G is equivalent in computing a maximum flow in a flow network constructed from G, where the flow between any node is restricted to an integer in the interval $[0, 1]$. If obtained

flow is equal to the number of constraint variables then such constraint is consistent.

By the definition of the constraint a value in domain of a constraint variable is consistent if there is a maximum flow in the flow network obtained from G which contains arc (x_i, v_j). Given a maximum flow f we can compute the consistency of any value v_j of a variable x_i by resolving if there exists a strongly component in the residual graph of f which involves at least three nodes, which contains nodes representing both v_j and x_i.

It is clear that the classical constraint of difference only needs information about different strongly connected components in an adequate graph. However, to justify removal of a value from a domain of a variable we not only need to know that two nodes are in different components but also we need to explain why. In order to do this we need to keep track on the links between different components. Figure 2.5 illustrates such case. To avoid removal of the link between component $i + 1$ and i, at least one of the links marked with dashed lines must be present.

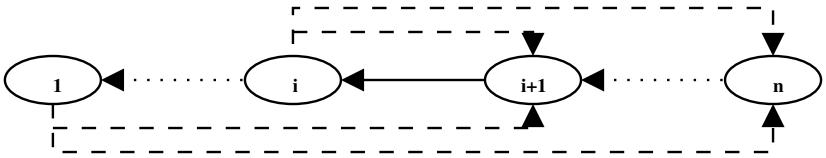

Figure 2.5: Explanation for removing the edge between components $i + 1$ and i: to avoid the removal at least one of dashed lines must be present

Similar explanations are also used to maintain consistency of constraints, which use more general flow formulation, namely flow constraint [9]. This constraint is meant to maintain a feasible flow through the defined network. The explanations generated here are based on the properties of flows. The first property, so called flow conservation property, states that for any node in the flow graph except the source and the sink, the amount of incoming flow must be equal to the amount of flow leaving such node. This generates a straightforward explanation for establishing lower and upper bounds for the flow through each individual arc in the graph.

The second property which is used for computing explanation is maximum flow/minimum cut property. The s–t cut is a partition of a flow graph into two subset of nodes S and $\overline{S}$ such that the source node $s \in S$ and the sink node $t \in \overline{S}$. The maximum flow/minimum cut property states that maximum flow

from s to t equals the minimum capacity among all s–t cuts. Given a maximum flow in a flow graph and using this property explanations of a maximum flow can be generated in terms of capacity and lower bounds of arcs in the cut. The same principle can be used for computing minimum flow in a graph.

The experimental evaluation performed on instances of rostering problems shows that the explained version of flow constraint is slower than the classical one. However in a more general aspect, explanations provide the tool for ex. interaction with a user, which can be very valuable in finding solution to dynamic problems.

It is also evident that it is very hard to define some general method for dynamic global constraints, without taking into consideration specific features of the special purpose algorithms used for computing consistency and filtering of specific global constraints.

2.6 Recurrent Dynamic Constraint Satisfaction Problem and Stable Solutions

As mentioned in Section 2.5.3 a dynamic CSP model can be used to capture changes to a problem but it assumes that those changes are permanent. However, there are situations where changes in the current problem are only temporary. Machine breakdown and employee absence are examples of such temporary changes.

The sequential nature of DCSP is a major drawback when using DCSP to model problems where temporary changes occur. Because DCSP is a sequence of CSPs then any change which causes infeasibility of a solution demands solving a new CSP. Such a situation is highly undesirable. Ideally, one wouldn't like to lose the original solution in the first place, which would forestall all extra search and other undesirable aspects of solution failure.

To remedy this Freuder and Wallace introduce the notion of Recurrent Dynamic CSP (RDCSP). This model takes into consideration temporary, recurrent changes to the problem [66, 67]. The aim of the model is to find solutions which are *stable*, i.e. which remain valid after the problem was altered.

The following criteria were used for evaluating the performance of the model:

1. efficency in finding a new solution,
2. solution similarity or consistency, i.e. a new solution should share as many values as possible with the old one, and

3. solution stability in face of problem alternation.

Since changes to the problem are unpredictable and can occur during a search, i.e. before a new solution, which replaces the lost one, is found, a heuristic repair method was chosen. This method is combined with min–conflicts heuristics extended by a random walk strategy. Min–Conflicts heuristic is defined as follows [40]:

Definition 2.6.1.
Given: *A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables* ***conflict*** *if their values violate a constraint.*
Procedure: *Select a variable that is in conflict, and assign it a value that minimizes number of conflicts. Break ties randomly.*

A random walk strategy makes an assignment to the variable in conflict regardless of whether the new assignment is better or worse than the previous one. The assignment is made according to the fixed probability. Randomization of assignment aims at getting min–conflicts out of local minima. This method is reported to be effective in finding global minima for random CSP [65].

Using hill climbing, which is a general iterative repairment technique, also makes it possible to incorporate the changes in the scheduling problem which appear during search for new solution.

The general hill climbing algorithm was presented in the Algorithm 1 in Section 2.5.4. The Algorithm 6 presents hill climbing technique for stable solutions.

The basic procedure of search must be extended by collecting information about changes in environment and incorporating those changes into the search procedure. Before we describe how it is done we need to present the basic properties of RDCSP. They are as follows:

- Values and/or variables may become insignificant or inconsistent and constraints may be added temporarily.
- There are differences in likelihood of changes which can occure,
- Those differences are assumed to be known *a priori*.

Collecting information about the changes will help to avoid choosing values for assignment that are likely to make the current solution invalid in the future. Information about changes is collected as they occur and is used to decide which value to assign to a given variable during subsequent search. The procedure of gathering information can be carried out in following ways:

Algorithm 6 Basic hill climbing for stable solutions

{PREPROCESSING}
For each successive variable
choose a value that minimizes conflicts with variables already assigned

{HILL CLIMBING }
repeat
1. randomly select variable x with ≥ 1 conflicts
2. with probability p
choose value v at random from the domain of x
2'. with probability $1 - p$
find all min–conflicts values in domain of x
choose value v at random from min–conflicts set
3. assign v to x

until cutoff time reached or complete solution found

1. changes may be traced directly by e.g. recording inconsistent values, or

2. solution loss resulting from the problem change may be traced and related to the values participating in solutions of varying stability.

The method described in [66] tracks changes by counting the number of times that an element changes. Those counts can be related to relative frequencies of changes, which in turn gives information about the probability that given change will occur in the future. Another essential feature of counting changes is that it provides a total ordering among the variables, whose assignments have become inconsistent.

Methods for incorporating the gathered information into the search consider two cases.

The first one is that of value loss since the count of value loss, or "penalties" relate directly to min–conflict procedure. The method adds the penalty for each value to its current conflict at the point where conflicts of the values in domain are compared. This is called `penalty-adding`, Algorithm 7a.

Since using penalties in their original form can bias the hill climbing procedure heavily, they are rescaled relatively to the penalty of the variable being repaired.

The second method, `penalty selection` in Algorithm 7b, works as

follows. After the set of min–conflicts is chosen in standard way, the value from this set is chosen at random, but the assignment is delayed unless its penalty is less than some criterion or cutoff value. This process is repeated until an acceptable value is found or all values have been tested.

In both cases, a selection or cutoff method is used with the random walk, i.e. if a value was chosen at random, then values are chosen successively and at random from the remaining, untested subdomain until one that matches the selection criterion is found. If no such value is found the random walk procedure is terminated.

Algorithm 7 Penalty ajustment for min–conflicts heuristics

a.PENALTY–ADDING
  2'. with probability $1 - p$
    find all values in domain of x with minimum (conflicts + scaled penalty)
     choose value v at random from this biased min–conflicts set

b. PENALTY SELECTION
  2'. with probability $1 - p$
    find all min conflicts values in domain of x
    repeat
      select and remove value v at random from min–conflicts set
    until for $v <$ cutoff or no more values
    if penalty for $v <$ cutoff **then**
      choose v for assignment
    else
      assign a value chosen at random from original min–conflicts set
    endif

To avoid values that violate potential constraints with high penalties the constraint penalties can be incorporated into hill climbing by checking them during value selection, Algorithm 8. In this case, a value chosen from the min–conflict set is tested against potential constraints which are not part of the current problem. If none of the potential constraints are violated or if the penalties of violated constraints do not reach the cutoff value, then the tested value is accepted. Otherwise, another value is chosen. If no acceptable value was found, a value from min–conflicts set is chosen at random.

Such a penalty–based selection may in certain situation lead to overcon-

straining the hill–climbing procedure. In the experimental work presented in [66], the selection criterion was relaxed each time the hill–climbing procedure failed to find a solution after k seconds.

Algorithm 8 Adjustment to min–conflicts caused by constraint addition

```
SELECTION BASED ON CONSTRAINT PENALTIES
  2'. with probability 1 − p
      find all min conflicts values in domain of x
      repeat
        select and remove value v at random from min–conflicts set
        for each constraint not currently present
          if value is not compatible with other value(s) in constraint
          and constraint penalty > cutoff then
            reject value and exit for loop
          endif
      until v not rejected or no more values
      if v not rejected then
        choose v for assignment
      else
        assign a value chosen at random from original min–conflicts set
      endif
```

Wallace and Freuder present the experimental evaluation of stable solution strategies in [66]. They were tested with simulated Recurrent DCSP on random 3–color problems with 100 variables. The probability of value loss was set to high 0.3 and low 0.003. The probability that a domain value was associated with the higher probability of loss, p_p was set to 0.025. In the runtime phase there was 250 occasions of change in which the entire set of relevant elements was scanned deciding for each element if any change should be made to original problem. Both changes to the problem when hill climbing finds a new solution as well as periodical changes at fixed intervals regardless search state was investigated.

In the case of value loss with the values penalized after each deletion both penalty startegies guided the hill climbing to a solution with appreciably fewer "bad" solutions. In addition, adding a penalty to the number of conflicts resulted in solutions of one magnitude better than without penalty. Nevertheless, adding penalties was showed to be more costly in terms of run time. In contrast, for the penalty selection strategy there were no consistent differences between

penalty and non penalty conditions across problems. However, the large difference in run time favours the penalty condition as often as the control.

Compared to ”restarting” the solver after assignment loss, solution reuse was very successful in avoiding bad values in solutions.

Experiments on using constraint addition procedure were carried out using the same problem setup. The average degree of improvement in this case was somewhat better than with value deletion, but the mean run time was 2–3 times greater under penalty conditions. Solution similarity was high in both penalty and non–penalty conditions, with the avarage difference by 0.01 in each tested case.

Experimental evaluation shows that the penalty function improves the general measure of solution quality. Nevertheless, there exists a tradeoff between solution quality and a run time. Reported experiments show, that the same problem configurations run under penalty condition has the mean run time 2–3 times greater. Solution similarity was high in both penalty and non–penalty conditions, differing at average by 0.01 in each case.

It was also concluded that tracking the value loss is an efficient way to gather information for finding stable solutions. Even in the situations where assumption about convergence of underlying probabilities are clearly violated penalties served as a bias for hill climbing procedure led in the direction of more stable solutions.

Finally, it can be concluded, that the stable solution framework is the only attempt within CSP framework to gather information about the frequence and the nature of the changes in the problem and use it later to solve it. The idea of guiding a search for solution to an altered problem by using the information about probability of the changes may be used to effectively prune the search tree of the problem.

2.7 Dynamic scheduling problems

In this section we introduce some general scheduling problems as examples of constraint satisfaction problems. First, a general model for static scheduling problem, Kernel Resource Feasibility Problem, is introduced. Then, Minimal Perturbation Problem as an example of dynamic constraint satisfaction problem is described. Both examples are used in Section 2.9 to illustrate dynamic constraint solving by so called probing methods.

2.7.1 Kernel Resource Feasibility Problem

Different classes of scheduling problems can be generalized as a kernel resource feasibility problem (KRFP)[20]. In the KRFP the goal is to fix start and end times of activities such that the quantities of available resources are not over–allocated.

Basic definitions

Definition 2.7.1. *A **Kernel Resource Feasibility Problem** is a 5-tuple* $(\mathcal{A}, \mathcal{R}, quantity, \mathcal{TC}, T_{max})$*, where* $\mathcal{A}$ *is a set of* n *activities,* $\mathcal{R}$ *is a set of* m *resource types* $r_1, \ldots, r_m$ *,* $quantity$ *is a resource function* $quantity : \mathcal{R} \rightarrow \mathbb{N}$*,* $\mathcal{TC}$ *is a set of temporal constraints and* $T_{max} \in \mathbb{N}$ *is a latest point of scheduling horizon.*

The KRFP contains three major components: activities, resources and time constraints.

The activities of the KRFP are represented as a set of atomic activities $\mathcal{A} = \{A_1, \ldots, A_n\}$. The following assumptions characterize activities of KRFP:

1. they require exactly one resource;
2. they are non–interruptible but the duration may vary;
3. they consume a quantity of resource which does not change through the duration.

Each activity $A_i \in \mathcal{A}$ consists of five elements: r_i which is the name of the required resource, the resource area variable $area_i$, the variable $quantity_i$ representing the quantity of resource r_i, the discrete start point variable s_i and the discrete end point variable e_i.

Temporal constraints ($\mathcal{TC}$) are of one of the following forms:

1. $u \; R \; c$ (bounding constraints)
2. $u \; R \; v \pm c$ (distance constraints)
   where: $R \in \{=, <, >, \leq, \geq\}, u, v \in \bigcup_{a_i \in A} \{s_i, e_i\}$

A solution to KRFP is an assignment of values to the variables in $\mathcal{A}$ where the following constraints are satisfied:

- The activity constraints:

$$\forall A_i \in \mathcal{A} : area_i = quantity_i \times (e_i - s_i) \tag{2.1}$$

- The temporal constraint in $\mathcal{TC}$, including the constraints relating the start and end time of each activity, as well as constraints enforcing schedule horizon: $\{0 \leq s_i \leq T_{max}, 0 \leq e_i \leq T_{max} : A_i \in \mathcal{A}\}$.
- Let $\forall r \in \mathcal{R} : \mathcal{A}_r = \{A_i : A_i \in \mathcal{A} \wedge r_i = r\}$. The resource constraints:

$$\forall r \in \mathcal{R}, \forall t \in \{0, \ldots, T_{max}\} : \; quantity(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} quantity_i \tag{2.2}$$

Representing KRFP as a Constraint Satisfaction Problem

The activity constraints of (2.1) and the temporal constraints $\mathcal{TC}$ can be directly represented as a CSP constraints according to the model presented by El–Kholy and Richards in [20].

In this model, a new resource quantity variable Q_r is introduced for every $r \in \mathcal{R}$. This variable corresponds to the maximum quantity of this resource used over the schedule horizon. For each activity $A_i \in \mathcal{A}_r$, a variable Q_{rs_i} counts the quantity of resource r used at start time s_i. Variable Q_r denotes the maximum of these quantities $Q_r = \mathsf{MAX}\{Q_{rs_i} : A_i \in \mathcal{A}_r\}$. The variables of the Q_{rs_i} are defined in terms of Booleans. A Boolean $B_{s_i A_j}$ is introduced for each pair of activities $A_i, A_j \in \mathcal{A}_r$ and denotes a situation when activity A_j overlaps with s_i.

$$\forall r \in \mathcal{R}, \forall A_i, A_j \in \mathcal{A}_r : B_{s_i A_j} = \begin{Bmatrix} 1 & \mathsf{iff}\ s_j < s_i \wedge s_i < e_j \\ 0 & \mathsf{otherwise} \end{Bmatrix} \tag{2.3}$$

Linked with the corresponding Q_{rs_i}, these Boolean variables link temporal and resource reasoning and the following constraint can be applied:

$$\forall r \in \mathcal{R}, \forall A_i in \mathcal{A}_\nabla : Q_{rs_i} = \sum_{A_j \in \mathcal{A}_r} B_{s_i A_j} \cdot quantity_j \tag{2.4}$$

Even if each Q_{rs_i} is bounded directly by the maximum resource quantity $quantity(r)$ via the constraints $Q_{rs_i} \leq Q_r$ and $Q_r \leq quantity(r)$, introducing variables Q_r will allow us to modify the KRPF by adding constraints.

2.8 Minimal Perturbation Problem

A problem which often arises in a dynamic scheduling environment is creating a minimally disruptive schedule, where disruptiveness is measured as the

distance between two solutions. In the case of scheduling under resource constraints this problem can be transformed into a Resource Utilization Problem (RUP). A hybrid method for solving this problem was presented by El Sakkout et al. in [54, 57, 55].

Definition 2.8.1. *A **minimal perturbation problem** Π is a 5-tuple (Θ, α_Θ, C_{del}, C_{add},δ) where Θ is a CSP, α_Θ is a solution to Θ, C_{del} and C_{add} are constraint removal and addition sets and δ is a distance function, which evaluates the distance between two solutions. This function is used to measure the degree of perturbation.*

A complete assignment is a solution to Π if and only if it is a solution to $\Theta' = (Z, D, C_{\Theta'})$, where $C_{\Theta'} = (C_\Theta \setminus C_{del}) \cup C_{add}$. A solution is optimal if and only if $\delta(\alpha_\Pi, \alpha_\Theta)$ is minimal.

In the context of Definition 2.8.1 a Resource Utilization Problem may be defined as follows:

Definition 2.8.2. *A **Resource Utilization Problem** is a minimal perturbation problem ($\Theta_i, \alpha_i, C_{del}, C_{add}, \delta$)where:*

- *Θ is any KRFP ($\mathcal{A}, \mathcal{R}, \mathcal{TC}, T_{max}$) modeled as a CSP, such that*
  - *there is only one resource type $r(R = \{r\})$ and Q_r is the resource quantity variable for that resource type*
  - $\forall A_i \in \mathcal{A} : quantity_i = 1$
- *α_Θ is a solution to Θ;*
- *$C_{del} = \emptyset$, $C_{add} = \{Q_r \leq c\}$, such that $c < quantity(r)$;*
- $\delta(\alpha_{\Theta'}, \alpha_\Theta) = \sum_{u \in \bigcup_{a_k \in A} \{s_k, e_k\}} |\alpha_{\Theta'}(u) - \alpha_\Theta(u)|$

The definitions above model the problems in terms of Dynamic Constraint Satisfaction Problems described in Section 2.5.3 and create the base of an array of methods described in Chapter 2.9. However, as mentioned in Section 2.6, the Dynamic CSP model is based on the assumption of permanent changes in the scheduling environment. Some of the instances of MPP, like a rescheduling caused by machine failure or by delays in a public transportation system, can contain only temporary changes. It seems more natural to model those cases using the Recurrent Dynamic CSP paradigm. However, as far as we know, there is no investigation on modeling MPP in the context of recurrent dynamic CSPs.

2.9 Probing methods

Most real world scheduling problems which has been addressed in recent years, including those coping with dynamic changes, have characteristics of large scale, complex, optimization problems. An attempt has been made to address those problems by decomposing them into multiple subproblems. The optimal solutions to the subproblems are rarely compatible. Since the optimal solutions of different subproblems are invariably incompatible, other ways of solving subproblems, which make such sub–solutions globally consistent, are being explored. This research topic belongs to an area known as "hybrid algorithms".

This chapter presents most of the work done on hybrid algorithms in the context of dynamic scheduling. The algorithms are complete tree search algorithms which, at every search node, implement a repair step. The repair is performed on *super–optimal* assignments, i.e. assignments which are optimal with respect to the objective function, but which are only partially consistent.

The probing methods presented in this chapter are not intrinsically dynamic. The computation of each consecutive problem is performed from scratch. However, the probing methods can be easily extended with functions which e.g. measure the distance between solutions to consecutive problems of DCSP and can be easily used to handle generic dynamic problems like e.g. minimum perturbation problem etc.

Since the probing method evolved from constraint backtrack we start with a presentation of this method.

2.9.1 Constraint Backtrack

The constraint backtrack algorithm presented in [57] is an extended version of the resource feasibility algorithm from [46]. The algorithm contain an aspect of temporal optimization not present in [46]. The outline of the Constraint Backtrack algorithm is presented here as Algorithm 9.

The backtracking procedure is initiated by parameter `MonitoredConstrs`, which is a set of violated constraints. The algorithm is based on simple depth–first search which calls `PushConstrStore` and `PopConstrStore`. The `PushConstrStore` pushes a decision represented by a constraint onto a constraint store stack and triggers the local consistency propagation. The `PopConstrStore` undoes the decision and the propagation based on it.

Constraints which are subject to contention are filtered by the `constraint_filter` procedure. Filtered out constraints have no impact on resource fea-

Algorithm 9 Constraint Backtrack

```
Require: Set of monitored constraints MonitoredConstrs
  ContentionConstrs = constraint_filter(MonitoredConstrs)
  if ContentionConstrs = ∅ then
    return TRUE
  else
    Constr = select_constraint(ContentionConstrs)
    Decision = select_decision(Constr)
    if PushConstrStore(Decision) then
      if constraint_backtrack(MonitoredConstrs) then
        return TRUE
      end if
    end if
    PopConstrStore
    if PushConstrStore not(Decision) then
      if constraint_backtrack(MonitoredConstrs) then
        return TRUE
      end if
    end if
    PopConstrStore
    return FALSE
  end if
```

sibility in this search branch and are omitted. If all the conflicts are removed the resource feasibility phase passes control over to the temporal optimization phase.

The procedure `select_constraint` selects from the filtered constraint set *ContentionConstraints* the constraint *Constr* with the greatest potential for conflicts. At this point the objective is to reduce conflicts in *Constr*. On backtracking the ordering constraint of `Decision` is revoked and replaced by its negation. If the negation also fails then it is revoked and the algorithm backtracks to an earlier decision.

The algorithm applies arc–B consistency propagation on arithmetic constraints, see Section 2.5.1. The decision is propagated by an interplay between resource and temporal constraints. The algorithm applies an additional look–ahead check: after some resource was not found to be a subject of contention at the latest search decision, a resource usage profile is built for the time horizon interval which a given activity must span over. Other activities are assumed to

take minimal span over the same interval. If the resource usage exceeds the capacity at any time point in this interval, a failure is signaled and the algorithm backtracks to the previous decision point. This procedure was found to be important for solving KRFP problems, which relies on computing resource overlap at activity start times.

The empirical experiments and comparisons with other techniques (see [57]) show that Constraint Backtracking algorithm is quite ineffective at obtaining the minimum and proof of optimality. The propagation methods described above fail to prune efficiently and the objective function does not reduce the search space in a satisfying manner.

The experiments find Constraint Backtracking relatively ineffective in finding the optimum and prove of optimality. The propagation methods are shown to fail in performing efficient pruning in the search tree. In particular, since implications of individual search decisions with respect to the cost are not discovered until most of the variables are fixed, the optimization function fails to efficiently reduce the search space.

2.9.2 Probe Backtrack Algorithms

Probe backtracking is an extended version of constraint backtracking, where the backtrack search procedure is supported by look–ahead procedures, so called *probe generators*, generating potentially good assignments, *probes*.

The purpose of creating a probe is to direct backtrack search and limit the size of the search space. After the probe is created the search concentrates on regions where the probe violates constraints. The main difference between the constraint backtrack search and the probe backtrack search is that the probe backtrack search calls the procedure `obtain_probe_assignment`. It represents a call to the probe generator used to focus search decision at this search node. The only conditions which must be guaranteed by the prober is satisfying the variables' domain constraints. It is preferable that assignments generated by the prober are of good quality, usually super–optimal with respect to the objective function if such a function exists.

The result returned from the probe makes it possible to filter and select monitored constraints. Similarly to the filtering process in constraint backtracking, constraints which are not subject to a conflict are filtered out. Moreover, any constraint which is satisfied by a tentative assignment returned by the prober is removed from the set of violated constraints.

To exemplify the filter mechanism consider the Resource Utilization Problem from Section 2.8. Let $X[Dv]$ to represent the variable X with domain D

Algorithm 10 Probe Backtrack Search

```
Require: Set of monitored Constraints Monitored Constrs
  Assignment = obtain_probe_assignment
  ViolatedContentionConstrs = constraint_filter(MonitoredConstrs,Assignment)
  if ViolatedContentionConstrs == ∅ then
    return TRUE
  else
    Constrs = select_constraint(ViolatedContentionConstrs,Assignment)
    Decision = select_decision(Constr,Assignment)
    if PushConstrStore(Decision) then
      if probe_backtrack(MonitoredConstrs) then
        return TRUE
      end if
      PopConstrStore
      if PushConstrStore not(Decision) then
        if probe_backtrack(MonitoredConstrs) then
          return TRUE
        end if
        PopConstrStore
        return FALSE
      end if
    end if
  end if
```

and a probe value v. A constraint

$$Q_{rs_i}[\{0..1\}1] = B_{s_i\mathcal{A}_j}[\{0,1\}0] + B_{s_i\mathcal{A}_k} : [\{0,1\}0] + B_{s_i\mathcal{A}_l}[\{0,1\}1]$$

is removed even if in contention, because its satisfied by the probe assignment.

Even if constraints satisfied by current probe may be later invalid, it is assumed that the current probe will lead to a solution. If all constraints are satisfied after the probe a solution is found and probe backtrack terminates.

Otherwise a decision is made about the constraint `Constr` which has been violated by the probe assignment to force new probes and obtain assignments that are closer to satisfying `Constr`. Selecting decisions that allow the prober generator to return the same probe would not progress the search.

Some necessary heuristics are used to select a decision about choosing an appropriate constraint *Constr*. In the RUP studied in [57] contention reduction is achieved by forcing apart two temporal variables in the fashion similar

to `CB`. The secondary heuristics is to order decision constraints using least–commitment or estimating the impact of a new ordering constraint on the assignment returned by the probe.

The probe backtracking algorithm generates a probe closely based on the old solution, which is only partially consistent with the new problem constraints. With every call to `obtain_probe_assignment` the local consistency propagation prunes the domain of the temporal variables. Since *MonitoredConstrs*' parameter includes only resource constraints the temporal constraints are not monitored for violation, which makes pure probe backtracking algorithm incomplete.

Completeness of the algorithm is restored by introducing an intermediate phase between the probe backtracking and temporal optimization phases to relieve any remaining contention from resources that were filtered out.

Nevertheless, since the number of temporal constraints increases at every decision step the probes generated in the probe backtracking phase may be of little of none relevance because the tentative values for remaining variables, even if still super–optimal, may violate so many temporal constraints that they no longer provide accurate information about feasible solution.

2.9.3 Unimodular probing

The unimodular probing algorithm [54] combines constraint programming techniques such as constraint propagation with linear programming optimization. It relies on a constraint programming platform which can invoke an LP solver and extract the information about the optimal solution computed by the LP solver.

The Linear Programming solver can handle only a set of linear constraints and treats all its variables as continuous, relaxing any discrete constraints. It computes an optimum value of a given objective function and records the values of the variables in this optimum.

The earlier integrations of constraint programming and linear programming, like those described in [2] and [50], pass the whole set of constraints into the LP solver. The unimodular probing algorithm, in contrast, passes to the LP solver only a restricted class of constraints which has a special property called *total unimodularity*. The consequence of this property is that the optimum solution is guaranteed to be integer–valued. A set of constraints with total unimodularity property is referred to as an "easy set".

Below we define total unimodularity and describe its characteristics. It follows definitions given in [44], but it can be found in any book on operations

research.

Definition 2.9.1. *An $m \times n$ matrix A is **totally unimodular** (TU) if the determinant of each square submatrix of A is equal to $0,1$ or -1.*

Let $P(b)$ represent the polyhedron of solutions to the set of linear inequalities $Ax \leq b$.

Theorem 2.9.1. *If A is totally unimodular, then $P(b) = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$ is integral for all $b \in \mathbb{Z}^n$.*

The proof of the Theorem 2.9.1 may be found in e.g. [44].

This property is applied by unimodular probing algorithm on an easy set of constraints to assure that the solution returned by the probe is an optimal integer solution. The probing procedure proceeds by applying local consistency techniques to the constraints of the problem. After a solution was found by the prober, the search procedure generates the constraints in the full set violated by the unimodular probe. If there are no violated constraints then the search found a solution. Otherwise, the algorithm selects a violated constraint, and rules out the present unimodular probe by imposing an additional easy constraint. If a choice of possible repairs exists then the algorithm backtracks through these choices in its search for optimality. When a repair has been selected, local consistency methods derive further easy constraints which follow from the choice. We will explain the propagation mechanism on KRFP example later on.

The Algorithm 11 on page 57 gives an outline of the unimodular probe algorithm.

Below we give an example how the unimodular probing algorithm can be applied on the Resource Feasibility Problem defined in Section 2.8.

The first task of constructing a unimodular probing algorithm is the identification of a suitable easy set with the total unimodularity property. In a Linear Programming problem, the set of constraints and variables is represented by a constraint matrix, where the rows represent constraints, and columns represent variables, with the element e_{ij} of the matrix denoting the coefficient of variable j in a particular constraint i.

Sufficient conditions for unimodularity of constraint matrix are as follows:

1. All the variable coefficients are 0,1,-1,
2. No more than two nonzero coefficients appear in each row (column).
3. The columns (rows) of the matrix can be partitioned into two subsets Q_1 and Q_2 such that:

(a) If a row (column) contains two nonzero coefficients with the same sign, one element is in each of the subsets.

(b) If a row (column) contains two nonzero elements of opposite sign, both elements are in the same subset.

For a proof see [44].

The temporal constraints of KRFP satisfy the sufficient conditions for the TU property if the set of variables (columns) is partitioned into a subset S_1 containing all temporal variables and an empty subset S_2. Since the characteristics and representation of $\mathcal{TC}$ is identical for KRFP and RUP, then the temporal constraints of the RUP constitute a totally unimodular set. For a proof see [54].

The Resource Utilization Problem perturbation function δ is defined to be an absolute change over the temporal variables. If x is a temporal variable then the absolute change in x is $d_x = |x - c|$, where c represents the value of x in previous solution. Since d_x, which is a non–linear function, is introduced in the objective function the following constraints are added:

$$d_x \geq x - c \tag{2.5}$$

$$d_x \geq c - x \tag{2.6}$$

Even if (2.6) violates the sufficient conditions for total unimodularity this characteristics is preserved on the incremental addition of new variable d_x and the constraints (2.5) and (2.6), which is proved in [57].

The experimental evaluation of the unimodular probing algorithm was done on a set of commercial aircraft utilization problems. The performance of the algorithm was compared with specialized constraint programming algorithm (a detailed description of the algorithm may be found in [46]. A modified version of the algorithm is given in Section 2.9.1) and a mixed integer programming (MIP) search. In comparison with pure constraint programming algorithm unimodular probing is capable to find an optimal solution for many problems where the pure constraint algorithm fails.

In comparison with MIP search unimodular probing achieves the substantial reductions in the number of search nodes. Another advantage of the unimodular probing algorithm over MIP techniques lies in the search ordering heuristics. MIP sees all violations in terms of integrity of problem variables that results in facing large amount of violations which are vary similar in their nature. Unimodular probing faces instead much fewer number of violations, but they seem much more heterogenous in their character. Moreover, the non–discrete values returned by the LP–solver for the relaxed MIP problem have

limited importance in a discrete context, which results in quite poor information about search choices.

In contrast, unimodular probing gives a relatively high level view of the sources of violation, which enables the use of meaningful heuristics for repair selection and bottleneck prioritization.

2.9.4 Local probing

Local search is a search method based on the idea that an exhaustive exploration of the search space of a problem is in most cases not necessary in order to find a feasible or optimal solution. A solution can be found by exploring assignments which lie in the neighborhood of current assignments.

Definition 2.9.2. *A neighborhood* $\mathbf{N}(\mathbf{x}, \sigma)$ *of a solution* $\mathbf{x}$ *is a set of solutions that can be reached from* $\mathbf{x}$ *by some simple operation* σ.

Examples of such operations are adding or removing an object from the solution.

If a solution $\mathbf{y}$ is better than any other solution in its neighborhood $\mathbf{N}(\mathbf{y}, \sigma)$ then $\mathbf{y}$ is a local optimum with respect to this neighborhood [47].

Preliminary research on local search as a probing method was presented in [33]. The authors evaluate hill–climbing search combined with several heuristics: limited variable search and limited shift search, both based on minimal discrepancy search [28], and minimal perturbation search [57], where the neighborhood operator correspond to the perturbation function δ in MPP, see Definition 2.8.1 in Section 2.8.

In the limited variable search the search decisions are variable assignments. The neighborhood solution is searched by letting one variable change. If a feasible solution is not found after exploring all possibilities the discrepancy limit is increased by one. This is continued until a feasible solution is found or the whole search is explored.

Limited shift search changes the number of variable re–assignments. The measure of discrepancy is the absolute distance of the new assignment from the basic solution. The variables for re–assignment are chosen in random order. The new value which a chosen variable takes is the one which gives a biggest improvement.

The minimum perturbation search heuristic assigns a value to one variable and uses linear programming to assign the remaining variables in a way which minimizes changes to previous assignment. The objective function and the

minimal perturbation constraints are modeled as follows:

$$min \sum_{i=1}^{N} d_{x_i} \tag{2.7}$$

s.t.

$$d_{x_i} \geq x_i - c_i \qquad i = 1, 2, \ldots, N \tag{2.8}$$

$$d_{x_i} \geq c_i - x_i \qquad i = 1, 2, \ldots, N \tag{2.9}$$

where $d_x = |x - c|$ represents the absolute change between variable x and its initial value c and N is the total number of noninstantiated variables. The objective function in (2.7). is linear and can be solved vary quickly.

A performance evaluation for a local prober was conducted on the set of benchmarks for resource feasibility problems [30](see Section 2.8). Since the heuristic value selection in the Minimal Perturbation Search tends to lead hill–climbing to local minima, a *semi–random* value selection was added to the neighborhood operator. This heuristics divides the variable domain into the set of values which potentially can improve and those which can deteriorate a solution. Values which improve the solution are explored first.

In addition tests with neighborhood operators using depth–first search were conducted for comparison. These tests show that minimum perturbation search is the best choice for quickly finding a feasible solution to the problem and outperforms both minimum discrepancy heuristics. Depth first search performs almost as well as minimum perturbation search.

As mentioned before the investigations presented in the paper are preliminary, there is for example no comparison between using a local prober and those presented in [57] or [54].

2.9.5 Hybridization based on Benders decomposition

The idea of decomposing an optimization problem into two subproblems which can be solved by cooperating solvers, presented in Section 2.9.3, is based on the fact that the cost function involved in the optimization process is linear or can be approximated by a linear or piecewise linear functions. Since linear programming offers efficient constraint solvers, which can quickly return optimal solutions to the problems, it seems advantageous to combine constraint and linear programming to solve complex optimization problems.

The form of hybridization referred in this section is based on the concept of a “master” problem, for which the optimal solution is found, and another

subproblems which interact with the master problem. In the simplest case the subproblem examines the latest optimal solution of the master problem and determines whether this optimal solution violates any constraint of the subproblem. In case of such a violation the subproblem returns one or more alternative linear constraints to the master problem, to reduce the probability that such a violation occurs again. One of these constraints is added to the master problem and a new optimal solution is found. To prove global optimality each of the returned alternatives is added to the master problem on different branches of the search tree.

Using operation research language this would correspond to *"row generation"* in contrast to *column generation*. The unimodular probing method described in Section 2.9.3 is such a "row generation" method.

Another form of hybridization based master/slave problem concept is *Benders decomposition*. The idea of Benders decomposition was first presented in [5] and generalized in [24]. Integration of Benders decomposition and constraint programming was described in [21] and [58].

Benders decomposition is presented below. The hybrid algorithm is listed as Algorithm 12 on page 44.

Consider the linear program $\mathbf{P}$ given by;

$$\begin{aligned} \mathbf{P}: \ & min\, \mathbf{f^T x} + \sum_{i=1}^{I} \mathbf{c_i^T y_i} && \\ & subject\ to\ \mathbf{G_i x} + \mathbf{A_i y_i} \geq \mathbf{b_i} && \forall i \\ & \mathbf{x} \in \mathcal{D}_{\mathcal{X}} && \\ & \mathbf{y_i} \geq 0 && \forall i \end{aligned} \tag{2.10}$$

Fixing $\mathbf{x}$ to some value $\mathbf{x^k}$ results in a linear program in $\mathbf{y_i}$ which can have a special structure or can be easy to solve. This program may be partitioned as follows:

$$\begin{aligned} \mathbf{P}: \quad & \min_{\mathbf{x}\in\mathcal{D}_{\mathcal{X}}} \left\{ \mathbf{f^T x} + \sum_{i=1}^{I} (min\ \{\mathbf{c_i^T y_i} : \mathbf{A_i y_i} \geq \mathbf{b_i} - \mathbf{G_i x}, \mathbf{y_i} \geq \mathbf{0}\}) \right\} \\ & = \min_{\mathbf{x}\in\mathcal{D}_{\mathcal{X}}} \left\{ \mathbf{f_T x} + \sum_{i=1}^{I} (max\{\mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x}) : \mathbf{u_i A_i} \leq \mathbf{c_i}, \mathbf{u_i} \geq \mathbf{0}\})) \right\} \end{aligned} \tag{2.11}$$

where the inner optimization has been dualized. If the $U_i = \{\mathbf{u_i} : \mathbf{u_i Ai} \leq \mathbf{c_i}, \mathbf{u_i} \geq \mathbf{0}\}$ is non–empty for each i there exists an optimal so-

lution to each inner optimization or it is unbounded along extreme rays. By letting $u_i^1, \ldots, u_i^{t_i}$ and $d_i^1, \ldots, d_i^{s_i}$ be the extreme points and directions of U_i equation (2.11) can be rewritten as a mixed integer *Master Problem* $\mathbf{MP}$:

$$\begin{aligned} \mathbf{MP}: \quad & min\ z = \mathbf{f^T x} + \sum_{i=1}^{I} \beta_i \\ & s.t.\ \beta_i \geq \mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x})\ \ \forall i\ \ \forall k \\ & \mathbf{0} \geq \mathbf{d_i^l}(\mathbf{b_i} - \mathbf{G_i x}) \quad \forall i\ \ \forall l \\ & \mathbf{x} \in \mathcal{D}_{\mathcal{X}} \end{aligned} \tag{2.12}$$

where β_i is a bound on the optimal value that depends on x called *Benders cut*.

Since there may exist many extreme points in the direction of each U_i which has its impact on number of constraints in 2.12 the master problem must be relaxed. If for some relaxed problem $\mathbf{RMP^k}$ the optimal relaxed solution $(z^k, \mathbf{x^k})$ satisfies all the constraints of (2.12) then $(z^k, \mathbf{x^k}, \mathbf{y_1^k}, \ldots, \mathbf{y_I^k})$ is an optimal solution of (2.10). Otherwise there exists some constraint or *Benders cut* in (2.12) which is violated for $\mathbf{x} = \mathbf{x^k}$. The violated element is then added to $\mathbf{RMP^k}$ creating $\mathbf{RMP^{k+1}}$, the ground for the next iteration.

The Benders cut is determined by fixing $\mathbf{x} = \mathbf{x^k}$ in (2.11):

$$\begin{aligned} \mathbf{SP_i^k}: \ & max\ \beta_i^k = \mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x^k}) \\ & subject\ to \quad \mathbf{u_i A_i} \leq \mathbf{c_i} \\ & \mathbf{u_i} \geq \mathbf{0} \end{aligned} \tag{2.13}$$

If any subproblem $\mathbf{SP_i^k}$ has an unbounded optimal solution for $\mathbf{x^k}$ then the primal of subproblem is infeasible for $\mathbf{x^k}$. If any subproblem $\mathbf{SP_i^k}$ is infeasible for $\mathbf{x^k}$ then its infeasible for any $\mathbf{x}$. In these cases the *Homogenous Dual* to the primal of the subproblem is considered:

$$\begin{aligned} & max\ \mathbf{u_i}(\mathbf{b_i} - \mathbf{G_i x^k}) \\ & subject\ to \quad \mathbf{u_i A_i} \geq \mathbf{0} \\ & \mathbf{u_i} \geq \mathbf{0} \end{aligned} \tag{2.14}$$

This problem is always feasible since $\mathbf{u_i} = \mathbf{0}$ is a solution and can have an unbounded optimum when the primal is infeasible and finite optimal solution when the primal is feasible. In case of an unbounded solution a cut $\mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x}) \leq 0$ which corresponds to an extreme direction of $U_i' = \{\mathbf{u_i} : \mathbf{u_i A_i} \leq \mathbf{0}, \mathbf{u_i} \geq \mathbf{0}\}$ is obtained.

The complete Benders decomposition is presented as Algorithm 12 on page 58.

The classical Benders decomposition can be generalized for the problems with nonlinear constraints and nonlinear objective function. In most general form the original problem:

$$\begin{aligned} \mathbf{P}: \; & minf(f_1(\mathbf{x}, \mathbf{y_1}), \ldots, f_I(\mathbf{x}, \mathbf{y_I})) \\ & subject\ to\ g_i(\mathbf{x}, \mathbf{y_i}) \geq \mathbf{b_i} \quad \forall\, i \\ & \mathbf{x} \in \mathcal{D}_\mathcal{X} \\ & \mathbf{y_i} \in \mathcal{D}_\mathcal{Y} \qquad \forall\, i \end{aligned} \tag{2.15}$$

is decomposed into a master problem:

$$\begin{aligned} \mathbf{MP}: \; & min \;\; z = f(\mathbf{x}, \beta_1, \ldots, \beta_I) \\ & subject\ to \quad \beta_i \geq \beta_i^k(\mathbf{x}) \qquad \forall i\, \forall k \\ & \qquad\qquad \mathbf{0} \geq \beta_i^l(\mathbf{x}) \qquad \forall i\, \forall l \\ & \qquad\qquad \mathbf{x} \in \mathcal{D}_\mathcal{X} \end{aligned} \tag{2.16}$$

and subproblems:

$$\begin{aligned} \mathbf{SP_i^k}: \; & min \quad f_i(\mathbf{x^k}, \mathbf{y_i}) \\ & subject\ to \quad g_i(\mathbf{x^k}, \mathbf{y_i}) \geq \mathbf{b_i} \\ & \qquad\qquad \mathbf{y_i} \in \mathcal{D}_\mathcal{Y} \end{aligned} \tag{2.17}$$

An evaluation of the algorithm (see [21]) was performed on instances of the minimal perturbation problem from [30]. Correct and optimal solutions were returned but the performance was of one magnitude slower than for the unimodular probing described in [54]. The reason for such a behavior may be the composition of benchmarks. Benders Decomposition proves to be very efficient in case the problem breaks down into a master problem and multiple subproblems. Since the benchmarks investigated in the paper involve a single kind of resource they do not have an apparent decomposition into multiple subproblems, which may be one of the reasons why this algorithm performs slower then unimodular probing.

2.10 Conclusions

This section describes the state-of-the-art in the field of dynamic constraint solving. The Constraint Satisfaction framework for dynamic problems was

presented and techniques which can be used to solve those problems were described. Also an overview of cooperating solvers and so called hybrid methods for complex dynamic problems was given.

Several future research directions in the area have been discussed. Of these, the idea of cooperative solvers which combines methods known from Operation Research and Constraint Programming seems most promising. We believe that cooperating solvers can be used to solve constraint optimization problems in general. The current investigation of hybridization methods does not fully take advantages of the dynamic nature of the problem. Any changes in the environment creates a new Constraint Satisfaction Problem which is solved from scratch. Propagating changes in configuration of the problem in context of cooperative solvers should be investigated further.

As it has been shown, many of the problems which include temporary changes, e.g., Minimal Perturbation Problem, are usually modelled as Dynamic CSPs. The possibility to model those problems using the Recurrent DCSP framework should be investigated.

Moreover, further research on the methods for propagating changes for Dynamic Constraint Satisfaction Problems should be conducted.

On the other hand there is no much work, that we are aware of, on modeling dynamic problems using the framework of the global constraints. This approach will be investigated for a class of global constraints in Paper D of this thesis.

Finally, methods which gather the information about the nature of the changes to a dynamic problem and use the gathered information to guide the search for solution to the new problem should be further investigated.

Algorithm 11 Unimodular probing search

Require: Set of new constraints to be added C_{new}
  $C_{cpnew} =$ push_cp(C_{new})
  if ($C_{cpnew} \neq FALSE$) **then**
    $C_{lpnew} =$ obtain_easy_constraints($C_{new} \cup C_{cpnew}$)
    $S_{lp} =$ push_lp(C_{lpnew})
    if ($S_{lp} \neq FALSE$) **then**
      $C_{conf} =$ violated_constraints(S_{lp})
      if ($C_{conf} = \emptyset$) **then**
        S_{lp}
      else
        $C_{repair} =$ select_repair_choice(C_{conf}, S_{lp})
        $S_{lp} =$ unimodular_probing_search(C_{repair})
        if ($S_{lp} \neq FALSE$) **then**
          return S_lp
        else
          $S_{lp} =$ unimodular_probing_search($NOT(C_{repair})$)
        end if
        if ($S_{lp} \neq FALSE$) **then**
          return S_{lp}
        end if
      end if
    end if
    pop_lp
  end if
  pop_cp
  return S_{lp}

Algorithm 12 Benders Decomposition Algorithm

1: Initialization: From the original linear program $\mathbf{P}$ (2.10) construct the *relaxed master problem* $\mathbf{RMP^0}$ (2.12) with the initial constraint set $\mathbf{x} \in \mathcal{D}_\mathcal{X}$ and set $k = 0$.

2: Iterative step: From the current relaxed problem $\mathbf{RMP^k}$ with the optimal solution $(z^k, \mathbf{x^k})$ construct $\mathbf{RMP^{k+1}}$ with optimal solution $(z^{k+1}, \mathbf{x^{k+1}})$: fix $\mathbf{x} = \mathbf{x^k}$ in $\mathbf{P}$ and solve the resulting subproblems $\mathbf{SP_i^k}$ (2.13) according to following cases:

- $\mathbf{SP_i^k}$ is primal unbounded for some i – stop with the original problem having unbounded solution.
- $\mathbf{y_i^k}, \mathbf{u_i^k}$ are, respectively, primal and dual solutions of subproblem $\mathbf{SP_i^k}$ with objective values β_i^k for each i.
  - **if** $\sum_{i=1}^{I} \beta_i^k = z^k$ **then** stop with $(z^k, \mathbf{x^k}, \mathbf{y_1^k}, \ldots, \mathbf{y_I^k})$ as the original solution to the original problem.
  - **if** $\sum_{i=1}^{I} \beta_i^k > z^k$ **then** add the *Benders cuts* $\beta_i \geq \mathbf{u_i^k}(\mathbf{b_i} - \mathbf{G_i x})$ to $\mathbf{RMP^k}$ to form the new relaxed master problem $\mathbf{RMP^{k+1}}$ set $k = k + 1$ and return to step 2.
- $\mathbf{SP_i^k}$ is dual unbounded or both primal and dual unfeasible for some i – find an extreme direction $\mathbf{d_i^k}$ of the homogeneous dual leading to unboundness; add the cut $\mathbf{d_i^k}(\mathbf{b_i} - \mathbf{G_i x}) < 0$ to $\mathbf{RPM^k}$ to form a new relaxed master problem $\mathbf{RMP^{k+1}}$, set $k = k + 1$ and return to step 2.

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In *Actes des Journees Francophones de Programmation end Logique, Lille*, 1992.

[2] H. Beringer and B. De Backer. Satisfiability of boolean formulas over linear constraints. In *IJCAI*, pages 296–304, 1993.

[3] N. Beldiceanu and E.Contejean. Introducing global constraints in chip. *Methematical Computation Modelling*, 20(12):97–123, 1994.

[4] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Principles and Practice of Constraint Programming*, pages 52–66, 2000.

[5] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[6] C. Bessiere. Arc-consistency for non-binary dynamic CSPs. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 23–27, Vienna, Austria, August 1992. John Wiley & Sons.

[7] C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 592–598, 1995.

[8] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI/IAAI*, pages 310–318, 1998.

[9] A. Bockmayr, N. Pisaruk and A. Aggoun Network flow problems in constraint programming. In *Principles and Practice of Constraint Programming 2001*, pages 196–210, 2001

[10] W. Y. Chiang and M. S. Fox. Protection against uncertainty in a deterministic schedule. In *Proceedings 4th International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 184–197, Hilton Head Island, 1990.

[11] P. Chatalic. Incremental techniques and prolog. Technical Report TR-LP-23, European Computer-Industry Research Centre, June 1987.

[12] P. Chatalic. IMPRO: an environment for incremental execution in prolog. Technical Report TR-LP-42, European Computer-Industry Research Centre, May 1989.

[13] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[14] J. Carlier and E. Pinson. Adjustments of heads and tails for the job-shop scheduling problem. *European Journal of Operational Research*, 78:146–161, 1994.

[15] A.J. Davenport and J.Ch. Beck. Managing uncertainty in scheduling: a survey. Preprint, 2000.

[16] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proc. of the Twelth National Conference on Artificial Intelligence (AAAI-94)*, pages 1098–1104, Seattle, WA, 1994. AAAI Press.

[17] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, pages 37–42, 1988.

[18] J. de Kleer. A comparison of ATMS and CSP techniques. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 290–296, Detroit, MI, USA, August 1989. Morgan Kaufmann.

[19] J. Doyle. A truth maintenance system. In Bonnie Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 496–516. Morgan Kaufmann, Los Altos, California, 1981.

[20] A. El-Kholy and B. Richards. Temporal and resource reasoning in planning: the parcplan approach. In *Proc. ECAI-96*, 1996.

[21] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, Lecture Notes in Computer Science, pages 1–15. Springer, 2001.

[22] H. Gao. Building robust schedules using temporal protection(an empirical study of constraint based scheduling under machine failure uncertainty). Master's thesis, Department of Industrial Engineering, University of Toronto, 1995.

[23] E. Gaudin, N. Jussien and G. Rochart. *Implementing explained global constraints* CP04 Workshop on Constraint Propagation and Implementation (CPAI'04), pp. 61–76, 2004

[24] A.M. Geoffrion. Generalized benders decomposition. *Journal of Optimization theory and Applications*, 4(10):237–260, 1972.

[25] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems*. PhD thesis, University of Pittsburgh, 1996.

[26] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993. (electronic journal).

[27] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. In *IEEE Real-Time Systems Symposium*, pages 120–129, 1995.

[28] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proc. of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 607–615. Morgan Kaufman, 1995.

[29] P. Van Hentenryck and T. Le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9:257–275, 1991.

[30] Introduction to resource utilization benchmarks. Available at www.icparc.ic.ac.uk/˜hhe/RFPBenchmarks/new_benchmark_intro.html, 2000.

[31] W. Kocjan Dynamic scheduling. State of the art report. Technical Report, Swedish Institute of Computer Science, 2000.

[32] P. Kreuger, M. Carlsson, T. Sjöland, and E. Åström. Sequence dependent task extensions for trip scheduling. Technical report, Swedish Institute of Computer Science, 2001.

[33] O. Kamarainen, H. El Sakkout, and J. Lever. Local probing for resource constrained scheduling. In *International Conferences Constraint Programming & Logic Programming. Workshop Proceedings. CoSolv'01: Cooperative Solvers in Constraint Programming.*, pages 73–86, 2001.

[34] O. Lhomme. Consistency techniques for numeric CSPs. In Ruzena Bajcsy, editor, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 232–238, Chambery, France, 1993. Morgan Kaufmann.

[35] L. Lovasz and M.D. Plummer. Matching theory. *North-Holland mathematic studies*, 121, 1986.

[36] V. Leon and S. Wu. Storer: Robustness measures and robust scheduling for job shops, 1994.

[37] A.K. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[38] D. McAllester. Truth maintenance. In R. Smith and T. Mitchell, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 2, pages 1109–1116. AAAI Press, 1990.

[39] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[40] S. Minton, M. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[41] U. Montanari. Networks of constraints fundamental properties and applications to picture processing. *Information Sciences*, 66(7):95–132, 1974.

[42] U. Montanari. Optimization methods in image processing. In *Proceedings IFIP*, pages 727–732, North-Holland, 1974.

[43] M. Maher and P. Stuckey. Expanding query power in contraint logic programming languages. In *Proceedings of the North American Conference on Logic Programming (NACLP–89), Cleveland, Ohio, U.S.A*, pages 20–36, 1989.

[44] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Ltd., 1988.

[45] M. Ohki, A. Takeuchi, and K. Furukawa. A framework for interactive problem-solving based in interactive query revision. In E. Wada, editor, *Proc. 5th Conf. on Logic Programming'86*, Springer LNCS 264, pages 137–146, 1986.

[46] D. Pothos. A constraint-based approach to the bristish airways schedule re-timing problem. Technical Report 97/04-01, IC-Parc, Imperial College, 1997.

[47] C.R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill Book Company, 1995.

[48] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[49] G. Rochart, N. Jussien and F. Laburthe Challenging Explanations for Global Constraints CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03), pp. 31-43, 2003

[50] R. Rodosek, M. Wallace, and M. Hajian. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations Research*, 1998.

[51] N. Sadeh. *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[52] J. Sgall. On-line scheduling - a survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms*. Springer-Verlag, Berlin, 1997. Available from citeseer.nj.nec.com/sgall97line.html.

[53] N. Sadeh, S. Otsuka, and R. Schnelbach. Predictive and reactive scheduling with the micro-boss production scheduling and control system. In *Proc. IJCAI-93 Workshop on Knowledge–based Production Planning, Scheduling, & Control*, Chambery, France, Aug 1993.

[54] H. El Sakkout, T. Richards, and M. Wallace. Unimodular probing for minimal perturbance in dynamic resource feasibility problems. In *Proc. of the CP97 workshop on Dynamic Constraint Satisfaction*, 1997.

[55] H. El Sakkout, T. Richards, and M. Wallace. Minimal perturbation in dynamic scheduling. In Henri Prade, editor, *ECAI98. 13th Europen Conference on Artificial Intelligence*. John Wiley & Sons, Ltd., 1998.

[56] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI/IAAI 1999*, pages 163–168, Orlando, Florida, USA, July 1999. AAAI Press / The MIT Press.

[57] H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints. An International Journal. Special Issue on Industrial Constraint-Directed Scheduling*, 5(4):359–388, October 2000.

[58] E. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, Lecture Notes in Computer Science, pages 16–30. Springer, 2001.

[59] Tsang, E. *Foundation of Constraint Satisfaction*. Academic Press, 1993.

[60] M. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing*, 4:287–304, 1986.

[61] G. Verfaillie and T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications, Amsterdam, The Netherlands*, pages 1–8, 1994.

[62] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 307–312, Menlo Park, CA, USA, July 31–August 4 1994. AAAI Press.

[63] G. Verfaillie and N. Jussien Constraint Solving in Uncertain and Dynamic Environment: A Survey *C*onstraints, 10, pp. 253-281, 2005

[64] D.L. Waltz. Understanding line drawing of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[65] J.R. Wallace. Analysis of heuristic methods for partial constraint satisfaction problems. In M. Maher and J.-F. Puget, editors, *Principle and Practice of Constraint Programming - CP'98*, Lecture Notes on Computer Science No. 1520, pages 482–496. Springer, 1998.

[66] R.J. Wallace and E.C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Principles and Practice of Constraint Programming*, pages 447–461, 1998.

[67] R.J. Wallace and E.C. Freuder. Representing and coping with recurrent change in dynamic constraint satisfaction problems. In *CP'99 Workshop on Modelling and Solving Soft Constraints*, 1999.

Chapter 3

Paper B: Filtering Methods for Symmetric Cardinality Constraint

Waldemar Kocjan and Per Kreuger

Jean-Charles Régin, Michel Rueher (Eds.): Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings. Lecture Notes in Computer Science 3011 Springer 2004, ISBN 3-540-21836-X, pages 200-208

Abstract

The symmetric cardinality constraint is described in terms of a set of variables $X = \{x_1, \ldots, x_k\}$, which take their values as subsets of values $V = \{v_1, \ldots, v_n\}$. It constraints the cardinality of the set assigned to each variable to be in an interval $[l_{x_i}, c_{x_i}]$ and at the same time it restricts the number of occurrences of each value $v_j \in V$ in the sets assigned to variables in X to be in an other interval $[l_{v_j}, c_{v_j}]$. In this paper we introduce the symmetric cardinality constraint and define set constraint satisfaction problem as a framework for dealing with this type of constraints. Moreover, we present effective filtering methods for the symmetric cardinality constraint.

3.1 Introduction

The symmetric cardinality constraint is specified in terms of a set of variables $X = \{x_1, \ldots, x_k\}$, which take their values as subsets of $V = \{v_1, \ldots, v_n\}$. The cardinality of the set assigned to each variable is constrained by the interval $[l_{x_i}, c_{x_i}]$, where l_{x_i} and c_{x_i} are non–negative integers. In addition, it constraints the number of occurrences of each value $v_j \in V$ in the sets assigned to variables in X to be an interval $[l_{v_j}, c_{v_j}]$. Both l_{v_j} and c_{v_j} are non–negative integers.

The symmetric cardinality constraint problems arise in many real–life problems. For example, consider an instance of a project management problem. The main task of the problem is to assign personnel with possibly multiple specialized competences to a set of tasks, each requiring a certain number of people (possibly none) of each competence. In this instance we consider a project consisting of 7 activities numbered from 1 to 7, each demanding a number of persons to be accomplished. An activity which demands minimum 0 persons is optional.

There are 6 members of personnel which can be assigned to this project. Each of those persons, here referred to as x with a respective index, is qualified to perform respective activities as shown in Fig.3.1. A non-zero value of a lower bound indicates that members of the staff represented by the variable must be assigned to some activities in the project.

The goal is to produce an assignment which satisfy the following constraints:

- every member of staff must be assigned to a minimum and maximum number of activities in the project,
- every activity must be performed by a minimum and maximum number of persons,
- each person can be assign only to an activity he/she is qualified to perform and, by symmetry, each activity must be performed by qualified personnel.

In this paper we show how such problem can be modeled as a constraint satisfaction problem. First, in Section 3.2 we give some preliminaries on graphs and flows. Then, in Section 3.3 we define set constraint satisfaction problem and give a formal definition of the symmetric cardinality constraint. The following section, 3.4, gives a method for checking consistency of symmetric

	1	2	3	4	5	6	7
x_1	1	1	0	0	0	0	0
x_2	0	1	1	0	0	0	0
x_3	1	0	1	0	0	0	0
x_4	0	1	0	1	0	0	0
x_5	0	0	1	1	1	1	0
x_6	0	0	0	0	0	1	1

person	activities
x_1	2..3
x_2	0..2
x_3	0..1
x_4	1..1
x_5	0..3
x_6	0..2

task	persons
1	1..3
2	1..2
3	0..1
4	1..2
5	1..3
6	0..2
7	1..1

Figure 3.1: Project assignment specification. In the left table, a 1 indicates that a person represented by x_i is qualified to perform corresponding activity in the project. The table in the center and to the right specifies the number of activities each person can perform and the number of persons required by each activity.

cardinality constraint. Finally, we describe a filtering method for symmetric cardinality constraint.

3.2 Preliminaries

3.2.1 Graph

The following definitions are mainly due to [1].

A *directed graph* $G = (X, U)$ consists of a set of nodes (vertices) X and arcs (edges) U, where every pair (u, v) is an ordered pair of distinct nodes. An *oriented graph* is a directed graph having no symmetric pair of arcs.

A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values. In this paper we do not make distinction between terms "network" and "directed network".

An arc (u, v) connects node u with node v, i.e. in directed graph it is an arc oriented from node u to node v. A *path* in a graph G from v_1 to v_k is a sequence of nodes $[v_1, \ldots, v_k]$ such that each (v_i, v_{i+1}) is an arc for $i \in [1, \ldots, k-1]$. The path is *simple* if all its nodes are distinct. A path is a *cycle* if $k > 1$ and $v_1 = v_k$.

A subgraph of a directed graph G, which contains at least one directed path from every node to every other node is called a *strongly connected component of G*.

3.2.2 Flows

Let N be a directed network in which each arc e is associated with two non–negative integers $l(e)$ and $c(e)$ representing lower bound and a capacity of flow on e. A flow $f(e)$ on arc e represents the amount of commodity that the arc accommodates. More formally:

Definition 3.2.1. *A flow in a network N is a function that assigns to each arc e of the network a value $f(e)$ in such way that*

1. *$l(e) \leq f(e) \leq c(e)$, where $l(e)$ is a lower bound of the flow in the arc and $c(e)$ is a capacity of e*
2. *for each node p in the network N it is true that $\sum_n f(e(n,p)) = \sum_r f(e(p,r))$ where $e(x,y)$ is an arc from node x to y*

The second property is known as a conservation law and states that the amount of flow of some commodity incoming to each node in N is equal the amount of that commodity leaving each node.

Three problems from the flow theory are referred to in this paper:

- *the feasible flow problem* which resolves if there exists a flow in N which satisfies lower bound and capacity constraints for all arcs in N.
- *the problem of maximum flow from m to n* which consists of finding a feasible flow in N such that the value $f(m,n)$ is maximum
- *the problem of minimum flow from m to n* which consists of finding a feasible flow in N such that the value $f(m,n)$ is minimum.

It is a well known fact that if the lower bounds and capacities of a flow problem are integral and there exists a feasible flow for the network, then the maximum and minimum flows between any two nodes flows are also integral on all arcs in the network. Hence, if there exists a feasible flow in a network there also exists an integral feasible flow. In this paper when we refer to a feasible flow we always mean an *integral* feasible flow.

We refer in this paper also to the *residual network*, which is a network representing the utilization and remaining capacity in the network with respect to a flow f.

Definition 3.2.2. *Given a flow f from s to t in the network N, the residual network for f, denoted by $R(f)$, consists of the same set of nodes as N. The arc set of $R(f)$ is defined as follows. For all arcs (m,n) in N*

- *if* $f(m,n) < c(m,n)$ *then* (m,n) *is an arc of* $R(f)$ *with residual capacity* $res(m,n) = c(m,n) - f(m,n)$*,*
- *if* $f(m,n) > l(m,n)$ *then* (n,m) *is an arc of* $R(f)$ *with residual capacity* $res(n,m) = f(m,n) - l(m,n)$*,*

3.3 Set Constraint Satisfaction Problem

We define a set constraint satisfaction problem as follows.

Definition 3.3.1. *A set constraint satisfaction problem (sCSP) is a triple* (X, D, Cs) *where*

1. $X = \{x_1, \ldots, x_n\}$ *is a finite set of variables.*
2. $D = \{D_1, \ldots, D_n\}$ *is a set of finite sets of elements such that for each* i*,* x_i *takes as value a subset of* D_i*.*
3. Cs *is a set of constraints on the values particular subsets of the variables in* X *can simultaneously take. Each constraint* $C \in Cs$ *constrains the values of a subset* $X(C) = \{x_{C_1}, \ldots, x_{C_k}\}$ *of the variables in* X *and may be thought of as a subset* $T(C)$ *of the Cartesian product* $= C_{C_1} \times \ldots, \times C_{C_k}$ *where each* $C_{C_i} = \{C \mid C \subseteq D_{C_i}\}$*.*

Let

$$D(C) = \bigcup_{i \in \{i \mid x_i \in X(C)\}} D_i$$

be the set of values that can be taken by any variable in $X(C)$. Furthermore, for a given assignment P, let $P(x_i)$ denote the value assigned to the variable x_i by P and $\#(x_i, P)$, the cardinality $|P(x_i)|$ of the set $P(x_i)$ and for any constraint C and element $v_j \in D(C)$, $\#(v_j, C, P)$ denote the number of occurrences of v_j in the values assigned by P to the variables in $X(C)$, i.e:

$$\sum_{x_i \in X(C)} \begin{cases} 1 \text{ if } v_i \in P(x_i) \\ 0 \text{ otherwise} \end{cases}$$

Definition 3.3.2. *A sCSP* $\langle X, D, Cs \rangle$ *is* consistent *if and only if there exists an assignment* P *with the following properties:*

1. *For each variable* $x_i \in X$ *with domain* D_i*, the value* $P(x_i)$ *assigned to* x_i *by* P *must be a subset of* D_i*.*

2. *For each constraint $C \in Cs$ and each variable in $X(C) = \{x_{C_1}, \ldots, x_{C_k}\}$ the tuple $\langle P(x_{C_1}), \ldots, P(x_{C_k})\rangle \in T(C)$.*

Moreover, a value $v \in D(x_{C_i})$ for x_{C_i} is consistent with C iff $\exists P(P(X(C)) \in T(C))$ such that v is an element in the value $P(x_i)$.

An n–ary constraint can be seen in terms of its *value graph* ([2]), i.e the bipartite graph $G(C) = (X(C), D(C), E)$, where for all $x \in X(C), v \in D(C), (x_i, v) \in E$ iff $v \in D_i$. This graph establishes an immediate correspondence between any assignment P and a special set of edges in a value graph.

We formulate this notion in the following proposition.

Proposition 3.3.1. *For any $C \in Cs$ every $P(X(C))$ corresponds to a subset of edges in $G(C)$ and the number of edges connecting $x_i \in X(C)$ with any $v_j \in D(C)$ is equal to the cardinality of the subset $P(x_i)$.*

3.4 Consistency of the Symmetric Cardinality Constraint

We define the symmetric cardinality constraint as follows.

Definition 3.4.1. *A symmetric cardinality constraint is a constraint C over a set of variables $X(C)$ which associates with each variable $x_i \in X(C)$ two non-negative integers l_{x_i} and c_{x_i}, and with each value $v_j \in D(C)$ two other non-negative integers l_{v_j} and c_{v_j} such that a restriction of an assignment P to the variables in $X(C)$ is an element in $T(C)$ iff*
$\forall i\, (l_{x_i} \leq \#(x_i, P) \leq c_{x_i})$ and $\forall j\, \left(l_{v_j} \leq \#(v_j, C, P) \leq c_{v_j}\right)$.

From the symmetric cardinality constraint we propose to build a particular oriented graph, which we denote $N(C)$. This extends the value network of the global cardinality constraint as described in [3] to handle sets of nonnegative cardinality assigned to the variables. Then, we will show an equivalence between the existence of a feasible flow in such graph and the consistency of the symmetric cardinality constraint.

Let C be a symmetric cardinality constraint, the *value network* $N(C)$ of C is an oriented graph with a capacity and a lower bound on each arc. The value network $N(C)$ is obtained from the value graph $G(C)$ by

- orienting each edge of $G(C)$ from values to variables. Since each value can occur in a subset assigned to a variable at least 0 and at most 1 time for each arc $(v, x) : l(v, x) = 0, c(v, x) = 1$

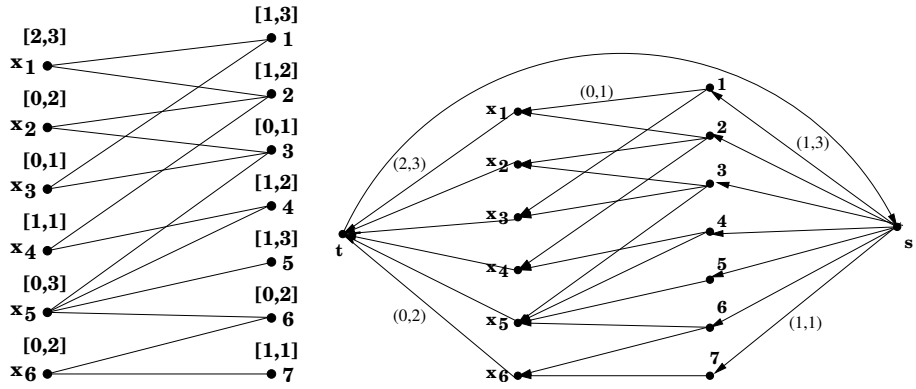

Figure 3.2: Value graph for assignment problem from Figure 3.1 and its value network

- adding a source node s and connecting it which each value. For every arc $(s, v_i) : l(s, v_i) = l_{v_i}, c(s, v_i) = c_{v_i}$
- adding a sink node t and an arc from each variable to t. For each such arc $(x_i, t) : l(x_i, t) = l_{x_i}, c(x_i, t) = c_{x_i}$ [1]
- adding an arc (t, s) with $l(t, s) = 0$ and $c(t, s) = \infty$

Proposition 3.4.1. *Let C be a symmetric cardinality constraint and $N(C)$ be the value network of C. The following properties are equivalent:*

- *C is consistent;*
- *there exists a flow from s to t which satisfies lower bounds and capacities of the arcs in $N(C)$.*

Proof. Suppose that C is consistent then $T(C) \neq \emptyset$. Consider $P \in T(C)$. We can build a function f in $N(C)$ as follows:

1. $\forall x_i \in X(C), f(x_{c_i}, t) = \#(x_{c_i}, P)$
2. $\forall x_i \in X(C), f(v, x_i) = 1$ if v appears in the subset $P(x_i)$, otherwise $f(v, x_i) = 0$

[1] Actually, the orientation of the graph has no importance. Here, we have chosen the same orientation as in [3]

3. $\forall v_j \in D(C), f(s, v_j) = \#(v_j, C, P)$

Since C is consistent then $\forall x_i \in X(C), 0 \leq l_{x_i} \leq \#(x_i, P) \leq c_{x_i}$ and $\forall v_j \in D(C), 0 \leq l_{v_j} \leq \#(v_j, C, P) \leq c_{v_j}$, which satisfies the lower bound and capacity constraint for the flow $f(v_i, x_i)$. Furthermore from (1)-(3) follows that v_j must appear $\#(v_j, C, P)$ times in $P(X(C))$, which means that $\sum_{i=1}^{k} f(v_j, x_i) = f(s, v_j)$ which satisfies the conservation law for all $v_j \in D(C)$. From this and (2) follows that for each $x_i \in X(C)$ the number of arcs with flow value 1 entering x_i is equal to $\#(x_i, P)$. According to (1): $\forall x_i \in X(CS), f(x_i, t) = \#(x_i, P)$ which satisfies the conservation law for all $x_i \in X$.

On the other hand, suppose there exists a feasible flow f from s to t. Since $f(v_j, x_i) = 1$ or $f(v_j, x_i) = 0$ and by the conservation law $f(s, v_j) = \sum_{i=1}^{k} f(v_j, x_i)$ then for each v_j the number of edges with the flow of value 1 leaving v_j is equal $f(s, v_j)$. Consequently, the number of $x_i \in X(C)$ connected with each v_j by an arc with a flow equal to 1 is equal to $f(s, v_j)$. Furthermore, due to the conservation law, the number of arcs for which $f(v_j, x_i) = 1$ entering each x_i is equal the value of $f(x_i, t)$. Thus the set of arcs such that $f(v_j, x_i) = 1$ corresponds to a set of edges in the value graph. By the capacity constraint $\forall v_i \in D(C) : l_{v_i} \leq f(s, v_i) \leq c_{v_i}$ and by the conservation law $f(s, v_i) = \sum_x f(v_i, x) \leq c_i$ therefore $l_i \leq \#(v_i, C, P) \leq c_i$. Similarly, $\forall x_i \in X : l_{c_{x_i}} \leq \sum_{i=1}^{k} f(v_j, x_i) \leq c_{x_i}$ and by the conservation law $f(x_i, t) = \sum f(v_j, x_i) \leq c_{x_i}$, therefore $l_{x_i} \leq \#(x_i, P) \leq c_{x_i}$, thus C is consistent. □

This proposition gives a way of checking the consistency of a symmetric cardinality constraint by computing a feasible flow in $N(C)$. Different algorithms for computing feasible flows are given in the literature on flow theory, e.g. in [1].

In the next section we will show how to implement filtering of symmetric cardinality constraint by considering certain properties common to all feasible flows in the value graph.

3.5 Filtering Algorithm for Symmetric Cardinality Constraint

Let f be a feasible flow in a network N, $R(f)$ be a residual graph for f. If there is a simple path p from n to m in $R(f) - \{(n, m)\}$, then we can obtain

a new feasible flow f' in N such that $f'(m,n) > f(m,n)$ (see [4]). We call such a path p an augmenting path. Similarly, if there exists a simple path p from m to n in $R(f) - \{(m,n)\}$, then we can obtain a new feasible flow f' such that $f'(m,n) < f(m,n))$. We refer to such simple path p as a reducing path.

Moreover, the maximum and minimum flow are defined as follows ([3]).

Definition 3.5.1. *A flow f from m to n is* maximum *if and only if there is no augmenting path from m to n for f.*

A flow f from m to n is minimum *if and only if there is no reducing path from m to n for f.*

The following Theorem 3.5.1 gives a way of determining if an arbitrary arc in N is contained in any feasible flow in the network. The theorem is similar to Theorem 4 from [3], but here the computation is performed on the residual graph of f which includes both (t,s) and, in the case when $f(t,s) > 0$, also (s,t).

Theorem 3.5.1. *Let N be a network for which each arc is associated with two non-negative integers, f be an arbitrary feasible flow in N, $R(f)$ be the residual graph for f and (m,n) be an arbitrary arc in N. For all feasible flows f' in N, $f'(m,n) = f(m,n)$ if and only if neither (m,n) nor (n,m) are contained in any simple cycle in $R(f)$ involving at least three nodes.*

Proof. If (m,n) is not contained in a simple cycle in $R(f)$ involving at least three vertices it means that there is no augmenting path from n to m for f. By the definition 3.5.1, the flow f is the maximum flow from m to n.

If (n,m) is not contained in a simple cycle in $R(f)$ which involves at least three nodes then there is no reducing path from n to m in N, so by the definition 3.5.1 f is the minimum flow from m to n.

Similarly, if (t,s) is not contained in a simple cycle in $R(f)$ with at least 3 nodes, then there is no augmenting path from s to t and by Definition 3.5.1 f is a maximum flow in N. Moreover, if (s,t) is not contained in a simple cycle in $R(f)$ involving at least three nodes then there is no reducing path for f in $R(f)$ and by Definition 3.5.1 f is a minimum flow in N. □

Let $C \in Cs$ be a symmetric cardinality constraint and f be an arbitrary feasible flow in $N(C)$. By Proposition 3.4.1 a value v of a variable x is not consistent with C if and only if there exists no feasible flow in $N(C)$ which contains (v,x). So, by the Theorem 3.5.1 if $f(v,x) = 0$ and (v,x) is not contained in a simple cycle in $R(f)$ involving at least three nodes then the value

v of a variable x is not consistent with C. Furthermore since two nodes m and n can be contained in the such a cycle only if they belong to the same strongly connected component in $R(f)$, we can determine if a value for a variable is consistent with a symmetric cardinality constraint using an algorithm that search for strongly connected components in a graph, e.g. the algorithm described in [6].

This leads to the following corollary:

Corollary 3.5.1. *The consistency of any value v of variable x with symmetric cardinality constraint C can be computed by finding an arbitrary feasible flow f from s to t in $N(C)$ and by computing strongly connected components in $R(f)$.*

3.6 Notes on Complexity

The complexity of the proposed filtering algorithm is dominated by the computation of a feasible flow in the value network of the symmetric cardinality constraint. Methods for finding feasible flows have in the worst case the same complexity as that of finding a maximum flow, which is $O(n^2m)$ (see e.g. [1, 5]), where n is the number of nodes (i.e. number of variables, $|X|$, plus the number values, $|D(C)|$) and m is the number of arcs which, for a bipartite graph of the type used in the value graph, is bounded by $(\frac{n}{2})^2 + n$.

The complexity of finding strongly connected components in the residual graph using the method proposed in [6] is $O(m + n)$. This gives worst case complexity for filtering of the symmetric cardinality constraint of $O((|X| + |D(C)|)^4)$, which is the same as for the global cardinality constraint introduced in [3].

3.7 Conclusion

In this paper we have introduced the symmetric cardinality constraint derived from the global cardinality constraint. Moreover, we have formalized set constraint satisfaction problem and defined symmetric cardinality constraint in the context of this problem. We have also presented efficient methods for filtering domains of symmetric cardinality constraint.

Acknowledgment

The work on this paper was supported by Swedish Institute of Computer Science and Mälardalen University. The authors wish to thank Nicolas Beldiceanu, Irit Katriel, Markus Bohlin and the anonymous reviewers for their helpful comments on this paper.

Bibliography

[1] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows. Theory, algorithms and applications. Prentice–Hall Inc. (1993)

[2] Laurière, J.-L.: A language and a program for stating and solving combinatorial problems. Artificial Intelligence, **10** (1978) 29–127

[3] Régin, J.-Ch.: Generalized Arc Consistency for Global Cardinality Constraint.Proc. of the Fourteenth National Conference on Artificial Intelligence (AAAI-96) (1996)

[4] Lawler, E.: Combinatorial Optimization: Network and Matroids, Holt, Rinehart and Winston (1976)

[5] Ahuja, R.K., Kodialam, A., Mishra, A.K., Orlin, J.B.: Computational Investigations of Maximum Flow Algorithms. European Journal of Operational Research, **97** (1997) 509–542

[6] Tarjan, R.E.: Depth–First Search and Linear Graph Algorithms. SIAM J. Computing, **1** (1972) 146–160

Chapter 4

Paper C: Symmetric Cardinality Constraint with Costs

Waldemar Kocjan, Per Kreuger, Björn Lisper

MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-167/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, December 2004

Abstract

The symmetric cardinality constraint is described in terms of a set of variables $X = \{x_1, \ldots, x_k\}$, which take their values as subsets of $V = \{v_1, \ldots, v_n\}$. It constrains the cardinality of the set assigned to each variable to be in an interval $[l_{x_i}, u_{x_i}]$ and at the same time restricts the number of occurrences of each value $v_j \in V$ in the union of the sets assigned to variables in X to be in an other interval $[l_{v_j}, u_{v_j}]$. In this paper we extend the symmetric cardinality constraint with a function which associate with each value of each variable a cost and constrains the sum of all costs associated with assigned values. We also give an algorithm for computing the consistency of a symmetric cardinality constraint with costs and describe filtering methods for this constraint.

4.1 Introduction

The symmetric cardinality constraint, introduced in [1], is specified in terms of a set of variables $X = \{x_1, \ldots, x_k\}$, which take their values as subsets of $V = \{v_1, \ldots, v_n\}$. The cardinality of the set assigned to each variable is constrained by an interval $[l_{x_i}, u_{x_i}]$, where l_{x_i} and u_{x_i} are non–negative integers. In addition, it constraints the number of occurrences of each value $v_j \in V$ in the union of the sets assigned to variables in X to be an interval $[l_{v_j}, u_{v_j}]$. Both l_{v_j} and u_{v_j} are non–negative integers.

The symmetric cardinality constraint is an extension of the global cardinality constraint [2] on sets. While global cardinality constraint allows us to model instances of matching problems where one variable can be matched with only one value, but the number of the occurrences of a value must be in a given interval, the symmetric cardinality constraint enables to assign to a variable a set of values with a cardinality described by an interval. The global cardinality constraint can be modeled using symmetric cardinality constraint by restraining the cardinality of a set assigned to any variable to exactly 1.

Later the global cardinality constraint was extended with a cost function ([3, 4]), which makes possible to model some assignments problems. Nevertheless, assignment problems, where a (possibly empty) set of values needs to be assigned to a variable in the problem, can not be handled easily.

Consider the following instance of a project management problem, which consists of the task of assigning a number of workers, with possibly multiple skills, to a set of activities, each requiring a number of workers (possibly none) with specified skills. Due to former experience of the members of the project we can approximate an amount of time necessary for respective worker to accomplish given activity. Moreover, a total amount of time which can be spent on the project is given.

The goal is to produce an assignment, which satisfies the following constraints:

- every member of a project must be assigned to a minimum and a maximum number of activities in the project
- every activity must be performed by a minimum and a maximum number of persons
- each person can be assigned to an activity he/she is qualified to perform and, by symmetry, each activity must be performed by a qualified personnel.

- the total time spend on accomplishing the project must not be exceeded.

Clearly, due to the fact that global cardinality constraint require exactly one value to be assigned to a variable, it can not be used to solve such a problem without introducing auxiliary constraints. On the other hand, we can easily model this problem by extending the symmetric cardinality constraint with a cost function.

In this paper we show how the symmetric cardinality constraint can be extended with a cost function and investigate if the filtering algorithms for the global cardinality constraint with costs can be applied to the symmetric cardinality constraint with cost.

The rest of the paper is organized as follow. The next section, 4.2, gives some preliminaries on graphs, flows and set constraint satisfaction problems. Section 4.3 briefly describes the symmetric cardinality constraint. Then, in Section 4.4 we give a formal definition of the symmetric cardinality constraint with costs and describe a method for computing its consistency. In the following section, 4.5, we describe an algorithm for computing a minimum cost flow in a network representing the constraint. The following Section 4.6 describes the filtering method for this constraint. Finally, Section 4.7 conclude this work.

4.2 Preliminaries

4.2.1 Graph

Definitions in this section follows the presentation in [5].

A *directed graph* $G = (N, E)$ consists of a set of nodes (vertices) N and arcs (edges) E, where every arc $(i, j) \in E$ is an ordered pair of distinct nodes. An *oriented graph* is a directed graph having no symmetric pair of arcs.

A graph $G = (N, E)$ is a *bipartite graph* if we can partition its node set into two subsets N_1 and N_2 so that for each arc $(i, j) \in E$ either $i \in N_1$ and $j \in N_2$ or $i \in N_2$ and $j \in N_1$.

A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values. In this paper we do not make any distinction between terms “network” and “directed network”.

An arc (i, j) connects node i with node j, i.e. in directed graph it is an arc oriented from node i to node j. A *path* in a graph G from v_1 to v_k is a sequence of nodes $[v_1, \ldots, v_k]$ such that each (v_i, v_{i+1}) is an arc for $i \in [1, \ldots, k-1]$. The path is *simple* if all its nodes are distinct. A path is a *cycle* if $k > 1$ and $v_1 = v_k$.

4.2.2 Flows

Let $G = \{N, E\}$ be a directed network in which each arc $(i, j) \in E$ is associated with two non–negative integers l_{ij} and u_{ij} representing lower and upper bound on a capacity of a flow through (i, j). A flow $f(i, j)$ on arc (i, j) represents the amount of commodity that the arc accommodates. More formally:

Definition 4.2.1. *A flow in a network G is a function that assigns to each arc (i, j) of the network a value $f(i, j)$ in such way that*

1. *$l_{ij} \leq f(i, j) \leq u_{ij}$, where l_{ij} and u_{ij} are lower and upper bounds on a capacity of (i, j)*
2. *for each node i in the network G it is true that $\sum_k f(k, i) = \sum_j f(i, j)$*

The second property is known as a conservation law and states that the amount of flow of some commodity incoming to each node in G is equal the amount of that commodity leaving each node.

In this paper we refer to *the feasible flow problem* which decides if there exists a flow in G which satisfies the capacity constraint for all arcs in G.

Moreover, we refer here to *the minimum cost flow problem* in which each arc $(i, j) \in E$ has an associated *cost* denoted as c_{ij}. For any flow f in G

$$cost(f) = \sum_{(i,j)\in E} c_{ij} \times f(i, j) \tag{4.1}$$

It is a well known fact that if the capacity bounds of a flow problem are integral and there exists a feasible flow for the network, then the maximum and minimum flows between any two nodes flows are also integral on all arcs in the network. Hence, if there exists a feasible flow in a network there also exists an integral feasible flow. Moreover, if there exists an integral feasible flow in G and all costs are integral then by Equation (4.1) the cost of f is also integral. In this paper when we refer to a feasible flow or a minimum cost flow we always mean an *integral* feasible flow and an *integral* minimum cost flow.

4.2.3 Set Constraint Satisfaction Problem

A set constraint satisfaction problem [1] is defined as a triple (X, D, Cs) where

- $X = \{x_1, \ldots, x_n\}$ is a finite set of variables.
- $D = \{D_1, \ldots, D_n\}$ is a set of finite sets of elements such that for each i, x_i takes as value a subset of D_i.

- Cs is a set of constraints on the values particular subsets of the variables in X can simultaneously take. Each constraint $C \in Cs$ constrains the values of a subset $X(C) = \{x_{C_1}, \ldots, x_{C_k}\}$ of the variables in X and may be thought of as a subset $T(C)$ of the Cartesian product $= C_{C_1} \times \ldots, \times C_{C_k}$ where each $C_{C_i} = \{C \mid C \subseteq D_{C_i}\}$.

The set constraint satisfaction problem differs from the constraint satisfaction problem as described in e.g. [6] by the fact that in general the value, which can be assigned to any $x_i \in X$, is in the powerset of D_i.

Definition 4.2.2. *A set constraint satisfaction problem is consistent if and only if there exists an assignment P with the following properties:*

1. *For each variable $x_i \in X$ with domain D_i, the value $P(x_i)$ assigned to x_i by P must be a subset of D_i.*
2. *For each constraint $C \in Cs$ and each variable in $X(C) = \{x_{C_1}, \ldots, x_{C_k}\}$ the tuple $\langle P(x_{C_1}), \ldots, P(x_{C_k})\rangle \in T(C)$.*

Let $X(C)$ be the set of constraint variables and $D(X(C))$ their domains. A value $v \in D(x_i)$ for x_i is consistent with C iff $\exists P(P(X(C)) \in T(C))$ such that v is an element in the value $P(x_i)$.

A value graph [7] of a constraint C is a bipartite graph $GV(C) = (X(C) \cup D(X(C)), E)$, where $(x, v) \in E$ if and only if $v \in D_x$.

4.3 Symmetric Cardinality Constraint

The symmetric cardinality constraint is specified in terms of a set of variables $X = \{x_1, \ldots, x_k\}$, which take their values as subsets of $V = \{v_1, \ldots, v_n\}$. The cardinality of the set assigned to each variable is constrained by the interval $[l_{x_i}, u_{x_i}]$, where l_{x_i} and u_{x_i} are non–negative integers. In addition, it constrains the number of occurrences of each value $v_j \in V$ in union of the sets assigned to variables in X to be in an interval $[l_{v_j}, u_{v_j}]$. Both l_{v_j} and u_{v_j} are non–negative integers.

More formally, the symmetric cardinality constraint is defined as follows.

Definition 4.3.1. *A symmetric cardinality constraint is a constraint C over a set of variables $X(C)$ which associates with each variable $x_i \in X(C)$ two non-negative integers l_{x_i} and u_{x_i}, and with each value $v_j \in D(X(C))$ two other non-negative integers l_{v_j} and u_{v_j} such that a restriction of an assignment P to the variables in $X(C)$ is an element in $T(C)$ iff*

$\forall i \, (l_{x_i} \leq \#(x_i, P) \leq u_{x_i})$ *and* $\forall j \, \left(l_{v_j} \leq \#(v_j, C, P) \leq u_{v_j}\right)$,
where $\#(x_i, P)$ *is the cardinality of the set assigned to* x_i *by* P *and* $\#(v_j, C, P)$ *is the number of variables to which* v_j *is assigned by* P.

Consistency of a symmetric cardinality constraint C is achieved by computing a feasible flow in a particular value network $N(C)$ obtained from a value graph of C by

- orienting each edge of $N(C)$ from values to variables. Since each value can occur in a subset assigned to a variable at least 0 and at most 1 time for each arc $(v, x) : l(v, x) = 0, u(v, x) = 1$
- adding a source node s and connecting it which each value. For every arc $(s, v_i) : l(s, v_i) = l_{v_i}, u(s, v_i) = u_{v_i}$
- adding a sink node t and an arc from each variable to t. For each such arc $(x_i, t) : l(x_i, t) = l_{x_i}, u(x_i, t) = u_{x_i}$
- adding an arc (t, s) with $l(t, s) = 0$ and $u(t, s) = \infty$

The complexity of computing the consistency of a symmetric cardinality constraint is the same as the complexity of computing a feasible flow in $N(C)$, which in worst case is the same as for computing a maximum flow in $N(C)$. Computing a maximum flow in a network depends on the used algorithm (see [9] for comparison of different maximum flow algorithms), but it can be approximated to $O(n^3)$, where n is a number of nodes in a network. This gives a complexity relative to the number of constraint variables, $|X|$, and the size of the union of all their domains $|D(X(C))|$, of $O((|X| + |D(X(C))|)^3)$ time.

Given a flow f in $N(C)$ we can obtain *the residual graph* for f, i.e. the network representing utilization and remaining capacity in the network with respect to f, as follows.

Let $N(C)$ be a value network of C, the residual graph of $N(C)$ with respect to a flow f, denoted by $R(f)$, is a graph with the same set of nodes as $N(C)$. The arc set of $R(f)$ is defined as follows.

- if $f(i, j) < u_{ij}$ then $(i, j) \in R(f), r_{ij} = u_{ij} - f(i, j)$
- if $f(i, j) > l_{ij}$ then $(j, i) \in R(f), r_{ji} = f(i, j) - l_{i,j}$

r_{ij} denotes the residual capacity of (i, j) The residual capacity of a path p, denoted by $r(p)$, is a minimum value r_{ij} for all $(i, j) \in p$.

A value v_j of a variable x_i is inconsistent with C if and only if there is no feasible flow in $N(C)$ which contains a flow on arc (v_j, x_i). Given a feasible

flow f, a flow on (v_j, x_i) is the same in any feasible flow f' if and only if neither (v_j, x_i) nor (x_i, v_j) is contained in a strongly connected component in $R(f)$ involving at least three nodes (see Theorem 1 in [1]). Thus, if a flow on (v_j, x_i) is equal to 0, and (v_j, x_i) is not contained in a strongly connected component in $R(f)$ involving at least three nodes, then, the value v_j of x_i is inconsistent with C.

There exist a number of algorithms for computing strongly connected components in a graph (see [10] p. 560 for references). The algorithm given in [11] computes strongly connected components in $O(n + m)$ time, where n is a number of nodes and m is a number of edges in a graph. In terms of symmetric cardinality constraint $n = |X(C)| + |X(D(C))|$ and $m = |X(C)| + 2 * |X(D(C))| + 2$, thus, the complexity of filtering the domains of the variables in C is $O(|X| + |X(D(C))|)$ time.

4.4 Consistency of Symmetric Cardinality Constraint with Costs

The symmetric cardinality constraint with costs extends the symmetric cardinality constraint with a cost function.

Definition 4.4.1. *A cost function on a variable set X is a function which associates with each element $v_j \in D(x)$ a non–negative integer denoted by $cost(x_i, v_j)$.*

The following gives a definition of symmetric cardinality constraint with cost as a conjunction of a symmetric cardinality constraint and a sum constraint.

Definition 4.4.2. *A symmetric cardinality constraint with cost is a constraint C over a set of variables $X(C)$ which associates with each variable $x_i \in X(C)$ two non-negative integers l_{x_i} and u_{x_i}, with each value $v_j \in D(X(C))$ two other non-negative integers l_{v_j} and u_{v_j} a cost function on each $x_i \in X(C)$ and an integer H such that a restriction of an assignment P to the variables in $X(C)$ is an element in $T(C)$ iff*

- $\forall i \, (l_{x_i} \leq \#(x_i, P) \leq u_{x_i})$
- $\forall j \, \left(l_{v_j} \leq \#(v_j, C, P) \leq u_{v_j}\right)$.
- $\sum_{i=1}^{|X(C)|} cost(x_i, P) \leq H$

where $\#(x_i, P)$ is the cardinality of the set assigned to x_i by P, $\#(v_j, C, P)$ is the number of variables to which v_j is assigned by P and $cost(x_i, P) = \sum_{v_j \in D(x_i)} cost(x_i, v_j)$ for each v_j assigned to x_i by P.

To achieve consistency of a symmetric cardinality constraint with costs we extend the value network of symmetric cardinality constraint, $N(C)$, with the cost function by

- associating with each arc $(v_j, x_i) \in N(C)$ a cost $c_{v_j,x_i} = cost(x_i, v_j)$
- associating with every arc (s, v_j) $c_{sv_j} = 0$
- associating with every arc (x_i, t) $c_{x_i t} = 0$
- associating with (t, s) $c_{ts} = 0$

Note that the value network of a symmetric cardinality constraint with costs is independent of H.

The following proposition defines consistency of a symmetric cardinality constraint with costs.

Proposition 4.4.1. *Let C be a symmetric cardinality constraint with cost and $N(C)$ be a value network of C. The following properties are equivalent.*

- *C is consistent*
- *there exists a flow from s to t in $N(C)$ which satisfies lower and upper bounds of capacities on the arcs in $N(C)$ and with cost less than or equal to H.*

Proof. Assume that C is consistent, thus $T(C) \neq \emptyset$. Consider $P \in T(C)$. We can build a function f in $N(C)$ with the following properties:

1. $\forall x_i \in X(C), f(x_i, t) = \#(x_i, P)$
2. $\forall x_i \in X(C), f(v_j, x_i) = 1$ if v_j appears in the subset $P(x_i)$, otherwise $f(v_j, x_i) = 0$
3. $\forall v_j \in D(X(C)), f(s, v_j) = \#(v_j, C, P)$
4. $cost(f) = \sum_{i=1}^{|X(C)|} cost(x_i, P) \leq H$

Properties (1) – (3) are the feasibility properties of a symmetric cardinality constraint and are proved in [1]. By these properties, if C is consistent then there exists a feasible flow f in $N(C)$. Recall from Equation (4.1) that the cost of a flow is a sum of products of costs associated with an arc and an amount of flow on respective arc. Since the cost associated with arcs other then arcs between $x_i \in X$ and $v_j \in D(X(C))$ is equal to 0 and, by Property 2, flow f from any v_j to any x_i is equal to 1 if v_j is in a subset $P(x_i)$ then the cost of such flow f is equal to the sum of cost of all $x_i \in X$ in P. Consequently, if C is consistent then $\sum_{i=1}^{|X(C)|} cost(x_i, P) \leq H$ and $cost(f) \leq H$, which proves Property 4.

On the other hand, assume that there exists a feasible flow f from s to t in $N(C)$ with cost lower than or equal to H. Since f is feasible, then $\forall x \in X(C) l_{x_i} \leq \#(x_{C_i}, P) \leq u_{x_i}$, $f(x_i, t) = \#(x_i, P)$ and $\forall v \in D(X(C)) l_{v_j} \leq \#(v_j, C, P) \leq u_{v_j} \wedge f(s, v_j) = \#(v_j, C, P)$, and the set of arcs such that $f(i, j) = 1$ corresponds to the set of edges of value graph of a symmetric cardinality constraint with costs, which is proved in [1]. Since the arc cost for any arc (s, i) and (i, t), which includes direct arcs between s and t, is equal to 0 and the flow between any i and j can be at most 1, then $cost(f)$ is equal to the sum of the costs of the arcs between i and j where $f(i, j) = 1$. Since the set of such arcs corresponds to the set of arcs in the value graph of C thus $cost(f) = \sum_{i=1}^{|X(C)|} cost(x_i, P)$. Consequently, if $cost(f) \leq H$ so is $\sum_{i=1}^{|X(C)|} cost(x_i, P)$, which proves Proposition 4.4.1. □

This proposition gives a way of computing the consistency of a symmetric cardinality constraint with costs by computing a feasible flow f in $N(C)$ and checking if $cost(f) \leq H$. Since H is independent of $N(C)$ every time this equality does not hold we would need to verify if there exists an other feasible flow f' in $N(C)$ such that $cost(f') < cost(f)$ and $cost(f') \leq H$.

On the other hand, if a feasible flow f in $N(C)$ is a minimum cost flow we can verify that C is consistent if $cost(f) \leq H$, otherwise C is inconsistent.

4.5 Minimum Cost Flow

There exists several methods of obtaining minimum cost flow in a network. A survey of such methods is given in [5]. Here we describe a simple variant of successive shortest path algorithm from [5].

Given a value network for a symmetric cardinality constraint with costs and a flow f, we can build the residual graph $R(f)$ in the same way as for the

symmetric cardinality constraint and extend it with arc costs by

- associating with each arc (i, j) with free capacity its residual cost $rc(i, j) = cost(i, j)$
- associating with each arc (j, i) representing a flow which exceeds the lower bound of (i, j) a residual cost $rc(j, i) = -cost(i, j)$.

For any path p the residual cost of p, denoted $rc(p)$ is a sum of residual costs of all arcs $(i, j) \in p$.

Algorithm 13 Minimum Cost Flow

1: Start with the zero flow f.
2: Pick an arc (i, j) such that $f(i, j)$ violates the lower bound for the flow from i to j.
3: Find p a shortest path from j to i in $R(f) - \{(i, j)\}$.
4: Obtain a new flow f' from f by sending a flow along p and set $f = f'$
5: Goto 2

The algorithm, listed here as Algorithm 13, computes a minimum cost flow in N by repeatedly choosing an arc (i, j) with a flow, which violates the lower bound constraint, and computes a shortest path p from j to i with respect the costs of an arc. After a path p is found a new flow f' is obtained by sending a flow along p. If, at some point, there is no path for the current flow then there is no feasible flow in N. Otherwise, obtained flow is feasible and is a minimum cost flow.

The complexity of the algorithm is dependent on the complexity of the implemented shortest path algorithm. However, the most powerful shortest path algorithms require that the costs on all arcs in a graph are non-negative.

The standard method for transforming all arcs costs to non-negative values is to use costs relative to costs associated with incident nodes of an arc. These costs are usually referred to as *reduced costs* and the costs associated with incident nodes are referred to as *node potentials*. More formally,

Definition 4.5.1.

- *a potential function is a function π which associates with each node $i \in N$ a number $\pi(i)$, which is referred to as a node potential*
- *with respect to the node potentials, the reduced cost c_{ij}^{π} of an arc (i, j) in $R(f)$ is defined by $c_{ij}^{\pi} = rc(ij) - \pi(i) + \pi(j)$*

Given a minimum cost flow f the potential function $\pi^f(i) = d_j^f(i)$, where $d_j^f(i)$ represents the shortest path distance in $R(f)$ from node j to every node $i \in N$. Starting with the zero flow,f^0, all $\pi^{f^0}(i) = 0$ and consequently all $c_{ij}^{\pi} = rc(i,j)$. After each iteration of the algorithm, which computes shortest paths from given node j to every other node in the graph, the potential of each node i is updated with $\pi'(i) = \pi(i) - d(i)$. Then, the cost of each arc in the residual graph can be converted to non-negative reduce cost as in Definition 4.5.1. For justification of this method see [5].

The fastest algorithm for computing shortest path in a graph with non-negative arc lengths is Dijkstra algorithm implemented with Fibonacci heaps (see [12, 5] for comparison). The algorithm requires $O(m + n\ log\ n)$ time, where m is the number of arcs and n is the number of nodes in a graph. The number of iterations is bounded by the number of lower bounds on the flow on each arc, which is $\sum l(i,j)$. Thus, the complexity of the shortest paths algorithm can be bounded by $O(\sum l(i,j) \times (m + n\ log\ n))$ time.

In terms of the symmetric cardinality constraint $n = |X(C)|+|D(X(C))|+2$ and $m = \sum_{i=1}^{|X(C)|} D(x_i) + |X(C)| + |D(X(C))| + 2$. The number of iterations is bounded by $\sum_{i=1}^{|X(C)|} l_{x_i} + \sum_{j=1}^{|D(X(C))|} l_{v_j}$, that is by the sum of lower bounds imposed on the cardinality of all variables and lower bounds imposed on the number of occurrences of all values in the union of the sets assigned to the variables of the constraint.

4.6 Filtering of Symmetric Cardinality Constraint with Costs

By Proposition 4.4.1, a symmetric cardinality constraint with costs C is consistent if and only if there is a feasible flow in the value network of C, $N(C)$, which cost is less than or equal to H. Thus, if there is a feasible flow in $N(C)$ with an overall cost less than or equal to H which contains a flow along (v_i, x_i) then the value v_i of x_i is consistent with C. Assuming that C is a consistent symmetric cardinality constraint with costs and f is a minimum cost flow in $N(C)$ we can formulate the following proposition.

Proposition 4.6.1. *A value v_j of a variable x_i is inconsistent with C if and only if for all feasible flows f in $N(C)$*

1. $f(v_j, x_i) = 0$ *or*

2. $d_{R(f)-\{(x_i,v_j)\}}(x_i, v_j) > H - cost(f) - rc(v_j, x_i)$

Proof. Property 1 states that the value v_j of the variable x_i is inconsistent with C if there is no feasible flow in $N(C)$ containing (v_j, x_i) which is proved in Proposition 4.4.1

Furthermore, it is proved (see [8], p. 130) that if there exists a path p from node j to node i in $R(f) - \{(j, i)\}$ then the flow f' obtained from f by sending k units of flow along p has a cost $cost(f') = cost(f) + k(rc(i, j) + d_{R(f)-\{(j,i)\}}(j, i))$. Since the maximum amount of flow which can be pushed through any (i, j) is equal to 1 we obtain $cost(f') = cost(f) + rc(i, j) + d_{R(f)-\{(j,i)\}}(j, i)$. By Proposition 4.4.1 C is consistent if cost of a feasible flow is lower than or equal to H, thus if $cost(f') > H$ than C is inconsistent. By substituting $cost(f')$ we obtain $cost(f)+rc(i, j)+d_{R(f)-\{(j,i)\}}(j, i) > H$, which gives $d_{R(f)-\{(x_i,v_i)\}}(x_i, v_j) > H - cost(f) - rc(v_j, x_i)$. This proves the second property. □

By Proposition 4.6.1, given an arc (v_j, x_i) such that $f_{v_j x_i} = 0$, if there exists a path from x_i to v_j in $R(f) - \{(x_i, v_j)\}$ then there exists a feasible flow in $N(C)$ containing (v_j, x_i). Moreover, if $d_{R(f)-\{(x_i,v_i)\}}(x_i, v_i) \leq H - cost(f) - rc(v_i, x_i)$ then such a flow has a cost less than or equal to H. Thus, a value v_j of a variable x_i is consistent with C.

Note that due to a special structure of the residual graph obtained from a flow in $N(C)$, if $f(v_j x_i) = 0$ then $R(f)$ does not contain (x_i, v_j). Thus, no modification of $R(f)$ is required in order to perform this computation. The special structure of the residual graph will also insure that a simple path from x_i to v_j will contain at least tree nodes.

Given a minimum cost flow f in a value network of a symmetric cardinality constraint $N(C)$, the consistency of each value $v_j \in D(x_i)$ is verified in $O(|X(C)| \times (m + n\ log\ n))$ time, which is the same as for filtering the global cardinality constraint with costs ([2, 4]).

4.7 Conclusion

In this paper we have introduced the symmetric cardinality constraint with cost. Moreover, we have presented methods for computing its consistency and methods for filtering the domains of its variables. We show that the time complexity for verifying the consistency of a symmetric cardinality constraint with costs and for filtering domains of constraint variables is the same as for the global cardinality constraint with costs.

Other variants of the symmetric cardinality constraint with cost, like those constraining the cost associated with the values assigned to individual variables etc., can be easily derived from the constraint described here.

Bibliography

[1] Kocjan, W., Kreuger, P.: Filtering Methods for Symmetric Cardinality Constraint. Proceedings of 1th Int. Conf. CPAIOR (2004)

[2] Régin, J.-Ch.: Generalized Arc Consistency for Global Cardinality Constraint. Proc. of the Fourteenth National Conference on Artificial Intelligence (AAAI-96) (1996)

[3] Régin, J.-Ch.: Arc Consistency for Global Cardinality Constraints with Costs. Principles and Practice of Constraint Programming — CP'99, Conference Proceedings, (1999) 390–404

[4] Régin, J.-Ch.: Cost–Based Arc Consistency for Global Cardinality Constraints. Constraints, **7** (2002) 387–405

[5] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows. Theory, algorithms and applications. Prentice–Hall Inc., (1993)

[6] Tsang, E.: Foundation of Constraint Satisfaction Academic Press (1993)

[7] Laurière, J.-L.: A language and a program for stating and solving combinatorial problems. Artificial Intelligence, **10** (1978) 29–127

[8] Lawler, E.: Combinatorial Optimization: Network and Matroids, Holt, Rinehart and Winston (1976)

[9] Ahuja, R.K., Kodialam, A., Mishra, A.K., Orlin, J.B.: Computational Investigations of Maximum Flow Algorithms. European Journal of Operational Research, **97** (1997) 509–542

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Second Edition, The MIT Press (2001)

[11] Tarjan, R.E.: Depth–First Search and Linear Graph Algorithms. SIAM J. Computing, **1** (1972) 146–160

[12] Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest Paths Algorithms: Theory and Experimental Evaluation. Mathematical Programming **73** (1996) 129–174

Chapter 5

Paper D: Maintaining Consistency of Dynamic Cardinality Constraints with Costs

Waldemar Kocjan, Per Kreuger, Björn Lisper

Revised and extended version of

MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-168/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, January 2005

Abstract

This paper introduce a novel method for maintaining consistency of cardinality constraints in context of dynamic constraint satisfaction. The presented method adopt sensitivity analysis for feasible and minimum cost flows underlying all cardinality constraints. Moreover, we discuss the problem of maintaining consistency of a value with a cardinality constraint by maintaining multiple shortest paths in the residual graph of a constraint.

5.1 Introduction

Within constraint programming we have seen the development a number of methods that adopt and extend assignment models known from optimization and operation research for constraint consistency and filtering. Global constraints which deal with the cardinality of the variables and values in a problem are typical examples of such methods.

To this class of constraints belongs, e.g., the constraints of difference [20] and the global cardinality constraint [21], as well as their weighted versions [4, 11] resp. [22, 23]. These models have been extended to set constraint satisfaction problems by the introduction of the *symmetric cardinality constraint* [15] and its weighted version [16]. The newest constraint in this class is cardinality matrix constraint introduced in [24]. This constraint is more general than those mentioned above, however, it does not attribute costs to the values associated with domains of constraint variables.

Cardinality constraints can be used to model and effectively solve a variety of problems such as sports scheduling [10], rostering [18], Latin square problems (ex. [24]) and Sudoku puzzle [25], to name a few.

Like most of the methods used in constraint programming, those mentioned above deal mainly with problems which demand only a single solution, and they assume completely known and persistent problem parameters. Unfortunately, this is seldom the case in real life problems. Many assignments problems need to be re-computed due to changes to, e.g., the costs of assignments or in the bounds of individual variables, and even after adding and removing variables.

Constraint Programming problems in which parameters change over time are usually solved by recomputation from scratch [6, 14]. Some other methods focus on modifying the current solution [3, 28]. Dynamic global constraints are subject of [2] and [13]. Paper [2] deals with constraint of difference which is a subject of monotonic changes and relies on backtracking in context of Constraint Logic Programming. Paper [13] deals with flow constraint and constraint of difference excluding the cost function.

Here we introduce a method for maintaining consistency of cardinality constraints with costs in the context of dynamic constraint satisfaction. The methods presented here are based on sensitivity analysis of the feasible and minimum cost flows underlying every cardinality constraint.

This paper is organized as follows. In Section 5.2 we describe some basic concepts in flow theory and constraint satisfaction. Section 5.3 gives a brief description the symmetric cardinality constraint with costs, and presents meth-

ods for checking the consistency of the constraint and for filtering the domains of the constrained variables. Section 5.4 introduces basic notions of sensitivity analysis, and the following sections, 5.5 – 5.7 describe how sensitivity analysis can be used to maintain consistency of a cardinality constraint as well as for maintaining the optimality of an underlying minimum cost flow in the value network of a cardinality constraint. Following Section 5.9 discusses the issue of maintaining consistency of each value with the modified constraint. The performance of the methods presented is then evaluated on a set of benchmarks. Results of evaluation are presented in Section 5.10. Finally, Section 5.11, concludes the paper and discusses future work.

5.2 Preliminaries

5.2.1 Graphs

Definitions in this section follows the presentation in[1].

A *directed graph* $G = (X, E)$ consists of a set of nodes (vertices) X and arcs (edges) E, where every arc $(u, v) \in E$ is an ordered pair of distinct nodes.

An arc (u, v) connects node u with node v, i.e. in directed graph it is an arc oriented from node u to node v. A *path* in a graph G from v_1 to v_k is a sequence of nodes $[v_1, \ldots, v_k]$ such that each (v_i, v_{i+1}) is an arc for $i \in [1, \ldots, k-1]$. The path is *simple* if all its nodes are distinct.

5.2.2 Network Flows

Definition 5.2.1. *A* network $G = (X, E)$ *is a directed graph, in which each arc* $e \in E$ *is associated with two non-negative integers* l_e *and* u_e *representing the lower and upper bounds, respectively, on the flow on* e*.*

The upper bound on the flow on an arc is also referred to as the *capacity* of the arc.

A flow f_e on an arc e represents the amount of commodity that the arc accommodates. More formally:

Definition 5.2.2. *A* flow f *in a network* (X, E) *is a function that assigns to each arc* $(i, j) \in E$ *a value* f_{ij} *in such way that for each node* $p \in X$*,*

$$\sum_{(i,p)\in E} f_{ip} = \sum_{(p,j)\in E} f_{pj}$$

This property is known as a *conservation law* and states that for each node in G, the incoming amount of flow of some commodity equals the amount of that commodity leaving that node.

Definition 5.2.3. *A flow f in a network (X, E) is* feasible *if, for each $e \in E$, it holds that $l_e \leq f_e \leq u_e$.*

Associate furthermore with each arc $(i, j) \in E$ a *cost* c_{ij} that denotes the cost per unit of flow on the arc. For any flow f in G, we define

$$cost(f) = \sum_{(i,j)\in E} c_{ij} * f_{ij} \tag{5.1}$$

A *minimum cost flow problem* is the problem of finding a feasible flow in G with minimal cost.

In this paper we assume that all the parameters (c_{ij}, l_e and u_e) of a flow problem are integral, which guarantees the existence of an integral minimal solution to the problem. (For the integrality property of minimum cost flow problems see, e.g., [1, p. 318]).

The *residual graph* $R(f)$ is a graph representing the utilization and remaining capacity in the flow graph with respect to a flow f.

Definition 5.2.4. *Given a flow f from s to t in the network G, the* residual graph *for f, denoted $R(f)$, has the same set of nodes as G. The arc set of $R(f)$ is defined as follows. For all arcs (i, j) in G,*

- *if $f_{ij} < u_{ij}$ then (i, j) is an arc of $R(f)$ with residual capacity $r_{ij} = u_{ij} - f_{ij}$, and cost c_{ij}*
- *if $f_{ij} > l_{ij}$ then (j, i) is an arc of $R(f)$ with residual capacity $r_{ji} = f_{ij} - l_{ij}$ and cost $-c_{ij}$*

Moreover, the *potential function* and the *reduced costs* are defined as follows.

Definition 5.2.5. *A* potential function *is a function π which associates with each node $i \in G$ a number $\pi(i)$, which is referred to as a node potential. With respect to the node potentials, the* reduced cost c^{π}_{ij} *of an arc (i, j) in $R(f)$ is defined by $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$.*

For minimum cost flows node potentials are typically defined by $\pi(i) = -d$, where d is a shortest path distance from a given source to i.

5.2.3 Constraint Satisfaction

A *constraint satisfaction problem (CSP)* is a triple (X, D, C) where X is a finite set of variables $\{x_1, \ldots, x_n\}$, $D = \{D_1, \ldots, D_n\}$ is a set of finite domains, where D_i is a set of values for the variable $x_i \in X$, and $C = \{C_1, \ldots, C_n\}$ is a set of constraints between variables [27].

The symmetric cardinality constraint used in this paper is defined in context of the *set constraint satisfaction problem (sCSP)*, which differs from CSP by the fact that each $x_i \in X$ is assigned a *subset* of elements in D_i [15].

Let $\mathcal{P}$ be a (set) constraint satisfaction problem. A *dynamic constraint satisfaction problem* $DCSP = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ [7] is a sequence of (s)CSPs such that any $\mathcal{P}_i$ differs from the consecutive $\mathcal{P}_{i+1}$ by a set of added and a set of deleted constraints.

Note, that a dynamic constraint satisfaction problem is a sequence of static problems. Any constraint can be added to or deleted from the problem after the solution to the previous problem was found.

5.3 The Symmetric Cardinality Constraint with Costs and Other Cardinality Constraints.

In this section we will briefly describe the symmetric cardinality constraint with costs, which we will use later to illustrate the sensitivity analysis. A detailed description of this constraint can be found in [16].

Definition 5.3.1. *A symmetric cardinality constraint with cost is a constraint C over a set of variables $X(C)$ and a set of values obtained from the domains of these variables:*

$$D(X(C)) = \bigcup_{x \in X(C)} D_x$$

The constraint associates with each value $v_j \in D(X(C))$ two non-negative integers l_{v_j} and u_{v_j}, with each variable $x_i \in X(C)$ two other non-negative integers l_{x_i} and u_{x_i} and with the occurrence of a value $v_j \in D_{x_i}$ in a set assigned to a variable x_i a cost $cost(x_i, v_j)$. Furthermore it associates a fixed integer limit H on the total cost of the constraint as defined below.

1. $\forall i \, (l_{x_i} \leq \#(x_i, P) \leq u_{x_i})$
2. $\forall j \, \left(l_{v_j} \leq \#(v_j, C, P) \leq u_{v_j}\right)$.

3. $\sum_{i=1}^{|X(C)|} \sum_{v_j \in D_{x_i}} cost(x_i, v_j) \leq H$

where $\#(x_i, P)$ is the cardinality of the set assigned to x_i by P and $\#(v_j, C, P)$ the number of variables to which v_j is assigned by P.

The symmetric cardinality constraint with costs extends the symmetric cardinality constraint of [15] with a cost associated with each individual assignment. The constraint can also be seen as an extension of the global cardinality constraint with costs [23] into the set constraint satisfaction setting.

For each global cardinality constraint with costs, there is an equivalent symmetric cardinality constraint with costs where the cardinality of each constraint variable is restricted to be in the interval $[1, 1]$. Similarly, for a constraint of difference with costs there is a symmetric cardinality constraint with costs where the bounds of each variable are restricted to $[1, 1]$, the bounds of each value is restricted to $[0, 1]$, and $H = \infty$.

The cardinality matrix constraint [24] implements multiple symmetric cardinality constraint. This constraint can easily be extended with a cost function by using multiple symmetric cardinality constraint with costs.

The first step of verifying the consistency of a symmetric cardinality constraint with costs C is to build its value network $N(C)$.

Definition 5.3.2. *A value network of a symmetric cardinality constraint is a network obtained by*

1. *Adding a node for each variable $x_i \in X(C)$, a node for each value in $v_j \in D(X(C))$, a source node s and a sink node t.*
2. *Adding an arc (s, v_j) from s to each node v_j and bounding the flow on it to be between l_{v_i} and u_{v_i} at zero cost $c_{sv_i} = 0$.*
3. *Adding an arc from each node $v_i \in D_{x_i}$ to x_i, and bounding the flow on each such arc to be between 0 and 1 at a cost equal to that of the occurrence of v_i in x_i.*
4. *Adding an arc (x_i, t) from each node x_i to t and bounding the flow on it to be between l_{x_i} and u_{x_i} at zero cost $c_{x_i t} = 0$.*
5. *Adding an arc from t to s and bounding the flow on this arc to be between 0 and ∞ at a zero cost $c_{t,s} = 0$.*

The following theorem relates consistency of a symmetric cardinality constraint to a flow in its value network.

Theorem 5.3.1. *A symmetric cardinality constraint with costs C is consistent if and only if there exists a feasible flow f^C in the value network $N(C)$ such that $cost(f^C) \leq H$.*

Proof. See [16] for proof. □

In practice, computing consistency of a symmetric cardinality constraint with costs C is done by computing a minimum cost flow in $N(C)$. Several algorithms can be used. For the state of the art in computing minimum cost flow see [1].

In this paper we assume that minimum cost flows are computed using the *successive shortest path algorithm*. Briefly, the successive shortest path algorithm searches for the shortest path from node j in an arc (i, j), whose lower bound on the flow is not satisfied, iteratively to every other node in the graph. After each iteration, the algorithm augments the flow on (i, j) along the shortest path from j to i and updates the residual graph relative to the newly computed flow. Moreover, the potential of each node k, initially set to 0, is updated by subtracting its distance from j. For a detailed description see [1], pp. 320-324.

The best known algorithm for computing shortest path is Dijkstra's algorithm with Fibonacci heaps implementation. Using this algorithm, the worst time complexity for verifying the consistency of C becomes

$$O((\sum_{i=1}^{|X|} l_{x_i} + \sum_{j=1}^{|D|} l_{v_j}) \times (m + n\ log\ n))$$

where m is the number of edges and n the number of nodes in the value network. Dijkstra's algorithm requires that the arcs in the network have non-negative costs, which can be ensured by transforming original costs into reduced costs (see [16] for details).

A minimum cost flow implies that the reduced cost of any arc in the graph is non-negative ([1], p. 308-309, see also Theorem 5.4.2 in Section 5.4). Given a minimum cost flow f^C in the value network of $N(C)$, we can compute the consistency of each value v_j in x_i by computing the shortest path from x_i to v_j in the residual graph of f^C, and checking if there exists a feasible flow containing (v_j, x_i) with cost lower than or equal to H.

5.4 Sensitivity Analysis

The purpose of sensitivity analysis is to determine changes in the solution resulting from the changes in the problem parameters. Here, we are interested in a sensitivity analysis of the *consistency* of a symmetric cardinality constraint with costs, as well as a sensitivity analysis of a *minimum cost flow* in the value network of the constraint. The results apply also to global cardinality constraints with costs and constraints of difference with costs, since these are instances of the symmetric cardinality constraint with costs.

Let $\mathcal{C} = \{C_1, \ldots, C_z\}$ be a dynamic constraint satisfaction problem consisting of a sequence of symmetric cardinality constraint with costs. For each $C_q \in \mathcal{C}$ we will denote its value network by $N(C_q)$, the flow in its network by f^{C_q}, and the residual graph of its flow by $R(f^{C_q})$.

Recall from Section 5.3 that a symmetric cardinality constraint with costs C_q is consistent iff there exists a feasible flow f^{C_q} in $N(C_q)$ such that $cost(f^{C_q}) \leq H$, where H is a non-negative integer. Moreover, a flow is feasible if it satisfies the lower and upper bound constraints of each arc in the graph. Consequently, a sensitivity analysis for consistency of a symmetric cardinality constraint with costs is an analysis of how changes to the parameters of a problem influence the feasibility of the flow in the value network of the constraint.

Computing a feasible flow is performed by augmenting and reducing a flow on arcs in a graph. Given a feasible flow we can compute a new one using the following theorem [17].

Theorem 5.4.1. *Let f be a feasible flow in a network N and $R(f)$ be a residual graph for f. If there is a simple path p from n to m in $R(f) - \{(n,m)\}$ then we can obtain a new feasible flow f' in N such that $f'_{mn} > f_{mn}$. Such a path is referred to as an augmenting path.*

Moreover, if there is a simple path p from m to n in $R(f) - \{(m,n)\}$ then we can obtain a new feasible flow f' in N such that $f'_{mn} < f_{mn}$. Such a path is referred to as a reducing path.

If no augmenting path for f_{mn} exists then f_{mn} is a *maximum flow* through (m,n). Similarly, if there is no reducing path for f_{mn} then f_{mn} is a *minimum flow* through (m,n).

Performing sensitivity analysis of consistency of a cardinality constraint requires information about the value network of the constraint and the feasible flow found during the computation. To make this information accessible we can store the residual graph resulting from computing a feasible flow in the value network of the constraint. Let n denote a number of nodes in the graph

and m number of edges. Storing information about the residual graph requires at most $O(n+m)$ space, where $n = |X| + |D| + 2$ and m in the worst case is equal to $\sum_{i=1}^{|X|} |D_{x_i}| + 2|D| + 2|X| + 2$.

In many situations it is interesting to maintain the minimum cost flow in the value network of a cardinality constraint. This is the case when a cardinality constraint is a subject to an objective function which minimizes the cost of the occurrence of all values, or when we are interested in maintaining the consistency of each value with the constraint. In the second case, maintaining a minimum cost flow makes it possible to use more powerful algorithms for computing shortest paths in the graph.

There are several methods for verifying the optimality of a flow in a graph (see [1], p. 306–315). Here we will use the *reduced cost optimality condition*.

Theorem 5.4.2. ***(Reduced Cost Optimality Condition).*** *A feasible flow f is a minimum cost flow if and only if some set of node potentials π satisfy the following reduced cost optimality conditions:*

$$c_{ij}^{\pi} \geq 0 \textit{ for every arc } (i,j) \textit{ in } R(f) \tag{5.2}$$

The reduced cost c_{ij}^{π} of an arc (i,j) indicates that augmenting or reducing a flow through (i,j) with 1 unit of commodity will change the total cost of the flow in a graph with c_{ij}^{π} (see [1], p. 43–44).Consequently, if $c_{ij}^{\pi} < 0$ then augmenting a flow through (i,j) might reduce the total cost of the flow in the graph with c_{ij}^{π}. Thus, if there exists a path p from j to i in $R(f) - \{(j,i)\}$ then, by Theorem 5.4.1, we can obtain a new feasible flow f' by augmenting a flow along p. Moreover, if $d_{ji} < -c_{ij}^{\pi}$ then $cost(f') < cost(f)$. If p is a shortest path from j to i then $d_{ji} + c_{ij}^{\pi}$ is a greatest possible reduction of $cost(f)$.

If no reducing path p for (i,j) exists then, by Theorem 5.4.1, there is no feasible flow in the graph containing (i,j) thus (i,j) is not contained in a minimum cost flow. Similarly, if $d_{ji} \geq -c_{ij}^{\pi}$ then augmenting a flow through (i,j) does not reduce the total cost of f. In these cases the reduced costs optimality conditions are maintained by adjusting potential of nodes with c_{ij}^{π}. See [1] p. 339 for justification and pp. 320-324 on relation between node potentials and minimum cost flow.

To perform a sensitivity analysis based on Theorem 5.4.2 we need additional information about the previously computed node potentials for each node of the graph. Storing this information requires an additional $|X|+|D|+2$ space.

In the following Sections 5.5 – 5.8 we show how the sensitivity analysis can be used to maintain the consistency of a cardinality constraint with costs and a minimum cost flow in the value network of the constraint.

5.5 Changes to the Cost

By Definition 5.3.2 a change to the cost of an appearance of a value v_i in x_i corresponds to a change of the cost associated with arc (v_i, x_i) in the value network of a constraint.

Let C_q, C_{q+1} be two consecutive cardinality constraints with costs and let $cost(C_q, P)$ denote $\sum_{i=1}^{|X(C)|} \sum_{v_j \in D_{x_i}} cost(x_i, v_j)$ assigned by P in C_q. Assume that C_q is consistent and that C_{q+1} differs from C_q only by a change in the cost of the occurrence of value v_i in x_i. By Definition 5.3.1 if C_q is consistent then there exists an assignment P which satisfies the cardinality and occurrence restrictions, and such that $cost(C_q, P) \leq H$.

Proposition 5.5.1. *The modified constraint C_{q+1} is inconsistent if and only if the cost of an occurrence of any value v_j in x_i such that $f^{C_q}_{v_j x_i} = 1$* increases *with $k > H - cost(C_q, P)$ and there is no path from v_j to x_i in $R(f^{C_q}) - \{(v_j, x_i)\}$ of length $d_{v_j x_i} \leq H - cost(C_q, P) - cost(x_i, v_j)$.*

Proof. By Theorem 5.3.1 v_j appears in a set assigned to x_i by P if there exists, corresponding to P, a feasible flow in the value network of C_q in which $f^{C_q}_{v_j x_i} = 1$. Thus, by Definition 5.3.1, if $f^{C_q}_{v_j x_i} = 0$ then $cost(C_{q+1}, P) = cost(C_q, P)$ and the modified constraint is consistent.

Moreover, if $f^{C_q}_{v_j x_i} = 1$, and the cost of an appearance of v_j in x_i decreases with any $k \leq cost(x_i, v_j)$, then $cost(C_{q+1}, P) \leq (C_q, P) \leq H$. Thus, the modified constraint is consistent. Since, by Definition 5.3.1, $cost(x_i, v_j)$ is non-negative it can be decreased at most with $k = cost(x_i, v_j)$.

Trivially, If $f^{C_q}_{v_j x_i} = 1$ and $cost(x_i, v_j)$ increases with $k \leq H - cost(C_q, P)$ then $cost(C_{q+1}, P)$is lower than or equal to H, thus the modified constraint is consistent.

If $cost(x_i, v_j)$ increases with $k > H - cost(C_q, P)$, then the sum of the costs assigned by P is greater than H. However if there exists a simple path from v_i to x_i of length $d_{v_j x_i}$ in $R(f^{C_q}) - \{(v_j, x_i)\}$ then, by Theorem 5.4.1, a new feasible flow in the value network of the constraint can be obtained by reducing the flow through (v_j, x_i). Moreover if $d_{v_j x_i} \leq H - cost(C_q, P) - cost(x_i, v_j)$ then the cost of the obtained flow is lower than H. Thus, by Theorem 5.3.1, the modified constraint is consistent.

On the other hand, if no such path exists then the flow through (v_j, x_i) can not be reduced, which implies that there is no assignment P, which satisfies Property 3 of Definition 5.3.1. Similarly, if the length of the path $d_{v_j x_i} > H - cost(C_q, P) - cost(x_i, v_j)$, then there is no flow in the value network

of the modified constraint whose cost is lower than or equal to H. Thus by Theorem 5.3.1, there exists no assignment which satisfies Property 3 of the Definition 5.3.1, which indicates inconsistency of the modified constraint. □

This theorem gives us a method for verifying the consistency of a cardinality constraint with dynamically changed costs. However, in some situations it is interesting to maintain the minimum cost flow in the value network of a constraint. To maintain such a flow we use the sensitivity analysis as described in Section 5.4.

First, note that changing the cost of the occurrence of a value with k units changes the cost of the corresponding arc with the same number of units (Definition 5.3.1). By Theorem 5.4.2, when decreasing the cost of the occurrence of a value v_j in x_i we are concerned with recomputing the minimum cost flow in the value network of a cardinality constraint only if the reduced cost of the corresponding arc becomes lower than 0.

Proposition 5.5.2. *Let f^{C_q} be a minimum cost flow in the value network of $N(C_q)$.*

1. *If $f^{C_q}_{v_j x_i} = 1$ then f^{C_q} is a minimum cost flow in the value network of the modified constraint.*

2. *If $f^{C_q}_{v_j x_i} = 0$ then a minimum cost flow in the value network of the modified constraint is obtained by augmenting a flow through (v_j, x_i) along the shortest path from x_i to v_j in $R(f^{C_q}) - \{(x_i, v_j)\}$ if the length $d_{x_i v_j}$ of this path is less than $k - d_{v_j x_i}$.*

Proof. If $f^{C_q}_{v_j x_i} = 0$ and there is no path from x_i to v_j in $R(f^{C_q}) - \{(x_i, v_j)\}$ then, by Theorem 5.4.1 there exists no feasible flow in the value network of the constraint which contains (v_j, x_i). If such a path exists, then redirecting the flow along this path will reduce the total cost of the flow if and only if $d_{x_i v_j} + d_{v_j x_i} - k < 0$, that is: $d_{x_i v_j} < k - d_{v_j x_i}$.

If $f^{C_q}_{v_j x_i} = 1$ then, by Definition 5.2.4 and 5.3.2, there exists an arc (x_i, v_j) in $R(f^{C_q})$ such that $c_{x_i v_j} = -c_{v_j x_i}$. If $cost(x_i, v_j)$ decreases then the cost $c_{x_i v_j}$ increases and by Definition 5.2.5 also $c^{\pi}_{x_i v_j}$ increases. Since $c^{\pi}_{x_i v_j}$ is non-negative for the original problem it is positive for the modified problem. Thus, by Theorem 5.4.2, f^{C_q} is the minimum cost flow in the value network of the modified constraint. □

If no path from v_j to x_i exists optimality conditions are maintained by increasing potential of v_j with k. In a case when the cost the found path does

not reduce the total cost of the flow the potential of nodes are updated with computed distances.

Decreasing the cost of an arc (v_j, x_i) where $f^{C_q}_{v_j x_i} = 0$, in a way that forces rerouting of the flow through (v_j, x_i), changes the total cost of the flow with $d_{x_i v_j} + c^{\pi}_{v_j x_i} - k$ where $c^{\pi}_{v_j x_i} - k < 0$. If decreasing the cost of (v_j, x_i) with k units, when $f^{C_q}_{v_j x_i} = 1$, then the total cost of the flow will also decrease with k.

The following proposition forms the basis for a method for recomputing a minimum cost flow, in the value network of a constraint where the cost of the occurrence of a value in a subset increases.

Proposition 5.5.3. *Let f^{C_q} be a minimum cost flow in the value network of $N(C_q)$ and let the cost of the occurrence of a value v_j in a subset x_i increase with k units.*

1. *If $f^{C_q}_{v_j x_i} = 0$ then f^{C_q} is the minimum cost flow for also the modified constraint.*
2. *If $f^{C_q}_{v_j x_i} = 1$, and the shortest path from v_j to x_i in $R(f^{C_q}) - \{(v_j, x_i)\}$ has length $d_{v_j x_i} < c_{v_j x_i} + k$, then a new minimum cost flow is obtained by reducing the flow through (v_j, x_i) along this path. If no such path exists, or if the length of the shortest path $d_{v_j x_i} > c_{v_j x_i} + k$, then f^{C_q} is a minimum cost flow for the modified problem.*

Proof. If $f^{C_q}_{v_j x_i} = 0$ and f^{C_q} is a minimum cost flow then $c^{\pi}_{v_j x_i} \geq 0$ and $c^{\pi}_{v_j x_i} + k > 0$ which satisfies the reduced costs optimality conditions of Theorem 5.4.2.

If $f^{C_q}_{v_j x_i} = 1$ and there is no path from v_j to x_i in $R(f^{C_q}) - \{(v_j, x_i)\}$ then, by Theorem 5.4.1, the flow through (v_j, x_i) can not be reduced without violating its feasibility. If such a path exists, but its length is greater then or equal to $c_{v_j x_i} + k$, then the cost of the flow obtained by redirecting the flow along this path is greater than or equal to $cost(f^{C_q}) + k$; thus, the obtained flow is not a minimum cost flow. On the other hand, if a path which satisfies this properties exists, then a new flow can be obtained by redirecting the flow from (v_j, x_i) along this path. Moreover, if this path is a shortest path from v_j to x_i, then the obtained flow is a minimum cost flow in the value network of the modified constraint. □

In the case when $f^{C_q}_{v_j x_i} = 1$ and $f^{C_{q+1}} = f^{C_q}$, the cost of $f^{C_{q+1}}$ is equal to $cost(f^{C_q}) + k$. Such case requires decreasing the potential of nodes in the graph. When the flow through (v_j, x_i) is reduced, $cost(f^{C_{q+1}}) = cost(f^{C_i}) + d_{v_j x_i} - c_{x_i v_j}$.

Theorem 5.5.1. *If the cost of an occurrence of value v_j in x_i is changed, then the consistency of the cardinality constraint can be checked, and the minimum cost flow in its value network maintained in $O(m + log\ n)$ time, where $m = \sum_{i=1}^{|X|} |D_{x_i}| + |X| + |D| + 2$ and $n = |X| + |D| + 2$.*

Proof. By Proposition 5.5.1, verifying the consistency of a cardinality constraint modified by changing the cost of the occurrence of a value in a subset assigned to a variable yields in the worst case one iteration of the shortest path algorithm. Similarly, by Proposition 5.5.2 and 5.5.3, restoring the minimum cost flow in the value network of the modified constraint is achieved by one iteration of the same algorithm.

Dijkstra's algorithm, implemented with Fibonacci heaps, has time complexity $O(m+n\ log\ n)$. In the residual graph of a flow in the value network of a cardinality constraint $m = \sum_{i=1}^{|X|} |D_{x_i}|+|X|+|D|+2$ and $n = |X|+|D|+2$ which proves the theorem. □

Note that recomputing the consistency of a cardinality constraint from scratch using the successive shortest path algorithm, which is one of the best algorithms for computing a minimum cost flow (see [1] for comparison) requires time $O((\sum_{i=1}^{|X|} l_{x_i} + \sum_{j=1}^{|D|} l_{v_j}) \times (m + n\ log\ n))$.

5.6 Changes to Cardinality Bounds

By Definition 5.3.2 a change to a lower or upper bound of the cardinality of a variable x_i corresponds to a change of the lower and upper bounds imposed on the flow from x_i to t. Similarly, a change of the cardinality bound for a value v_j corresponds to a change of the bounds imposed on the flow through (s, v_j) in the value network of a cardinality constraint.

First assume that the lower bound of the cardinality of x_i decreases with k units. Trivially, since C_q is consistent then the flow f^{C_q} is feasible. Moreover f^{C_q} also satisfies the lower bound on the flow through (x_i, t) in the value network of the modified constraint. Thus, by Theorem 5.3.1, the modified constraint is consistent.

Furthermore, if f^{C_q} is a minimum cost flow in $N(C_q)$ and the residual capacity $r_{tx_i} > 0$, then f^{C_q} satisfies the reduced costs optimality conditions for (x_i, t) in $R(f^{C_q})$. By Theorem 5.4.2 f^{C_q} is the minimum cost flow in the value network of the modified constraint.

However, if r_{tx_i} in $R(f^{C_q})$ is equal to 0, decreasing the lower bound of x_i will introduce a new arc (t, x_i) in the residual graph of f^{C_q} with capacity k and

with the reduced cost $c^{\pi}_{tx_i}$ (Definition 5.2.4). Such a situation corresponds to changing a "virtual" reduced cost of the introduced arc with $\pm c^{\pi}_{tx_i}$, and a new minimum cost flow can be computed using the methods described in Section 5.5.

Similarly, if C_q is consistent and the upper bound of the cardinality of x_i increases, then the modified constraint is consistent. As in the case above, a new minimum cost flow in the value graph of the modified constraint is computed only if increasing this bound introduces a new arc (x_i, t) into the residual graph of f^{C_q}. Again, recomputing the minimum cost flow is done by methods described in Section 5.5.

Consider now the case where the lower bound of x_i increases with k units:

Proposition 5.6.1. *The modified constraint is inconsistent if and only if there is no sequence of $k - r_{t,x_i}$ successive paths from t to x_i in $R(f^{C_q}) - \{(t, x_i)\}$ such that $\sum_{i=1}^{k-r_{t,x_i}} d^i_{tx_i} \leq H - cost(f^{C_q})$.*

Proof. If $k \leq r_{tx_i}$ then the flow through (x_i, t) is feasible also for the modified problem.

If there exist $k - r_{t,x_i}$ such paths then, by Theorem 5.4.1 there exists a feasible flow in the value network of the modified constraint. Moreover, if $\sum_{iter=1}^{k-r_{tx_i}} d^{iter}_{tx_i} \leq H - cost(f^{C_q})$ then the cost of obtained flow is lower than or equal to H. Thus, by Theorem 5.3.1 the modified constraint is consistent. □

Furthermore, modifying the lower bound of the cardinality of x_i with $k \leq r_{tx_i}$ does not influence the reduced cost of (t, x_i). Thus, if f^{C_q} is a minimum cost flow in $N(C_q)$, then it is also the minimum cost flow in the value network of the modified constraint. If $k > r_{tx_i}$, then, if each one of the $k - r_{tx_i}$ successive paths is a shortest path, then at each iteration f^{C_q} will be augmented with a cheapest flow. Consequently the obtained flow will be the minimum cost flow in the value network of the modified constraint.

When a constraint is modified by *decreasing* the upper bound of the cardinality of a variable, then the consistency of the new constraint can be checked in the same way as when the lower bound is increased. The proof is similar to the proof of Proposition 5.6.1. Even the minimum cost flow in the value network of the modified constraint is maintained by computing $k - r_{x_i t}$ shortest paths from x_i to t in $R(f^{C_q}) - \{(x_i, t)\}$.

The rules described above apply by symmetry to the changes in bounds restraining the number of occurrence of each value. For each variable v_j a change to the bounds of its occurrence corresponds to the bounds imposed on the flow through respective (s, v_j) (see Definition 5.3.2).

Theorem 5.6.1. *The consistency of a cardinality constraint C_q, modified by changing the bounds of the cardinality of a variable or the bounds of an occurrence of a value with $\pm k$ units, can be done in $O(|k| \times (m + n\ log\ n)$ time. The same goes for maintaining the minimum cost flow in the value network of the modified constraint.*

Proof. As shown above, verifying and recomputing the consistency of a cardinality constraint, modified by changing the bounds of the cardinality of a variable by $\pm k$, requires at most k iterations of the shortest path algorithm. The same number of iterations of the algorithm is required when modifying the number of occurrences of a value.

Moreover, it is shown that if f^{C_q} is a minimum cost flow in $N(C_q)$, then by augmenting and reducing flows along each shortest path a new minimum cost flow is obtained. □

5.7 Adding and Removing Variables and Values

Removing a variable x_i can be seen as reducing the bounds of its cardinality to the interval $[0, 0]$. This corresponds to modifying the bounds imposed on the flow through (x_i, t) to be in the interval $[0, 0]$, as well as modifying the bounds of each (v_j, x_i) such that $v_j \in D_{x_i}$ is in the same interval. In this case, verifying consistency of the modified constraint as well as computing a new minimum cost flow is done in the same way as for decreasing the upper bound of the cardinality of a variable (see Section 5.6).

Moreover, it can be proved that due to properties of the value network of a cardinality constraint, reducing the flow through (x_i, t) to 0 will reduce the flow through each arc (v_j, x_i) to 0.

Adding a variable x_i with the cardinality $[l_{x_i}, u_{x_i}]$ and domain D_{x_i} can be seen as modifying the cardinality bounds from $[0, 0]$ to $[l_{x_i}, u_{x_i}]$, which in turns modifies the bounds imposed on the flow through (x_i, t) and bounds of the flow on each (v_j, x_i) such that $v_j \in D_{x_i}$ to $[0, 1]$ and its cost to $cost(x_i, v_j)$. In this case verifying consistency of the modified constraint, as well as recomputing the minimum cost flow in the value network of the modified constraint, is done in the same way as for increasing the lower bound of the cardinality of a variable from 0 to l_{x_i} units.

Note however, that if $l_{x_i} = 0$ we need to verify that each of the arcs added to the value graph satisfies the reduced costs optimality conditions. If the reduced cost of any arc violates those conditions then the new minimum cost flow is computed by the methods described in Theorem 5.5.2.

Removing and adding a value from/to a constraint corresponds to decreasing and increasing, respectively, the bounds imposed on the occurrence of such a value. Consequently, checking consistency of the modified constraint, as well as maintaining a minimum cost flow in its value network, is done by methods corresponding to the ones described in Section 5.6.

Theorem 5.7.1. *If a cardinality constraint with costs C_q is modified by removing a variable or value, then recomputing consistency of C_q and maintaining a minimum cost flow in $N(C_q)$ requires $O(\#(z,P) \times (m + n\ log\ n))$ time, where $\#(z,P)$ is a cardinality of the set to a removed variable assigned by P in C_q or number of occurrence of a removed value in sets assigned by P in C_q.*

Recomputing consistency of a cardinality constraint with costs C_q and a minimum cost flow in $N(C_q)$ modified by adding a variable or a value requires $O(u_z \times (m + n\ log\ n))$ time, where u_z is the upper bound of the cardinality of the introduced variable or the upper bound of the number of occurrence of the introduced value.

Proof. By the correspondence between the removal of a variable and restraining the upper bound on the flow through (x_i, t), since the new upper bound of the flow through (x_i, t) equals 0, all the flow through (x_i, t) has to be reduced. By Theorem 5.3.1 the amount of flow through (x_i, t) corresponds to the cardinality of the set assigned to x_i by P in C_q.

By the same correspondences we can prove that removing a value v_j requires a number of iterations of the shortest path algorithm equal to the number of occurrences of v_j in the sets assigned to the variables of C_q by P.

Consider now the case of adding a variable. In the worst case augmenting the flow on each (v_j, x_i) reduces the total cost of the flow in the value network of the modified constraints. However, by Definition 5.3.2 the amount of flow through (x_i, t) is limited by the upper bound of the cardinality of x_i.

Similarly, the amount of flow which can be augmented to (s, v_j), where v_j is an added value, is limited by the upper bound of the number of occurrences of v_j. □

As mentioned in Section 5.3 recomputing a symmetric cardinality constraint from scratch requires $O((\sum_{i=1}^{|X|} l_{x_i} + \sum_{j=1}^{|D|} l_{v_j}) \times (m + n\ log\ n))$ time. It is clear that as long as $\sum_{i=1}^{|X|} l_{x_i} + \sum_{j=1}^{|D|} l_{v_j}) - \#(x,P) > 0$, where x is a variable removed from the problem, we will have better time complexity than for computing the consistency of the constraint from scratch. The same applies to the case of removing a value from a constraint.

Moreover, if the amount of flow, required in the value network of a symmetric cardinality constraint with costs, is greater than the upper bound of the introduced variable or value, the time required to recompute the consistency of such a constraint, by the means presented in this section, will be shorter than the time for recomputing the modified constraint from scratch. The same applies to recomputing the minimum cost flow in the value network of the modified constraint.

For global cardinality constraints with costs and constraints of difference with costs, the methods presented here will always have better time complexity.

5.8 Changes to the Global Cost Limit

Changing the global cost limit H does not have an influence neither on the minimum cost flow in the value network of the constraint nor on the shortest path distances between any pair of nodes in the network.

The value of H can decrease with at most $H - cost(f^{C_q})$ units without violating consistency of C_q. If C_q is consistent and H increases with any number of units then C_{q+1} is consistent even for the modified problem.

5.9 Filtering

Given a consistent symmetric cardinality constraint C_q, the consistency of each value v_j of a variable x_i with C_q is verified by establishing if there exists a feasible flow in the value network of C_q with cost lower than or equal to H such that the flow on (v_j, x_i) is equal to 1 (see Section 5.3). Typically, it is done by computing a minimum cost flow in the value network of C_q and shortest path distances from every x_i every v_j.

One possible way of dealing with consistency of values with constraint is to maintain all–pairs shortest paths in the residual graph. The main advantage of this approach would be that it diminishes the space requirements in comparison with the former approach. However, maintaining all–pairs shortest paths has several drawbacks. One is that current algorithms for dynamic all–pairs shortest path problems are quite ineffective for general graphs.

The best dynamic all–pairs shortest path algorithm so far is one presented in [9]. The algorithm is based on the property that given a shortest path from i_1 to i_n every subpath of this path is also the shortest path from i_1 to a given i_i. While generating shortest path distances, the algorithm stores each subpath of currently shortest path between two nodes. It is said, that such a subpath

is *potentially uniform*, i.e. can be a subpath in the shortest path between two nodes.

In a case, when any change to the graph occurs, all the paths, which contains some modified edge, are removed from the solution to the problem and the algorithm recomputes the shortest path by merging existing subpaths into a potential shortest path.

Another drawback of this algorithm is the necessity of maintaining a huge number of potential shortest paths and their subpaths, which slows down the performance of the algorithm. To limit the number of scanned subpaths a time stamp, indicating a time point in which given path was potentially uniform, is attached to each subpath. While recomputing shortest paths, only potential shortest paths up to a given time point are considered. However, it was experimentally proved [8] that time stamping paths has no bigger influence on the performance of this algorithm.

Another, more straightforward method, would be to maintain $|X|$ graphs, where each such graph would maintain single source shortest path from each $x_i \in X$ to every other node in the graph. In the naive implementation this would increase the space complexity of the problem with $O(|X| * (m + n))$, where m and n are the number of arcs, resp. the number of nodes in the residual graph of a consistent constraint. The space complexity of this approach can be significantly reduced by maintaining a distance matrix between every pair of nodes, which gives a space complexity $O(n^2)$ or in terms of cardinality constraint $((|X| + |V| + 2)^2)$.

Recall from Section 5.3, that v_j of x_i is consistent with C_q if $f^{C_q}_{v_j x_i} = 1$ or otherwise if $cost(f^{C_q}) + d_{x_i v_j} \leq H$. The distance $d_{x_i v_j}$ is measured relatively to the reduced costs of arcs [16]. Thus, let $c^{\rho}_{x_i v_j} = c^{\pi}_{x_i v_j}$ denote the cost of (x_i, v_j) in terms of the shortest path problem and let $c^{\sigma, x_i}_{x_i v_j}$ be the reduced cost of (x_i, v_j) with respect to the shortest path from x_i to v_j. For any (i, j) in the graph, we define

$$c^{\sigma,l}_{ij} = c^{\rho}_{ij} + d_{li} - d_{lj} \tag{5.3}$$

which can be derived from the optimality conditions for shortest path problems [1]. This basically formulates the all–pairs shortest path problem as a problem of pushing n commodities from each node to every other node in a graph and should be studied more thoroughly in context of multicommodity flows.

By optimality conditions for the shortest path problem and Equation (5.3) an arc (i, j) is contained in a shortest path from l to j if $c^{\sigma,l}_{ij} = 0$.

Now, we will consider, how changes to the c^{ρ}_{ij}, in our case induced by changes to c_{ij} in the value network of a cardinality constraint, affect all–pairs

shortest path distances. The method presented here is similar to the method for maintaining single source shortest paths in undirected graphs described in [12].

Assume, that the shortest paths from l are represented by a minimum spanning tree rooted at l. For each $(i,j) \in T(l)$ holds that $c_{ij}^{\sigma,l} = 0$. Assume that c_{ij} decreases with k units, thus for each l, $c_{ij}^{\sigma,l}$ decreases with k units. If (i,j) is contained in $T(l)$ then (i,j) is already contained in some shortest paths from l, and the distances to all nodes in the subtree of $T(l)$ rooted at j should be adjusted with $-k$. However, if (i,j) is not contained in $T(l)$, we need to check if the new reduced cost for any (i,j) becomes lower than 0 then (i,j) is contained in $T(l)$. This implies that subtrees roted at j for any $T(l)$ which will contain (i,j) must be updated with a distance $c_{ij}^{\sigma,l} - k$.

Similarly, when the cost of an arc increases with k and $(i,j) \notin T(l)$ then $T(l)$ is a shortest path tree rooted at l for the modified problem. Otherwise, if there exists an edge (p,j) such that $c_{ij}^{\sigma,l} + k > c_{pj}^{\sigma,l}$ then (p,j) is contained in $T(l)$. Consequently, the change to the shortest path distances are propagated along the subtree of $T(l)$ rooted at j.

In case when an arc is inserted to a graph, if the reduced cost of the inserted arc is lower than 0 then this arc is contained in $T(l)$ and the changes to the distances are propagated in the appropriate subtree of $T(l)$. Otherwise, no propagation of new distances is necessary.

If some arc (i,j) is deleted from a graph, then either a new path is found along the cheapest, relatively to source l, edge incoming to j or, if (i,j) is not contained in $T(l)$ which does not violate optimality of $T(l)$.

The correctness of the method described above can be proved from the definition of reduced costs for the shortest path trees and shortest path optimality conditions. For a closer description of sensitivity analysis for shortest paths and minimum spanning trees see [26, 19].

In a case, when the shortest path tree is modified due to cost change, insertion or deletion, it is important to limit the number of scanned edges while searching for an arc which restores the optimality of a shortest path tree. The authors of [12] solve this problem by maintaining a priority queue of incoming and outgoing edges for each node in the graph. This increases the space complexity of their algorithm with $2 * m$, where m is the number of edges in the graph. However, in the case of maintaining all–pairs shortest path it would require $2 * n * m$ space to store all such information.

To limit the space usage, we sort all incoming and outgoing edges of each node according to their cost. This method is based on observation that many

shortest path trees in graphs of cardinality constraints share common subtrees. This is especially visible in the case of global cardinality constraint and constraint of difference, where there is only one outgoing edge from a node representing a variable.

5.10 Computational Evaluation

We evaluate the presented methods on the global cardinality constraint with costs as defined in [22, 23]. The size of the generated benchmarks varies between instances from 50 variables and 300 values to instances with 1000 variables and 2500 values. The value networks for these instances vary in size from 352 to 3502 nodes and between 15 thousands edges and ca. 2.5 millions edges. The residual graphs for computed instances varies in size during the computation since the number of edges between value nodes and the source node can be doubled in the worst case. Even the number of edges between variable nodes and sink can increase with the same amount.

In our implementation, a graph is a set of edges, and every node in a graph is a list of pointers to the incoming and outgoing edges of a node. Although, such representation of a graph does not require much memory, the space requirements can be reduced even more by representing a graph as an array of nodes and edges as pointers between elements of the array.

The consistency of a constraint is established by finding the minimum cost flow using the successive shortest path algorithm, which in turn uses the Fibonacci heap implementation of the Dijkstra algorithm. This implementation is proved to have the best theoretical bounds and it performs well in practice [5]. While computing minimum cost flow each iteration of the shortest path algorithm is terminated when the shortest path to the specific target node is found (see [1], pp. 323–324).

The results presented in figures 5.1 and 5.2 shows the times for computing the consistency of a dynamic global cardinality constraint with costs. The problem size is expressed in terms of the number of variables times the number of values in the problem. The evaluation was conducted on an IBM ThinkPad X30 with a Mobile Intel Pentium III CPU–M 1200 MHz and 512 MB RAM.

Figure 5.1 shows the average times for restoring consistency of a gcc with costs by updating the minimum cost flow in the value network of a constraint. The tests has been conducted for one change at the time. The times given in the figure are the worst computing times for a single change.

The times given in Figure 5.1 refer to situations where the optimality of

Problem size	Restart	Cost Change	Bound Change	Add/remove a value to/from a domain	Add/remove variable/value
50 × 300	0.02 s	0.001 s	0.002 s	0.002 s	0.003 s
50 × 500	0.03 s	0.002 s	0.003 s	0.01 s	0.003 s
50 × 1000	0.07 s	0.006 s	0.006 s	0.02 s	0.007 s
100 × 300	0.03 s	0.003 s	0.017 s	0.003 s	0.01 s
100 × 500	0.08 s	0.006 s	0.011 s	0.01 s	0.01 s
100 × 1000	0.15 s	0.012 s	0.051 s	0.01 s	0.02 s
300 × 400	0.3 s	0.015 s	0.17 s	0.013 s	0.02 s
300 × 500	0.36 s	0.018 s	0.19 s	0.03 s	0.09 s
300 × 1000	0.5 s	0.04 s	0.2 s	0.03 s	0.09 s
500 × 800	1.2 s	0.067 s	0.53 s	0.004 s	0.08 s
500 × 1000	1.6 s	0.075 s	0.83 s	0.005 s	0.2 s
500 × 1500	1.8 s	0.12 s	0.9 s	0.025 s	0.2 s
1000 × 1500	3.3 s	0.19 s	0.7 s	0.02 s	0.5 s
1000 × 2000	6.5 s	0.21 s	0.9 s	0.02 s	0.5 s
1000 × 2500	8.4 s	0.37 s	1.4 s	0.027 s	0.9 s

Figure 5.1: Restoring optimality of the flow in the value network of a global cardinality constraint by iterations of Dijkstra's algorithm.

the flow is violated and needs to be restored by multiple iterations of Dijkstra's algorithm. These times includes also the time for pushing the flow along the found path. In a situation, when the optimality of the flow through an arc is not violated, the computing time is limited to just checking if a flow through this arc and its reduced costs satisfy lower bounds and the reduced costs optimality conditions. In such cases the computing time is limited to few microseconds.

The times given in column 6 of Figure 5.1 are mostly the times for re-computing the constraints after adding a value to domains of several variables. Computing consistency of a gcc after adding a value is usually more costly than after adding a variable. It is because adding a variable often requires only one iteration of Dijkstra algorithm. Similarly, when a variable is removed only one iteration of the shortest path algorithm is required. The times required for adding and removing a variable are similar to those of adding and removing a value from domain of a variable in column 5.

Figure 5.2 shows computational times for the filtering phase of a global

Problem size	From scratch	Dynamic
50 × 300	0.16 s	0.017 s
50 × 500	0.17 s	0.017 s
50 × 1000	0.2 s	0.017 s
100 × 300	0.16 s	0.01 s
100 × 500	0.18 s	0.01 s
100 × 1000	0.19 s	0.01 s
300 × 400	0.19 s	0.016 s
300 × 500	0.19 s	0.016 s
300 × 1000	0.2 s	0.19 s
500 × 800	0.19 s	0.3 s
500 × 1000	0.19 s	0.9 s
500 × 1500	0.15 s	0.9 s
1000 × 1500	0.2 s	1 s
1000 × 2000	0.2 s	1.2 s
1000 × 2500	0.21 s	1.2 s

Figure 5.2: Computing times for the filtering phase.

cardinality constraint with costs for changes to a constraint which does require redirecting the flow in the value network of a constraint. It is clear, that the method presented in Section 5.9 performs better than executing the filtering phase from scratch for small instances of gcc. However, for larger instances the relative performance degrades. This degradation is caused by a need of multiple traversing dynamically allocated linked lists representing arcs of a graph. The performance of the algorithm can be easily improved by implementing a graph with data structures requiring less dynamically allocated memory.

5.11 Conclusions and Future Work

In this paper, we have shown, how the consistency of a number of cardinality constraints with costs can be maintained by means of sensitivity analysis for minimum cost flows. We show, both by complexity analysis and computational evaluation that the introduced method outperforms recomputing a cardinality constraint from scratch.

Moreover, we show how the consistency of values with a cardinality constraint for small instances can be maintained using dynamic single source short-

est paths algorithm, under the assumption that a modification of a constraint does not influence the flow in the residual graph in a constraint.

We also discuss issues related to maintaining the all pairs shortest paths in a directed graph. We believe, that algorithms dealing with this problem need to be further improved in order to be effective in practice. Such improvements needs to be done by improving the effectiveness of accessing data structures maintained by such algorithms or by reformulating the model used for computing this problem. In this context we believe that formulating dynamic all pairs shortest paths as a problem of dynamic multicommodity flow might be worth further study.

Bibliography

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, 1993.

[2] R. Barták. Dynamic global constraints in constraint logic programming. Technical Report ITI Series 2001-018, 2001.

[3] R. Barták, T. Müller, and H. Rudová. A new approach to modelling and solving minimal perturbation problems. In *Recent Advances in Constraints*, LNAI 3010, pages 233–249. Springer Verlag, 2004.

[4] Y. Caseau and F. Laburthe. Solving various weighted matching problems with constraints. In Gert Smolka, editor, *Proceedings Third International Conference on Principles and Practice of Constraint Programming*, pages 17–31. Springer-Verlag LNCS 1330, 1997.

[5] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[6] A.J. Davenport and J.Ch. Beck. Managing uncertainty in scheduling: a survey. Preprint, 2000.

[7] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, pages 37–42, 1988.

[8] C. Demetrescu, S. Emiliozzi, and G.F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. url: http://www.dis.uniroma1.it/d̃emetres/.

[9] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In ACM, editor, *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing, San Diego, CA, USA, June 9–11, 2003*, pages 159–166, New York, NY, USA, 2003. ACM Press.

[10] K. Easton, G. Nemhauser, and M. Trick. Sports scheduling. In J. Leung, editor, *Handbook of Scheduling: Models, Algorithms and Performace Analysis*. CRC Press, 2004.

[11] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In *Principles and Practice of Constraint Programming*, pages 189–203, 1999.

[12] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest path trees. *Journal of Algorithms*, 34:251–281, 2000.

[13] E. Gaudin, N. Jussien and G. Rochart. *Implementing explained global constraints* CP04 Workshop on Constraint Propagation and Implementation (CPAI'04), pp. 61–76, 2004

[14] W. Kocjan. Dynamic scheduling. state of the art report. Technical Report T2002-28, Swedish Institute of Computer Science, November 12, 2002.

[15] W. Kocjan and P. Kreuger. Filtering methods for symmetric cardinality constraint. In *Proc. 1th Int. Conf. CPAIOR*, pages 200–208, 2004.

[16] W. Kocjan, P. Kreuger, and B. Lisper. Symmetric cardinality constraint with costs. Technical report, MRTC, Mälardalen University, 2004.

[17] J.-L. Laurière. A language and a program for stating and for solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[18] M. Milano, editor. *Constraint and Integer Programming: Toward the Unified Methodology*. Kluwer, 2003.

[19] R. Ramaswamy, J.B. Orlin, and N. Chakravarti. Sensitivity analysis for shortest path problems and maximum capacity path problems in undirected graphs. Technical Report 4391–02, MIT Sloan School of Management, 2004.

[20] J.-Ch. Régin. A filtering algorithm for constraints of difference in csps. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[21] J.-Ch. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 209–215, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[22] J.-Ch. Régin. Arc consistency for global cardinality constraints with costs. In *Principles and Practice of Constraint Programming (CP-99)*, pages 390–404, 1999.

[23] J.-Ch. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, (7):387–405, 2002.

[24] J.-Ch. Régin and C. Gomes. The cardnality matrix constraint. In *Internatonal Conference on Principle and Practice of Constraint Programming, Proceedings*, LNCS, pages 572–587, 2004.

[25] H. Simonis. Sudoku as a constraint problem. Available at http://www.icparc.ic.ac.uk/ hs/sudoku.pdf, 2005.

[26] R.E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters*, 14(1), 1982.

[27] E. Tsang. *Foundation of Constraint Satisfaction*. Academic Press, 1993.

[28] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 307–312, Menlo Park, CA, USA, July 31–August 4 1994. AAAI Press.