

Mälardalen University Licentiate Thesis
No.27

Design and Implementation of a Graph-Based Constraint Model for Local Search

Markus Bohlin

April 2004



MÄLARDALEN UNIVERSITY

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Markus Bohlin, 2004
ISSN 1651-9256
ISBN 91-88834-43-3
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Errata

Updated April 30, 2004.

References to corrections are given in page and line numbers, not counting headers.

Page	Line	Reads	Correction
i	4	<i>primitive</i> constraints	<i>primitive</i> constraints, often binary
4	11	virtual cost	composed cost
7	15	durations for	durations of
9	24	label l_b is	label l_a is
	25	l_a , written as	l_b , written as
10	6	constraints $c_S \in C$ where S is a subset of X .	constraints $c_S \in C$.
	13	$\text{DOM}(x_1) = \{v_{1,1}, v_{1,2}, \dots, v_{1,q}\},$	$\text{DOM}(x) = \{v_1, v_2, \dots, v_q\},$
11	14	$\dots \wedge \forall x \in \text{VAR}(\ell). x \in \text{DOM}(x)$	$\dots \wedge \forall \langle x, v \rangle \in \ell. x \in \text{VAR}(\ell) \rightarrow v \in \text{DOM}(x)$
12	5	where $s, e \in c$ for some c	where $s, e \in \text{VAR}(c)$ for some c
	6	$\{s\} = c$.	$\{s\} = \text{VAR}(c)$.
	15	we loose	we lose
13	1	are natural to represent	are naturally represented
	21	$\forall v. \text{SATISFIES}(\ell', C) \rightarrow \dots$	$\forall \ell'. \text{SATISFIES}(\ell', C) \rightarrow \dots$
14	6	$\{c \mid c \in C \wedge \neg \text{SATISFIES}(\ell', c)\} \succ \{c \mid c \in C \wedge \text{SATISFIES}(\ell, c)\}$	$\{c \mid c \in C \wedge \text{SATISFIES}(\ell', c)\} \succ \{c \mid c \in C \wedge \text{SATISFIES}(\ell, c)\}$
	25	row position ... at column i .	column position ... at row i
15	1	placed on a row:	placed on a column:
	2	$\forall i, j, i \dots$	$\forall i, j, i \dots$

21	14	exponential in	exponential to
23	8	all cities and minimizes the sum	all cities, minimizing the sum
26	21	relation undirected	relation directed
27	16	any state $u \in S$ equals	any state $u \in S$ contains
	18	$\dots \equiv \forall u \in S. TC_T(u) = S$	$\dots \equiv \forall u \in S. TC_T(u) \supseteq S$
29	18	we discuss is	we discuss are
31	13	and CS be $\dots SP(X) \rightarrow CS$	and W be $\dots SP(X) \rightarrow W$
33	14	Note also that that	Note also that
34	18	minimize $f(C, v)$	minimize $f^b(C, v)$
35	14	The conflict level CL on	The conflict level CL of a variable x on
	20	current cost $f_c(x)$	current cost $f_c(v)$
36	3	a variable for	a variable x for
	28	$\dots \wedge \forall u, v \in T. f(u) = f(v)$	$\dots \wedge \forall u, v \in L. f(u) = f(v)$
38	14	PLATEAU $_{f,T}$	PLATEAU $_{T,f}$
	15, 16	a minimum L_1 \dots another minimum L_2 with lower level than L_1 .	a minimum L \dots another minimum L' with lower level than L .
	30	\dots PLATEAU $_{f,T} \wedge \neg$ MINIMUM $_{f,T}(L)$	\dots PLATEAU $_{T,f} \wedge \neg \exists u \in B_T(L). f(u) = f(L)$
49		Note that we use a vector	representation of the arc sets.
	6	$\dots k \in \{1, \dots, A \} \wedge \dots$	$\dots k \in \{1, \dots, A \} \wedge \dots$
56	7	$\dots U[a]$	$\dots U_a$
	29	$\dots f(p_i)$	$\dots f(p_i)$
57	14	$\dots = \{(y, X[i] \mid \dots$	$\dots = \{(y, X[i] \mid \dots$
	15	$\dots = \{X[i], y \mid \dots$	$\dots = \{X[i], y \mid \dots$
59	11	sum of the subconflicts	sum of the subconflict
	13	$\dots = \sum_{p \in g(A')} \dots$	$\dots = \sum_{p \in pf(A')} \dots$
60	18	$\dots = \max(L[i] - p_i , \dots$	$\dots = \max(0, L[i] - p_i , \dots$
61	11	$\dots i \in \{1, \dots, X \} \wedge \dots$ a index	$\dots i \in \{1, \dots, X \} \wedge \dots$ an index
62	2	j this the	j is the
63	13	denote the end time.	denote the duration.
66	14	START-ACTIVE(t, y) $\equiv t \geq v(y.s) \wedge t < v(y.s) + v(y.d)$	START-ACTIVE(x, y) $\equiv v(x.s) \geq v(y.s) \wedge v(x.s) < v(y.s) + v(y.d)$

69	35	the capacity of the constraint k	the capacity k of the constraint
73	4	with is focus vertex	which is focus vertex
76	19	at time $s.a + d.a - \ell$	at time $x.s + x.d - \ell$
86	28, 29	traversal time from ... arrival a_{i+1} at ... stop time s_{i+1}	traversal time t_i from ... arrival a_i at ... stop time s_i
88	13	partition = Leaving;	combined = Leaving;
92	2	total cost by	total cost by ...
95	9	$v[x \mapsto a]$ where $a \neq v(a)$	$v[x \mapsto a]$ where $a \neq v(x)$
	21	contains any vertex in Y .	contains any vertex in Y_X .
	22	... = true]	... $\neq fc(A_Y[k], v[x \mapsto a])]$
96	4	$\Delta A \leftarrow \{(x, V[v(x)]), (x, V[a])\}$	$\Delta A \leftarrow \{(x, V[v(x)]), (x, V[a])\}$
	26	and f_o	and cv_o
101	27	cost = Entering;	combined = Entering;
	31	an offset vector M.	an offset vector m.
103	19, 20	We however managed to decrease the number of iterations by roughly 50%, but on the two	On the two
104	17, 19	oldCost	oldV
111	1	,oldV;	,oldV,nonimproving=0;
112	6	cost = Entering ;	combined = Entering ;
118	8, 9	form of <i>course-grain</i> ... <i>fine-grain</i>	form of <i>coarse-grained</i> ... <i>fine-grained</i>
119	2, 3	cost of the constraints in which x of the constraints in which x is ... summation is used	cost of the constraints in which x is ... summation are used

Abstract

Local search has during the last years evolved into a powerful technique for solving large combinatorial problems, often outperforming complete algorithms. The classical approach for generic constraint solving in local search is to provide a set of *primitive* constraints, which in turn can be used to form more complex combinatorial structures. Unfortunately, for several combinatorial structures there is no decomposition into binary constraints which is acceptable in terms of space and/or time complexity. Global constraints have been introduced in local search as time and space efficient modelling components, capturing the properties of common combinatorial substructures.

In this thesis we propose a *compositional* approach for global constraint design and implementation for local search. Traditionally, global constraints have been implemented as monolithic entities, often using a low-level language and requiring in-depth knowledge of the constraint system itself. In this work we propose to use graph structures, filters and cost components to create global constraints in a high-level C++ framework called *Composer*. The composed constraints can then be used for constraint solving in a generic, domain-independent local search solver. We show the theoretical model of the framework, and give algorithms for incrementally updating the costs and conflict levels of the constraints. We also show how to compose several well-known global constraints, and demonstrate by experimental results that a compositional approach at global constraint modelling is not only possible in practice, but also highly competitive with existing low-level implementations of constraint-based local search.

To Veronica

Preface

Looking back after three years of graduate studies, I can honestly say that although it has sometimes been hard work, it has also been very interesting, rewarding, and fun. I've had a great time, and I've met a lot of new people, of whom some will continue to be my friends for life.

First of all, I would like to thank my colleagues and friends Jan Carlson and Waldemar Kocjan for their company during these years. They have also endured with me during seemingly endless technical and, well, not-so-technical, discussions during our longer-than-normal coffee-breaks. Thanks, guys.

I am very grateful for the good advice and support my advisor Per Kreuger has given me during my licentiate studies. I would also like to thank my main supervisor Björn Lisper for support, help, and lots of different perspectives.

The people at IDT, especially Markus Nilsson, Johan Erikson, Mikael Sandberg, and the other people at the Computer Science Lab, as well as my colleagues at SICS; Martin Aronsson, Jan Ekman, Anders Holst, and my boss Björn Levin to mention a few, all deserve my deepest thanks.

To my former colleagues and class-mates Lars Bruce, Andreas Sjögren and Xavier Vera; We've had some great times, and I will always remember the days back then.

I also thank my father Björn, my mother Anna-Greta, my sister Maria, and my brother Martin, for their endless support and love.

Finally, and most of all, I am eternally grateful to Veronica Ståhl, for being the love of my life.

Markus Bohlin
Västerås, April 2, 2004

Contents

1	Introduction	1
1.1	Introduction	1
1.1.1	Thesis Organization	4
1.1.2	Publications	4
1.1.3	Contributions	5
2	Constraint Satisfaction	7
2.1	Basics of Constraint Satisfaction	7
2.1.1	Constraint Satisfaction	8
2.1.2	Constraint Graphs	11
2.1.3	Optimization and Partial CSP's	13
2.2	Motivation	14
2.2.1	Some Benchmark Problems	14
2.2.2	Configuration Problems	18
2.2.3	Scheduling and Planning	19
3	Constraint-Based Local Search	21
3.1	Basics of Local Search	22
3.1.1	The Travelling Salesman Problem	22
3.1.2	Iterative Improvement	24
3.1.3	Costs and Cost Functions	24
3.1.4	Neighborhoods and Transitions	26
3.1.5	Neighbor Selection	28
3.2	Constraint-Based Local Search	28
3.2.1	Transitions and the Neighbor Relation	31
3.2.2	Cost and Conflicts	34
3.3	Local Minima and Plateaus	36
3.4	Plateau Traversal and Avoidance	39

3.4.1	Flat Transitions	40
3.4.2	Randomization	40
3.4.3	Exhaustive Plateau Search	41
3.4.4	Tabu Search	41
3.4.5	Dynamic Weighting	42
4	A Model for Global Constraints	45
4.1	Introduction	45
4.1.1	Chapter Outline	46
4.2	A Model of Global Constraints	46
4.2.1	Constraint Representation	47
4.2.2	Cost Computation	48
4.2.3	Notation and Constraint Arguments	49
4.2.4	Vertex Generators	50
4.2.5	Graph Structures	51
4.2.6	Filter Constraints	53
4.2.7	Regular Cost Functions	54
4.2.8	Combined Costs and Subcosts	55
4.2.9	Conflict Computation	57
4.3	Capacity Constraints	59
4.3.1	The <code>capa</code> Constraint	59
4.3.2	The <code>gcc</code> Constraint	61
4.3.3	The <code>wcc</code> Constraint	61
4.3.4	The <code>nbdiff</code> Constraint	62
4.4	Non-Overlapping Scheduling Constraints	63
4.4.1	The <code>ordered-tasks</code> Constraint	63
4.4.2	The <code>serialized</code> Constraint	64
4.4.3	The <code>disjoint-tasks</code> Constraint	65
4.5	Cumulative Scheduling Constraints	65
4.5.1	A Filter Constraint for Cumulative Scheduling	66
4.5.2	Removing Symmetrical Arcs	67
4.5.3	The <code>cumulative-1</code> Constraint	68
4.5.4	The <code>cumulative</code> Constraint	71
4.5.5	Implicit Task Splitting	72
4.5.6	The <code>cumulative</code> Constraint with Cyclic Time	76

5	A Global Constraint Library	79
5.1	Introduction	79
5.1.1	Host Environment	81
5.1.2	Chapter Outline	82
5.2	The <i>Composer</i> Architecture	83
5.2.1	Incremental Computation of Expressions	84
5.2.2	State and Non-state Variables	85
5.2.3	Incremental Functions	85
5.2.4	Motivation	86
5.3	Constraint Composition Using <i>Composer</i>	88
5.3.1	Introduction	88
5.3.2	Composition Language	89
5.4	Incremental Cost and Conflict Computation	92
5.4.1	Total Cost and Conflicts	94
5.4.2	Individual Constraints	95
5.5	The <i>n</i> -Queens Problem	100
5.5.1	A Constraint Model	100
5.5.2	A First Local Search Algorithm	102
5.5.3	Results	103
5.5.4	A Swap-Based Local Search Algorithm	104
5.5.5	Results	106
5.6	The Progressive Party Problem	107
5.6.1	The Model	108
5.6.2	A Local Search Algorithm	110
5.6.3	Results	112
6	Related Work	117
6.1	Localizer and Friends	117
6.2	INC	118
6.3	Adaptive Search	118
6.4	Wsat(OIP)	119
6.5	GCSP	120
6.6	The Min-Conflict Heuristic	121
6.7	More Related Work	122
7	Conclusions	125
7.1	Conclusions	125
7.2	Future Work	126

List of Figures

2.1	The constraint graph $(\{a, b, c\}, \{\{a, b\}, \{a, c\}, \{b, c\}\})$. . .	12
2.2	The constraint graph for the 3-queens problem, where $c_1 \equiv x_1 \neq x_2 \wedge x_2 - x_1 \neq 1 \wedge x_2 - x_1 \neq -1$, $c_2 \equiv x_1 \neq x_3 \wedge x_3 - x_1 \neq 1 \wedge x_3 - x_1 \neq -1$, $c_3 \equiv x_2 \neq x_3 \wedge x_3 - x_2 \neq 1 \wedge x_3 - x_2 \neq -1$. The corresponding multigraph, with multiple edges and constraints between variables, is shown to the right.	15
2.3	Using the <code>alldiff</code> constraint in the n -queens problem to restrict assignments of queens to the same column. . . .	16
2.4	Using a modified <code>alldiff</code> constraint with constant terms in the n -queens problem. The two constraints restrict assignments of queens to the same left/right diagonal. . . .	17
3.1	The effect of a 2-interchange transition on a tour in solving the TSP. The original tour is to the left and the resulting tour to the right. Observe that the direction of traversal has changed in the lower part of the tour.	23
3.2	Five typical maximal plateaus in a visualization of a 2-dimensional CSP where height represent costs of assignments.	37
3.3	Illustration of typical plateaus in a 1-dimensional CSP. . . .	39
4.1	Two equivalent schedules which yield different cost unless corrected.	68

4.2	The number of binary overlaps of the infeasible schedule for a <code>cumulative-1</code> constraint with capacity 2 is equal to the cost of the <code>serialized</code> constraint on the right, where some tasks have been merged.	71
4.3	Splitting of tasks into tasks with height 1.	72
5.1	Design of the constraint composition framework. A constraint object is instantiated by selecting a structural component, a filter constraint and a cost component. Also, each composed constraint inherits properties from an abstract constraint class used by all constraints in <i>Composer</i>	81
5.2	Overview of the architecture of <i>Composer</i>	83
5.3	The dependency DAG for the flattened form of the incremental expression $x = f(g(Y), Y*(Z+10)) - 20$	85
5.4	A <code>serialized</code> constraint specification in <i>Composer</i>	88
5.5	The primitive <code>alldiff</code> constraint.	90
5.6	The compiled primitive <code>alldiff</code> constraint.	90
5.7	A bipartite <code>alldiff</code> constraint with subcosts.	92
5.8	A first model of the <i>n</i> -queens problem.	101
5.9	A bipartite <code>alldiff</code> constraint with an offset vector <i>M</i>	101
5.10	A model of the progressive party problem.	110
5.11	A bipartite <code>capa</code> constraint with subcosts.	112

List of Tables

4.1	The vertex generators used to produce the vertices in the constraint network for a global constraint.	51
4.2	The graph structures used to produce the arcs in the constraint network for a global constraint. Each graph structure takes either one or two vectors of vertices, and connects these using directed arcs.	52
4.3	The <code>alldiff</code> constraint.	55
4.4	Partition functions used to partition final arc sets A' for subcost computation. X is the vertex vector.	57
4.5	The <code>capa</code> constraint.	60
4.6	The <code>gcc</code> constraint.	60
4.7	The <code>wcc</code> constraint.	62
4.8	The <code>nbdiff</code> constraint.	63
4.9	The <code>ordered-tasks</code> constraint.	64
4.10	The <code>serialized</code> constraint.	64
4.11	The <code>disjoint-tasks</code> constraint.	65
4.12	The <code>alt-serialized</code> constraint.	69
4.13	The <code>cumulative-1</code> constraint.	70
4.14	The <code>split-cumulative</code> constraint.	72
4.15	The <code>cumulative</code> constraint.	76
4.16	The <code>cyclic-cumulative</code> constraint.	76
5.1	Experimental results in CPU time for 11 runs of the n -Queens Problem.	103
5.2	Experimental results in CPU time of 11 runs of the n -Queens Problem using the second model and conflict minimization.	106

5.3	Experimental results in CPU time of 11 runs of the n -Queens Problem using the second model and conflict improvement.	107
5.4	Configuration of the Progressive Party Problem. Each boat i has a total capacity k_i (the total number of people allowed on board at the same time), and a crew size c_i . The spare capacity s_i of a boat is formed by subtracting the crew size from the total capacity, $s_i = k_i - c_i$	108
5.5	Progressive Party Problem analysis for different host configurations. S is the total spare capacity of the host boats, and C is the total number of guest crew members in the instance.	109
5.6	Median CPU time of 101 runs of the Progressive Party Problem using restart solving.	113
5.7	Median iterations of 101 runs of the Progressive Party Problem using restart solving, in units of 10^3 iterations.	113
5.8	Median CPU time of 101 runs of the Progressive Party Problem using incremental solving.	114
5.9	Median iterations of 101 runs of the Progressive Party Problem using incremental solving, in units of 10^3 iterations.	115

List of Algorithms

5.1	Incremental cost computation of the difference $\Delta f_c(v, x, a)$ between the old and the new cost.	95
5.2	Incremental update of $\Delta cf(\Delta A)$ for a regular cost function.	97
5.3	Incremental update of a combined cost function $\Delta cf(\Delta A)$.	99
5.4	Search component for the n -queens puzzle using assign- ment transitions.	102
5.5	Search component for the n -queens puzzle using value swaps.	104
5.6	The search component for the progressive party problem.	111

Chapter 1

Introduction

1.1 Introduction

The constraint satisfaction problem (CSP) is a well-known problem in computer science. Informally, a CSP can be defined as follows.

Given a set of variables and constraints on these, is there an assignment of values to the variables, such that all constraints are satisfied?

A special case of a CSP is the *propositional satisfiability* problem, or SAT for short. SAT can in turn informally be defined as follows:

Given a set of Boolean variables and logical clauses over these, is there an assignment to the Boolean variables such that all clauses are satisfied?

CSP and SAT has been studied in numerous books and articles. Both CSP and SAT are core problems in computing theory and mathematical logic. SAT was actually the first problem shown (by Cook in [10]) to be NP-complete. This can be seen as an indication of the theoretical importance of these problems. Graph coloring, scheduling, planning, and assignment problems have also been formulated as SAT and/or CSP instances with much success. Methods for solving CSP and SAT are also fundamental in solving many practical problems including automated reasoning, computer-aided design and manufacturing, machine vision, databases, robotics, integrated circuit design, and computer network design.

The study of systematic methods for constraint satisfaction evolved from general systematic search methods in the field of artificial intelligence during the 70's and 80's. A key step toward constraint satisfaction was the development of *logic programming* languages like Prolog. In fact, many constraint satisfaction systems are extensions of logic programming, so-called *constraint logic programming* (CLP) systems.

Conventional constraint satisfaction methods have been shown to work well on a large number of problems from real life, like scheduling, planning and resource allocation problems. Unfortunately, these methods are in general time- and memory consuming, and because of this, they are not always suitable. For example, a *dynamic* planning problem is a planning problem, where the parameters change dynamically during the execution of the plan. This calls for a planner, which is able to recover from changes in the plan within reasonable time limits. Tasks like this are not obviously well-suited for regular constraint satisfaction methods.

Another approach for combinatorial problem solving is to use incomplete search methods such as local search. Heuristic methods based on local search for solving constraint problems have evolved during the last fifteen years into becoming one of the most powerful techniques for solving large combinatorial problems, often outperforming complete algorithms. The classical approach for generic constraint solving in local search is to provide a set of *primitive* constraints, which in turn can be used to form more complex combinatorial substructures [73]. A natural choice is to restrict the user to binary constraints only. Unfortunately, for several common combinatorial structures there simply do not exist a decomposition into binary constraints that is acceptable in terms of space and/or time complexity. Global constraints have been introduced in local search systems to provide time- and space efficient high-level components capturing common combinatorial substructures.

One local search approach to constraint satisfaction is called *iterative improvement*, in which a given candidate "solution" (which by no means actually has to solve the problem) is improved in an iterative process. The improvement is usually done by changing small parts of the solution, choosing the one that looks best at the moment. Iterative improvement and other related local search techniques also have the advantage that a solution always is available during the search, a property commonly known as *anytime behavior*.

In this licentiate thesis we study the fusion of local search and *global constraints*, which essentially are constraints encompassing a complex structural component of the problem. Theoretically, any constraint is simply a relation on the Cartesian product of the variables in the constraint. In practice, a global constraint is represented by a data structure together with specialized algorithms, which efficiently update the data structure of the global constraint. Traditionally, global constraints have been used mainly in systematic search frameworks, to prune the search space as much as possible during the search. However, in most local search frameworks, a global constraint acts only as an efficient representation of a large set of more primitive constraints.

Local search methods for constraint satisfaction, such as *steepest descent*, often aim at minimizing the number of violated constraints, by changing the value of a single variable in the problem. When introducing global constraints, this approach has the disadvantage that a global constraint, in effect encompassing the semantics of a large set of basic constraints, contribute as little to the number of violated constraints as, for example, a very simple constraint which imposes a restriction on a single variable. This is unfortunate because the search gets unbalanced, and the subproblem that the global constraint represents becomes difficult to solve.

In the main part of this thesis, we study the integration of global constraints into a local search framework, addressing the problems described above. We propose a *compositional* approach for modelling of global constraints themselves, which gives us a high-level tool for invention and modification of global constraints. This process is very common in real-life applications, where the provided global constraints often do not fit the problem exactly. In our approach, we parametrize global constraints over key properties of their structure. To do this we use a *generic graph model*, an approach that has previously been used successfully to model a large number of global constraints [4]. The main benefit of a parametrized model for global constraints is that practitioners can very easily experiment with different cost functions and different structures, and also create completely new global constraint, using composition of structural components, with minimum work. Also, the resulting constraints are evaluated very efficiently using highly-optimized incremental algorithms. The user does not have to care about keeping costs and conflict levels updated, which is handled by our implementation automatically.

1.1.1 Thesis Organization

This licentiate thesis is organized as follows. The first chapter gives a brief introduction to the problems addressed and investigated in this thesis. We also list some publications that have been written during the thesis work. The second chapter is an introduction to the constraint satisfaction domain and give some motivating examples of constraint models and problems. In the third chapter, we begin by introducing some concepts and formalism used for local search and constraint satisfaction. We define neighbors of a solution, local transitions and cost functions. In Chapter four, we introduce and motivate global constraints as costs on graphs in a local search context, and define some global constraints and the cost of these. We show the virtual cost for the common `alldiff` global constraint, and continue reasoning about costs for more complex global scheduling constraints. In Chapter five, we describe an implementation of the theoretic work of Chapter 4, and give some examples of problems that can be solved using this approach. We also give experimental results on some benchmark problems. In Chapter six, we describe some alternative approaches at constraint satisfaction using local search, and other related techniques. We conclude the thesis in Chapter seven with a summary of the contributions of the licentiate thesis work, and describe some possible future work.

1.1.2 Publications

The following papers have been published during the thesis work and form the basis of this thesis.

1. *Constraint Satisfaction using Local Search*, SICS Technical Report T2002:07.
2. *Improving Cost Calculations for Global Constraints in Local Search*, poster paper, in Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming, September 2002, Ithaca, New York.
3. *Designing Global Scheduling Constraints for Local Search: A Generic Approach*, SICS Technical Report T2002:20.
4. *Composing Global Constraints for Local Search*, in Proceedings of the 15th International Conference on Applications of Declarative

Programming and Knowledge Management (INAP), March 4-6, 2004, Fraunhofer FIRST, Berlin, and University of Potsdam.

1.1.3 Contributions

I believe that this licentiate thesis has contributed with the following.

- *A formalization of constraint-based local search.* We give formal definitions of many of the concepts of local search, which traditionally has been treated in a rather ad-hoc manner.
- *A theoretical framework for global constraint cost and conflict computation in local search.* We show how cost functions on structured graphs of filter constraints can be used to model global constraints. We also show how costs and conflict levels can be computed both from scratch and incrementally in this framework. Several important global constraints are modeled in the framework to show its expressiveness.
- *An implementation of a local search solver and a global constraint library, based on the theoretical model.* We have implemented the theoretical model for global constraints, and integrated the framework in a generic local search solver. We have also modeled and solved the Progressive Party Problem and the n -queens problem, and compares our results with other results reported for constraint-based local search implementations.

Chapter 2

Constraint Satisfaction

2.1 Basics of Constraint Satisfaction

Constraint satisfaction basically consists of assigning values to variables while meeting certain requirements (constraints). This declarative approach at problem modelling is very generic, and constraint problems are often concise and easy to understand. To clarify the concept of constraint satisfaction we will give two examples of problems that are easily modeled using CSP's.

Example 1 A graph coloring problem consists of assigning a given set of colors to the vertices in a graph, such that no adjacent vertices (which are connected via an arc) get the same color. Graph coloring problem can quite easily be represented as a CSP. In this case, we let the vertices of the graph and the colors correspond to variables and values respectively. The arcs of the graph can be modeled using a constraint that prohibits assignments of the same value to adjacent variables.

Example 2 In scheduling, constraints apply to variables representing the starting times and durations for tasks. In addition, constraints may be imposed on the non-overlapping of certain activities in the schedule – for example, the simultaneous use of a single resource may be forbidden.

The most common constraint satisfaction model is the *finite domain constraint model* (FD), where each variable has a domain of finite size. In this thesis we will only discuss FD constraint models.

2.1.1 Constraint Satisfaction

We now define the class of finite domain constraint satisfaction problems (CSP). For more information, see the books by Marriott and Stuckey [38], Tsang [72], and Apt [2]. To discuss constraint models in detail, we first need to define some common concepts. We begin by defining the *domain* of a variable in a CSP.

Definition 1. The *domain* of a variable is the set of possible values that can be assigned to the variable. If x is a variable, then $\text{DOM}(x)$ denotes the domain of x .

The next step is to define concepts regarding assignments of values to variables. We do this by first defining an assignment of one single variable, called a *label*.

Definition 2. A *label* is a tuple $\langle x, v \rangle$ where x is a variable and $v \in \text{DOM}(x)$ is a value in the domain of x .

More informally, a label is the state of a variable x after assignment of the value v . The process of assigning values to variables using a constructive method is often called *labeling*¹. The notation $\text{VAR}(\ell)$ will be used to denote the set of variables assigned by the label ℓ . We continue by formalizing the simultaneous assignment of several tuples.

Definition 3. A *k-compound label* ℓ is a set of tuples with size k , simultaneously assigning k values to k unique variables.

We use $\text{VAR}(\ell)$ to denote the variables in a compound label or an assignment. Compound labels are mostly used to assign values to a subset of the variables in a problem. In contrast to a *complete assignment*, assigning all variables in a problem, we refer to a compound label that does not necessarily assign all variables as a *partial assignment*.

Next, to reason about constraint satisfaction problems, we need to formalize the concept of a constraint. We use the notation c_S to denote the constraint c on the set S of variables.

Definition 4. A *constraint* c_{x_1, x_2, \dots, x_k} on the variables x_1, x_2, \dots, x_k is a set of k -compound labels for x_1, x_2, \dots, x_k , representing the compound labels that satisfy the constraint.

¹The term *enumeration* is also common, in particular in the systematic constraint satisfaction community.

We say that a constraint c_{x_1, x_2, \dots, x_k} has *arity* k , and refer to constraints with $k = 1, 2$ and 3 respectively as *unary*, *binary* and *ternary*. As an example of a constraint, $\text{GREATER-THAN}(x, y)$ is a relation that can be seen as a binary constraint, with the intuitive meaning that $x > y$. If the domains of x and y are both $\{1, 2, 3\}$, then the binary constraint $c_{x,y}$ representing GREATER-THAN is conceptualized by the set of 2-compound labels that satisfy the constraint, which are shown below.

$$\begin{aligned} c_{x,y} &= \{\ell_1, \ell_2, \ell_3\} \\ \text{where} \\ \ell_1 &= \{\langle x, 2 \rangle, \langle y, 1 \rangle\} \\ \ell_2 &= \{\langle x, 3 \rangle, \langle y, 1 \rangle\} \\ \ell_3 &= \{\langle x, 3 \rangle, \langle y, 2 \rangle\} \end{aligned}$$

Definition 4 states that a constraint is a set of compound labels. This representation of a constraint is hardly ever used in practice for space and efficiency reasons. When discussing constraint satisfaction we will sometimes talk about other representations of constraints, where the tuples that satisfy it is not readily available.

Definition 5. The *variables* of a constraint c_S , denoted as $\text{VAR}(c_S)$, is the set S of variables in the constraint.

$$\text{VAR}(c_S) = S$$

Next, we define the binary relation SATISFIES between a compound label and a constraint. To do this we first need to define projections of compound labels.

Definition 6. A compound label ℓ_b is a *projection* of a compound label ℓ_a , written as $\text{PROJECTION}(\ell_a, \ell_b)$ if and only if ℓ_a is a subset of ℓ_b .

$$\text{PROJECTION}(\ell_a, \ell_b) \equiv \ell_a \subseteq \ell_b$$

Definition 7. A compound label ℓ *satisfies* a constraint c if and only if there is some compound label in c that is a subset of ℓ .

$$\text{SATISFIES}(\ell, c) \equiv \exists \ell' \in c : \text{PROJECTION}(\ell', \ell)$$

This intuitively means that a partial assignment satisfies a given constraint if all variables involved in the constraint are assigned values that are compatible with the constraint.

For example, the compound label $\ell = \{\langle x, 3 \rangle, \langle y, 1 \rangle\}$ satisfies the constraint c representing the relation GREATER-THAN, where c is the set

$$\{\{\langle x, 2 \rangle, \langle y, 1 \rangle\} \{\langle x, 3 \rangle, \langle y, 1 \rangle\} \{\langle x, 3 \rangle, \langle y, 2 \rangle\}\}$$

of 2-compound labels that satisfy the constraint.

Definition 8. A compound label ℓ *satisfies* a set of constraints C if and only if ℓ satisfies all constraints $c_S \in C$ where S is a subset of X .

$$\text{SATISFIES}(\ell, C) \equiv \forall c \in C. \text{SATISFIES}(\ell, c)$$

Having defined the important concepts of constraints and variable assignments, we can now define constraint satisfaction problems themselves.

Definition 9. A *constraint satisfaction problem* is a triple (X, D, C) where X is a finite set of variables $\{x_1, x_2, \dots, x_n\}$, D is a function mapping each variable $x \in X$ to a set of values $\text{DOM}(x) = \{v_{1,1}, v_{1,2}, \dots, v_{1,q}\}$, and C is a finite set of constraints $\{c_1, c_2, \dots, c_m\}$.

We will assume that n and m represent the number of variables and the number of constraints respectively of a certain CSP.

In contrast to the partial assignments defined in 3, we define complete assignments as follows.

Definition 10. A *complete assignment* for a CSP (X, D, C) is an n -compound label ℓ where $n = |X|$ and $X = \text{VAR}(\ell)$.

Definitions 3 and 10 together state that if ℓ is a complete assignment, then ℓ is also a *partial* assignment. However, every partial assignment is not necessarily a complete assignment. In Chapter 4 we will look on another representation of assignments useful in the context of local search.

Definition 11. The *state space* $S(Y)$ of a set of variables $Y \subseteq X$ and a CSP (X, D, C) is the set of all possible assignments for Y :

$$S(Y) = \{\ell \mid \text{VAR}(\ell) = Y \wedge \forall y \in Y. v(y) \in \text{DOM}(y)\}$$

We call a state space where $Y = X$ the *total* state space for a CSP (X, D, C) .

Definition 12. The *state powerspace* $SP(Y)$ of a set of variables $Y \subseteq X$ and a CSP (X, D, C) is the set of all possible assignments for $Z \subseteq Y$:

$$SP(Y) = \{\ell \mid \text{VAR}(\ell) \subseteq Y \wedge \forall z \in \text{VAR}(\ell).v(z) \in \text{DOM}(z)\}$$

It is important to realize that the *total* state space is the state space for all variables in a problem, that is, all possible total assignments of all variables in the problem. The *state powerspace* of a set of variables Y is the union of all state spaces on subsets of Y , or alternatively put, all possible partial assignments of any variables in Y . Finally, the *total state powerspace* is then all possible partial assignments of any variables in the problem.

We can now define the concept of solutions to CSP's.

Definition 13. A *solution* to a CSP (X, D, C) is a compound label ℓ that satisfies C , where for all tuples $\langle x, v \rangle$ in ℓ , v is in the domain of x .

$$\text{SOLUTION}(\ell, (X, D, C)) \equiv \text{SATISFIES}(\ell, C) \wedge \forall x \in \text{VAR}(\ell).x \in \text{DOM}(x)$$

2.1.2 Constraint Graphs

Any given CSP $\mathcal{P} = (X, D, C)$ can be represented as a *constraint graph*². In a constraint graph, each arc represents a constraint³. Because an arc in a graph by definition is a pair of vertices, we first need to generalize regular graphs to accommodate k -ary edges.

Definition 14. A *hypergraph* is a tuple (V, \mathcal{E}) where V is a set of vertices and \mathcal{E} is a set of *hyperedges*. A hyperedge $\epsilon \in \mathcal{E}$ is a set of vertices.

Having defined hypergraphs, we can now define the *constraint hypergraph* for a given CSP.

Definition 15. A *constraint hypergraph* for a CSP is the hypergraph (X, \mathcal{E}) where X is the set of variables in the problem and \mathcal{E} is a set of hyperedges, where each hyperedge $\epsilon_i \in \mathcal{E}$ corresponds to a constraint c_i , and contains the set of variables that are present in c_i .

²In fact, they are represented by an *annotated* constraint graph, where additional information about constraints are annotated to the edges of the graph.

³If several constraints are on the same variables, these can be combined into one single constraint representing the conjunction of the constraint on the variables in question.

The concept of constraint hypergraphs can be restricted to *constraint graphs*, which are ordinary graphs with binary and unary edges only.

Definition 16. A *constraint graph* for a CSP is the undirected graph (X, E) where X is the set of variables in the problem and E is the union of the set of edges $\{s, e\}$ where $s, e \in c$ for some c , and the set $\{s\}$ where $\{s\} = c$.

The constraint graph of a general CSP \mathcal{P} is also called a *primal graph* of \mathcal{P} . As an example of a constraint graph, consider the inequality problem, where $X = \{a, b, c\}$, $D = \{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 3\}, c \mapsto \{1, 2, 3\}\}$ and $C = \{a \neq b, a \neq c, b \neq c\}$. This problem can be visualized as the constraint graph $(\{a, b, c\}, \{\{a, b\}, \{a, c\}, \{b, c\}\})$, as shown in Figure 2.1.

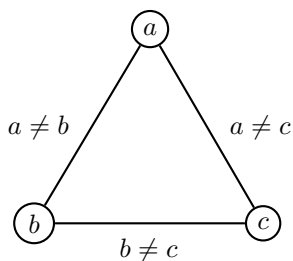


Figure 2.1: The constraint graph $(\{a, b, c\}, \{\{a, b\}, \{a, c\}, \{b, c\}\})$

Unfortunately, by transforming a constraint hypergraph into a regular constraint graph, we lose structural information about the original constraints. This information can potentially be used to prune the search space more aggressively by using specialized techniques for certain non-binary constraints. Worse yet, there are constraint hypergraphs that cannot simply be transformed into primal graphs. For example, consider the *parity problem* on n Boolean variables x_1, x_2, \dots, x_n . In this problem, we have one single constraint on the n variables, which only contains the n -compound labels where $x_1 \oplus x_2 \oplus \dots \oplus x_n = \mathbf{true}$ and \oplus is the logical *exclusive or* operator with the normal properties, defined as $x \oplus y \equiv (x \vee y) \wedge (\neg x \vee \neg y)$. Parity information is not local to any two variables. Therefore, the constraint hypergraph for this problem has no semantic-preserving primal graph.

2.1.3 Optimization and Partial CSP's

Many real-life problems are natural to represent using a set of constraints coupled with a function ordering the solutions to the problem with respect to their usefulness. The constraint satisfaction problem is a model without any gradation of solutions. To cope with this we extend the CSP model to accommodate for an *objective function* grading solutions to the problem.

Definition 17. A *constraint satisfaction optimization problem* (CSOP) is a quadruple (X, D, C, f) , where (X, D, C) is a CSP and $f : \text{DOM}(x_1) \times \text{DOM}(x_2) \times \dots \times \text{DOM}(x_n) \rightarrow \mathbb{R}$ is a function mapping each complete solution tuple to a numerical value.

The objective of solving a CSOP is to find the best (lowest or highest, rated by f) complete assignment that satisfies all constraints.

The following definition applies to minimization problems. Maximization problems can be modeled as a minimization problem where f is the negation of the objective function of the maximization problem.

Definition 18. A *solution* to a CSOP (X, D, C, f) is a complete assignment ℓ that satisfies C , and where for all other complete assignments ℓ' that satisfies C , it holds that $f(\ell) \leq f(\ell')$.

$$\begin{aligned} \text{SOLUTION}(\ell, (X, D, C, f)) &\equiv \\ \text{SATISFIES}(\ell, C) \wedge & \\ \forall v. \text{SATISFIES}(\ell', C) \rightarrow f(\ell) \leq f(\ell') & \end{aligned}$$

Many problems are easily expressed as constraint satisfaction problems, but sometimes we may still be interested in compound labels that satisfy not all but *some* of the constraints in the problem. Next, we extend the model of CSP's to formalize this concept into *partial constraint satisfaction problems*.

Definition 19. A *partial constraint satisfaction problem* (PCSP) is a quadruple (X, D, C, \succ) , where X is a set of variables $\{x_1, x_2, \dots, x_n\}$, $D : X \rightarrow d$ is a function mapping each variable $x \in X$ to a set of objects d , C is a finite set of constraints on a subset of variables in X and \succ is a partial ordering among subsets of C .

The relation \succ imposes an ordering of importance on the constraints, so that we can satisfy as important constraints as possible. Intuitively,

$C' \succ C''$ means that the constraints in C' are more important than those in C'' .

Definition 20. Given a PCSP $\mathcal{P} = (X, D, C, \succ)$, a compound label ℓ of all variables in X is a *solution* to \mathcal{P} if and only if there is no compound label $\ell' \neq \ell$ such that

$$\{c \mid c \in C \wedge \neg \text{SATISFIES}(\ell', c)\} \succ \{c \mid c \in C \wedge \neg \text{SATISFIES}(\ell, c)\}$$

2.2 Motivation

Many real-life problems can easily be expressed in CSP terms. In this section, we give a few examples of problems that can either easily be formulated as CSP's or, in the cases where the formulation is too complex to explain here, have successfully been formulated as CSP's.

2.2.1 Some Benchmark Problems

In this section we take a closer look on a small set of combinatorial problems traditionally used to test constraint systems; the *n-queens problem*, the *magic square problem* and the *magic sequence problem*. These problems have in common that they are easily modeled and straightforward in design, and therefore of mostly theoretical interest. Nevertheless, modelling these problems using constraints show the great generality of a constraint-based approach at problem solving.

The n-Queens Problem

The *n-queens* problem is simply the problem of placing n queens on an $n \times n$ chessboard, so that no two queens attack each other. The *n-queens* problem is a classical search problem used to test both backtracking search and local-search based algorithms.

We now model the *n-queens* problem using CSP notation. First, observe that each solution to the *n-queens* problem must have the queens placed on different rows and columns. Let the variables x_i denote the row position of the queen located at column i . The *N-queens* puzzle can then be formalized as follows.

Each queen placed on a row: $x_i \in \{1, 2, \dots, n\}$
 No two queens on the same row: $\forall i, j, i \neq j \rightarrow x_i \neq x_j$
 No two queens on the same diagonal: $\forall i, j, i \neq j \rightarrow i - j \neq x_i - x_j$
 $\forall i, j, i \neq j \rightarrow i - j \neq x_j - x_i$

As an example, the 3-queens problem can be expressed as the system of inequalities below. Some redundant constraints have been removed to clarify the model.

$$\begin{array}{l}
 x_1, x_2, x_3 \in \{1, 2, 3\} \\
 x_1 \neq x_2 \quad x_2 - x_1 \neq 1 \quad x_2 - x_1 \neq -1 \\
 x_1 \neq x_3 \quad x_3 - x_1 \neq 2 \quad x_3 - x_1 \neq -2 \\
 x_2 \neq x_3 \quad x_3 - x_2 \neq 1 \quad x_3 - x_2 \neq -1
 \end{array}$$

We can visualize the 3-queens problem using the constraint graph representation from Section 2.1.2. This graph is shown in Figure 2.2.

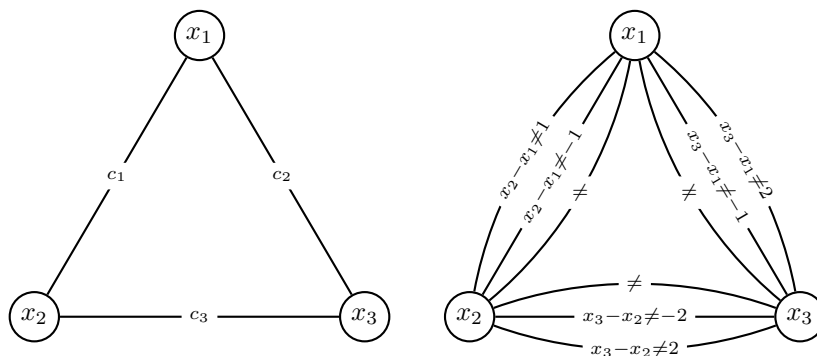


Figure 2.2: The constraint graph for the 3-queens problem, where $c_1 \equiv x_1 \neq x_2 \wedge x_2 - x_1 \neq 1 \wedge x_2 - x_1 \neq -1$, $c_2 \equiv x_1 \neq x_3 \wedge x_3 - x_1 \neq 2 \wedge x_3 - x_1 \neq -2$, $c_3 \equiv x_2 \neq x_3 \wedge x_3 - x_2 \neq 1 \wedge x_3 - x_2 \neq -1$. The corresponding multigraph, with multiple edges and constraints between variables, is shown to the right.

In solving the n -queens problem in practice, a mathematical model based on binary arithmetic constraints such as the one above is very inef-

efficient – the number of constraints is quadratic to the number of queens used. Using the global⁴ constraint `alldiff` (introduced by Régin in [52]) which simply state that a set of variables should take distinct values, we can represent a third of the constraints in the model as a single global constraint. This is illustrated in Figure 2.3.

The formulation is then reduced to:

$$\begin{aligned} & x_1, x_2, x_3 \in \{1, 2, 3\} \\ & \text{alldiff}(\{x_1, x_2, x_3\}) \\ & x_2 - x_1 \neq 1 \quad x_2 - x_1 \neq -1 \\ & x_3 - x_1 \neq 2 \quad x_3 - x_1 \neq -2 \\ & x_3 - x_2 \neq 1 \quad x_3 - x_2 \neq -1 \end{aligned}$$

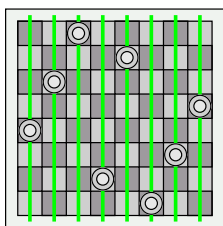


Figure 2.3: Using the `alldiff` constraint in the n -queens problem to restrict assignments of queens to the same column.

We can actually model the n -queens problem using only 3 constraints if we use a generalized version of the `alldiff` constraint, accepting a constant term in addition to a variable. The constraint then states that the value of each variable modified by the constant should be different from the value of all other variables, modified by their constants. This is illustrated in Figure 2.4. We show the new compact model below.

$$\begin{aligned} & x_1, x_2, x_3 \in \{1, 2, 3\} \\ & \text{alldiff}(\{x_1, x_2, x_3\}) \\ & \text{alldiff}(\{x_1 + 1, x_2 + 2, x_3 + 3\}) \\ & \text{alldiff}(\{x_1 - 1, x_2 - 2, x_3 - 3\}) \end{aligned}$$

In Section 5.5 we use the model above using our local-search constraint tool *Composer* to solve the n -queens problem.

⁴A *global* constraint is a usually non-binary constraint for a complex property with a specialized algorithm for achieving a level of *consistency* [72].

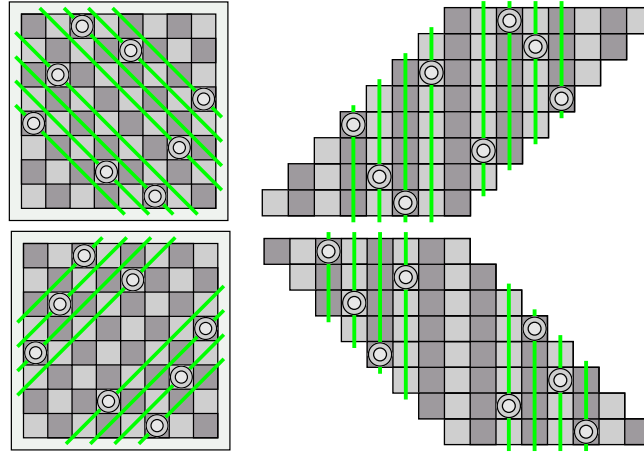


Figure 2.4: Using a modified `alldiff` constraint with constant terms in the n -queens problem. The two constraints restrict assignments of queens to the same left/right diagonal.

The Magic Squares Problem

A magic square with order n is a $n \times n$ matrix, containing the integers $1, 2, \dots, n^2$. In a magic square, each row, column and main diagonal must equal the same sum. The problem is to find magic squares for a given n , and can be formalized as follows. A magic square A is an n by n integer matrix for which it holds that the values in $1, 2, \dots, n^2$ occur exactly once, and

$$\begin{aligned} \text{MAGICSQUARE}(A) &\equiv \exists k : \\ &\forall i \in \{1, \dots, n\}. \sum_{j=1}^n a_{i,j} = k \\ &\forall j \in \{1, \dots, n\}. \sum_{i=1}^n a_{i,j} = k \\ &\sum_{i=1}^n a_{i,i} = k \\ &\sum_{i=1}^n a_{i,n-i} = k \end{aligned}$$

The constant k , which is the sum of all rows, columns and the two diagonals, can easily be computed to $k = n(n^2 + 1)/2$. We have not yet modeled and solved the magic squares problem in our implementation, but previous results by other researchers show that it is certainly possible to do so using local search [9, 41].

The Magic Sequence Problem

The magic sequence problem is similar to the magic squares problem in Section 2.2.1. A magic sequence of length n is a sequence of integers x_0, x_1, \dots, x_{n-1} between 0 and $n - 1$ such that for all integers $i \in \{0, \dots, n - 1\}$, the number i occurs exactly x_i times in the sequence. For instance

$$6, 2, 1, 0, 0, 0, 1, 0, 0, 0$$

is a magic sequence since 0 occurs 6 times, 1 occurs twice, 2 occurs once, and so forth.

The magic sequence problem can be modeled as follows. A magic sequence \mathbf{a} is an integer vector of size n for which it holds

$$\text{MAGICSEQUENCE}(\mathbf{a}) \equiv \forall i \in \{0, 1, \dots, n - 1\}. a_i = \sum_{j=1}^n \text{EQUAL}(i, a_j)$$

where the function EQUAL is defined as

$$\text{EQUAL}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

2.2.2 Configuration Problems

Configuration is the design of an artifact, which is to be assembled from predefined components, which can only be connected according to some restrictions. The goal is to select a set of components such that the specification of the artifact is satisfied. Configuration problems arise in many practical applications of the areas of sales, engineering and manufacturing.

Constraint systems are well suited to model configuration problems. Typically, the variables represent components or attributes of the artifact, and the values represent different selections for a component or

attribute. The constraints are used to form rules for selection of the components.

Because of the generality of configuration problems, we will not investigate practical constraint models for this topic. Instead, we refer the reader to [14], where configuration problems and models using local search are investigated.

2.2.3 Scheduling and Planning

Scheduling and planning problems are well-suited to formulation using constraint models. For example, work by Kreuger et al. [35, 34], Fox [15], Sadeh et al. [56, 57], Le Pape [48], Smith et al. [66, 67, 68], Davenport and Tsang [12], El Sakkout and Wallace [58], and Kamarainen [33] all address scheduling and planning using constraints. Often, scheduling problems are formulated with specialized constraints that encompass the functionality of a large set of “basic” constraints, as described in the articles by Beldiceanu [6] and Régis [52, 54].

The Progressive Party Problem

The progressive party problem is a well-known benchmark problem in the constraint programming community, and have been used in several generic constraint-based local search methods as well [73, 20, 42]. The problem can be described informally as follows.

An evening party is to be organized in the setting of a yachting rally. It has been decided that the visiting boats are going to visit the boats of the organizers in turn. The crew of a host boat serves the guests on the visiting boats, and every half-hour the guest boat move to a new host boat. This will go on for a given number of time periods. Also, no guest boat is allowed to visit the same host boat twice, and two crews must never meet more than once.

More detail of the Progressive Party Problem can be found in Section 5.6, where we show how to solve the progressive party problem using an implementation of constraint-based local search.

Below we now present two problems taken from CSPLib [24], an on-line combinatorial problem library for constraint-based solving approaches.

The Social Golfer Problem

The social golfer problem is described in CSPLib as follows.

The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion.

The problem can be generalized to that of scheduling m groups of n golfers over p weeks, such that no golfer plays in the same group as any other golfer twice.

The Car Sequencing Problem

The car sequencing problem is mentioned in work by Parrello and Kabat [49], Dinçbas et al. [13], Gent [21], Régis [54] and Lee et al. [37]. In CSPLib it is described as follows.

A number of cars is to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option.

Chapter 3

Constraint-Based Local Search

In this chapter we will discuss how discrete combinatorial problems can be solved using *local search methods*. In the context of constraints and combinatorial problems, local search has nothing to do with searching a local physical area. Instead, algorithmic local search refers to the restricted subspace of the total search space the algorithm investigates. As we will see in later chapters, local search is a highly efficient family of search techniques for constraint problems.

The standard method of solving generic constraint problems is to use a backtracking search, combined with pruning techniques to reduce the search space. Completeness is achieved because the backtracking search guarantees that a solution will be found if one exists. This property is of course desirable, but also comes with a high price: because algorithms of this type must guarantee that all solutions can be found, it is not uncommon that execution times are in practice exponential in the size of the problem.

On the other hand, heuristic methods for constraint satisfaction are based on incomplete search algorithms, sacrificing completeness for speed¹. In comparison with complete search, a restricted local search, concentrated on promising areas of the state space, can in many cases provide superior performance in terms of time and space requirements. In this

¹Note that for the search to prove infeasibility or optimality of a problem we still need complete search.

chapter, we discuss theory and some generic methods for heuristic search, and then apply these on local search in a constraint-based environment.

3.1 Basics of Local Search

The exact formalization of a local search algorithm includes specifying algorithmically the subset of the state space to be investigated. For example, *iterative improvement* is an algorithm subclass of local search in which a single candidate solution is *improved* iteratively to find a good enough solution. We will refer to the improvement as a *transition* from one state to another. The initial solution is often randomly generated or generated according to some orthogonal heuristic. Thus, the search space of iterative improvement can be described by 1) an initial solution, and 2) an improvement strategy. Very common is also an explicit *utility function*, computing the “goodness” of a candidate solution.

Another example of what can be considered local search are so-called *genetic algorithms*, which have much in common with iterative improvement methods. In a genetic algorithm (GA), elements from a set of *genotypes* are randomly modified (*mutation* on GA language) and/or combined (*crossover*) with each other, resulting in an offspring set of genotypes. A genotype represents a solution candidate – in many cases, the genotype is simply a bit string representing the values (*alleles* in GA theory) of the problem variables (*genes*). Note the similarity to iterative improvement using multiple solution candidates instead of a single one.

3.1.1 The Travelling Salesman Problem

As an example of a practical application of local search, consider solving the well-known Travelling Salesman Problem (TSP) using local search. The problem can be stated informally as follows. Imagine a salesman who has to visit all the cities in a country, returning at the end of the trip to the city where he/she started. Because there is a cost (gasoline, etc.) associated with traveling between each pair of cities, the salesman wants to find a tour that is as short as possible.

We now state TSP formally. First, we need to define some auxiliary terms. A *path* in a graph (V, E) is a sequence $\langle v_0, v_1, \dots, v_k \rangle$ of vertices $v_i \in V$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$. The path *contains* the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. The path is *simple* if all vertices in the path are distinct. A path forms

a *cycle* if $v_0 = v_k$ and the path contains at least one edge. A simple cycle that contains each vertex in V is a *Hamiltonian cycle*, which we also refer to as a *tour*.

We associate a real-valued constant distance $c(i, j)$ to each pair (i, j) of vertices. Formally, TSP is defined as a pair $\langle G, c \rangle$ where $G = (V, E)$ is a complete graph² and $c : V \times V \rightarrow \mathbb{R}$ is a distance function from edges to reals. The goal of TSP is to find a Hamiltonian cycle p in G visiting all cities and minimizes the sum of the distance of the edges in p . We call a TSP *undirected* iff $\forall i, j \in V. c(i, j) = c(j, i)$ and *directed* otherwise.

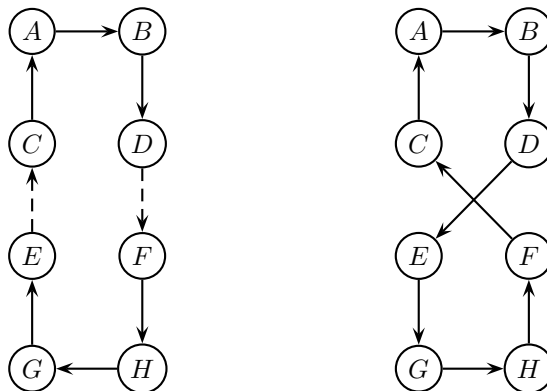


Figure 3.1: The effect of a 2-interchange transition on a tour in solving the TSP. The original tour is to the left and the resulting tour to the right. Observe that the direction of traversal has changed in the lower part of the tour.

A common approach for solving TSP using local search is to start with a Hamiltonian cycle p^0 , constructed using some heuristic, and then improve the tour using interchanges of edges in the tour. A very common interchange operator is the edge interchange operator of [11] (the 2-interchange transition), which deletes two edges from the tour, thus breaking it into two paths. Then, the two paths are recombined to form a new tour in the other possible way. See Figure 3.1 for an example of a 2-interchange transition.

An extension of the 2-interchange transition is the 3-interchange tran-

²A complete graph is a graph where all pairs of vertices are connected via an edge.

sition, which examines interchanges of 3 edges of the tour. Using these transitions, we investigate resulting tours with reduced length until no interchange yields any improvement. The resulting tour is called either *2-optimal* or *3-optimal*, depending on the type of transition used.

3.1.2 Iterative Improvement

Iterative improvement is a technique for local search where we locally improve a state (or *solution candidate*), according to the value³ of the state. This reparation is typically continued in several iterations, until either a sufficiently good state is found, or after a fixed number of iterations, when we abort the search. The initial state is commonly chosen using either a heuristic or simply at random.

In the rest of this thesis, we will often discuss states in the search. As a convention, we will use the letters u and v to denote such states (and later assignments).

3.1.3 Costs and Cost Functions

We continue in this section by formally defining the meaning of a *cost function*, which is applied to a state to yield its *cost*. We will use the term “cost” throughout this text. As an example, in constraint-based local search, we use a cost function based on the violated constraints in the current state as a measure of value.

In general, the cost of a state is taken from some *cost set*. In order to compare different states with each other, we want the cost set to be totally ordered by a relation:

Definition 21. A *cost set* W is a set which is totally ordered by a relation \leq .

A total order on a set W by a relation \leq has the following properties.

- $\forall x \in W. x \leq x$ (reflexivity)
- $\forall x, y \in W. x \leq y \wedge y \leq x \rightarrow x = y$ (antisymmetry)
- $\forall x, y, z \in W. (x \leq y \wedge y \leq z) \rightarrow x \leq z$ (transitivity)
- $\forall x, y \in W. x \leq y \vee y \leq x$ (linear order property)

³Some commonly used terms for this are *cost*, *penalty*, *violation level* and *utility*.

The by far most common cost set is \mathbb{N} ordered by the usual \leq relation. This is also the cost set we use in our local search implementation as described in Chapter 5. However, it is important to realize that other cost sets can be useful. Consider for example a heuristic that should as usual minimize the violation of the constraints used, but also minimize the unused capacity in a resource allocation problem. A sensible cost set for this would be \mathbb{N}^2 together with a user-defined relation \lesssim , where we let the first attribute of an element in \mathbb{N}^2 represent the violation of the constraints, and the second the unused capacity of the resources. We can now define the relation \lesssim as follows.

$$\forall (u_1, u_2), (v_1, v_2) \in \mathbb{N}^2. (u_1, u_2) \lesssim (v_1, v_2) \leftrightarrow u_1 < v_1 \vee (u_1 = v_1 \wedge u_2 < v_2)$$

If we consider local search with real-valued variables, the cost set can be generalized to \mathbb{R}^2 instead.

In the following definition we use S to denote the set of possible states in a problem specification, similar to Definition 11 of state space in Chapter 2.

Definition 22. Let S be a state space of a problem and W a cost set. A *cost function* f is a function with type $S \rightarrow W$ which map states to costs.

The difference between a cost function and an objective function (from Section 2.1.3) is that a cost function is used to rate *states*, which do not need to satisfy the constraints in the problem. Instead, the cost is a measure of to which extent a solution satisfies the constraints. Orthogonally to this, an objective function is used to rate *solutions* to the problem. We make this distinction because we are mainly interested in solving satisfaction problems, although a cost function could of course also model the objective value of solutions.

We refrain from defining states formally for now, because they can in essence represent any possible collection of mutable data in a model of a problem. Later in this thesis we will concentrate on the special case where a state is an assignment of values to a set of variables.

In designing a local search method, the first thing usually done is to specify a cost function. There often exist quite natural measures of a state in the given problem, which can be used as a cost function, but the natural choice may not be the optimal cost to use in practice. For example, in constraint-based local search, an often used and natural cost

function is to use the number of violated constraints for a given state. However, if the problem contains very few constraints, or if some constraints span a large percentage of the total variables, this cost function will not be very useful. The result is that we cannot differ between states that satisfy the constraints partially to different levels. A more accurate representation of cost in these cases would be to use a cost for each complex constraint based on a set of equivalent, less complex, constraints. In Section 4.2 we develop this idea further.

3.1.4 Neighborhoods and Transitions

The next important design decision made when implementing a local search procedure based on iterative improvement is to decide upon a *transition function*, to be applied to the current state in each iteration. The transition function simply transforms a state into another state, and is usually parametrized over some parts of the state to be modified. The transition function induces a *neighbor relation* on the state space of the problem – we say that a state v is a *neighbor* of (or *adjacent* to) a state u if application of the transition function (with some parameters) on u yields v . Note that this relation is *directed*. It is common that neighbor relations are defined as undirected relations. The motivation for using directed neighbor relations is that we need the direction when analyzing adaptive memory-based transitions. These transitions restrict visits to previously investigated states, making the neighbor relation undirected in practice.

In this thesis we also do not require the neighbor relation to be reflexive and symmetric⁴ – in fact, irreflexivity is often a useful property, and introducing Tabu search, breakout and other types of adaptive memory often give us an asymmetric neighbor relation. Also, we will frequently use a loose definition of a relation and allow this relation to change over time in order to model more elaborate neighborhoods.

Formally, a neighbor relation is simply a binary relation on the set of possible states, induced by the transition used. We use the notation $u \xrightarrow{T} v$ to denote that v is a neighbor of u given the transition T , and $u \xrightarrow{T^*} v$ to denote that we can reach v in zero or more steps from u using T . We use the term *neighborhood* to denote the states reachable in a single transition from a given state.

⁴A relation R is *reflexive* if $R(a, a)$, *irreflexive* if $\neg R(a, a)$, and *symmetric* if $R(a, b) \leftrightarrow R(b, a)$, for all a, b .

Definition 23. The *neighborhood* $\text{NBH}_T(u)$ for a transition function T and a state u is the set of states which are neighbors of u via T .

$$\text{NBH}_T(u) = \{u' \mid u \xrightarrow{T} u'\}$$

A property we are often interested in when using local search methods is that any two states in the state space are reachable from each other. This corresponds to that we (theoretically) can visit all states of the problem. We can express this property formally by first defining the *transitive-reflexive closure* of our neighbor relation for a given state.

Definition 24. The *transitive-reflexive closure* $\text{TC}_T(u)$ of a state u and a transition function T is the set of states which are reachable from u using T in zero or more steps.

$$\text{TC}_T(u) = \{u' \mid u \xrightarrow{T^*} u'\}.$$

Using this definition we can now state the wanted property easily as follows.

Definition 25. A transition function T , on a state space S , is *covering* S if and only if its transitive-reflexive closure of any state $u \in S$ equals S .

$$\text{COVERING}_T(S) \equiv \forall u \in S. \text{TC}_T(u) = S$$

We will sometimes refer to a transition covering the state space implicitly defined from the context simply as a *state space covering* transition.

Another interesting attribute of the neighbor relation is which states can be reached in a given number $n \geq 0$ of steps. We define the *distance* between two states u and v with regard to a neighbor relation as the minimum number of transitions that has to be done to get to v from u .

Definition 26. The *distance* $\text{DIST}_T(u, v)$ between two states u and v is

$$\text{DIST}_T(u, v) = \begin{cases} 1 & \text{if } u \xrightarrow{T} v, \\ 1 + \min_{u' \in \text{NBH}_T(u)} (\text{DIST}_T(u', v)) & \text{otherwise.} \end{cases}$$

Note that the distance between two states is at least 1, and in particular that $\text{DIST}(u, u) \neq 0$. We chose this definition to be able to analyze the cyclic behaviour of local search.

3.1.5 Neighbor Selection

In theory, in each iteration of local search we first generate a representation of the neighborhood of the current state using all parameter instances for the transition function used, and then select the next state using some criteria on the transition. The most primitive selection used is to try all transitions, yielding all neighbors, and then select either the *first* generated neighbor that improves the current cost or the *best* possible neighbor. In practice however, generating and evaluating possibly all neighbors to the current state can be too expensive, in terms of execution time, due to the size of the neighborhood. Real implementations of local search algorithms usually involve a custom-made neighbor generation and selection phase for efficiency reasons.

Another orthogonal technique for gaining a more efficient neighborhood analysis phase is to use *incremental computation* of the cost of the neighbors. This usually involves computing only the difference in cost that the application of the transition on the current state yields. We will discuss this topic further in Section 5.4 and in Chapter 5.

3.2 Constraint-Based Local Search

In this section we will apply the methods outlined above to constraint problems. Although we focus on satisfaction problems, optimization problems solved using local search in practice use the same or very similar techniques. It is easy to add an objective function for optimization of the search along with the cost of the constraints. Optimization is however not covered by this thesis.

We begin by formally defining assignments as they are usually used as states in constraint-based local search. Although there exist many variations of what an assignment really is, we will use one based on *functions*.

Definition 27. An *assignment* v for a set of variables Y and a CSP (X, D, C) , where $Y \subseteq X$, is a function $v : Y \rightarrow V$ mapping each variable in Y to a value.

We also have *domains* on all variables in the problem. We can view the domains of the variables as unary constraints on these, and indeed one approach at handling domains in local search is to treat them as any other constraint. In this thesis we will instead use a more rigid model,

where the domain constraints of the problem are guaranteed to always remain satisfied. The advantage of this is that we can concentrate on a smaller state space than if the domain constraints were soft, which in turn allows us to limit the amount of memory used to maintain constraints and costs during the search. We call an assignment that fulfills all domain constraints *domain conforming*, and restrict our search to such states.

Definition 28. A *domain conforming* assignment is a state in which all variables are assigned values in the domain of the corresponding variable.

$$\text{DCONF}(u) \equiv \forall x \in \text{VAR}(u). u(x) \in \text{DOM}(x)$$

Note that the definitions above are different from the one given in Definition 3 used as an assignment in the CSP theory. We use the notation $\ell(u)$ to refer to the compound label definition of an assignment u . We still use Definition 1 for the domain of the variables in the problem.

Most assignments used in local search has the additional property that $Y = X$, and as in Chapter 2 we will refer to such assignments as *total*. In this thesis we will almost exclusively use total assignments. However, many of the topics that we discuss is also applicable to search using partial assignments.

We use x and y , with optional subscripts, to denote individual variables in the following text. $\text{VAR}(u)$ is used to denote the set of variables in the assignment u and we use $u(x)$ to denote the value of the variable x in the assignment u . Values are usually denoted by a, b, c etc., optionally with subscripts. For an explicit assignment we use the notation

$$\{x_1 \mapsto a_1, x_2 \mapsto a_2, \dots, x_n \mapsto a_n\},$$

denoting a function mapping the variable x_1 to the value a_1 , x_2 to a_2 etcetera. We also use

$$u[x_1 \mapsto a_1, x_2 \mapsto a_2, \dots, x_n \mapsto a_n]$$

to denote the function obtained by simultaneously replacing the mapping of the distinct variables x_1, x_2, \dots, x_n to the values a_1, a_2, \dots, a_n in u . For example, assuming that we have the assignment $u = \{x \mapsto 1, y \mapsto$

$2, z \mapsto 3\}$ we can deduce the following.

$$\begin{aligned} u(y) &= 2 \\ u[y \mapsto 4] &= \{x \mapsto 1, y \mapsto 4, z \mapsto 3\} \\ u[y \mapsto 4](y) &= 4 \\ u[w \mapsto 4] &= \{x \mapsto 1, y \mapsto 2, z \mapsto 3, w \mapsto 4\} \end{aligned}$$

Replacing a mapping of a variable which did not previously exist in the function naturally adds the mapping to the function.

We also use $\|u\|_1$ as notation for the norm $\sum_{x \in \text{VAR}(u)} u(x)$ of an assignment u , where the '+' operator is defined on the values of the variables in $\text{VAR}(u)$. Subtraction and addition of two assignments u and v is defined if $\text{VAR}(u) = \text{VAR}(v)$ as variable-wise subtraction and addition of the values in the states, given that the operator '-' is also defined on the possible values. For example, $u - v$ yields

$$\{x_1 \mapsto u(x_1) - v(x_1), x_2 \mapsto u(x_2) - v(x_2), \dots, x_n \mapsto u(x_n) - v(x_n)\}$$

where $n = |\text{VAR}(u)| = |\text{VAR}(v)|$.

Given a constraint problem, we have a set of variables, a set of constraints, and domains on all variables in the problem. The objective is to find an assignment of the variables (in the corresponding domains) such that all constraints are satisfied. A common technique for constraint-based local search is to define a *cost function* for the constraints in the problem. The cost function assigns a cost to each assignment, which we then can vary according to a strategy to try to find a solution to the problem. Usually the cost is 0 when we have a solution to the constraints in the problem.

The state space of a CSP is defined in Definition 11. In local search it is often the case that we want to restrict the search to a subset of the state space where a set of constraints are satisfied. We refer to such constraints as *structural constraints*, and extend Definition 11 of the state space of a constraint problem to take into account this notion.

Definition 29. The *state space* $S_{C'}(Y)$ of a set of variables $Y \subseteq X$ and a CSP (X, D, C) , with respect to a subset $C' \subseteq C$ of the constraints in the CSP, is the set of all possible assignments for Y such that all constraints in C' are satisfied:

$$S_{C'}(Y) = \{v : Y \rightarrow \bigcup_{y \in Y} \text{DOM}(y) \mid \forall y \in Y. v(y) \in \text{DOM}(y) \wedge \text{SATISFIES}(\ell(v), C')\}$$

As before, we are often interested in the case of a *total* state space, where $Y = X$, with respect to some structural constraints C' .

We also call a transition function on a state space domain conforming, if it is not possible to break the domain constraints using the transition:

Definition 30. A transition function T on a state space S is *domain conforming* on S if and only if its transitive-reflexive closure of any state $u \in S$ contains only domain conforming states.

$$\forall u \in S. \forall v \in \text{TC}_T(u). \text{DCONF}(v)$$

We now formally define a *cost function* used in the search to assign costs to assignments.

Definition 31. Let $SP(X)$ be the state powerspace of possible assignments for the variables in a constraint satisfaction problem (X, D, C) and CS be a cost set. A *cost function* $f : SP(X) \rightarrow CS$ maps each possible assignment to an element in CS .

3.2.1 Transitions and the Neighbor Relation

Recall from Section 3.1 that the transition function is used to transform a state into the neighbors of the state by varying certain parameters. In constraint-based local search, the transition function is commonly parametrized over the variables of the problem. Formally, the transition function is a function on an assignment and parameters controlling which variables should be varied, returning a new state. We use the earlier notation of replacing mappings of variables for transition functions.

As an example of a transition function for use in constraint-based local search, the transition of swapping the value of two variables in the assignment u is defined as follows.

$$\text{SWAP}(u, x, y) = u[x \mapsto u(y), y \mapsto u(x)]$$

The SWAP transition has been shown to be very efficient for certain combinatorial problems, see for example [69, 20, 19]. Also, proper set up combinatorial problems with `alldiff` constraints can in some cases be simplified by removing some `alldiff` constraints and using SWAP transitions, ensuring that all states investigated respects the constraints anyway.

Observe that SWAP does *not* always result in domain conforming states. As an example of this, two variables x and y where $\text{DOM}(x) \neq$

$\text{DOM}(y)$ cannot in general be swapped without breaking the domain-conforming property on the resulting state. We can use conditional statements to restrict the transition function to domain conforming states. This allows a swap to take effect only if the values to be swapped are in the respective domains of the variables. Below is the corrected domain conforming SWAP transition.

$$\text{SWAP}(u, x, y) = \begin{cases} u A & \text{if } u(y) \in \text{DOM}(x) \wedge u(x) \in \text{DOM}(y) \\ u & \text{otherwise.} \end{cases}$$

where $A = [x \mapsto u(y), y \mapsto u(x)]$.

However, in general the SWAP transition is still not a state-space covering transition. Starting in any state u we can only visit the part of the state space where the variables take as values a permutation of the original values of u . Take as example any `alldiff` constraint c . For any CSP's where some of the domains differ, the SWAP transition is not state-space covering with respect to c . We correct this later by combining modifications of the swap and the assignment transitions.

The simplest domain-conforming and state-space covering neighbor function for use in integer constraint satisfaction is the *1-modification* transition function, in which we add or subtract 1 from/to the value of a single variable, assuming that the resulting value is still in the domain of the variable. This transition has been used previously in [73] and is an extension of the *flip* transition of SAT local search, where the value of a Boolean variable is simply negated. Formally, the 1-modification transition function is defined as

$$\text{MODIFY}_1(u, x, m) = \begin{cases} u & \text{if } u(x) + m \notin \text{DOM}(x) \\ u[x \mapsto u(x) + m] & u(x) + m \in \text{DOM}(x) \end{cases}$$

where the parameter $m \in \{-1, 1\}$ is used to denote the actual modification of the current value. Note the domain-inclusion tests to ensure that the result of the transition is a domain conforming assignment. We can easily extend the transition function to investigate a change of at most n by changing the allowed values of m to $\{-n, \dots, -1, 1, \dots, n\}$.

The related ASSIGN transition is probably the most used neighbor construct for generic constraint-based local search. It is natural to define, clear and simple, and works surprisingly well on many combinatorial problems, despite its lack of structure-capturing properties. Several pro-

blems are solved using this transition, or variants of it, in work by [42], Codognet and Diaz [9], Galinier and Hao [20] and Nareyek [45].

We define the ASSIGN transition as follows.

$$\text{ASSIGN}_n(u, x_1, a_1, \dots, x_n, a_n) = \begin{cases} u & \text{if } \exists i, j. i \neq j \wedge x_i = x_j \\ u & \text{if } \exists i. a_i \notin \text{DOM}(x_i) \\ u \ A & \text{otherwise} \end{cases}$$

where $A = [x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$.

Once again we check for domain inclusion of the changed value to ensure a result respecting the domains of the variables. Note that we allow the neighbor to take equal values for all or some of its variables. We could force the ASSIGN transition to assign different values to all variables present, but this would break the covering property of this transition, and would not improve evaluation time significantly in practice.

Note also that that all variables assigned are distinct. The most common incarnation of the ASSIGN transition is the special case where $n = 1$, and a single variable is considered for change. The reason for this is that the neighborhood size expands rapidly as n is increased – often, a compromise between high guidance and rapid evaluation is to set $n = 1$.

Theorem 1. *For any n where $1 \leq n \leq m$ where m is the number of variables in the problem, the n -ASSIGN transition covers the state space S of all possible assignments.*

Proof. In each iteration we are given the possibility to change n variables to any values in their domains. We can therefore reach any assignment in $\lceil \frac{m}{n} \rceil$ iterations, where m is the number of variables whose values differ from the current assignment. \square

As noted before, the SWAP transition is not a state space covering transition. The reason for this is twofold. First, the SWAP transition allows us to, in the best case, traverse a state space containing a permutation of the assignments in the initial state. Second, the state space we can traverse is even more restricted if the domains of the variables are different – in this case, we cannot even traverse all permutations, since a value of one variable may not be feasible for another variable.

One way to correct this problem is to use two transitions interleaved with each other. If we choose to use both the SWAP and the ASSIGN₁ transition, the properties of ASSIGN₁ holds here as well.

3.2.2 Cost and Conflicts

In this section we look closer on common cost functions used in constraint-based local search. Since we are interested in *generic* local search for constraint problems, we will exclusively use cost functions parameterized over the problem instance used, which in the case of constraint-based local search means a cost function which is parametrized over the constraints in the problem.

The most obvious generic cost function is simply the number of violated constraints in the problem. We name this cost function the *basic cost function*. We can formalize the basic cost function as follows.

Definition 32 (Basic Cost Function). The cost function f^b defined on a set of constraints C is defined as

$$f^b(C, v) = |S| \text{ where } S = \{c \mid c \in C \wedge \neg\text{SATISFIES}(\ell(v), c)\}.$$

Remember from Definition 7 in Section 2.1.1 that the predicate SATISFIES is true if and only if the assignment v satisfies the constraint c . Using the cost function to measure the value of assignments (assuming that lower values equal better assignments), any given CSP can be stated as a minimization problem

$$\begin{aligned} &\text{minimize} && f(C, v) \\ &\text{subject to} && v_i \in \text{DOM}(x_i), 1 \leq i \leq n \end{aligned}$$

where n is the number of variables in the problem. Note also that we can tell if a given assignment v is a solution to the problem by testing if $f^b(C, v) = 0$, because this means that no constraints are violated.

The basic cost function is very primitive and not suited to constraint problems in which global constraints are used. To remedy this we now define the *extended cost function*, which we use in practice in the problem solving parts of Chapter 5.

Definition 33 (Extended Cost Function). The cost function f^e defined on a set of constraints C is defined as

$$f^e(C, v) = \sum_{c \in C} w_c f_c(v)$$

where w_c is a weight and $f_c(v)$ is the constraint cost of c .

We use weights w_c for the constraints to trim the balance of the local search. This makes it possible to increase the weight on constraints that are hard to satisfy. Increasing the relative weight of a constraint makes it more rewarding to satisfy this specific constraint. Also, the weight makes it possible to balance the total cost, by changing the weight on constraints whose cost is not directly comparable with the cost of other constraints. Observe that in the special case where $w_c = f_c = 1$ for all constraints, the extended cost function is equal to the basic cost function. We will address the problem of defining f_c for specific constraints later on.

We define the *conflict level* $\text{CL}(C, v, x)$ of a variable $x \in \text{VAR}(v)$ in a state v to be the (possibly weighted) sum of the conflicts on x , for all violated constraints in which x is present.

Definition 34 (Conflict Level). The conflict level CL on a set of constraints C is defined as

$$\text{CL}(C, v, x) = \sum_{c \in C} w_c \text{CL}_c(v, x)$$

where w_c is the constraint weight of c , and $\text{CL}_c(v, x)$ is the conflict level of x for the constraint c , given the assignment v .

The conflict level of a variable x for a constraint c indicates the extent to which the current value $v(x)$ of x contributes to the current cost $f_c(x)$ of the constraint. Typically, the conflict level of a variable in a constraint is therefore dependent on both the cost and the internal structure of the constraint.

To exemplify conflict levels, assume an `alldiff` constraint with a cost equal to the number of corresponding unique binary inequality constraints that are broken by a given assignment. For example, for an `alldiff` constraint over the variables x, y, z, w , an assignment

$$\{x \mapsto 1, y \mapsto 1, z \mapsto 1, w \mapsto 1\}$$

would have a cost of 6, corresponding to the unique violated binary constraints

$$x \neq y, x \neq z, x \neq w, y \neq z, y \neq w, z \neq w$$

where the assignment

$$\{x \mapsto 1, y \mapsto 1, z \mapsto 2, w \mapsto 2\}$$

would have a cost of 2, corresponding to the broken constraints

$$x \neq y, z \neq w$$

The conflict level of a variable for the `alldiff` constraint could be defined as the number of other variables that take the same value as x . This conflict level is equal to the amount the cost could be decreased if x were assigned a non-conflicting value instead of its current value.

The conflict levels for the variables are important in problems where the evaluation time of the neighborhood is high. In these cases, a pre-processing stage where a small subset of the neighborhood is selected for further evaluation can be very useful. In constraint-based local search, the conflict level of a variable is often a good estimate of which variables are most critical to modify. We use conflict levels to decrease the size of the neighborhood significantly when solving some of the benchmark problems in Chapter 5.

3.3 Local Minima and Plateaus

In finite-domain constraint-based local search, the discrete formulation of the problem as a CSP sometimes makes the search spend considerable amounts of time searching areas where no cost improvement is possible. Also, since the search is based on local improvement, the search often gets stuck in “basins” which are locally optimal, but do not represent a solution to the problem. This behavior is well-known, and has been reported for SAT by Frank et al. [17] and Hampson and Kibler [29].

Using the formal treatment of neighborhoods and transitions in Section 3.1.4 we can define the characteristics of a special state subset, which we call a *plateau*, where a primitive local search algorithm will get stuck.

Definition 35. A *plateau* with respect to a transition T and a cost function f is a nonempty state space L where (1) T is covering L , and (2) all states have equal cost.

$$\text{PLATEAU}_{T,f}(L) \equiv |L| > 0 \wedge \text{COVERING}_T(L) \wedge \forall u, v \in T. f(u) = f(v)$$

Note that we do not require a plateau to be the largest possible plateau at one location, although in general these are the plateaus we are interested in. This also means that most plateaus contain a lot of sub-plateaus. We will define maximal plateaus later.

Figure 3.2 shows a 2-dimensional constraint problem with five maximal plateaus, where height represents cost. For a 1-dimensional problem, Figure 3.3 also illustrates three maximal plateaus, L_1 , L_2 and L_3 .

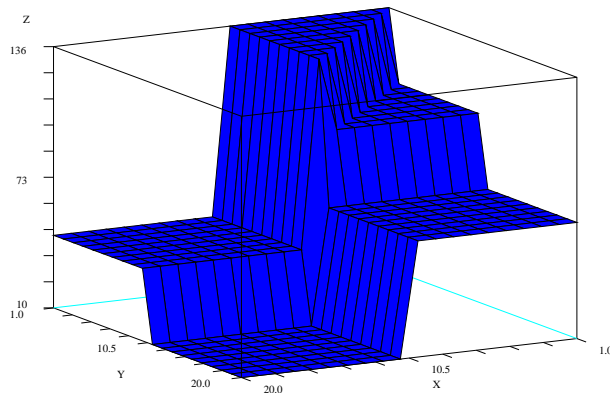


Figure 3.2: Five typical maximal plateaus in a visualization of a 2-dimensional CSP where height represent costs of assignments.

Discussing plateaus in the search space, we are often interested in the specific cost of the states in the plateau. We call this the *level* of a plateau, and define it as follows.

Definition 36. The *level* $f(L)$ of a plateau L is the value of the cost function f for an arbitrary state in L .

The level of a plateau is well-defined due to that a plateau by definition contains at least one state. To further define the special traits of plateaus, we need to investigate the states that are adjacent to the plateau, but not themselves members of the plateau. We call the set of such states the *border* of the plateau.

Definition 37. The *border* $B_T(L)$ of a plateau L with respect to the transition T is the set of states that are (1) not elements of L , and (2) are adjacent, with respect to T , to at least one state in the plateau.

$$B_T(L) = \left(\bigcup_{u \in L} \text{NBH}_T(u) \right) \setminus L$$

The border may be used to find out which kind of plateau we are dealing with. If there are no states in the border of a plateau for which the cost of the state and the level of the plateau are equal, we say that the plateau is a *maximal* plateau.

$$\text{PLATEAU}_{T,f}(L) \wedge \neg \exists u \in B_T(L). f(u) = f(L)$$

A plateau where all states in the border have a higher cost than the level of the plateau is a *minimum*, and would appear in three-dimensional space, with height representing cost, as a valley in the cost surface. Two minima are illustrated as L_1 and L_3 in Figure 3.3.

Definition 38. A *minimum* with respect to a cost function f is a plateau where all states in the border of the plateau has higher cost, measured by f , than the level of the plateau:

$$\begin{aligned} \text{MINIMUM}_{f,T}(L) \equiv \\ \text{PLATEAU}_{f,T}(L) \wedge \forall u \in B_T(L). f(u) > f(L) \end{aligned}$$

Definition 39. A *local minimum* is a minimum L_1 where there exists another minimum L_2 with lower level than L_1 .

$$\begin{aligned} \text{LOCAL-MINIMUM}_{f,T}(L) \equiv \\ \text{MINIMUM}_{f,T}(L) \wedge \exists L' : \text{MINIMUM}_{f,T}(L') \wedge f(L') < f(L) \end{aligned}$$

In Figure 3.3, L_1 is a local minimum. If a minimum is not a local minimum, it is a *global minimum*. For decision problems that have a solution, global minima are trivial to detect – an assignment \hat{u} is in a global minimum if $f(C, \hat{u}) = 0$ for the set of constraints C in the problem. This is under the assumption that the cost function f used is 0 when no constraints are violated.

In Figure 3.3, the plateau L_3 is a global minimum. Note that it is possible for a search space to have several global minima, and also that for optimization problems in general, global minima cannot easily be detected in reasonable time.

Definition 40. A *bench* is a maximal plateau that is not a minimum.

$$\text{BENCH}_{f,T}(L) \equiv \text{PLATEAU}_{f,T}(L) \wedge \neg \text{MINIMUM}_{f,T}(L)$$

Definition 40 states that a bench has at least one assignment with lower cost in the border. This implies that a bench may be exited by

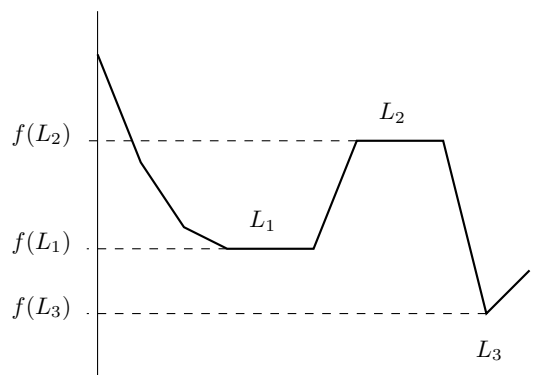


Figure 3.3: Illustration of typical plateaus in a 1-dimensional CSP.

a sufficiently elaborate search algorithm. A bench L_2 is illustrated in Figure 3.3.

In the empirical studies on random SAT instances by Frank et al. [17] and Hampson and Kibler [29], most local minima tended to be small, but the largest local minimum in problem instances with 100 variables often exceeded 10000 assignments. In these publications, local minima were also more frequent than benches, and benches tended to be much larger than local minima.

In other publications by Larrabee and Tsuji [36] and Monasson et al. [46], local minima were more frequently found in random problem instances generated in the *phase transition* of satisfiability problems. Phase transition occurred when the number of clauses m divided by the number of variables n roughly equals 4.3. Random problem instances in phase transition have been shown to be especially hard to solve.

3.4 Plateau Traversal and Avoidance

Local search has been used as a heuristic method for solving hard satisfaction and optimization problems for quite some time. These early approaches for using local analysis and search were tightly coupled with the specific problem instance, and hence the possibilities for reuse were not obvious. However, in the last 15 years several local search techniques usable for local minima avoidance and plateau traversal, applicable to

a larger class of problems, have been developed. In this section we will take a closer look on some of the most influential techniques.

3.4.1 Flat Transitions

If the local search only selects improving transitions, the search will terminate with an empty neighborhood at the first plateau. One of the most obvious strategies to escape benches is then simply to allow transitions that neither increase nor decrease the cost of the search. This allows the search to traverse a flat plateau, which in the case of a bench hopefully leads the search to the edge of the bench, making escape possible. This strategy has been shown to be practically a necessity for many problems, and is used by Wu and Wah [76], Shang and Wah [65], Schuurmans and Southey [60], Hoos and Stutzle [30], Galinier and Hao [20], Walser [73] Frank et al. [17], Cha and Iwama [7], Morris [44], Selman et al. [64] and Ekelin and Olovsson [14].

The first algorithms for SAT local search⁵ simply gave up when discovering that no assignment in the neighborhood had a lower cost than the current assignment, resorting to restart strategies or random walk (Section 3.4.2). Gent and Walsh showed [23] that for random satisfiability problems, the performance of local search degrades significantly when forbidding such non-improving transitions.

A transition that does not affect the cost is called a *flat transition*. The conventional way of handling plateaus is to pick a random flat transition, hoping that the sequence of randomized flat transitions will eventually allow the search to escape the plateau. The downside of this approach is that no guarantee that we will escape a bench can be given. Certain transitions will probably be searched numerous times, and no detection of inescapable plateaus is possible with this method. Therefore, flat transitions must be complemented with other techniques to be useful for traversing plateaus.

3.4.2 Randomization

It is generally agreed that randomization of certain parameters may help local search procedures overcome local minima. Stochastic behavior may be introduced in numerous ways, one of the most basic being to introduce *random restarts* in the search after a fixed number of transitions. Walser

⁵See Chapter 1 for an informal introduction to SAT.

[73], Selman et al. [63], Gent and Walsh [22, 23] and Gu et al. [28] take this approach.

Another common randomization strategy is to introduce *random walk* in the search. Random walk is the occasional random transition (or transitions) in the search space, the probability in each iteration for taking a random transition depending on a parameter typically supplied by the user of the search algorithm.

A third possibility is to change the neighborhood of an assignment according to a probabilistic distribution. In the well-known WalkSAT algorithm, introduced by Selman et al. [63], a successor assignment is selected by picking an unsatisfied clause at random, and from this clause selecting promising variables.

3.4.3 Exhaustive Plateau Search

One approach to escape plateaus may be to do a complete exploration of the plateau, as reported by Frank et al. [17] and Hampson and Kibler [29]. Thus, we can detect if the plateau is a bench and may be escaped at all. This also means that we are guaranteed to escape all benches.

Either the plateau can be searched depth-first or breadth-first. However, Hampson and Kibler [29] concludes that for random 3-SAT problems, the plateaus when the number of Boolean variable n were greater than 50 generally are too big to be searched exhaustively with breadth-first search, at least within reasonable time. To our knowledge, there has been no investigation if depth-first search is tractable for exploring large plateaus. Also, plateau search for general CSP problems may be intractable, simply because of the increased search space due to the in general non-Boolean domains of the variables.

3.4.4 Tabu Search

A common technique to diversify the search when traversing plateaus is to keep in memory the assignments already tried. This would give us a way of avoiding multiple visits to a single assignment. Glover and Laguna [25, 26] refer to this approach as *tabu search*, which is in fact a general search technique. Tabu search is closely related to another class of methods for search behavior learning called *nogood recording* [55].

Because the potential search space for a CSP can be huge, all transitions already traversed are not kept in memory. For example, a relatively

small problem with $n = 10$ variables, where each variable has a finite domain containing 20 values, has a search space of $20^{10} \approx 1.024 \cdot 10^{13}$ possible assignments. In this example, it is infeasible to keep even 1% of the possible assignments in memory. Instead, state memory is kept in a circular list of size t . Also, usually the inverse⁶ of the transitions done are stored instead of the actual assignments. This approach makes the inverse of the past t transitions forbidden. The parameter t is called the *tenure size* of the list. Such a list of forbidden transitions is called a *tabu list*. In general constraint-based local search, the data stored in the tabu list are variable-value pairs.

To avoid forbidding improving transitions that happen to be in the tabu list, we add an *aspiration criterion*: if a transition that is in the tabu list would yield an improvement, we disregard the fact that the transition is forbidden and select it anyway. This situation can for example arise if a sequence of transitions $a, b, \neg a$ is investigated, where $\neg a$ is the inverse of a and therefore tabu. Aspiration will occur if the result of these three transitions is a completely new state that, due to cost irregularity, is better than any previously investigated assignment.

Galinier and Hao [20] suggest using tabu search with constraint weighting and individual cost of each constraint (a measurement of how far the constraint is from being satisfied). Galinier [18] also reports results of using tabu search for MAX-CSP⁷.

3.4.5 Dynamic Weighting

One class of highly successful techniques to handle local minima and to some extent benches, especially for SAT, is based on the observation that some constraints (clauses for SAT) are violated more frequently than others in a plateau. The idea is to modify the cost function with added dynamic *weights* for each constraint. This approach was first introduced as a technique for SAT by Morris [44], and was further developed by Selman and Kautz [62], Frank et al. [16] and Wah et al. [65, 75, 76]. Dynamic weighting for general constraint satisfaction is mentioned in a paper by Galinier and Hao [20] in the context of tabu search. Note also that this method has many things in common with the *exponentiated*

⁶The purpose of Tabu search is to prevent the search from visiting a state already explored. Storing the inverse makes detection of transitions that may lead back to a visited state trivial

⁷In MAX-CSP, the goal is to satisfy as many constraints in a given CSP as possible.

subgradient method by Schuurmans et al. [61].

Modifying the extended cost function f^e from Definition 33 with dynamic constraint weights one yields the *dynamic weight cost function* f^w defined as follows.

Definition 41 (Dynamic Weight Cost Function). The cost function f^w defined on a set of constraints C is defined as

$$f^w(C, v, \Lambda) = \sum_{c \in C} \lambda_c w_c f_c(v)$$

where w_c is a constant weight, $f_c(v)$ is the constraint cost of c , and $\Lambda = \{\lambda_1, \lambda_2, \dots\}$ is a set of dynamic weights, each λ_c associated with constraint c .

The notation for the constraint weights is taken from the article by Shang and Wah [65], where the constraint weights are called *discrete Lagrangian multipliers* and a mathematical foundation for SAT local search based on clause weights is given. The constraint weights as presented here are a generalization of the SAT method used in [65] to handle general constraint satisfaction problems.

The idea of the dynamic constraint weight approach is to increase the weights on the constraints that remain unsatisfied at a plateau. This will eventually make an unsatisfied constraint c satisfied as the weight λ_c increases, thus forcing the search trajectory to leave the local minimum and continue the search.

The weights may be updated either at a plateau or after each iteration. Frank [16] concludes that updating of the weights at each iteration is the approach converging fastest for SAT. Contrary to this result, Shang and Wah [65] come to the conclusion that the weights should be updated only at plateaus and local minima.

The weights are either updated additively (as in the articles by Wu et al. [76, 74, 75]) or multiplicatively (as done by Schuurmans et al. [60, 61]). Also, to prevent Λ from becoming unbalanced, some normalization mechanism is necessary.

Chapter 4

A Model for Global Constraints

4.1 Introduction

In this chapter, we investigate a model usable to express the structural properties of global constraints, and how to use this to compute a cost. Historically, the most common approach at constraint-based local search has been to express all properties of a solution to the problem using a set of primitive constraints [73]. For example, to express that a set of n variables should all take disjoint values, the classical model is to use $n(n - 1)/2$ binary inequalities between all variables. This technique is possible for many global constraints, but for some more complex constraints like the family of cumulative scheduling constraints, this is not possible in practice, because of either a combinatorial explosion in the size of equivalent primitive constraints, or due to restrictions on the actual primitive constraints used.

Recently, the usability of local search constraint satisfaction has improved radically with the introduction of global constraints [45, 20]. In this work a separate cost representing the degree of violation for each global constraint is computed. The sum of the cost of the constraints in the model is then used to form the total cost of the problem.

The problem with this way of handling global constraints is twofold. First, the construction of a cost function for a global constraint is far from trivial, and requires knowledge of the implementation of the solver

in question. Second, the cost function of all constraints, global and primitive, must be comparable with each other, or else the solver efficiency will suffer from the resulting imbalance.

In this chapter we present a new approach at designing global constraints for local search. We define a global constraint using *filter constraints* (taking the inverse role of primitive constraints), which are applied to a graph of arcs and vertices, created from the constraint data and the current instantiation of the variables in the problem. The filter constraints correspond to a set of primitive constraints, and therefore provide global constraint costs based on a common basis. This makes designing constraints whose costs are compatible with each other an easier task.

4.1.1 Chapter Outline

The chapter is organized as follows. We first describe the model of global constraints as cost functions on structured networks of filter constraints, and give an overview of how the cost computation can be done from scratch. We use the global constraint `alldiff` as a running example for how we can model a global constraint using our approach. Next, we discuss how the conflicts of the variables in a constraint can be computed from scratch. We continue by demonstrating the flexibility and usability of our approach by modelling constraints from three important global constraint groups using the methods in this thesis. One of these families consist of global constraints for cumulative scheduling, which we discuss in more detail. As part of this we introduce a special subcost function, which can be used to reduce evaluation time for cumulative scheduling constraints.

4.2 A Model of Global Constraints

In this section we describe the model of global constraints using filter constraints on a graph, constructed from the constraint parameters.

The term *global constraint* was invented to name nontrivial constraints spanning a varying number of variables, such as the well-known global constraints *element*, *alldiff*, and *cumulative*. However, several global constraints cannot easily be directly represented using primitive constraints.

No formal definition of a global constraint exists and informally

speaking, any non-binary constraint can be considered a global constraint [3]. By tradition, global constraints are often associated with nontrivial constraints with specialized algorithms for domain reduction in a constraint programming environment. In a local search context however, a global constraint is a software component with the following characteristics:

- The constraint is dependent only on a set of input variables, a current state, and auxiliary information passed to the constraint at an initialization phase
- The constraint can on demand produce a *cost*, reflecting to which level the constraint is violated in the current state. A violation degree of zero is traditionally used for a satisfied constraint.
- The constraint can, given a variable present in the constraint, produce a *conflict level* for the variable. This level measures the number of conflicts in the constraint that the given variable is involved in. Another way to express this is that the conflict level indicates how much of the violation level of the constraint the variable causes in the current state.

Since constraint evaluation take place each iteration, the constraints should be optimized to compute their costs and conflict levels as fast as possible.

4.2.1 Constraint Representation

We represent a global constraint as a *directed graph* on which we post a *filter constraint*, and a *cost function* on the filtered graph, which computes a cost for the constraint. The goal is to have a constraint cost that is comparable with the cost of a set of primitive constraints. In general, we assume that if a constraint is satisfied, it has a cost of 0. Formally, we can describe a global constraint by the following four components:

1. One or more *vertex generators*, creating the vertices in the graph,
2. A *graph structure*, maintaining a set of arcs on the vertices,
3. A *filter constraint*, attached to the arcs in the graph, and
4. A *cost function* that computes the final cost and the conflict levels of the variables for the constraint.

This division of a global constraint into components gives us an expressive high-level model, which also can be implemented efficiently, as shown in Chapter 5. Beldiceanu pioneered the representation of a global constraint as a graph with certain properties in [4], where a classification of global constraints and their declarative meaning is given. Here we extend this framework for direct use in a constraint-based local search implementation, presented in Chapter 5. The main difference in the constraint model is that in this thesis, we use a cost function instead of the *graph properties* of [4] to express the global properties of the constraint. The cost function in our work computes a cost for the constraint, as opposed to the graph properties of [4], which expresses when the constraint is satisfied or violated. The concept of *dynamic global constraints* of [4] is not used in this thesis – instead, costs can be constructed using a *combined cost*, forming a cost from several *subcosts*. This is an extension of the dynamic properties used in [4], and is necessary to compute a complete cost for many global constraints.

We are also primarily interested in that the framework of this thesis can be efficiently implemented as a direct component usable to model global constraints. As a part of this we also investigate *incremental* cost and conflict computation of global constraints, two areas of great importance in a local search implementation.

4.2.2 Cost Computation

A naive cost computation from scratch for a constraint can be done as follows. The algorithm sketched below would be very inefficient if it was used in each iteration of the local search procedure. It is only used for initialization of the cost and conflict levels of a constraint. The algorithm used after initialization is incremental, and described in detail in Section 5.4.

1. Create the vertices of the graph from the input:

$$\begin{aligned} V_1 &= vg_1(a_1, a_2 \dots) \\ V_2 &= vg_2(b_1, b_2 \dots) \\ &\vdots \end{aligned}$$

where V_i are the resulting vertex vectors, vg_i are the vertex generators, and a_1, \dots, b_1, \dots are some of the arguments to the constraint. These can be domain variables, vectors, integers, pairs, etc.

2. Create the arcs on the vertices from step 1:

$$A = gs(V_1, V_2, \dots)$$

where A is an *arc set*, gs is the graph structure used, and V_1, V_2, \dots are the set of vertices used as input to the graph structure.

3. Filter the arc set by applying a filter constraint:

$$A' = [A[k] \mid k \in \{1, \dots, |A|\} \wedge fc(A[k], v) = \mathbf{true}]$$

where A' is a *final arc set*, fc is a function taking an arc and the current assignment v and returning a Boolean. Sometimes, the filter constraint is also dependent on other constant parameters as well. We do not show this here for simplicity.

4. Apply the cost function to the resulting final arc set:

$$c = cf(A')$$

where c is the final cost of the constraint and cf is a cost function.

We use a different notation for the cost function cf used in this chapter as a final step in computing the cost for a constraint, and the cost function f_c of chapter 3, associated with a global constraint c . The former take a final arc set as argument, while the latter take an assignment as argument.

Conflicts for the variables are also computed using the components for cost computation. Because this involves the actual components of the constraint to a higher degree, we describe the conflict computation later in Section 4.2.9.

4.2.3 Notation and Constraint Arguments

We use meta-variables to refer to subsets of vertices and arcs produced by the generators. As an example of our model, we use the global constraint `alldiff`, which ensures that a set of variables take disjoint values. We will use a model for the `alldiff` constraint based on the direct binarization of the constraint into inequalities. This is not the most efficient representation of the constraint, but serves the purpose of demonstrating how to model a constraint in the framework presented in this thesis. In Chapter 5, a more efficient model of cardinality constraints is investigated.

We specify the arguments of a global constraint as follows.

Constraint: `alldiff`
Arguments: `X : vec dvar`

For our `alldiff` constraint, the specification says that the constraint takes a vector (array) of domain variables as argument. In general, an argument to the global constraint is specified as $x : \mathfrak{t}$ where x is a variable and \mathfrak{t} is the type of x . In some cases we will use arguments with polymorphic types, specified as $x : t$. This means that t is a type variable for x , and denotes any possible type. The two basic types we use are integer domain variables, denoted as `dvar`, and integer constants denoted as `int`. We also have two constructed types; records, denoted as $\langle \mathbf{a} : t_1, \mathbf{b} : t_2, \dots \rangle$, and vectors, denoted as `vec t`. Here \mathbf{a} and \mathbf{b} are names used to refer to the attributes of the record, and $t, t_1, t_2 \dots$ refer to any constructed or basic type. We use the syntax $x.p$ to project the attribute named p from the argument x of record type.

4.2.4 Vertex Generators

The vertex generators produce vertices for the constraint network used to represent a global constraint. A vertex generator takes some of the arguments to the global constraint, and generates a set of vertices. Normally, the arguments are vectors of variables. Each global constraint has at least one vertex generator. The different vertex generators we use in this thesis are shown in Table 4.1. The `IDENTITY` vertex generator simply returns the argument as vertex vector. We will usually leave out the vertex generator specification if it is of this type, and use the argument directly as a reference to the vertices in the constraint.

The `ATTRIBUTE` vertex generator creates a vertex vector by extracting the given attribute with name p from all elements of the vector. The `EXPAND` vertex generator on a vector of type `vec t1` uses an additional function $M : t_1 \rightarrow \text{vec } t_2$ to generate a new vector of type `vec t2`, by applying M on all elements in the vector, and appending the results. The vertex generator `UNION` creates a vertex vector by applying the supplied function $M : t_1 \rightarrow \text{vec } t_2$ to all elements in the input vector, and appending only those elements that are not present in the already constructed vector. The effect is the same as to use `EXPAND`, and then remove duplicate elements, leaving the first occurrence.

In the specification of `alldiff`, we could use the vertex generators as follows.

Vertices: `V=IDENTITY(X)`

IDENTITY(L)	=	L
ATTRIBUTE($[a_1, \dots, a_n], p$)	=	$[a_1.p, \dots, a_n.p]$
EXPAND($[a_1, \dots, a_n], M$)	=	APPEND($M(a_1), \dots, M(a_n)$)
UNION($[a_1, \dots, a_n], M$)	=	APPEND($M(a_1), A_2, A_3, \dots, A_n$)
		where $A_2 = M(a_2) - M(a_1), A_3 = (M(a_3) - M(a_2)) - M(a_1), \dots$

Table 4.1: The vertex generators used to produce the vertices in the constraint network for a global constraint.

We can leave out the `IDENTITY` vertex generator in this case. We therefore use the identifier X to refer to the set of nodes in the continued specification of `alldiff`.

4.2.5 Graph Structures

Graph structures produce the directed arcs of the constraint network. In this thesis we use binary arcs only. The reason for this is that binary filter constraints are quite expressive and allow modelling of many global constraints [5]. A graph structure takes either one or two vectors of vertices, and produces a set of arcs between the vertices. The graph structures used in this work are described in Table 4.2. In an implementation of the framework described here, explicit representation of the arcs would be space and time inefficient. In Chapter 5, which presents an implementation based on the graph model of global constraints presented in this chapter, we use a more efficient implicit representation of arcs to decrease memory and time requirements.

As an example of using graph structures, we now continue the specification of the `alldiff` constraint by stating the arc generator to be used. X is the vector of vertices (and variables) generated using the vertex generator `IDENTITY` in Section 4.2.4.

Structure: $A = \text{CLIQUELT}(X)$

The specification above takes the vector X and uses the `CLIQUELT` graph structure to produce a set A of arcs between all tuples of variables, where the leaving vertex is present before the entering vertex in the vector X . For the `alldiff` constraint, if $X = [a, b, c, d]$, the generated set of arcs

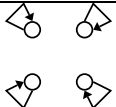
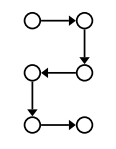
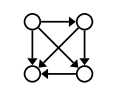
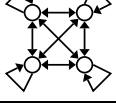
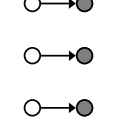
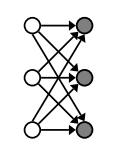
Name	Arcs generated	Example
LOOP	Arcs (x_i, x_i) on all vertices x_i .	
PATH	Arcs between all adjacent pairs (x_i, x_{i+1}) of vertices in the vector.	
CLIQUELT	Each vertex x_i in the vertex vector is connected to a vertex x_j in the vertex vector if and only if $i < j$.	
CLIQUE	The set of arcs (x_i, x_j) where x_i, x_j is in the vertex vector.	
PRODUCTEQ	Arcs between vertex x_i in the first vertex vector, and vertex w_i second vector. The vectors must have the same length.	
PRODUCT	Arcs from each vertex in the first vector to each vertex in the second vector.	

Table 4.2: The graph structures used to produce the arcs in the constraint network for a global constraint. Each graph structure takes either one or two vectors of vertices, and connects these using directed arcs.

A from `CLIQUELT` would be

$$A = \{(a, b), (a, c), (a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e), (d, e)\}.$$

This corresponds directly to the binarization of the `alldiff` constraint into inequalities.

4.2.6 Filter Constraints

A *filter constraint* is a primitive constraint that is applied on the arcs in the graph of the constraint. The roles of the filter constraints are to filter the set of arcs, as an intermediate step in computing the cost of the global constraint. We use general unary and binary constraints as filter constraints in this thesis, but in theory any arity for the filter constraints is of course possible. Unary constraints are posted on binary arcs leaving and entering the same vertex, and are represented as binary constraints. Formally, we define a filter constraint fc for a CSP (X, D, C) as a function $fc : A \times S(X) \rightarrow \text{bool}$, taking an arc from the arc set A as produced by the graph structure, and an assignment v from the state space $S(X)$, and returning a Boolean value.

In this thesis, a filter constraint is specified in terms of two vertex meta-variables, denoted as x_1 and x_2 , which specify the vertex the arc is leaving and entering respectively. We do this to be able to express the semantics of the filter constraint function fc . We specify the semantics using arithmetic, numerical relations, logical negation, conjunction, and disjunction. When posting a filter constraint, we also specify a source set of arcs, to which the filter constraint should be attached. This generalizes to filter constraints of higher arity, where the vertex placeholders are enumerated as x_1, x_2, x_3, \dots .

In the first part of this chapter, we will use equality filter constraints exclusively, corresponding to primitive inequality constraints. Later on when discussing scheduling constraints, we will form more complex intersection filter constraints, operating on tasks. As an example of a filter constraint, we denote an equality filter constraint on two variables simply as $v(x_1) = v(x_2)$.

We now continue with an example of the filter constraints in the specification of the `alldiff` constraint. Because we are interested in how many pairs of variables are equal, we express the filter constraints on the arcs using an equality constraint as follows.

Filter: $A' = \{A : v(x_1) = v(x_2)\}$

This specifies that binary equality constraints should be applied on all arcs in A . We now have an encapsulated binarization of the `alldiff` constraint, and the only thing left to do is to compute a cost on the filtered arcs.

4.2.7 Regular Cost Functions

The last component we need in order to specify a global constraint formally is the *cost* we compute from the final graph, obtained when filtering the arcs from the graph structure and the vertex generators. The cost of a constraint is formalized by a *cost function* cf of type $AS \rightarrow \text{int}$, where AS is the set of all possible arc sets. In many cases we can construct a cost function directly from a *property* p on the arc set, and a *cost modifier* cm . Such a cost is called a *regular cost*. The property $p : AS \rightarrow \text{int}$ samples a trait of an arc set (AS is the set of possible arc sets) and returns an integer, and the cost modifier cm is a function of type $cm : \text{int} \rightarrow \text{int}$, which is applied to the integer result of the property. The result of this application is returned as the cost of the constraint.

The properties we use in this thesis are the cardinality property $|A|$, computing the size of an arc set A , and a weighted sum property $\sum_{(i,j) \in A} w(i,j)$, computing a sum of constant weights for a set of arcs. In the implementation, we provide specialized versions of the weighted sum property to decrease execution time. The two special versions we provide compute weighted sums only dependent on the leaving and the entering vertices respectively.

A cost modifier can be any function of the correct type. To get a cost that is comparable with other constraint costs, one must however select the cost modifier carefully. In our experiments we have used mostly threshold cost modifiers of the type $cm(x) = \max(0, x - k)$ for a constant k . The linear correspondence between the property value and the cost makes the cost closely related to the value of the property of the final arc set. This is important if we want to have a cost that is comparable with costs of other constraints. If the cost modifier is the identity function, i.e. $cm(x) = x$, we will leave this out and use the property directly to express the cost.

Now again consider the `alldiff` global constraint. We want to

express that the cost of the constraint is the number of violated binary inequalities. We can do this using the following specification.

$$\text{Cost: } |A'|$$

Here we simply use the cardinality of the set of final arcs A' , corresponding to how many equalities are satisfied – this is of course the same as counting the number of inequalities.

To conclude our example, the global constraint `alldiff` restricts a set of variables from taking the same values. We can express an `alldiff` constraint on n variables semantically with $n(n-1)/2$ binary inequalities, restricting each possible pair of unique variables from taking the same value. Using this observation, we base our model (shown in Table 4.3) of the `alldiff` constraint on binary equalities between the pairs of variables in the constraint.

Constraint:	<code>alldiff</code>
Arguments:	$X : \text{vec dvar}$
Structure:	$A = \text{CLIQUELT}(X)$
Filter:	$A' = \{A : v(x_1) = v(x_2)\}$
Cost:	$ A' $

Table 4.3: The `alldiff` constraint.

Since we do not specify a vertex generator, the `IDENTITY` vertex generator is implicitly used, constructing vertices directly from the arguments (variables) given to the constraint. The `CLIQUELT` graph structure gives us arcs between each pair of unique vertices. We use an equality constraint as the filter constraint, because we want to count the number of equal variable pairs to see if the constraint is violated or satisfied; if the number of satisfied equalities is zero, then the global `alldiff` constraint is satisfied, and if the constraint is violated, the cost for the constraint should be the number of equal variable pairs. We can express this using an `identity` cost function on the cardinality property of the set of filtered equality arcs.

4.2.8 Combined Costs and Subcosts

For several global constraints, the regular cost of the previous section is not generic enough to express a suitable cost. One example is the

gcc constraint¹ of [53]. This constraint takes a set of variables X and two sets of integers L, U denoting lower and upper capacity. Each lower and upper capacity is associated to a unique value in the union of the domains of the variables in X . The gcc constraint is satisfied when, for all values a , the number of variables taking a value a is in the interval (L_a, U_a) :

$$\forall a \in \bigcup_{x \in X} \text{DOM}(x). L_a \leq |\{x \mid x \in X \mid v(x) = a\}| \leq U[a]$$

An obvious way to compute a cost for this constraint is to accumulate several costs on the individual values, and then use the sum of these costs as the final cost of the constraint. In several other constraints as well, there is a need to be able to partition the cost computation into several smaller cost computations.

To remedy this situation, we introduce *combined costs* f_Σ and *subcosts*. A combined cost consists of a *partition function* pf and a subcost function f . The partition function pf partitions a final arc set A' into a vector of arc subsets $pf(A')$. The maximal size of this vector must be statically computable, and each arc set must have a fixed location in the vector. We can then apply the subcost function f on each arc set in $pf(A')$, and take the sum of the subcosts as the total cost of the constraint:

$$f_\Sigma(A') = \sum_{p \in pf(A')} f(p).$$

If we need to use the index of an arc subset, we will use the notation $\sum_{p_i \in pf(A')} f(p_i)$ where i is the index of the arc subset p_i . The subcost f is a normal cost function, and is therefore often regular and constructed using a property and a cost modifier. In the specification of a constraint, the notation to use a combined cost is shown below, with an example from the gcc constraint. In the specification of the cost for the constraint, we use A to refer to the final arc set.

$$\text{Partition : } \sum_{p_i \in \text{ENTERING}(A')} f(p_i)$$

In general, we are not forced to use addition to combine subcosts, and can use other operators that are associative, commutative and invertible. These properties are necessary for the incremental computation

¹gcc is an abbreviation of *Global Cardinality Constraint*.

to be efficient [77]. In this thesis we only use addition to combine sub-costs, although other operators, such as maximum and product, can certainly be useful. Note that this means that we use a similar notation for combined costs using addition, and weighted sum properties – their meaning should be clear from context.

The two partition functions that we use in this thesis are shown in Table 4.4. The ENTERING and LEAVING partition functions generate the sets of arcs that all enter/leave the same node x_i . This is usable in situations like the one described for the cardinality family of constraints above. We call the vertex that the arcs in an entering arc subset enters the *focus vertex* of the arc subset, and vice versa for the LEAVING partition function.

Partition Functions	
ENTERING[i]	$= \{(y, X[i] \mid (y, X[i]) \in A') \text{ for } i \in \{1, \dots, X \}\}$
LEAVING[i]	$= \{(X[i], y \mid (X[i], y) \in A') \text{ for } i \in \{1, \dots, X \}\}$

Table 4.4: Partition functions used to partition final arc sets A' for subcost computation. X is the vertex vector.

As an example of a constraint using subcosts, see the `gcc` constraint, described in Section 4.3.2.

4.2.9 Conflict Computation

In this section we take a closer look on how the conflicts $CL_c(v, x)$ on a variable x for a constraint c can be computed. The conflicts have to be collected for all constraints in the problem, to form the total conflict level of a variable. This is done using addition, as shown in Section 5.4.1.

The conflict level of a variable present in a constraint is computed at the same time as the cost of the constraint. The basic idea for this computation is to distribute the cost over the vertices in the final arc set to form the conflict level of a vertex. The cost is distributed to the vertices in the graph in the computation of the cost function, and the cost of a vertex is then in turn distributed equally on the variables that are used to form the vertex. In the common case where a vertex *is* a variable, this distribution is of course trivial. In the case where a vertex is a record, the full cost of the vertex is distributed to each element in

the record. Another way to do this would be to divide the cost by the total number of elements in the record before distribution.

The conflict level is computed differently depending on if the cost used for the constraint is *regular*, *combined*, or *specialized*. In the case of a specialized cost function, this function is responsible for computing both the cost and the conflict levels of the constraint.

Conflict Level on Regular Costs

For a constraint c with a cost function that is *regular*, we distribute the cost on the vertices present in the final arc set to form the conflicts. Computing the conflicts from scratch can be done as shown below.

1. The cost $cf(A')$ of the constraint is computed for the final arc set A' as before.
2. For each vertex x , the conflict level as contributed by the constraint c is computed as $CL_c(v, x) = cf(A') \frac{|A''|}{|A'|}$, where $A'' = \{a \mid a \in A' \wedge (a = (x, y) \vee a = (y, x))\}$.

For the `alldiff` constraint, if the final arc set for an assignment v and a vertex list of $[a, b, c, d, e, f]$ is

$$A' = \{(a, b), (a, c), (b, c), (d, e)\},$$

the cost f as computed by the cardinality property would be 4. This cost would then be distributed on the vertices as

$$\begin{aligned} CL_c(v, a) &= 4 \cdot 2/4 = 2 \\ CL_c(v, b) &= 4 \cdot 2/4 = 2 \\ CL_c(v, c) &= 4 \cdot 2/4 = 2 \\ CL_c(v, d) &= 4 \cdot 1/4 = 1 \\ CL_c(v, e) &= 4 \cdot 1/4 = 1 \\ CL_c(v, f) &= 4 \cdot 0/4 = 0 \end{aligned}$$

This corresponds exactly to the number of violated binary inequalities on a variable.

Conflict Level on Combined Costs

The conflict level $CL_c(v, x)$ of a variable x for a constraint c using a combined cost is computed differently. For such a constraint, the cost is distributed on the vertices present in the final arc set as shown below.

1. The partition function pf partitions the final arc set A' into a set of arc sets $pf(A')$.
2. The subcost f is applied on each arc set $p \in pf(A')$.
3. In the computation of f , the contributed subconflict level $CL_p(v, x)$ of x becomes available. $CL_p(v, x)$ is computed as for any normal cost.
4. For each vertex x , the conflict level $CL_c(v, x)$ on x as contributed by the constraint c is computed by taking the sum of the subconflicts levels for x :

$$CL_c(v, x) = \sum_{p \in pf(A')} CL_p(v, x)$$

4.3 Capacity Constraints

In this section we take a closer look on some global constraints used to compute capacity and cardinality.

4.3.1 The capa Constraint

The `capa` constraint [20] takes a vector of variables X and a vector of integers W with equal size, and two integers a and k . The constraint enforces that the sum of the weights of the variables taking the value a is less than or equal to k .

$$\sum_{i: v(X[i])=a} W[i] \leq k$$

We can state the constraint as a loop graph with the unary constraint $v(x_1) = a$. We use the `LOOP` graph structure to do this, and use a cost of the sum of the weights of the variables, reduced by k . The formulation is shown in Table 4.5.

Constraint:	capa
Arguments:	$X : \text{vec dvar}$ $W : \text{vec int}$ $a, k : \text{int}$
Structure:	$A = \text{LOOP}(X)$
Filter:	$A' = \{A : v(x_1) = a\}$
Cost:	$\max(0, (\sum_{(X[i], X[i]) \in A'} W[i]) - k)$

Table 4.5: The capa constraint.

Constraint:	gcc
Arguments:	$X : \text{vec dvar}$ $L : \text{vec int}$ $U : \text{vec int}$
Vertices:	$X = \text{IDENTITY}(X)$ $D = \text{UNION}(X, \text{DOM})$
Structure:	$A = \text{PRODUCT}(X, D)$
Filter:	$A' = \{A : v(x_1) = x_2\}$
Partition:	$\sum_{p_i \in \text{ENTERING}(A')} f(p_i)$
Subcost:	$f(p_i) = \max(L[i] - p_i , p_i - U[i])$

Table 4.6: The gcc constraint.

4.3.2 The gcc Constraint

The gcc constraint [53], where gcc is an abbreviation of *Global Cardinality Constraint*, takes a vector of domain variables X and two vectors of integers L, U of same size as the number of elements in the union of the domains of X . In our implementation, this union must form a closed interval of integers. The two vectors L, U contains lower and upper bounds for the values that can be assigned to the variables in X . The gcc constraint is satisfied when, for all values a , the number of variables taking a value a is in the interval (L_a, U_a) :

$$\forall a \in \bigcup_{i \in \{1, \dots, |X|\}} \text{DOM}(X[i]). L[a-l] \leq e(a) \leq U[a-l],$$

where v is an assignment, $e(a)$ is the number of variables taking the value a , $e(a) = |\{i \mid i \in \{1, \dots, |X|\} \wedge v(X[i]) = a\}|$, and we find a index $a-l$ for L and U by normalization with $l = \min(\bigcup_{i \in \{1, \dots, |X|\}} \text{DOM}(X[i])) + 1$.

The gcc constraint is specified in our graph model in Table 4.6. For a violated constraint, we use a subcost for each value that is proportional to the distance from the lower or upper bound of the constraint. We create the total cost by using a combined cost computing the sum of the ENTERING subcosts.

4.3.3 The wcc Constraint

The wcc constraint is a version of the gcc constraint from Section 4.3.2 where we limit the sum of the weights of the arcs for each value. This is also a generalization of the capa constraint. The constraint takes three vectors X, W, K as input; X contains domain variables, X and W has the same size, and K must have the same size as the number of elements in the union of the domains of X , which must form a closed interval of integers. K contains capacities for the values of the variables, and W contains vectors of integers, which have the same size as K , containing weights for the arcs of the constraint.

The constraint ensures that for each value a , the sum of the weights $W[i][a]$ of the variables x_i taking the value a is less than or equal to $K[a]$:

$$\forall a_j \in \bigcup_{x \in X} \text{DOM}(x). \sum_{x_i \in X \mid v(x_i) = a} W[i][j] \leq K[j],$$

where i, j are sequence numbers of the variables and the values respectively – j this the ordinal number in the closed interval formed from the domains of the variables.

Constraint:	wcc
Arguments:	$X : \text{vec dvar}$ $W : \text{vec vec int}$ $K : \text{vec int}$
Vertices:	$X = \text{IDENTITY}(X)$ $D = \text{UNION}(X, \text{DOM})$
Structure:	$A = \text{PRODUCT}(X, D)$
Filter:	$A' = \{A : v(x_1) = x_2\}$
Partition:	$\sum_{p_j \in \text{ENTERING}(A')} f(p_j)$
Subcost:	$f(p_j) = \max(0, (\sum_{(x_i, a_j) \in p_i} W[i][j]) - K[j])$

Table 4.7: The wcc constraint.

4.3.4 The nbdiff Constraint

The nbdiff constraint takes two vectors with equal size X, Y of variables and an integer k as arguments, and check that the number of variables with the same index taking the same value is less than or equal to k .

$$|\{i \mid v(X[i]) = v(Y[i])\}| \leq k$$

The name `nbdifferences` (abbreviated as `nbdiff`) for the constraint was introduced in [20] and is a bit misleading – the constraint actually makes sure that the number of *equalities* is less than or equal to k . To make the name of the constraint more natural, the meaning can be reformulated as

$$|\{i \mid v(X[i]) \neq v(Y[i])\}| \geq |X| - k.$$

We can represent the constraint as a graph with the variables as vertices, and arcs with an associated equality constraint $v(x_1) = v(x_2)$ between the tuples. We use the `IDENTITY` vertex generator, and the `PRODUCTEQ` graph structure to form the graph.

Constraint:	<code>nbdiff</code>
Arguments:	<code>X : vec dvar</code> <code>Y : vec dvar, k : int</code>
Structure:	<code>A = PRODUCTEQ(X, Y)</code>
Filter:	<code>A' = {A : v(x₁) = v(x₂)}</code>
Cost:	<code>max(0, A' - k)</code>

Table 4.8: The `nbdiff` constraint.

4.4 Non-Overlapping Scheduling Constraints

In this section we investigate some global constraints useful for non-overlapping, non-preemptive resource scheduling. The basic unit in scheduling is the *task*, which models an activity with an integer start time and a positive integer duration. To handle tasks we introduce the data type `task` as an alias for a record of type $\langle s : \text{dvar}, d : \text{dvar} \rangle$ where for a task x , $x.s$ denote the start time and $x.d$ denote the end time.

To express the different resource constraints used for scheduling and cumulative constraints, we will invent new filter constraints capturing these properties. The most basic filter constraint for scheduling is the `OVERLAP` constraint, equivalent to the temporal overlapping relation between two tasks x and y .

$$\text{OVERLAP}(x, y) \equiv v(x.s) < v(y.s) + v(y.d) \wedge v(y.s) < v(x.s) + v(x.d)$$

This filter constraint is used primarily for non-overlapping scheduling. We will consider other types of scheduling filter constraints in Section 4.5.

4.4.1 The *ordered-tasks* Constraint

The `ordered-tasks` constraint on a set of n tasks handles a set of $n - 1$ binary precedence relations between tasks on the form $v(X[i].s) + v(X[i].d) + K[i] \leq v(X[i + 1].s)$, where the vector K contains integers, corresponding to the setup time between the tasks. The constraint takes a vector of records $T : \text{vec } \langle t : \text{task}, k : \text{int} \rangle$ as argument. For an element in T , $T[i].t$ denote a task and $T[i].k$ denote the setup time between tasks. We handle this constraint by building a path including all vertices using the `PATH` graph structure. The vertices are produced

using the `IDENTITY` vertex generator. We then use a special-made filter constraint, returning the logical negation of the precedence relation above. We use a regular cost function; the identity cost modifier on the cardinality property.

Constraint:	<code>ordered-tasks</code>
Arguments:	$T : \text{vec } \langle t : \text{task}, k : \text{int} \rangle$
Structure:	$A = \text{PATH}(T)$
Filter:	$A' = \{A : v(x_1.t.s) + v(x_1.t.d) + x_1.k > v(x_2.t.s)\}$
Cost:	$ A' $

Table 4.9: The `ordered-tasks` constraint.

4.4.2 The *serialized* Constraint

The `serialized` global constraint ensures that the tasks in a vector do not overlap in time. A `serialized` constraint on n tasks is semantically equivalent to $n(n-1)/2$ non-overlapping constraints, restricting all possible pairs of tasks from overlapping. We use a negated variant of this as the basis for our model of the `serialized` constraint.

Constraint:	<code>serialized</code>
Arguments:	$T : \text{vec } \text{task}$
Structure:	$A = \text{CLIQUELT}(T)$
Filter:	$A' = \{A : \text{OVERLAP}(x_1, x_2)\}$
Cost:	$ A' $

Table 4.10: The `serialized` constraint.

The representation of `serialized` shown in Table 4.10, is similar to the one used for the `alldiff` constraint. We use the filter constraint `OVERLAP` on the arcs. As vertices of the constraint graph, we use the tasks themselves, using the `IDENTITY` vertex generator. Again, we use the `CLIQUELT` graph structure, and apply the `OVERLAP` filter constraint on the arcs. The rest of the formulation of `serialized` is in analogy with the model for `alldiff`.

In Section 4.5.1 we will investigate yet another possible way to model the `serialized` constraint in our framework.

4.4.3 The disjoint-tasks Constraint

The `disjoint-tasks` constraint takes two vectors of tasks, and ensures that no task in the first vector intersects with a task in the second vector. The formulation of the constraint as graph properties on a structured network is straightforward and shown in Table 4.11. We use the `IDENTITY` vertex generator to produce one vertex for each task. The graph structure used is `PRODUCT`, which produces an arc between each vertex in the first vector and each vertex in the second vector. We again use `OVERLAP` as the filter constraint, and count the number of overlaps as a measure on the cost of the constraint. The `disjoint-tasks` constraint is a variant of the `serialized` constraint [5].

Constraint:	<code>disjoint-tasks</code>
Arguments:	$T_1 : \text{vec task}$ $T_2 : \text{vec task}$
Structure:	$A = \text{PRODUCT}(T_1, T_2)$
Filter:	$A' = \{A : \text{OVERLAP}(x_1, x_2)\}$
Cost:	$ A' $

Table 4.11: The `disjoint-tasks` constraint.

4.5 Cumulative Scheduling Constraints

In this section we take a closer look on a set of global constraints commonly referred to as *cumulative constraints*. In these constraints, we are interested in restricting the number of tasks executing simultaneously. Formally, we associate a nonnegative integer height $x.h$ to each task x , and for all points in time t restrict the sum of the heights of the tasks X_t executing at t to be in an interval $\sum_{x \in X_t} x.h \leq k$, where k is the upper capacity bound. This is a generalization of a `serialized` constraint, for which the upper bound is 1, and all heights are 1.

To handle tasks with height we use the data type `htask`, an alias for a record of type $\langle s : \text{dvar}, d : \text{dvar}, h : \text{int} \rangle$. We will use the same notation as for vertices of type `task`, and also refer to the height of a task x as $x.h$.

4.5.1 A Filter Constraint for Cumulative Scheduling

The filter constraint OVERLAP is severely limited and is not general enough to handle more generic cumulative resource constraints. This is partially due to the fact that OVERLAP is a binary constraint on two tasks only and cannot handle more tasks combined with a resource which does not have an upper capacity bound of 1.

One possible solution for this is to introduce a filter which tests if a task is active at a certain time point. The only time points where the resource usage in a schedule will change are at the start and end times of the tasks. These can be considered the *critical time points* of resource schedules. The idea is captured by the START-ACTIVE filter, which is shown below. The START-ACTIVE filter constraint between two tasks is true whenever the second task is active at the time point when the first task becomes active.

$$\text{START-ACTIVE}(t, y) \equiv t \geq v(y.s) \wedge t < v(y.s) + v(y.d)$$

One problem with the approach above is that it introduces symmetrical filtered arcs between tasks due to the ordered nature of the arcs – the imaginary edges we want to count are unordered. We can remove such symmetries by using a total order on the tasks. Some of the symmetries are actually removed by using the partial order that the start times impose on the tasks. We show later how we can remove all symmetries by extending this order.

In the following discussion we restrict our attention to upper-bounded capacitated scheduling constraints with nonnegative heights. Given these assumptions, the start time points of the tasks are the only time points where the resource usage increases. For capacitated scheduling with upper bounds only, it is enough to investigate these to find out when the resource usage is higher than the available capacity.

We can now use START-ACTIVE as a filter, in order to form the sets of arcs, which are intersecting with the start time points of a tasks. The basic idea here is to compute, for each task start, the set of binary task collisions that is directly dependent on the start of the task. On these sets, we can then apply different cost modifiers removing the cost of the allowed number of filtered arcs, and capturing the violation of the constraint. This can be done using a connected graph on the tasks, with a combined cost on the leaving arc sets of the START-ACTIVE filter constraint. For the `serialized` constraint, this would correspond to

the following cost, where A' is the final arc set, essentially allowing the execution of at most one single task at each critical time point.

$$\sum_{p_i \in \text{LEAVING}(A')} \max(0, |p_i| - 1)$$

The basic idea is that each new task that becomes active corresponds to n binary task overlaps, where n is the number of tasks active at the start of the new task. We then subtract from n the number of allowed tasks at the time point.

4.5.2 Removing Symmetrical Arcs

If we use the START-ACTIVE filter constraint as above, we get a cost that is not directly comparable with primitive binary constraints for tasks that start at exactly the same time. This is because the time ordering is partial and cannot remove all symmetries – if n tasks start at the exact same time, we get n identical sets of leaving arcs.

As an example of this, consider the two situations shown in Figure 4.1. We have a vector of tasks $T = [TA, TB, TC]$. The situation to the left yields the following partition of the final arc set into a vector of leaving arc sets.

$$\begin{aligned} \text{LEAVING}[1] &= \{(TA, TA)\} \\ \text{LEAVING}[2] &= \{(TB, TA), (TB, TB)\} \\ \text{LEAVING}[3] &= \{(TC, TA), (TC, TB), (TC, TC)\} \end{aligned}$$

The cost expression for the `serialized` constraint results in a cost of $1 + 2 + 3 - 3 = 3$. In contrast, the situation to the right in Figure 4.1, which should be considered equal since the numbers of binary overlaps are the same, gives us the following set of leaving arc sets.

$$\begin{aligned} \text{LEAVING}[1] &= \{(TA, TA), (TA, TB), (TA, TC)\} \\ \text{LEAVING}[2] &= \{(TB, TA), (TB, TB), (TB, TC)\} \\ \text{LEAVING}[3] &= \{(TC, TA), (TC, TB), (TC, TC)\} \end{aligned}$$

The cost resulting from this situation is $3 + 3 + 3 - 3 = 6$, which is *not* equal to the number of binary overlaps. This is because the partial start time order does not break symmetries between tasks starting at the same

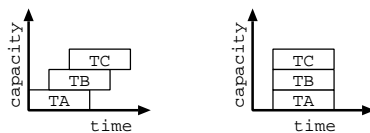


Figure 4.1: Two equivalent schedules which yield different cost unless corrected.

time – for example, the pairs (TA, TB) and (TB, TA) both correspond to the overlap $\{TA, TB\}$.

To remove the remaining symmetries we impose an arbitrary order \preceq on the tasks with equal start time, and include a task y with start time equal to the leaving task x in the leaving arc set if and only if $y \preceq x$. In our implementation we take this order from the order of the task vector, as generated by the vertex generator. This order can be determined statically for a task vector.

To incorporate the order in the framework we define the extended filter constraint $\text{START-ACTIVE}_{\preceq}$ as shown below.

$$\begin{aligned} \text{START-ACTIVE}_{\preceq}(x, y) &\equiv \text{STRICT-START}(v(x.s), y) \\ &\vee \text{EQUAL-START}(v(x.s), y) \wedge y \preceq x \end{aligned}$$

The new filter constraint first breaks symmetries using the STRICT-START filter:

$$\text{STRICT-START}(t, y) \equiv t > v(y.s) \wedge t < v(y.s) + v(y.d)$$

If the start times of the two tasks are equal,

$$\text{EQUAL-START}(t, y) \equiv t = v(y.s) \wedge t < v(y.s) + v(y.d),$$

we instead use the new ordering relation to break the remaining symmetries.

The new `serialized` constraint `alt-serialized` is formulated in Table 4.12.

4.5.3 The *cumulative-1* Constraint

In this section, we generalize the `serialized` constraint to the restricted k -capacity `cumulative-1`, where the

Constraint:	<code>alt-serialized</code>
Arguments:	$T : \text{vec task}$
Structure:	$A = \text{CLIQUE}(T)$
Filter:	$A' = \{A : \text{START-ACTIVE}_T(x_1, x_2)\}$
Partition:	$\sum_{p \in \text{LEAVING}(A')} f(p)$
Subcost:	$f(p) = \max(0, p - 1)$

Table 4.12: The `alt-serialized` constraint.

resource can handle at most k tasks at the same time. The semantics of the constraint prevent more than k tasks from executing at each time point.

Note again that if some tasks are active at a single time point t , they are also intersecting with each other. This allows us to state the semantics of `cumulative-1` as follows.

$$\begin{aligned} \text{cumulative-1}(T, k) \equiv \\ \forall s \in T. |\{t | t \in T \wedge \text{START-ACTIVE}_{\preceq}(s, t)\}| \leq k \end{aligned}$$

This ensures that for each critical time point, the number of overlapping tasks should be less than or equal to k . The total order \preceq is as before arbitrary and of no consequence for the semantics of the constraint – we use it only to break ties between tasks starting at the exact same time point.

The `cumulative-1` constraint is formulated using graph properties in Table 4.13. We use a cost for `cumulative-1` based on the number of tasks which are active at the start time of a tasks, which we then reduce by the number of active tasks that are allowed. This cost is similar to the one used for `alt-serialized` in the previous section. We produce the vertices of the constraint using the implicit `IDENTITY` vertex generator, and use the `CLIQUE` graph structure to get a graph with arcs between any pair (i, j) of vertices. We apply the `START-ACTIVE` filter constraint with the order from V on the arcs, and use a combined cost on the `LEAVING` arc subsets. We use a threshold cost on the cardinality of these arc subsets, reduced with at most k allowed active tasks, to form the total cost.

The cost of a `cumulative-1` constraint is comparable with the cost of a `serialized` constraint. This is because we once more count the number of binary task overlaps. When the capacity of the constraint k

Constraint:	<code>cumulative-1</code>
Arguments:	$T : \text{vec task}, k : \text{int}$
Structure:	$A = \text{CLIQUE}(T)$
Filter:	$A' = \{A : \text{START-ACTIVE}_T(x_1, x_2)\}$
Partition:	$\sum_{p \in \text{LEAVING}(A')} f(p)$
Subcost:	$f(p) = \max(0, p - k)$

Table 4.13: The `cumulative-1` constraint.

is greater than zero, the cost is reduced so that for any time point, k active tasks count only as one when computing overlaps. This is best demonstrated with an example.

We use a `cumulative-1` constraint with a capacity of $k = 2$ on a vector of tasks $T = [A, B, C, D, E, F]$. We use the `IDENTITY` vertex generator and get the vertex vector $V = [A, B, C, D, E, F]$. The graph structure `CLIQUE` generates a set of arcs $A = \{(x, y) | x, y \in V\}$.

On A we apply the binary constraint `START-ACTIVET(x_1, x_2)`, giving us the set of final arcs A' . Now suppose we have the total assignment of tasks with the following start- and end times.

Task	A	B	C	D	E	F
START	0	1	4	8	8	9
END	7	3	9	13	15	13

The execution profile of these tasks is shown to the left in Figure 4.2. To the right, a corresponding execution profile for a `serialized` constraint is shown. Note that the two costs should be the same. The application of the partition function `LEAVING(A')` on the set of final arcs A' yields the following vector of arc subsets.

$$\begin{aligned}
 \text{LEAVING}[1] &= \{(A, A)\} \\
 \text{LEAVING}[2] &= \{(B, A), (B, B)\} \\
 \text{LEAVING}[3] &= \{(C, A), (C, C)\} \\
 \text{LEAVING}[4] &= \{(D, C), (D, D)\} \\
 \text{LEAVING}[5] &= \{(E, C), (E, D), (E, E)\} \\
 \text{LEAVING}[6] &= \{(F, D), (F, E), (F, F)\}
 \end{aligned}$$



Figure 4.2: The number of binary overlaps of the infeasible schedule for a cumulative-1 constraint with capacity 2 is equal to the cost of the serialized constraint on the right, where some tasks have been merged.

Applying the subcost function $cf(p) = \max(0, |p| - k)$, where $k = 2$, on all arc subsets p_i yields

$$\begin{aligned}
 f(p_1) &= \max(0, |\{(A, A)\}| - 2) = 0 \\
 f(p_2) &= \max(0, |\{(B, A), (B, B)\}| - 2) = 0 \\
 f(p_3) &= \max(0, |\{(C, A), (C, C)\}| - 2) = 0 \\
 f(p_4) &= \max(0, |\{(D, C), (D, D)\}| - 2) = 0 \\
 f(p_5) &= \max(0, |\{(E, C), (E, D), (E, E)\}| - 2) = 1 \\
 f(p_6) &= \max(0, |\{(F, D), (F, E), (F, F)\}| - 2) = 1
 \end{aligned}$$

The total cost of the cumulative-1 constraint thus becomes $1+1 = 2$, which is the same as the one for a serialized constraint and the tasks shown to the right in Figure 4.2.

4.5.4 The cumulative Constraint

The cumulative constraint is a generalization of the cumulative-1 constraint, where we associate a nonnegative integer *height* $x.h$ with each task x . The constraint ensures for the given vector of tasks that for any point in time, the sum of the heights of the tasks executing at that time is less than or equal to the capacity k of the constraint.

A straightforward way to handle the cumulative constraint is to split each task x with a height greater than 1 into $x.h$ different tasks of height 1 with equal start and end times. The task splitting makes it possible to treat the constraint exactly as we did with the cumulative-1 constraint. In Figure 4.3 this approach is shown by an example.

The formulation of the cumulative constraint using this approach

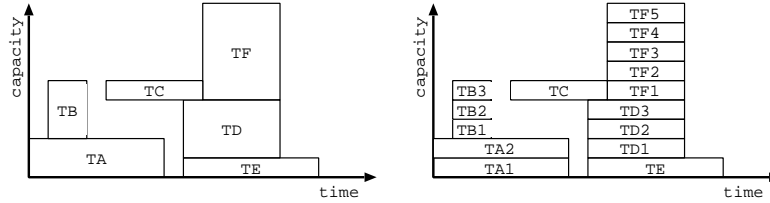


Figure 4.3: Splitting of tasks into tasks with height 1.

is shown in Table 4.14. Here, we use a similar approach as we did for the `cumulative-1` constraint. The difference is that we use tasks of type `htask` with an associated height, and that we use the `EXPAND` vertex generator to produce $x.h$ vertices for a task x using the `split` function shown below.

$$\text{SPLIT}(x : \langle s : \text{dvar}, d : \text{dvar}, h : \text{int} \rangle) = [\langle x.s, x.d, 1 \rangle, \dots, \langle x.s, x.d, 1 \rangle]$$

where $|\text{SPLIT}(x)| = x.h$.

Constraint:	<code>split-cumulative</code>
Arguments:	$T : \text{vec } \text{htask}, k : \text{int}$
Vertices:	$V = \text{EXPAND}(T, \text{SPLIT})$
Structure:	$A = \text{CLIQUE}(V)$
Filter:	$A' = \{A : \text{START-ACTIVE}_V(x_1, x_2)\}$
Partition:	$\sum_{p \in \text{LEAVING}(A')} f(p)$
Subcost:	$f(p) = \max(0, p - k)$

Table 4.14: The `split-cumulative` constraint.

4.5.5 Implicit Task Splitting

The downside of the expansion of tasks shown in the previous section is that the number of tasks we have to handle increase significantly. This can in turn have severe effects on performance. Therefore, we propose to handle the splitting of tasks *implicitly* by computing the numerical result of this operation. We do this by the specialized cost function `VSPLIT`.

This cost function takes a leaving arc subset on vertices of type `htask` and the maximum capacity k of the constraint as an extra parameter, and computes the cost of the corresponding split task set for the task with is focus vertex of the arc subset.

To see what cost `VSPLIT` computes, we first assume that we use the previous model of `cumulative` and that we have expanded the vertices (of tasks) V in question into a vector V' of *virtual* tasks, as in the previous section. As before we use the order from the vector of tasks to remove symmetries from the cost computations. We now have the following. For each task $t_i \in T$ with a height of $t_i.h$, we have $t_i.h$ virtual tasks $t_i^1, t_i^2, \dots, t_i^{t_i.h}$, each with a height $t_i^j.h$ of 1. We assume that the virtual tasks are ordered as $t_i^1 \preceq t_i^2 \preceq \dots \preceq t_i^{t_i.h}$, and also that if $t_i \preceq t_j$ then $\forall q \in \{1, \dots, t_i.h\}, r \in \{1, \dots, t_j.h\}. t_i^q \preceq t_j^r$ for all tasks t_i, t_j , that is, the order of the source tasks t_i, t_j extends to the virtual tasks as well. This order can be determined statically from the orders of the vectors that `SPLIT` produces, since $t.h$ is constant.

In the following discussion we assume that we use the same structure and cost as for the `split-cumulative` constraint. Each virtual task t_i^j is now the focus of a leaving arc subset $p_i^j = \text{LEAVING}(A')[\text{ord}(t_i^j)]$, where A' is the final arc set and $\text{ord}(t_i^j)$ denote the ordinal in \preceq for t_i^j , used for indexing in the vector of arc subsets. The cost computed for a task t_i should then be $\sum_{j=1, \dots, t_i.h} f(p_i^j)$ where f is the subcost function of the constraint. For the `cumulative` constraint we have that the subcost is $f(p) = \max(0, |p| - k)$. This gives us that the total cost for one task t_i should be

$$\sum_{j=1, \dots, t_i.h} \max(0, |p_i^j| - k).$$

Now let T^0 be the set of virtual tasks, except those expanded from t_i , which are active at t_i :s starting time point.

$$\begin{aligned} AT &= \{t_j \mid t_j \in T \setminus \{t_i\} \wedge \text{START-ACTIVE}_{\preceq}(t_i, t_j)\} \\ T^0 &= \text{EXPAND}(AT, \text{SPLIT}) \end{aligned}$$

According to our ordering \preceq we now have the following, where T^j is the set of tasks which are active and ordered before t_i^j when considering

virtual task t_i^j .

$$\begin{aligned}
T^1 &= T^0 \cup \{t_i^1\} \\
p_i^1 &= \{(t_i^1, x) \mid x \in T^1\} \\
T^2 &= T^1 \cup \{t_i^2\} \\
p_i^2 &= \{(t_i^2, x) \mid x \in T^2\} \\
&\vdots \\
T^j &= T^{j-1} \cup \{t_i^j\} \\
p_i^j &= \{(t_i^j, x) \mid x \in T^j\} \\
&\vdots \\
T^{t_i.h} &= A_{t_i.h-1} \cup \{t_i^{t_i.h}\} \\
p_i^{t_i.h} &= \{(t_i^{t_i.h}, x) \mid x \in T^{t_i.h}\}
\end{aligned}$$

Note that the leaving arc subsets p_i^j are unique. We also have that $|T^0| = \sum_{t_j \in T \setminus \{t\} \wedge \text{START-ACTIVE}(t_i, t_j)} t_j.h$, and from this we get that

$$\begin{aligned}
|p_i^1| &= |T^0| + 1 \\
|p_i^2| &= |p_i^1| + 1 = |T^0| + 2 \\
&\vdots \\
|p_i^j| &= |p_i^{j-1}| + 1 = |T^0| + j \\
&\vdots \\
|p_i^{t_i.h}| &= |T^0| + t_i.h
\end{aligned}$$

For the cumulative constraint, the costs computed for the virtual arc sets are

$$\begin{aligned}
f(p_i^1) &= \max(0, |T^0| + 1 - k) \\
f(p_i^2) &= \max(0, |T^0| + 2 - k) \\
&\vdots \\
f(p_i^i) &= \max(0, |T^0| + i - k) \\
&\vdots \\
f(p_i^{t_i.h}) &= \max(0, |T^0| + t_i.h - k)
\end{aligned}$$

The costs above have a very regular structure. This gives us the means to compute the expanded cost directly, without actually performing the task splitting. We get three cases for computing this value $\text{VSPLIT}_k(p_i)$ for a non-expanded arc subset p_i .

1. $|T^0| + t_i.h \leq k$, where the sum of the heights of the tasks in the leaving arc subset of t_i is less than or equal to k . In this case, the cost $f(p_i) = 0$.
2. $|T^0| \geq k$, where the sum of the heights of the tasks in the leaving arc subset of t_i excluding t_i itself is greater than or equal to k . In this case the cost is $f(p_i) = (t_i.h)(|T^0| - k) + (t_i.h)(t_i.h + 1)/2$, because $|T^0| \geq k$ implies that $f(p_i^j)$ is always positive:

$$\begin{aligned}
 f(p_i^1) &= |T^0| + 1 - k \\
 f(p_i^2) &= |T^0| + 2 - k \\
 &\vdots \\
 f(p_i^j) &= |T^0| + j - k \\
 &\vdots \\
 f(p_i^{t_i.h}) &= |T^0| + t_i.h - k
 \end{aligned}$$

3. $|T^0| < k < |T^0| + t_i.h$. In this case, for $j \leq k - |T^0|$, $f(p_i^j) = 0$, and for $j > k - |T^0|$, $f(p_i^j) > 0$, so we have the following.

$$\begin{aligned}
 f(p_i^1) &= 0 \\
 &\vdots \\
 f(p_i^{k-|T^0|+1}) &= |T^0| + j - k = 1 \\
 f(p_i^{k-|T^0|+2}) &= |T^0| + j - k = 2 \\
 &\vdots \\
 f(p_i^{t_i.h}) &= |T^0| + t_i.h - k
 \end{aligned}$$

Thus, the cost for this case is $f(p_i) = (|T^0| + t_i.h - k)(|T^0| + t_i.h - k + 1)/2$.

Since $|T^0| = \sum_{t_j \in T \setminus \{t\} \wedge \text{START-ACTIVE}(t_i, t_j)} t_j.h$ can be computed without expansion, we can compute the cost of VSPLIT for one task t_i without

expanding it. This approach avoids the negative effect on performance resulting from explicit expansion of the tasks, and instead computes the result of cost computation on this expanded set of tasks directly. This new formulation of the cumulative constraint is shown in Table 4.15. At present, VSPLIT is not incrementally computed.

Constraint:	<code>cumulative</code>
Arguments:	$T : \text{vec htask}, k : \text{int}$
Structure:	$A = \text{CLIQUE}(T)$
Filter:	$A' = \{A : \text{START-ACTIVE}_T(x_1, x_2)\}$
Partition:	$\sum_{p \in \text{LEAVING}(A')} f(p)$
Subcost:	$f(p) = \text{VSPLIT}_k(p)$

Table 4.15: The cumulative constraint.

4.5.6 The cumulative Constraint with Cyclic Time

We can further extend the family of cumulative constraints with a cyclic cumulative constraint called `cyclic-cumulative`. The constraint ensures that for all time-points $0 \leq i \leq \ell$, where ℓ is the cycle length, the sum of the heights of all tasks running at time i is less than or equal to the capacity k of the constraint. In addition, any task x ending after the cycle length should wrap around, and be considered to end its execution at time $s.a + d.a - \ell$.

Constraint:	<code>cyclic-cumulative</code>
Arguments:	$T : \text{vec htask}, k : \text{int}, \ell : \text{int}$
Structure:	$A = \text{CLIQUE}(T)$
Filter:	$A' = \{A : \text{CYCLIC-START-ACTIVE}_{(T, \ell)}(x_1, x_2)\}$
Partition:	$\sum_{p \in \text{LEAVING}(A')} f(p)$
Subcost:	$f(p) = \text{VSPLIT}_k(p)$

Table 4.16: The cyclic-cumulative constraint.

We describe the `cyclic-cumulative` constraint in Table 4.16. We can model the constraint by using the `CYCLIC-START-ACTIVE` filter constraint, which handles the eventual wrap around of tasks in time. The

CYCLIC-START-ACTIVE filter constraint can be described as below.

$$\begin{aligned} \text{CYCLIC-START-ACTIVE}_{\preceq, \ell}(A, B) &\equiv \text{START-ACTIVE}_{\preceq}(A, B) \\ &\vee v(x.s) < v(y.s) + v(y.d) - \ell \end{aligned}$$

The formulation is otherwise in analogy to the one for the `cumulative` constraint shown in Table 4.15.

In designing global constraints for local search, one should aim at using a cost for the constraints based on the same “common ground” as the primitive constraints in the system. We believe that a cost basis in the form of a set of primitive constraints, which all constraint costs are based on, fulfills this property. More specifically, we have designed the framework so that the cost of a global constraint is based on a set of satisfied filter constraints. We have decided to use generic unary and binary constraints as filter constraints.

To measure how many filter constraints are violated, we use the graph properties of a global constraint as input to the cost expression, in order to obtain a cost.

Chapter 5

A Global Constraint Library

5.1 Introduction

In this chapter we will investigate a library for modeling global constraints in constraint-based local search, based on the framework presented in Chapter 4. Global constraints in local search can be seen as high-level modelling components, designed to reduce evaluation time and space complexity. Traditionally, they have been implemented as monolithic entities, often using a low-level language and requiring in-depth knowledge of the constraint system itself. In this chapter we show a *compositional model* of global constraints, in which we use graph structures, filters and cost components to create global constraints using a high-level C++ framework called *Composer*. The composed constraints can then be used for constraint solving in a generic, domain-independent local search solver. We show how to compose several well-known global constraints, and also show by experimental results that a compositional approach for global constraint modelling is not only possible in practice, but also highly competitive with existing low-level constraint-based local search implementations.

The classical approach for generic constraint solving in local search is to provide a set of *primitive* constraints, which in turn can be used to form more complex combinatorial substructures [73]. A common choice is to restrict the user to binary constraints only. Unfortunately, for sev-

eral common combinatorial structures there simply is no decomposition into binary constraints that is acceptable in terms of space and/or time complexity. Global constraints exist in local search systems to provide time- and space efficient high-level components capturing common combinatorial substructures.

In the local search transition evaluation, we are interested in how a small change in a state can be re-evaluated efficiently. *Incremental evaluation* address this problem, and for global constraints it is often possible to write code for incremental evaluation of properties of the constraints. One such constraint is the well-known `alldiff` constraint, restricting all variables present to take disjoint values. The natural decomposition of this constraint reduces to $n(n - 1)/2$ binary inequality constraints, where n is the number of variables. The incremental evaluation of a decomposed `alldiff` constraint, where we assume a change of the value of one single variable, must consider exactly $n - 1$ binary constraints for re-evaluation, since a single variable is connected to $n - 1$ other variables via inequality constraints in such a decomposition. This gives us that re-evaluation has a time complexity of $O(n)$, which should be compared to the constant $O(1)$ time requirements for a global `alldiff` constraint. Also, a decomposition has the space complexity of $O(n^2)$, where a global constraint implementation of the `alldiff` constraint could be implemented using $O(n)$ space.

A compositional approach for modelling of global constraints gives us a high-level tool for *invention* and *modification* of global constraints. This process is very common in real-life applications, where the provided global constraints often do not fit the problem exactly. In our approach, we parametrize global constraints over key properties of their structure. To do this we use a *generic graph model*, an approach that has previously been used successfully to model a large number of global constraints [4]. The main benefit of a parametrized model of global constraints is that practitioners can very easily experiment with different cost functions and different structures, and also compose completely new global constraint with a minimum of work. Also, composition of constraints is not only simple to model, but the resulting constraints are evaluated very efficiently using highly-optimized incremental algorithms. The user does not have to care about keeping costs and conflict levels updated, which is handled by the *Composer* system automatically.

5.1.1 Host Environment

Although high-level constraint composition is a very attractive approach for global constraint implementation, there will always be a need for a strong host programming language as a support system for small custom-made implementation details of the constraint system. Therefore, we have implemented *Composer* as a template library for C++, which give us support of the full C++ language together with the standard template library. Also, this makes the integration of the constraint system into external software painless and transparent.

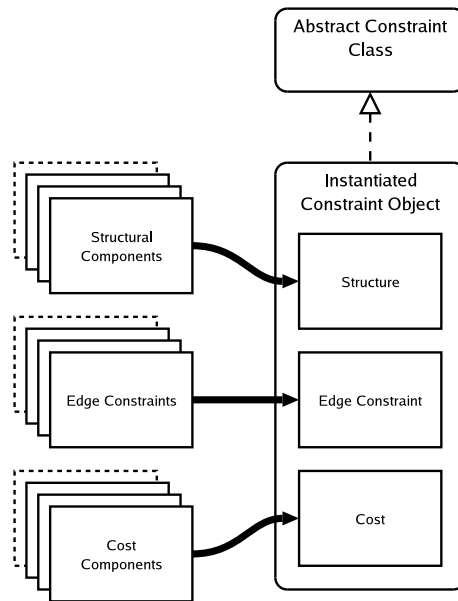


Figure 5.1: Design of the constraint composition framework. A constraint object is instantiated by selecting a structural component, a filter constraint and a cost component. Also, each composed constraint inherits its properties from an abstract constraint class used by all constraints in *Composer*.

The basic structure of a composed constraint in *Composer* is shown in Figure 5.1. A constraint is created using template instantiation of the classes used to represent components in the framework. The constraint is

represented by the structural class selected, which also inherits constraint generic properties from the abstract constraint class `Constraint`. If one wants to implement specialized constraints from scratch without the aid of *Composer*, this interface can be inherited to simplify the implementation.

The implementation of *Composer* as a template library has two main purposes. First, it provides the generality needed for such a component - it is easy to extend the library with more objects which will interact with the system, and second, it maintains the efficiency that is absolutely crucial in local search constraint solving. Using templates allow us to inline and eliminate function calls that would otherwise affect performance substantially. To use inheritance for component construction would require a virtual method call for each component evaluation, something that could be fatal for the solver performance. For example, in a single typical iteration it is not uncommon that a filter constraint is evaluated 10,000 times. Because the nature of the filter is to compute a simple expression, the actual virtual method call overhead would be a larger amount of the total execution time than the filter evaluation itself. This is of course unacceptable in a local search library, where main focus is on speed of evaluation. In addition, templates make inlining of these crucial method calls possible, something that further increase the performance of our method.

In complex template libraries, by nature the template instantiations can become quite messy. To overcome this obstacle and to simplify the interface to *Composer*, the user can also specify global constraints in a configuration file using a specification language. This language uses a syntax that is simplified from the complexities of template instantiation, and hides a lot of implementation-specific details that are of no interest to the user.

5.1.2 Chapter Outline

The rest of the chapter is organized as follows. First, Section 5.2 gives an overview of the architecture of the *Composer* local search system. Next, Section 5.3 gives an introduction to constraint composition in *Composer*. Section 5.3 and Section 5.4 discusses the algorithms for incremental update of the cost and conflict of a constraint. Sections 5.5 and 5.6 present two example applications, showing that constraint composition is indeed useful in practice.

5.2 The *Composer* Architecture

An overview of the architecture of *Composer* can be seen in Figure 5.2.

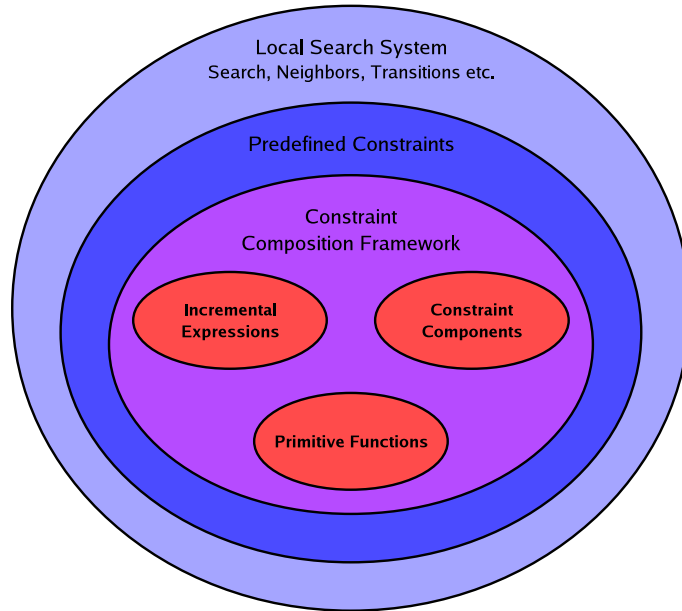


Figure 5.2: Overview of the architecture of *Composer*.

The *Composer* system can basically be divided in three layers.

1. The Constraint Composition Framework layer of *Composer* contains the implementation of incremental expressions, the components used to model global constraint, and some primitive functions that are used throughout the system.
2. The Predefined Constraints layer contains constraint that are already implemented, for example arithmetic constraints, and global constraints like those investigated in this thesis. It is important to point out that the predefined global constraints in this layer are composed as any other global constraint described in this thesis. Also, any monolithic global constraints would be located here – currently, no such constraints exist in *Composer*.

3. The Local Search System layer contains all the components that is needed to solve problems using local search on the constraint in the inner layers. Here, we have the mechanisms for basic local search like transitions, conflict and cost evaluation of a constraint system, state variables, etc. Also, metaheuristics like Tabu search and other history-based components are located here.

Most parts of the first and second layer has been discussed Chapter 4. The third layer is discussed in Chapter 3. In this section, we will therefore focus on the parts of the first layer, regarding incremental expressions, that have been left out.

5.2.1 Incremental Computation of Expressions

One of the cornerstones of *Composer* is *incremental evaluation of basic expressions* of the form

$$X = f(Y1, Y2, Y3, \dots, Yn)$$

where f is a generic function and $Y1, Y2, \dots, Yn$ are integer variables. A resulting directed acyclic graph (DAG) of functions on the form above has been shown previously to be well-suited to incremental evaluation [47, 1, 40]. In general, we assume that there exist an efficient algorithm to compute X once a single input variable Yi has changed. *Composer* provides many such functions useful for expressing relationship between variables in the problem model. Infix functions such as addition, subtraction and multiplication are also present for simplicity, and are converted into the form above. Also, observe that the expression $X = f()$ denotes that X should take the constant value computed by f (which take no arguments).

More complex expressions can also be used. These will be flattened into basic expressions on the form above by introducing new variables. For example, the expression

$$X = f(g(Y), Y*(Z+10)) - 20$$

is converted into the following expressions

$$\begin{aligned} X &= \text{sub}(T1, 20) \\ T1 &= f(T2, T3) \\ T2 &= g(Y) \end{aligned}$$

```
T3 = mult(Y,T4)
T4 = add(Z,10)
```

forming the dependency DAG shown in Figure 5.3.

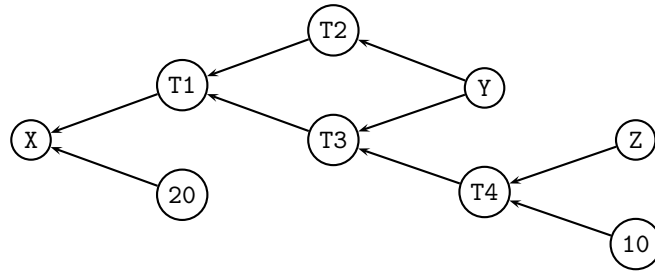


Figure 5.3: The dependency DAG for the flattened form of the incremental expression $X = f(g(Y), Y*(Z+10)) - 20$.

Only acyclic expressions are allowed. All basic expressions, constraints and variables as well as modifications are posted into a *model*, which keeps track of expression evaluation and updates, and also provides support methods for local search.

5.2.2 State and Non-state Variables

In *Composer* we distinguish between *state* and *non-state* variables. A state variable is subject to change from the search component. Non-state variables are simply entities whose values are entirely dependent on the values of the state variables in the model. In any incremental expression, the left-hand side variable must be a non-state variable, and the right-hand variables can be either state or non-state variables. The reason to differ between state and non-state variables is that *Composer* automatically computes a *conflict level* for each state variable X , indicating in how many conflicts X is present.

5.2.3 Incremental Functions

As an example of an incremental function, *Composer* provide the `vecOp` operator, which applies a given associative and commutative binary operator \oplus , for which an inverse \oplus^{-1} exist, on all its arguments $Y1, Y2, \dots, Yn$ and gives the result $Y1 \oplus Y2 \oplus \dots \oplus Yn$. The folding of \oplus over a set of

variables can be incrementally evaluated in time $O(f_{\oplus} + f_{\oplus^{-1}})$ where f_{\oplus} and $f_{\oplus^{-1}}$ is the time of computing $r = x \oplus y$ and $x = r \oplus^{-1} y$ respectively. For many common operators, such as addition and multiplication, this time is $O(1)$ giving such functions nice incremental properties.

For functions which do not have an inverse, such as \max , argmax ¹ and relatives, *Composer* provides efficient $O(\lg n)$ incremental implementations using a sorted structure.

We also include extreme value functions based on the usual `vecOp` operator. The worst case time complexity to compute the inverse of an extreme value operator such as \max is $O(n)$ where n is the number of arguments. To see this consider the case where we want to compute the maximum value of a set of variables S . We know that the last maximum was 6, and that a single variable $x \in S$ has changed from 6 to 3. To find the maximum of $S \setminus x$ we now need to investigate all elements of $S \setminus x$ to find the new maximum, and therefore the time complexity is $O(|S|)$.

However, considering that the only case in which we have to investigate all elements of $S \setminus x$ to find the new maximum is when the old maximum is *equal to the old value* of x , and the new value of x is *smaller* than the old value, this may still be a useful method – all other changes can be updated in constant time. The reasoning is similar for \min , argmax and argmin . *Composer* therefore provides two different methods for computing these functions - either using the inverse operator version `emin`, `emax` etc., or using a binary tree implementation `min`, `max` and so on.

5.2.4 Motivation

The usefulness of incremental expressions can be demonstrated by an example. In train scheduling [35] a train traversing a sequence of stations can be elegantly modeled using variables denoting departure d_i from station i , traversal time from station i to station $i + 1$, arrival a_{i+1} at station $i + 1$, and stop time s_{i+1} at station $i + 1$, and so on until the last station k . However, there exists a very tight relationship between these variables. Consider the case where the traversal time is known in advance and constant. We get the following equations.

$$\begin{aligned}\forall 1 < i \leq k. d_i &= a_{i-1} + s_{i-1} \\ \forall i. a_i &= d_i + t_i\end{aligned}$$

¹`argmax` computes the index to an argument that has a maximal value.

Because t_i is constant, we get a single departure variable d_1 and a set of stop time variables s_i on which all the other variables depend. Using incremental expressions, these relationships can be declaratively modeled and automatically used. The *Composer* system takes care of the handling of the dependent variables and the distribution of conflicts, so that state variables automatically are assigned a proper conflict level. The relationships above can be modeled in *Composer* as follows.

```
a[0] = d[0] + t[0];
for(int i=1;i<=k;i++)
{
    d[i] = a[i-1] + s[i-1] ;
    a[i] = d[i] + t[i] ;
}
```

The incremental back-bone of *Composer* also gives the global constraints an environment which helps in modelling many combinatorial problems. In fact, composed global constraints are compiled into special incremental functions and are fully integrated in the incremental framework of *Composer*. Because of this it also becomes very easy to post global constraints on non-state variables, which are efficiently computed by the system during run-time. For example, consider the train scheduling application above. We can for example impose that all trains on a single track should be serialized in time, which can be done as follows.

```
vector<vector<int> > start(Tracks);
vector<vector<int> > dura(Tracks);
for(int i=0;i<Traversals;i++)
{
    start[track[i]].push_back(d[i]);
    dura[track[i]].push_back(t[i]);
}
for(int i=0;i<Tracks;i++)
{
    serialized(model,start[i],dura[i]);
}
```

The code fragment above first creates two vectors for each track i , one with the start time and the other with the duration of all traversals on track i . Then, a `serialized` constraint is posted on the tasks

represented by the start and duration of the traversals on that track. Observe that `start[i]` is composed of non-state variables as shown in the previous example. *Composer* will now automatically assign a violation level to the state variables that `start[i]` depends on, making guided local search possible.

The composition of the `serialized` constraint is similar to the one used for the primitive `alldiff` constraint. The formulation of the `serialized` constraint used above is shown in Figure 5.4.

```
constraint Serialized (vec[var] x, vec[int] d)
{
    structure = Clique;
    filter    = TSOverlap<vector<int> >(d);
    partition = Leaving;
    subcost   = AtmostC<1>;
    property  = Card;
}
```

Figure 5.4: A `serialized` constraint specification in *Composer*.

In [40] an extended version of the incremental expressions used here is described, in which cycles in the dependency graph can in certain cases be resolved during runtime. This allows a limited form of recursive behavior of the equations and increase the expressiveness of the network somewhat. Presently, we do not support cyclic dependency graphs, but plan to incorporate this feature in the future. The main reason not to allow cyclic dependencies is that they require online detection and unnesting of cycles, imposing a substantial overhead on the evaluation of the incremental expressions.

5.3 Constraint Composition Using *Composer*

In this section we will discuss how to compose global constraints using the *Composer* template library. More on the formal model of global constraints can be found in Chapter 4.

5.3.1 Introduction

In the *Composer* model, a global constraint is parametrized over its *graph structure*, its *filter constraint*, and its *cost*. A global constraint is composed by selecting three components corresponding to the parametrized

properties above. The structure component uses the variables and parameters of the global constraint to construct an *arc set*. Although we often use binary arcs and constraints in *Composer* we are not limited to this and can use any uniform arity we want.

The filter component is then used to select a final set of *critical arcs* – the arcs that satisfy the filter constraint. We can then apply our *cost component* to the set of critical arcs to form the total cost of the constraint.

5.3.2 *Composition Language*

Constraints in *Composer* are either composed using the raw implementation template classes, or specified in a separate source file using the Constraint Composition Language. The specification file is then compiled into one C++ function for each constraint, in which all template instantiations and initializations are done automatically. The function also posts the constraint into a provided constraint model. The resulting code can then be compiled using a regular C++ compiler and included in the project.

A constraint is specified by a `constraint` block, defining the name of the constraint and necessary parameters for the components used. In comparison with C++ a simplified and slightly different syntax is used for the parameter description. A parameter is described by a type and an identifier, where the type is either a basic type (integer or variable) or a vector of types, denoted by `vec[T]`. All vectors are passed as a pointer by default, basic types are passed as value.

The composition of the constraint is done using a list of attribute descriptions – for example, the graph structure of the constraint is described by the `structure` attribute. An attribute is set by specifying an attribute, the `=` operator, and a value for the attribute (which is a component for use in the constraint). The value of an attribute is a component name, possibly extended with a parenthesized list of actual arguments to the component itself. The actual arguments are usual C-style expressions and are emitted as-is from the compilation.

The binarization of the `alldiff` constraint in *Composer* is shown in Figure 5.5. We use the variables of the constraint as nodes and form arcs between all subsets $\{x, y\}$ of two variables, using the `CliqueLt` structure. The equality constraint `VarEqVar` is used as filter component. Semantically this constraint is a binary constraint on two variables,

which is true if and only if the values of the two variables are equal. A linear threshold cost component `AtmostC<0>` with a threshold level of 0 on the cardinality of the resulting set is used as a total cost for the constraint. The component `AtmostC<k>` computes the value $\max(0, v - k)$ where v is a property value, in this case the cardinality of the final set of equality arcs in the graph.

```
constraint PrimAllDiff ( vec[var] x )
{
    structure = CliqueLt;
    filter    = VarEqVar;
    cost     = AtmostC<0>;
    property  = Card;
}
```

Figure 5.5: The primitive `alldiff` constraint.

The result of compiling the specification of Figure 5.5 is shown in Figure 5.6. Note that a lot of implementation detail has been hidden from the user, who only has to specify which components to use to form the global constraint. Still, the raw template library as provided by *Composer* is very high-level, and the translation of a specification file into C++ code is very straightforward. The very high level compositional approach at global constraint design is one of the main features of *Composer*.

```
void PrimAllDiff(Model &model, vector<state* > *x)
{
    typedef Card<AtmostC<0 > > cost_PrimAllDiff ;
    typedef CliqueLt<VarEqVar,
                    cost_PrimAllDiff> type_PrimAllDiff ;

    vector<int> Y1=toIntVec(x);

    type_PrimAllDiff * con =
        new type_PrimAllDiff(&Y1 ,&model) ;
    con->setFilter(VarEqVar(con));
    model->post(con);
}
```

Figure 5.6: The compiled primitive `alldiff` constraint.

Composer provides many other cost functions and properties, although we will mostly show variations of the ones above. Of course, the user is free to invent new cost functions and properties by writing new template classes.

The second type definition states that the `AllDiff` type is a composition of the `CliqueLt` graph structure, a `VarEqVar` filter, and the previously specified cost component `Cost`. The `CliqueLt` component forms arcs between all subsets $\{x, y\}$ of variables, and the `VarEqVar` constraint filters away all arcs for which the variables have disjoint values. The rest of the code instantiates the constraint with the composition classes. In this case we need to tell that the model in which the constraints should be posted is `model` and that the variables of the constraints are in `X`. The constraint above will act exactly as the binarization of the `alldiff` constraint – *Composer* will automatically compute the constraint cost and the conflict level of each present variable. In this case the `Card` component will compute a conflict level $c(X)$ for a variable `X` and a critical arc set A as $c(X) = \frac{2v|S|}{|A|}$ where $S = \{(X, y) \in A\} \cup \{(z, X) \in A\}$ and v is the cost of the constraint.

Although the model for `alldiff` shown in Figure 5.5 is brief and concise, it has one drawback. It is essentially a decomposition of the constraint into binary inequalities, and the space complexity of $O(n^2)$ and incremental time complexity of $O(b \cdot n)$, where b is the number of variable changes, is of course the same as for the decomposition. We can however do better than this, given that a more clever implementation of the `alldiff` constraint would have a space complexity of $O(n + v)$ and an incremental time complexity of $O(b)$, where v is the number of values for all variables. In *Composer* we can use the concept of *partitioned costs* to partition the cost of a constraint into smaller, more manageable parts. A partitioned cost is computed by combining several distributed cost computations in the final graph, which we refer to as *subcosts*. The exact distribution of subcosts is dependent on which combined cost we use. The subcost components themselves are also composed of a *property* and a *cost expression*. The property samples a trait of the arc set, returning an integer, and the cost expression is then applied to the integer, giving us the full cost of the constraint.

As we will see, using subcosts on all arcs entering or leaving a subset of the nodes of the graph is often useful. We therefore provide the two combined costs `Entering` and `Leaving`, each of which take a subcost component C as a template argument. The combined cost components

apply C on each subset A_i of arcs entering or leaving a single node respectively, and compute a total cost by incremental evaluation of $C(A_1) + C(A_2) + \dots + C(A_n)$. This can be done in time $O(b)$ where b is the number of changed subcosts.

In the specification language, the `subcost` attribute can in addition take an identifier enclosed in brackets to the left of the `=` operator, used for instantiation of the subcosts with different parameters depending on their order. This identifier is compiled into an integer variable, which at runtime will indicate the sequence number of the currently initialized subcost. This variable can then be used in the right-hand side expression of the attribute specification to assign specific parameters to the different subcosts.

For the `alldiff` constraint a straightforward representation is a bipartite graph with the variables as one node partition and the possible values as the other one. The arcs of the graph correspond to an assignment of a variable to a value. In such a model we can update the cost very efficiently by keeping track of how many variables are connected to a single value. In *Composer* we can model the `alldiff` constraint as an efficient bipartite structure with a combined cost as shown below. The `Bipartite` structure is an optimized version of the `Product` structure on graphs with variable and value nodes. The composition of the `alldiff` constraint below uses $O(n)$ space and $O(1)$ time for a single variable change.

```
constraint AllDiff ( vec[var] y )
{
    structure = Bipartite ;
    filter    = VarEqC ;
    combined  = Entering ;
    subcost[i] = QuadAtmostC<1>;
    property  = Card ;
}
```

Figure 5.7: A bipartite `alldiff` constraint with subcosts.

5.4 Incremental Cost and Conflict Computation

An important characteristic of local search is the efficiency of the cost computation. Typically, a neighbor is constructed by changing the value

of one or a small number of variables in the problem. One way of speeding up the cost computation is to take advantage of this small difference between a neighbor and the current solution in the computation of the new cost. This is the basis of incremental cost computation; recomputing only the parts of the cost that have actually changed [47, 1]. A local search procedure should use this technique to speed up the cost computation.

In this section we show how to use incremental cost computation for constraints described as cost functions on graphs in *Composer*, significantly reducing computation time. In general, we assume a change of the current assignment v to $v[x \mapsto a]$ for a variable x and a value a , where $a \neq v(x)$. Any change of v can be realized by applying transitions of this form in sequence.

Incremental computation can basically be divided in *coarse-grained* and *fine-grained* incrementality [77]. In coarse-grained incrementality, care is taken to only recompute sub-expressions that actually changes. In fine-grained incrementality, we try to modify the actual sub-expressions to obtain more efficient incremental properties.

The coarse-grained incrementality approach is taken in the papers of Alpern et al. [1], Michel and Hentenryck [42], and Yellin and Strom [77], where generic networks of expressions are incrementally evaluated. The nodes of these networks consist of variables or functions, depending directly on a set of arguments, also represented as nodes. Using this approach, one can guarantee that the minimum number of nodes are actually re-evaluated. In general, the networks have to be acyclic to be evaluated efficiently using incremental methods.

In *Composer*, the possibility to obtain coarse-grained incrementality is restricted to three levels; the *expression level*, where we form a network of expressions similar to those in the papers referred in the previous paragraph; the *global level*, where we compute the total cost and conflict level for all constraints, and the *constraint level*, where the cost and conflict level for a single constraint may be incrementally computed. The expression level is described in Section 5.2.1. We discuss the two other levels in Section 5.4.1 and 5.4.2.

In the fine-grained incrementality approach, care is taken to re-evaluate nodes in the network efficiently. For example, if one of the nodes computes a sum of its arguments, and only one of the arguments has actually changed, the new sum can be recomputed in constant time, instead of the linear time to the number of arguments that a complete re-evaluation

requires. Fine-grained incrementality is investigated in articles by Paige and Koenig [47] as well as in the papers of Michel and Hentenryck [42] and Yellin and Strom [77].

In *Composer*, fine-grained incrementality is used in all places where it can be motivated. For example, cardinality and plain and weighted sums are implemented using fine-grained incrementality, as shown for these properties in Section 5.4.2. It is important to point out that fine-grained incrementality can *not* be motivated for certain expressions – the actual cost of incremental computation should not exceed the cost of recomputing an expression. Therefore, we do not evaluate filter constraints and cost modifiers incrementally, since these tend to be very small and can be efficiently evaluated from scratch.

5.4.1 Total Cost and Conflicts

Coarse-grained incrementality on the local search engine level is obtained by recomputing the cost and conflict levels only for those constraints, which are posted on variables that have actually changed.

In our local search system, we use the extended cost function from Definition 33, which computes for an assignment the weighted sum of the costs of the individual constraints in the problem. Fine-grained incrementality is obtained by recomputing the sum as shown below.

$$f_n^e(C, v, x, a) = f_o^e + \sum_{c \in C \wedge x \in \text{VAR}(c)} w_c \Delta f_c(v, x, a),$$

where $f_n^e(C, v, x, a)$ denote the new extended cost of the problem given the transition $v[x \mapsto a]$. f_o^e is the old extended cost, which is saved in each computation, $\Delta f_c(v, x, a)$ is the difference between the new cost of the transition $v[x \mapsto a]$ for the constraint c , and the old cost of c , given the assignment v .

The conflict level is computed as in Definition 34. Using fine-grained incrementality, this can be computed as

$$\text{CL}_n(C, v, x, a) = \text{CL}_o^x + \sum_{c \in C \wedge x \in \text{VAR}(c)} w_c \Delta \text{CL}_c(v, x, a),$$

where $\text{CL}_n(C, v, x, a)$ is the new conflict level of x after the transition $v[x \mapsto a]$, CL_o^x is the old conflict level for x , saved after each computation, w_c is the constraint weight of c , and $\Delta \text{CL}_c(v, x, a)$ is the difference in

conflict level we get by the transition $v[x \mapsto a]$ for c , given the assignment v .

5.4.2 Individual Constraints

For global constraints to be practically useful in local search, we need to be able to incrementally evaluate the effect of variable changes on a constraint. To do this, we use an implementation model in where the graph structure is the *master* component of a global constraint, evaluating possible changes and propagating these to the other components.

In this model, an instantiated graph structure *is* a global constraint. For a transition $v[x \mapsto a]$ where $a \neq v(x)$, we can simply reapply the filter to those arcs that contain x , and incrementally recompute the cost of the transition $v[x \mapsto a]$. Only costs, subcosts and property values are updated incrementally – the status of the filter constraints and the cost modifiers are computed more efficiently from scratch.

On the constraint level, incremental computation for a constraint c and a variable assignment $v[x \mapsto a]$ is done as shown in Algorithm 5.1, which computes the difference $\Delta f_c(v, x, a)$ between the old and the new cost for the constraint. We do this by applying the incremental version of the cost function Δcf of the graph model on the set of arcs on vertices containing x , whose filter constraint has a changed Boolean state.

```

 $Y_x \leftarrow$  the set of vertices that contains  $x$ .
 $A_Y \leftarrow$  the set of arcs that contains any vertex in  $Y$ .
 $\Delta A \leftarrow [A_Y[k] \mid k \in \{1, \dots, |A_Y| \wedge fc(A_Y[k], v) = \mathbf{true}\}]$ , the set of arcs
in  $A_Y$  whose filter has changed state.
return  $\Delta cf(\Delta A)$ .

```

Algorithm 5.1: Incremental cost computation of the difference $\Delta f_c(v, x, a)$ between the old and the new cost.

For the PRODUCT graph structure, a more efficient computation of which arcs are affected can be done if we take into account the actual filter constraint used. If we have an equality filter constraint on a PRODUCT graph on variables and disjoint values, denoting that a variable take a certain value, we can compute the affected arcs more efficiently than in the general case. In this case, at most two arcs are affected by a transition $v[x \mapsto a]$: one arc connecting the variable to its old value, and one arc connecting the variable to its new value. We have the following

two cases.

1. If the values form an interval, we can use direct indexing of a vector V representing the value vertices to find the two arcs:

$$\Delta A \leftarrow \{(x, V[v(x)]), (x, V[a])\}$$

This can be done in $O(1)$ time.

2. If the values do not form an interval, we can represent them as a sorted sequence, and find the value vertices in $O(\log n)$ time, where n is the number of value vertices in the graph.

A hybrid method where we search a binary tree of intervals is also possible, but in the worst case this method also has a time complexity of $O(\log n)$. In our system, only the first case is implemented as the specialized graph structure `Bipartite`.

Algorithm 5.1 has a time complexity of $O(|A_Y| + g(|\Delta A|))$ where $g(|\Delta A|)$ is the time complexity for computing the cost difference of the affected arc set ΔA .

In computing incrementally the value of the cost function $\Delta cf(\Delta A)$, we have three cases depending on if the cost function is regular, combined, or specialized.

Regular Costs

For a regular cost function, $\Delta cf(\Delta A)$ is computed as shown in Algorithm 5.2. Here, the set of arcs A^+ , which became satisfied, and the set A^- , which became violated, are extracted. A new property value is then computed using the value difference $\Delta p(A^+, A^-)$ of the property, and the old property value p_o . The cost delta Δcv can then be obtained by computing the value of cost modifier cm for the new property value, and by subtracting the old cost value cv_o . During the process, we also update p_o and f_o , for use in the next computation.

The conflicts of the vertices are also computed in Algorithm 5.2. To do this, we need to keep track of the number of arcs, which satisfy the filter constraint, and how many of these arcs are connected to the vertices in the graph. This is done incrementally in the algorithm. Unfortunately, the actual number of conflicts on the vertices in the arc set have to be recomputed for all vertices in $A' \cup A^-$, where A' is the full set of arcs satisfying the filter. This is because the conflict of each vertex

```

/* added and deleted arcs */
 $A^+ \leftarrow \{a \mid a \in \Delta A \wedge \text{SATISFIED}(a) \wedge \neg \text{WAS-SATISFIED}(a)\}$ 
 $A^- \leftarrow \{a \mid a \in \Delta A \wedge \neg \text{SATISFIED}(a) \wedge \text{WAS-SATISFIED}(a)\}$ 
/* Update old property value */
 $p_o \leftarrow p_o + \Delta p(A^+, A^-)$ 
/* Compute cost difference */
 $\Delta cv \leftarrow cm(p_o) - cv_o$ 
/* Update old cost */
 $cv_o \leftarrow cm(p_o)$ 
/* Update # arcs in arc set */
 $arcs \leftarrow arcs + |A^+| - |A^-|$ 
/* Next step needed for conflict computation */
for all vertices  $x$  in  $A^+ \cup A^-$  do
    Update the number of final arcs  $a_x$  on  $x$ 
end for
/*  $A'$  is the set of arcs which currently satisfy the filter */
for all vertices  $x$  in  $A^- \cup A'$  do
    /* Compute conflict level difference */
     $\Delta CL_c(v, x) \leftarrow cv_o a_x / arcs - cl_x^o$ 
    /* Update old conflict level */
     $cl_x^o \leftarrow cv_o a_x / arcs$ 
end for
return  $\Delta cv$ 

```

Algorithm 5.2: Incremental update of $\Delta cf(\Delta A)$ for a regular cost function.

is dependent on the cost of the constraint, and also on the number of arcs that are connected to the vertex. In the general case, some of these parameters change in each iteration, implying that we have to recompute the conflicts for these vertices. We can exclude arcs not in $A' \cup A^-$ because they have not been in A' for at least two iterations. Therefore, their conflict contribution both was and is 0, and doesn't have to be recomputed. Given that the cost modifier can be computed in constant time, the worst-case time complexity of the general incremental update of a regular cost function is $O(f(|A^+| + |A^-|) + |A^-| + |A'|)$, where $f(|A^+| + |A^-|)$ is the time complexity of the property function. For the properties in this thesis, $f \in O(n)$. This makes the incremental update of the regular cost limited by the fact that we need to update the conflict levels of all vertices in the filtered and deleted arcs. If this conflict computation is disabled, the time complexity is reduced to $O(f(|A^+| + |A^-|) + |A^+| + |A^-|)$, which is at most linear to the number of changed arcs for the properties in this thesis.

For many constraints, the regular cost only works as a *subcost* which can improve the time complexity substantially. If the maximum number of arcs in the arc subset is limited to at most n , and only a constant number of subcosts are affected by a transition, the total complexity can be reduced to $O(n)$, which is perfectly acceptable for many problems.

In the implementation of this framework, the conflict level computation is also simplified in certain cases. If the cost computed is cardinality without a cost modifier, $cf(A') = |A'|$, the conflict level of Section 4.2.9 can be simplified to

$$CL_c(v, x) = cf(A') \frac{|A''|}{|A'|} = |A''|$$

where $A'' = \{a \mid a \in A' \wedge (a = (x, y) \vee a = (y, x))\}$. A'' is called a_x in Algorithm 5.2 and updated incrementally. Transforming this function to compute the difference in conflict level yield the following lines for updating the conflict level difference.

```

for all vertices  $x$  in  $A^- \cup A^+$  do
  /* Compute conflict level difference */
   $\Delta CL_c(v, x) \leftarrow a_x - cl_x^o$ 
  /* Update old conflict level */
   $cl_x^o \leftarrow a_x$ 
end for

```


In this way we can skip the actual conflict computation and only keep track of a_x . We only investigate the vertices in arcs in $A^- \cup A^+$ instead of $A^- \cup \Delta A$, reducing the time complexity of the algorithm to $O(f(|A^+| + |A^-|) + |A^+| + |A^-|)$, which again is at most linear to the number of changed arcs for the properties in this thesis. Today, the user can select a special version of the cost function to get this more efficient implementation, but automatic optimization of this computation is certainly possible.

Combined Costs

For a combined cost function, $\Delta cf(\Delta A)$ is computed as shown in Algorithm 5.3. First, the cost value delta Δcv is set to 0, and then the modified arc set $\Delta A = A^+ \cup A^-$ is partitioned using the partition function g . The cost delta Δcv is then updated by simply adding the cost delta $\Delta cf_p(p)$ to Δcv for each modified partition p . The conflict levels of the variables are distributed and updated as part of the subcost computation.

Using a combined cost can in some cases reduce incremental evaluation time significantly as noted in the previous section. The time complexity of Algorithm 5.3 is $O(|\Delta P| \cdot f(\max(|p|)))$ where $f(\max(|p|))$ is the time complexity for the subcost f for the maximum modified arc subset $p \in \Delta P$.

For certain graph structures and filter constraints, a constant number of arcs are updated for each transition, giving us a total time complexity of $O(f(1))$. See Section 5.4.2 for more info. If we can compute the subcosts $\Delta f_p(p)$ in linear time to the size of the change, we reduce the time complexity of the full constraint to $O(1)$. This can be done for certain cardinality constraints, see Section 5.4.2.

```

 $\Delta cv \leftarrow 0$ 
 $\Delta P \leftarrow g(\Delta A)$ 
for each  $p$  in  $\Delta P$  do
     $\Delta cv \leftarrow \Delta cv + \Delta f_p(p)$ 
end for
/*  $\Delta CL_p$  is computed and distributed in the subcost function */
return  $\Delta cv$ 

```

Algorithm 5.3: Incremental update of a combined cost function $\Delta cf(\Delta A)$.

Special Cost

Finally, if the constraint uses a special cost function, like the VSPLIT cost of Section 4.5.5, the corresponding function Δcf for computing the difference in incremental cost and conflicts, is used.

Incremental Property Computation

The values of the properties are also updated incrementally. This is often trivial. We show here how this can be done for the cardinality and weighted sum properties. The cardinality value of a set can be incrementally updated in constant time by computing

$$\Delta_{\text{CARD}}(A^+, A^-) = |A^+| - |A^-|,$$

and the corresponding computation for the weighted sum is

$$\Delta_{\text{WSUM}}(A^+, A^-) = \sum_{a \in A^+} w(a) - \sum_{a \in A^-} w(a).$$

Because w is constant, we can store it locally. The computation of $\Delta_{\text{WSUM}}(A^+, A^-)$ has a linear time complexity of $O(|A^+| + |A^-|)$.

5.5 The n -Queens Problem

In this section we will investigate a model and algorithms for solving the n -queens problem, a classical combinatorial puzzle that has been used extensively as an example in the constraints community. In this problem, n queens are to be placed on a chessboard of dimensions $n \times n$ so that no queen can be attacked by another queen. The simplest instance of the n -queens problem with a solution is the 4-queens problem.

All results were obtained on a 1 GHz Pentium III with 256 Mb of memory, running Redhat 9 with Linux kernel version 2.4.20. gcc version 3.2.2 dated 20030222 (Red Hat Linux 3.2.2-5) was used to compile the models.

5.5.1 A Constraint Model

We have solved the n -queens problem in two different ways – one using a simple 1-assignment neighborhood, and one using a more elaborate 2-swap neighborhood. Both models are similar in that each queen is

```

Model model;

statevector queen(Queens, "Queen_", 0, Queens-1);

vector<int> X(fromto(0, Queens-1));
vector<int> pos(fromto(0, Queens-1));
vector<int> neg(fromto(Queens-1, 0));

alldiff(model, &X);
alldiff(model, &X, pos);
alldiff(model, &X, neg);

model.randomize();
model.close();

```

Figure 5.8: A first model of the n -queens problem.

represented by a state variable with domain $0..n - 1$. The restriction on the positions of the queens significantly reduces the size of the state space. In the first model, we use three `alldiff` constraints to model that no two queens on the board may attack each other as shown in Figure 5.8. The constraint

```
alldiff(model, &X, M)
```

states that for all variable pairs (x_i, x_j) in X where $i \neq j$, the constraint $x_i + m_i \neq x_j + m_j$ should hold where $m_i, m_j \in M$. The *Composer* system automatically computes a cost and a conflict level for each composed constraint. The model presented here is the same as the one stated in Section 2.2.1.

```

constraint AllDiff ( vec[var] y ,vec[int] m )
{
  structure = Bipartite;
  filter = VarPlusEqC(m);
  cost = Entering;
  subcost[i] = QuadAtmostC<1>;
  property = Card ;
}

```

Figure 5.9: A bipartite `alldiff` constraint with an offset vector M .

A generic `alldiff` constraint with an offset vector is composed in *Composer* as shown in Figure 5.9. The difference from the `alldiff` constraint in Figure 5.7 is shown in boldface, and is simply a replacement of the trivial equality constraint $v(X) = k$ with an equality constraint with offset, $v(X) + c(X) = k$. This shows quite clearly the expressiveness of the global constraint composition approach of *Composer*. This new composition of the `alldiff` constraint also uses $O(n)$ space and $O(1)$ time for a single variable change.

5.5.2 A First Local Search Algorithm

The local search algorithm used for the first model of the n -queens problem is shown in Algorithm 5.4. Line 1 declares place-holders for a variable

```
int Q,V;
while(model.violation()!=0)
{
    Q = model.getMaxConflicting();
    V = model.getMinCostValue(Q);

    model.assign(Q,V);
}
```

Algorithm 5.4: Search component for the n -queens puzzle using assignment transitions.

and a value, `Q` and `V`. The next line states that the search should continue until no constraints are violated. Remember that *Composer* will update the constraint violations and conflict levels automatically using incremental algorithms. The next line

```
Q = model.getMaxConflicting();
```

fetches a random variable with maximum conflicts, and the line

```
V = model.getMinCostValue(Q);
```

finds the value `V` for `Q` which minimizes the violation of assigning `V` to `Q`. Finally,

```
model.assign(Q,V);
```

places queen `Q` on row `V` and updates all conflict levels and the violation of `model`.

Size n	Init (s)	Solve (s)	Iter.
8	0.06	0.06	30
16	0.05	0.02	465
32	0.05	0.02	152
64	0.06	0.04	218
128	0.06	0.05	193
256	0.08	0.21	368
512	0.08	0.87	671
1024	0.10	2.14	834
2048	0.12	7.81	1346
4096	0.21	27.35	2336
8192	0.36	100.39	4259
16384	0.76	386.06	8179

Table 5.1: Experimental results in CPU time for 11 runs of the n -Queens Problem.

5.5.3 Results

In Table 5.1 we show the best results obtained by running n -queens 11 times for each instance using our first model. The results are clearly competitive with those found in [9]. Compared with the results obtained by Michel and Hentenryck in [42], our results are similar. We however managed to decrease the number of iterations by roughly 50%, but on the two largest problem instances, *Composer* is marginally slower in terms of CPU time than what is reported by Michel and Hentenryck. However, the results for [42] are not directly comparable with our results. The results by Michel and Hentenryck are obtained on a 1.1 GHz Linux PC, compared to the 1 GHz Linux PC we used for our results. Also, a low-level monolithic implementation of global constraints was used in [42]. It is important to keep in mind that *Composer* is a fully generic global constraint composition library – in fact, *Composer* maintains a lot of data structures that are necessary to keep this flexibility.

We noted that the search was prone to becoming trapped in local minima and was not very robust in terms of consistency of results, which led us to try another method of solving the problem.

5.5.4 A Swap-Based Local Search Algorithm

For the second algorithm we decided to use variable swaps as the local transition instead of assignments. This approach has previously been shown to be very successful for several hard combinatorial problems [9, 20, 69]. If we initialize the queens as a random permutation of $0..Queens - 1$ and always use swaps between two variables as the transition, it is easy to see that the first `alldiff` constraint becomes redundant since it will be satisfied in all visited states. Thus, the second model only differs from the first in that we remove the first constraint, and use a random permutation instead of the fully random initialization.

```
int Qi,Qj;
int oldV,iter;

Tabuvector tabu(model.arity(),&iter);

for(iter=0; model.violation()!=0; iter++)
{
    Qi = model.getMaxConflicting(tabu);
    Qj = model.getMinCostSwapWith(Qi,tabu);

    oldCost = model.violation();

    model.swap(Qi,Qj);

    if(model.violation() >= oldCost)
    {
        tabu.insert(Qi,Tenure);
    }
}
```

Algorithm 5.5: Search component for the n -queens puzzle using value swaps.

In the new algorithm, in each iteration we first select at random one of the queens s with maximum number of conflicts. We then find the set of queens t which minimizes the resulting cost of swapping queen s and t . We then randomly select one of these queens and swap the positions of s and t . We also use a tabu component from *Composer* to diversify the search on plateaus and to escape local minima. In this case the tabu component consists of a vector T of size n in which element T_i stores

the iteration count when queen i was marked as tabu. This arrangement makes it possible to check if a queen q_i is tabu in constant time, simply by checking the current iteration count against T_i .

The code for solving the second model is shown in Algorithm 5.5. The line

```
Tabuvector tabu(model.arity(),&iter);
```

declares a tabu vector with size equal to the arity of the problem, which uses the integer variable `iter` as the iteration count. The next line

```
for(iter=0; model.violation()!=0; iter++)
```

initializes the iteration count, loops until the violation of the constraints in `model` is zero, and increases the iteration count by one each iteration. The lines

```
Qi = model.getMaxConflicting(tabu);  
Qj = model.getMinCostSwapWith(Qi,tabu);
```

first fetches a randomly chosen variable Q_i with maximum number of conflicts, disregarding the variables which are tabu in `tabu`, and then selects a second random variable Q_j for which swapping Q_i and Q_j yields the least violation possible, once again disregarding variables declared as tabu. Finally, the lines

```
oldV = model.violation();  
model.swap(Qi,Qj);  
if(model.violation() >= oldV)  
{  
    tabu.insert(Qi,Tenure);  
}
```

first saves the previous violation of the model, and then commits to the value swap between Q_i and Q_j . If the new violation is greater or equal to the previous one, we then declare Q_i tabu. We use a tabu tenure of 10 which we fixed with minor experimentation.

Previous work on the n -queens puzzle has been successful in using a first-descent strategy instead of a conflict minimizing one [69]. We therefore modified our search procedure to select an improvement immediately if one existed. The new procedure is equal to the one shown in Algorithm 5.5, except that we use the line

```
Qj = model.getImprovingSwapWith(Qi, tabu);
```

to select a second queen. This method works exactly as the one above, except that as soon as a swap which improves the current violation is found, the method returns with this one immediately.

5.5.5 Results

Size n	<i>Composer</i>				Adaptive search	
	Init (s)	Solve (s)	Total (s)	Iter.	Solve (s)	Iter.
8	0.03	0.00	0.03	2	0.00	6
16	0.03	0.00	0.03	11	0.00	9
32	0.04	0.00	0.04	40	0.00	14
64	0.04	0.00	0.04	25	0.00	22
128	0.04	0.00	0.04	29	0.00	36
256	0.04	0.01	0.05	54	0.01	63
512	0.04	0.04	0.09	121	0.06	116
1024	0.06	0.16	0.22	213	0.22	216
2048	0.08	0.84	0.93	421	0.82	409
4096	0.15	4.48	4.62	806	3.23	805
8192	0.27	20.82	21.09	1593	13.12	1577
16384	0.52	89.96	90.48	3153	59.25	3118
32768	1.01	371.89	372.9	6279	276.09	6215

Table 5.2: Experimental results in CPU time of 11 runs of the n -Queens Problem using the second model and conflict minimization.

The results of the swap-based steepest-descent algorithm is shown in Table 5.2 together with results obtained by using the *Adaptive Search* algorithm from [9]. The results reported are median results of 11 runs. As can be seen, the results show that using a generic framework for global constraints is not only feasible but also highly competitive with those results previously obtained in [42] and [9]. The results of the modified first-improvement algorithm shown in Table 5.3 also support this. Note that the number of iterations has increased but the runtime of the new algorithm is generally smaller than the conflict minimizing algorithm. Also, keep in mind that *Adaptive Search* uses a low-level encoding of the problem in contrast with *Composer*. The results of Table 5.2 and 5.3 clearly outperform those reported by Michel and Hentenryck

[42]. A probable explanation of this is that we use a neighborhood based on variable swaps, which is much more powerful than the variable assignment neighborhood of Michel and Hentenryck.

Size n	<i>Composer</i>			
	Init (s)	Solve (s)	Total (s)	Iter.
8	0.02	0.00	0.02	4
16	0.02	0.00	0.02	8
32	0.02	0.00	0.02	59
64	0.02	0.00	0.02	30
128	0.02	0.00	0.02	65
256	0.02	0.00	0.02	119
512	0.02	0.01	0.03	253
1024	0.03	0.02	0.05	498
2048	0.04	0.10	0.14	1014
4096	0.07	0.39	0.46	1975
8192	0.13	1.51	1.64	3975
16384	0.25	7.35	7.60	7942
32768	0.51	35.74	36.25	15906

Table 5.3: Experimental results in CPU time of 11 runs of the n -Queens Problem using the second model and conflict improvement.

5.6 The Progressive Party Problem

The progressive party problem is a well-known benchmark problem in the constraint programming community, and have been used in several generic constraint-based local search methods as well [73, 20, 42]. The problem can be described informally as follows. An evening party is to be organized in the setting of a yachting rally. The organizers have decided that the visiting boats will visit the organizers' boats (the *host boats*) in turn, where the crew of the host boat stays aboard and serves the guests on the *guest boat*. Every 30 minutes, each guest boat will move to a new host boat. A guest boat must never visit a host boat twice, but does not have to visit *all* host boats. Also, the capacities of the host boats must never be exceeded, lest the host boat sinks. This will go on for a given number of time periods. The organizers have also decided two guest crews must never meet more than once. In Table 5.4

the different boats and their crews and capacities are shown. We also show the *spare capacities* of each boat, which is simply the number of guests that can safely be taken aboard.

i	k_i	c_i	s_i	i	k_i	c_i	s_i	i	k_i	c_i	s_i	i	k_i	c_i	s_i
1	6	2	4	12	10	3	7	23	7	4	3	33	6	2	4
2	8	2	6	13	8	4	4	24	7	4	3	34	6	2	4
3	12	2	10	14	8	2	6	25	7	2	5	35	6	2	4
4	12	2	10	15	8	3	5	26	7	2	5	36	6	2	4
5	12	4	8	16	12	6	6	27	7	4	3	37	6	4	2
6	12	4	8	17	8	2	6	28	7	5	2	38	6	5	1
7	12	4	8	18	8	2	6	29	6	2	4	39	9	7	2
8	10	1	9	19	8	4	4	30	6	4	2	40	0	2	-2
9	10	2	8	20	8	2	6	31	6	2	4	41	0	3	-3
10	10	2	8	21	8	4	4	32	6	2	4	42	0	4	-4
11	10	2	8	22	8	5	3								

Table 5.4: Configuration of the Progressive Party Problem. Each boat i has a total capacity k_i (the total number of people allowed on board at the same time), and a crew size c_i . The spare capacity s_i of a boat is formed by subtracting the crew size from the total capacity, $s_i = k_i - c_i$.

5.6.1 The Model

We have solved the progressive party problem using the *Composer* system and composed global constraints. The problem instances we have considered are listed in Table 5.5, where the first 6 instances are well-known and previously studied in a local search context [42, 20]. Instance 7 to 11 are new and has not been investigated as far as we know. We will see that these instances become increasingly harder to solve. Also, we have tried to solve the progressive party problem for as many time periods as possible, making the problem increasingly hard.

The experiments were carried out on a 1 GHz Pentium III with 256 Mb of memory, running Redhat 9 with Linux kernel version 2.4.20. gcc version 3.2.2 dated 20030222 (Red Hat Linux 3.2.2-5) was used to compile the models.

The model we have used is shown in Figure 5.10. We use a matrix of states `boat` of dimension $p \times g$, where p is the number of periods to run the problem, and g is the number of guests available. `boat(i, j)`

Config	Host boats	S	C	ratio
A	1-12,16	100	92	0.92
B	1-13 (orig.)	98	94	0.96
C	1,3-13,19	96	92	0.96
D	3-13,25,26	98	94	0.96
E	1-11,19,21	95	93	0.98
F	1-9,16-19	93	91	0.98
G	1-11,21,22	94	92	0.98
H	1-9,16-18,22	92	90	0.98
I	1-9,15-17,22	91	89	0.98
J	1-11,22,23	93	92	0.99
K	1,3-11,21-23	91	90	0.99

Table 5.5: Progressive Party Problem analysis for different host configurations. S is the total spare capacity of the host boats, and C is the total number of guest crew members in the instance.

represents which host boat guest j is assigned to in period i . We also use the auxiliary vectors `size` which holds the crew size of the guests, and `spare` which holds the spare capacities of the host boats. The model consists of three types of composed global constraints. First, the `capa` constraint states that for all time periods, the capacity of the host boats should not be exceeded. The `capa` constraint is actually a restricted version of a generalized weighted cardinality constraint [42] (which is also easily expressed using *Composer*, see the `wcc` constraint in Section 4.3.3), and is shown in Figure 5.11. This constraint is equivalent with the inequalities

$$\forall v. \sum_{x \in X | x=v} w_x \leq c_v,$$

where X is the set of variables in the constraint. The `nbdiff` constraint states that in the two vectors X and Y , at most $k = 1$ pairs X_i, Y_i may be equal. The constraint is equivalent to the following.

$$|S| \leq k, S = \{i \mid X[i] = Y[i]\}$$

In this problem, we post a `nbdiff` constraint for all pairs of guest boats i, j where $i < j$. The constraint makes sure that the number of periods when boat i and j meet are less than or equal to 1.

```
statematrix boat(Periods, Guests);
vector<int> X(Periods), Y(Periods);
for(int p=0; p<Periods; p++)
{
    capa(model, boat(p), size, spare);
}
for(int i=0; i<Guests; i++)
{
    for(int j=0; j<Periods; j++)
    {
        X[j] = boat(j, i);
    }
    alldiff(model, &X);
}
for(int i=0; i<Guests; i++)
{
    for(int j=i+1; j<Guests; j++)
    {
        for(int k=0; k<Periods; k++)
        {
            X[k]=boat(k, i);
            Y[k]=boat(k, j);
        }
        nbdiff(model, &X, &Y);
    }
}
}
```

Figure 5.10: A model of the progressive party problem.

5.6.2 A Local Search Algorithm

The local search procedure used is shown in Algorithm 5.6. In short, we use a simple hill-climbing modification in which the best move is always selected, and where ties are broken randomly. As before we use the single-assign neighborhood and use a tabu structure, this time over the variables *and* values, to diverge the search. As parameters we use a tabu tenure of initially 2. This tenure is increased or decreased as the search proceeds in order to not hinder the exploration of new states. We also use a limited backtracking mechanism to improve performance on very hard problem instances. Every time a new best state is found, a state memory is cleared and initialized by the line

```
int G,H,iter,bestsofar,oldV;

Tabumatrix(model.arity(),Hosts,&iter,&bestsofar);

for(iter=0; model.violation()!=0; iter++) {
    oldV = model.violation();

    G = model.getMaxConflicting();
    H = model.getMinCostValue(tabu);

    model.assign(G,H);
    tabu.insert(G,H,Tenure);

    if(model.violation() < oldV && Tenure > 2)
        Tenure --;

    if(model.violation() >= oldV && Tenure < TenureLimit)
        Tenure ++;

    if(model.violation() < bestsofar) {
        model.hist_clear();
        model.hist_store();
        nonimproving=0;
    } else {
        nonimproving++;
        if(model.violation()==bestsofar)
            model.hist_store();
    }
    if(nonimproving>Backtrack)
        model.hist_restoreProportional();
    if(iter>MaxTries)
        model.randomize();
}
```

Algorithm 5.6: The search component for the progressive party problem.

```
constraint Capa ( vec[var] y, vec[int] c, vec[int] k )
{
    structure      = Bipartite;
    filter         = VarEqC ;

    /* sum of subcosts of entering arcs on all nodes */
    cost          = Entering ;

    subcost[i]    = AtmostVariant(k[i]);
    property      = WSum1<WVector>(c);
}
```

Figure 5.11: A bipartite capa constraint with subcosts.

```
model.hist_clear(); model.hist_store();
```

The state history component also keeps track of the frequency of the states, and whenever `Backtrack` transitions have been done, a state in the state memory is chosen with probability inversely proportional to the frequency of the state, and the search backtracks to this state. This will ensure that the search restarts from a good state and also that all currently best states are kept in memory for future backtracking reference.

To improve performance even more, we also use a restarting mechanism which restarts the search from scratch whenever `MaxTries` iterations have passed. We added this feature when we noted that the search got stranded in areas where no improvement seemed possible. With the restarting mechanism, we were able to get the local search to terminate with a solution in every sample run of every instances of the problem tested.

5.6.3 Results

Table 5.4 shows the different host configurations used when solving the progressive party problem using *Composer*. We fixed `Backtrack` to 2000 and `MaxTries` to 100000.

The resulting median CPU times and iterations of 101 sample runs of PPP are shown in Table 5.6 and 5.7. For the first 6 host configurations we see a clear improvement over the results reported in [73, 20, 42] using similar neighborhoods. The local search algorithm does not differ in any substantial way from those used in [20, 42], which clearly shows

Cfg	Periods									
	1	2	3	4	5	6	7	8	9	10
A	0.01	0.02	0.02	0.04	0.06	0.10	0.18	0.31	0.69	5.23
B	0.01	0.03	0.05	0.09	0.16	0.27	0.58	2.51	14.9	
C	0.01	0.03	0.05	0.10	0.15	0.27	0.57	2.28	22.2	
D	0.01	0.03	0.06	0.11	0.19	0.31	0.66	2.52	20.8	
E	0.02	0.05	0.11	0.20	0.40	1.14	6.94			
F	0.02	0.06	0.13	0.26	0.55	1.81	18.3			
G	0.02	0.05	0.11	0.22	0.47	1.41	46.1			
H	0.02	0.06	0.13	0.24	0.61	2.26	118			
I	0.02	0.06	0.14	0.31	1.11	49.5				
J	0.04	0.20	0.51	4.46						
K	0.04	0.18	0.62	3.03						

Table 5.6: Median CPU time of 101 runs of the Progressive Party Problem using restart solving.

Cfg	Periods									
	1	2	3	4	5	6	7	8	9	10
A	0.01	0.03	0.06	0.10	0.16	0.25	0.45	0.83	1.86	13.2
B	0.02	0.07	0.13	0.24	0.43	0.72	1.57	6.63	37.2	
C	0.02	0.07	0.14	0.26	0.40	0.73	1.57	6.20	55.2	
D	0.03	0.09	0.16	0.28	0.50	0.84	1.81	6.60	51.6	
E	0.05	0.14	0.30	0.55	1.14	3.25	17.8			
F	0.05	0.15	0.37	0.73	1.54	4.92	46.2			
G	0.05	0.13	0.29	0.59	1.33	3.92	116			
H	0.04	0.16	0.34	0.66	1.76	6.51	291			
I	0.05	0.16	0.39	0.88	3.16	124				
J	0.10	0.57	1.47	11.5						
K	0.11	0.54	1.78	8.09						

Table 5.7: Median iterations of 101 runs of the Progressive Party Problem using restart solving, in units of 10^3 iterations.

the feasibility of high-level constraint modelling using global-constraint composition. The results reported in our work and those reported in [73] are not directly comparable, since Walser uses linear constraints exclusively to model PPP. This also demonstrates the need for efficient global constraints in constraint-based local search.

Cfg	Periods									
	1	2	3	4	5	6	7	8	9	10
A	0.01	0.01	0.01	0.02	0.03	0.04	0.06	0.13	0.43	4.42
B	0.01	0.02	0.03	0.04	0.06	0.11	0.29	1.52	21.0	
C	0.01	0.02	0.03	0.04	0.06	0.11	0.30	1.92	52.1	
D	0.01	0.02	0.03	0.05	0.07	0.12	0.37	1.83	51.4	
E	0.02	0.03	0.05	0.08	0.22	0.58	7.04			
F	0.02	0.03	0.07	0.10	0.24	1.46	53.4			
G	0.02	0.03	0.05	0.09	0.24	1.05	60.1			
H	0.02	0.03	0.06	0.10	0.31	1.66	144			
I	0.02	0.04	0.06	0.14	0.59	60.3				
J	0.06	0.08	0.21	3.32						
K	0.07	0.10	0.24	4.31						

Table 5.8: Median CPU time of 101 runs of the Progressive Party Problem using incremental solving.

In Table 5.7 we see the number of iterations used to compute a solution for PPP using our local search algorithm. These results compare fairly well with those reported in [20, 19], although we have a higher iteration count where the number of periods are less than 9. This is probably due to the incremental nature of the local search algorithm in [20, 19]. Note also that on the hardest problem instance on period level 9, we have generally a lower iteration count than reported by Galinier and Hao, which can probably be accredited to our backtracking mechanism which focuses the search on promising areas.

We also tried a similar strategy as in [19] of incremental solving for PPP, where we first solved the problem for 1 time period, then for 2 time periods, this time reusing the found solution as an initial solution, then for 3 time periods, again reusing the found solution for 2 time periods, and so on. The variables were all randomly initialized in the beginning.

The results in CPU time for this approach is shown in Table 5.8. A comparison with the previous approach shows that for almost all hard problem instances of a configuration, where the number of time periods

Cfg	Periods									
	1	2	3	4	5	6	7	8	9	10
A	0.01	0.02	0.03	0.04	0.06	0.10	0.17	0.32	1.18	10.6
B	0.02	0.04	0.07	0.10	0.16	0.31	0.79	4.08	50.5	
C	0.02	0.04	0.06	0.10	0.15	0.30	0.81	4.68	126	
D	0.03	0.05	0.07	0.13	0.19	0.35	1.04	4.79	124	
E	0.05	0.09	0.14	0.23	0.62	1.65	17.2			
F	0.04	0.09	0.17	0.27	0.66	4.08	128			
G	0.04	0.09	0.14	0.24	0.66	2.98	146			
H	0.05	0.08	0.15	0.29	0.85	4.54	354			
I	0.05	0.10	0.16	0.40	1.67	148				
J	0.16	0.23	0.62	8.34						
K	0.19	0.30	0.69	11.0						

Table 5.9: Median iterations of 101 runs of the Progressive Party Problem using incremental solving, in units of 10^3 iterations.

was the maximum tried for that configuration, the incremental approach was slower than the previous approach. For instances where the number of time periods was not the maximum number tried for the configuration, the incremental approach was generally faster. The iteration results of Table 5.9 support this.

A possible explanation of these results might be that the more clever initial solution of the incremental solver algorithm gave the search a good starting point, which for easier problems was close to a solution. For harder problem instances, where the number of local minima are more frequent and the number of true solutions are fewer, the initial solution was too good, and the search became more prone to get trapped in local minima.

Chapter 6

Related Work

In this chapter we will discuss several related methods and systems for constraint satisfaction using local search.

6.1 Localizer and Friends

Localizer [40] is a full-blown language for local search. The language has over the years evolved into first Localizer++ [41], a C++ library, and later Comet [42], once again a language, but more based on object orientation than the original language. All the variants of Localizer are based on incremental expressions called *invariants*. These invariants are an extension of traditional incremental expressions, as found in [77, 1] as well as in our work, with cyclic invariants. Cyclic expressions increase the expressive power of incremental expressions to include limited form of recursion. However, no clear incremental advantage is gained by these structures. In traditional incremental expressions, the evaluation of an expression always yields a correct value due to strict acyclicity requirements. Cyclic invariants however introduce the possibility of run-time failure, and a cycle detection procedure must be included, thus degrading evaluation performance. In general, Localizer is more of an implementation language for local search procedures than our work, which is more aimed towards declarative constraint solving using local search.

6.2 *INC*

INC [77] is a language purely designed for incremental evaluation based on very high level languages such as SETL. In INC, a program is a network of nodes, representing incremental functions. This network forms a directed acyclic graph (DAG) of the nodes and their dependencies, and is topologically sorted before execution. This guarantee that at the evaluation of a node, all of its arguments have either already been evaluated (incrementally) or should not be evaluated at all. This yield a form of *course-grain* incrementality [1]. On the other hand, *fine-grain* incrementality is obtained by finite differencing techniques [47] used in the nodes themselves. For example, a sum can be recomputed efficiently if the last result and the changed input terms are available.

The data types of INC are the primitive types; integers, booleans, chars and reals, and constructed types. Two constructed types exist; tuples and multisets. A form of higher-order functions exists in the *hierarchical components* of INC, which is computational nodes with exchangeable subcircuits performing parametrized computation.

Composer and INC differ at a high level in that INC is a general-purpose language, not directly aimed at constraint satisfaction and local search, where *Composer* is a library specifically made for this purpose. At a more detailed level, *Composer* helps the global constraint designer and provides many features useful for local search, where no help is given by INC. Nevertheless, many features of INC would be highly useful in a local search setting. For example, the multiset types are very user-friendly and useful in modelling, although incremental operations on set types are in some cases too expensive.

6.3 *Adaptive Search*

In [9], the local search algorithm Adaptive Search is presented. This algorithm takes a CSP on a special form as input, and performs local search using single variable assignments and Tabu search. This algorithm is similar to what is used in Algorithm 5.6 for the progressive party problem. A problem is specified by a set of *variables* with associated domains, a set of *constraints* with associated error functions (more on this later), a *combination function*, and a *cost function* for minimization. Constraints are present mainly in the form of *error functions*, which in practice is the same as the cost of a constraint in *Composer*. The conflict

level for a variable x is computed by combination, using the combination function, of the costs of the constraints in which x of the constraints in which x is present. In practice, variations of summation is used as both cost and combination functions.

The main difference between *Composer* and Adaptive Search is that in the former, a full framework of incremental graph components is available to help the user in global constraint modelling, while in the latter the user has no help from the system, and is forced to re-implement efficient incremental error functions, combination functions and ad-hoc cost functions for use in the algorithm. Also, the user is stuck with the design choices made in Adaptive Search, and has no choice of local search algorithm to use. This can be a major drawback in certain problems, and make Adaptive Search practically useless for more structured problems such as TSP and harder constraint problems, which in practice requires extended neighborhoods and much more elaborate transitions than those provided in Adaptive Search.

6.4 Wsat(OIP)

The algorithm Wsat(OIP) developed by Walser [73] is an extension of the SAT local search algorithm Walksat [63] to handle more general integer optimization problems called *over-constrained integer programs* (OIP's). Global constraints are not used in the system.

An OIP consists of hard and soft inequality constraints, wherein the optimization objectives are represented by the soft constraints. If all inequalities are linear an OIP can be formulated in matrix notation as

$$A\mathbf{x} \geq \mathbf{b}, \quad C\mathbf{x} \leq \mathbf{d}, \quad \mathbf{x} \in \mathbf{D}$$

where $A\mathbf{x} \geq \mathbf{b}$ is the hard (regular) constraints, $C\mathbf{x} \leq \mathbf{d}$ is the soft (objective) constraints, and \mathbf{D} is a vector of domains.

Given a tuple $(A, \mathbf{b}, C, \mathbf{d}, \mathbf{D})$, the OIP minimization problem is

$$\begin{aligned} & \min_{\mathbf{x}} \|C\mathbf{x} - \mathbf{d}\| \\ & \text{subject to } A\mathbf{x} \geq \mathbf{b} \\ & \quad \mathbf{x} \in \mathbf{D} \end{aligned}$$

where $\|\mathbf{x}\| := \sum_i \max(0, v_i)$, and the objective is to find a feasible solution with minimal soft constraint violation.

The Wsat(OIP) algorithm basically works as follows:

1. Choose a violated constraint c at random. If both soft and hard constraints are violated, choose a hard constraint with probability p_{hard} .
2. Let S be the variable-value pairs that would improve c 's score. For integer variables, the values at most d steps away are considered.
3. Remove forbidden pairs from S and calculate a total score for each pair in the updated S . Forbidden pairs are typically tabu.
4. The local move that improves the total score most is chosen greedily. If several moves give the same improvement, choose the variable chosen i) least frequently, and ii) longest ago.
5. If the total score cannot be improved, choose with probability p_{noise} a random assignment from S , and with probability $1 - p_{noise}$ the best one.

The hypothetical total score of an assignment \mathbf{x} is calculated as

$$\text{score}(\mathbf{x}) = \|\mathbf{b} - A\mathbf{x}\|_{\lambda} + \|C\mathbf{x} - \mathbf{d}\|.$$

where the norm $\|\mathbf{x}\| = \sum_i \max(0, v_i)$ holds. The vector $\lambda \geq 0$ is used to weight the hard constraints, where $\|\mathbf{x}\|_{\lambda} = \sum_i \lambda_i \max(0, v_i)$. The soft constraints have no weights to keep the score of a feasible solution the same as the value of the cost function for the same solution. The score of a violated constraint α is calculated as $\lambda_{\alpha}(b_{\alpha} - \mathbf{a}_{\alpha}\mathbf{x})$ for a soft constraint, and $\mathbf{c}_{\alpha}\mathbf{x} - d_{\alpha}$ for a hard constraint. A tabu mechanism with tenure size t is used to diversify the search.

6.5 GCSP

The algorithm GCSP, described in articles by Clark et al. [8] and Tomov [71], is the analogy of the GSAT algorithm of Selman, Levesque and Kautz [64] and Gu [27, 28] for general constraint satisfaction. The GCSP algorithm begins by a random generated assignment of values to the variables, and then uses the steepest descent heuristic to find the single variable-value assignment that locally maximizes the number of satisfied constraints. After a fixed number of cycles, search is restarted from a new random assignment. The search continues until we find a solution or we have done a fixed number of restarts.

6.6 The Min-Conflict Heuristic

The *min-conflict heuristic* introduced by Minton et al. [43] is a general purpose heuristic for constraint satisfaction. The general intuition is to resolve conflicts in the current assignment by reassigning labels that are involved in constraints not satisfied.

Definition 42 (Conflict). Given a CSP $\mathcal{P} = (X, D, C)$, where C is a set of binary constraints, and a compound label ℓ , a label $\langle x, v \rangle \in \ell$ is in *conflict* with regard to (X, D, C) and ℓ if x is involved in a violated constraint.

$$\text{CONFLICT}_{\mathcal{P}, \ell}(\langle x, v \rangle) \equiv \langle x, v \rangle \in \ell \wedge \exists c_S \in C : x \in S \wedge \neg \text{SATISFIES}(\ell, c_S)$$

A variable x is in conflict if and only if there is a label $\langle x, v \rangle$ in conflict.

Definition 43 (The Min-Conflicts Heuristic). Select a variable that is in conflict and assign it a value that minimizes the number of conflicts. Break ties randomly.

The min-conflicts heuristic can be used either in a local search framework as the one we described in Section 3.1, or in a hybrid method based on complete backtracking.

Hill-climbing using the min-conflicts was first tried by Minton et al. [43]. The idea is simple. Start with a complete assignment, a hill-climbing algorithm would iteratively apply the heuristic for selecting the new variable to improve. This method can be extended with other techniques as well.

Min-conflicts Informed Backtracking

The min-conflict heuristic can also be applied to a backtracking scheme as described in Minton et al. [43]. Here the heuristic serves to guide the selection of variables and values during search. The basic idea is to use a regular backtracking method, augmented with a variable and value selection scheme which selects variables and values according to the min-conflicts heuristic. Note that a similar approach with a heuristic counting the number of constraint violations would be possible.

MC-log

MC-LOG by Clark et al. [8] is based on the min-conflict hill-climbing method described in Section 6.6, but adds the ability to escape local

minima. Unlike GCSP, MC-LOG does not consider all variables but instead selects randomly a variable in conflict with a constraint. The local neighborhood for this variable consists of alternative values for it. Unlike min-conflicts hill-climbing, MC-LOG does not select the neighbor that minimizes the number of conflicts, but ranks all neighbors according to the number of conflicts, and selects one according to a probabilistic selection function. This function is logarithmic so the 'best' value is chosen most often, but not exclusively as with min-conflicts. More precisely, the i th ranked value where $i = \text{INT}(\log_2(1/r)/w)$, r is a random number uniformly distributed in $[0, 1]$ and w is some fixed weighting.

Weak-commitment

The algorithm *weak commitment* (WC) is an extension of the informed backtracking algorithm introduced by Yokoo in [78]. In the weak-commitment algorithm, all variables are given tentative values as in the informed backtracking algorithm, and variables are added one by one to the consistent partial solution. Where a variable choice is never abandoned in informed backtracking (unless it turns out to be hopeless), in weak commitment we commit to the partial solution as long as it can be extended. When there exists no value for one variable that satisfies all constraints between the partial solution, we instead abandon the whole partial solution, and start constructing a new partial solution from scratch, using the current value assignment as new tentative initial values.

Nogood learning

In [55], Richards and Richards describes the effect of modifying weak-commitment with a nogood learning method called "learning-by-merging", and by adding forward-checking [72]. Also, the result of removing the nogood learning part of WC is demonstrated, somewhat surprisingly, to give better results than the WC with nogood learning on random problem instances of 3-SAT and graph 3-colorability, using constraint checks and steps as measure of performance.

6.7 More Related Work

Global constraints in local search has been investigated by Nareyek, who uses an approach where a selected constraint improves the current assignment in a local manner [45]. He also gives ad hoc costs for some global constraints including `ordered-tasks` and a version of the `serialized` constraint.

Galinier and Hao [20] describes a general constraint solver using local search. The solver handles a small set of global constraints, including `alldifferent`, `capa` and `nbdifferences`, and is based on hill-climbing and tabu search. Galinier and Hao give costs for the constraints used in their work and demonstrate it on some combinatorial problems. They also propose to use the minimum number of variables that need to be modified for a constraint to be satisfied as the cost of the constraint.

Petit, Régim and Bessière use cost-based filtering for systematic constraint programming [50]. The authors also propose two general cost policies for global constraints. The first is to use the minimum number of variables needed to be modified as the cost. The second is to use the number of violated binary constraints that can be used to represent the global constraint as cost. In our work we extend the second approach to constraints that are hard to represent using binary constraints only.

A classification of global constraints as graph properties on structured networks of elementary constraints [5] has previously been done. Beldiceanu [4] gives a framework for describing global constraints as properties of graphs. We discuss this work in Section 4.2.1.

Schaerf [59] experiments with a combination of local search and look-ahead techniques, from the systematic constraint solving area, for scheduling and constraint satisfaction.

Stützle [70] investigates the use of Tabu search [26] and the *min conflicts* heuristic [43] for random CSP's and graph coloring problems. He concludes that Tabu search gives better results for most random problem instances than random walk.

Jussien and Lhomme [32] introduces an algorithm called *path-repair*, based on local search combined with systematic search, operating on partial assignments using filtering and tabu-search to remember conflicts during the search.

Chapter 7

Conclusions

In this chapter we conclude our work and discuss some possible future work.

7.1 Conclusions

In this thesis we have investigated a framework for modelling of global constraints for local search as a cost function on a graph of filter constraints. We have presented a generic approach for the design of global constraints for in local search. We used a representation of a global constraint as a cost function on a structured network of filter constraints, and showed how to efficiently compute a cost for a constraint represented in this way. We also gave algorithms for incremental cost and conflict computation. To demonstrate the generality and usability of our approach we showed how to represent several important global constraints using our method. We believe that this work further increases the usability of constraint-based local search algorithms.

We also proposed a compositional approach for global constraint modelling and implementation, and shown that such an approach is not only feasible but also highly competitive with existing low-level, non-compositional approaches for constraint solving using local search. The main point of a compositional approach for constraint modelling is that it allows great flexibility in experimentation with modification and creation of new constraints at a very low run-time overhead. In compositional constraint design, global constraints are parametrized over three main

properties of their structure; their *graph structure*, their *filter constraint* and their *cost components*. We have also integrated the global constraint composition mechanism into a generic local search library, *Composer*, which provides much help in designing local search algorithms for combinatorial problems. Using the parametric approach at global constraint implementation, we have composed several useful global constraints using *Composer*. We showed that a compositional approach is very competitive with existing methods on two hard problems.

We believe that the results in this thesis clearly shows the benefits of a less monolithic and more modular approach at global constraint design for local search, and that such an approach can be competitive with existing low-level monolithic implementations of global constraints for local search.

To conclude, we think that composed global constraints in local search can significantly increase the usability of local search methods for modelling and solving of hard combinatorial problems in scheduling, planning and other related areas.

7.2 Future Work

We plan to proceed with this work by implementing a fully integrated local search language, capable of modelling the global constraints in this work and more, and to give empirical results showing the accuracy, generality and efficiency of our approach.

Several other future research directions in the area of local search and stochastic methods have also been discussed. We aim to investigate the following topics:

Applying local search to industrial problems We plan to investigate the behaviour of *Composer* on large-scale industrial problems in domains such as dynamic planning and scheduling tasks. This may lead to the development of more powerful generic components and constraints for constraint-based local search.

Investigating exhaustive plateau search As far as we know, no complete method for exploring plateaus except breadth-first search has been tried. Experiments with other search techniques such as depth-first search and A*-search might be more successful at exploration than breadth-first search.

Hybrid local search A number of approaches for combining heuristics and local search with traditional systematic techniques has been done [59, 43, 39, 32, 31, 51, 55]. We plan to investigate if the work we have done on cost calculations and improvement heuristics can be used in a systematic constraint solver.

Bibliography

- [1] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990.
- [2] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003.
- [3] Nicolas Beldiceanu. Personal communication.
- [4] Nicolas 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] Nicolas Beldiceanu. Global constraints as graph properties on structured networks of elementary constraints of the same type. Technical Report 2000/01-SE, SICS, 2000.
- [6] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in chip. *Mathematical Computation Modelling*, 20(12):97–123, 1994.
- [7] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *AAAI/IAAI, Vol. 1*, pages 332–337, 1996.
- [8] David A. Clark, Jeremy Frank, Ian P. Gent, Ewan MacIntyre, Neven Tomov, and Toby Walsh. Local search and the number of solutions. In *Principles and Practice of Constraint Programming*, pages 119–133, 1996.

- [9] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *SAGA*, pages 73–90, 2001.
- [10] Steven. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, Shaker Heights, Ohio, pages 151–158, 1971.
- [11] G. A. Croes. A method for solving traveling salesman problem. *Operations Research*, pages 791–812, 1958.
- [12] Andrew Davenport and Edward Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP)*, pages 345–357, april 1999.
- [13] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *European Conference on Artificial Intelligence*, pages 290–295, 1988.
- [14] Cecilia Ekelin and Martin Olovsson. Local search and configuration problems. Master’s thesis, Uppsala University, 1996.
- [15] Mark S. Fox. *Constraint-directed Search: A Case Study of Job-Shop Scheduling*. Morgan Kaufmann Publishers, 1987.
- [16] Jeremy Frank. Learning short-term weights for GSAT. In *IJCAI (1)*, pages 384–391, 1997.
- [17] Jeremy Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.
- [18] Philippe Galinier and Jin-Kao Hao. Tabu search for maximal constraint satisfaction problems. In *Principles and Practice of Constraint Programming*, pages 196–208, 1997.
- [19] Philippe Galinier and Jin-Kao Hao. Solving the progressive party problem by local search. In I.H. Osman S. Voss, S. Martello and C. Roucairol, editors, *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, chapter 29, pages 418–432. Kluwer Academic Publishers, 1998.

- [20] Philippe Galinier and Jin-Kao Hao. A general approach for constraint solving by local search. In *Proc. CP-AI-OR'00*, Paderborn, Germany, March 2000.
- [21] Ian P. Gent. Two results on car sequencing problems. Technical Report Technical Report APES-02-1998, University of St Andrews, School of Computer Science, APES Group, 1998.
- [22] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *AAAI*, pages 28–33, 1993.
- [23] Ian P. Gent and Toby Walsh. The search for satisfaction. Technical report, Dept. of Computer Science, University of Strathclyde, 1999.
- [24] Ian P. Gent, Toby Walsh, and Bart Selman, 2001. CSPLib: a problem library for constraints.
<http://www-users.cs.york.ac.uk/~tw/csplib>.
- [25] Fred Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operations Research*, pages 5:533–549, 1986.
- [26] Fred Glover and Manuel Laguna. Tabu search. In Colin R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Optimization*, chapter 3, pages 70–150. McGraw-Hill, 1995.
- [27] Jun Gu. Efficient local search of very large-scale satisfiability problems. *SIGART Bulletin*, 3(1):8–12, 1992.
- [28] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: a survey. In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, volume 35, pages 19–152, 1997.
- [29] Steven Hampson and Dennis Kibler. Plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. In *Workshop Notes: 2nd DIMACS Challenge*, 1993.
- [30] Holger H. Hoos and Thomas Stutzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

- [31] Yuejun Jiang, Thomas Richards, and Barry Richards. No-good backmarking with min-conflict repair in constraint satisfaction and optimization. In *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA, 1994.
- [32] Narendra Jussien and Olivier Lhomme. The path-repair algorithm. In Suzanne Heipcke and Mark Wallace, editors, *CP99 Post-conference workshop on Large scale combinatorial optimisation and constraints*, volume 4 of *Electronic Notes in Discrete Mathematics*. Elsevier Science, 2000.
- [33] Olli 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] Per Kreuger and Martin Aronsson. A constraint model for a cyclic time personnel routing and scheduling problem. Technical Report T2001:04, SICS, September 2001.
- [35] Per Kreuger, Mats Carlsson, Thomas Sjöland, and Emil Åström. Sequence dependent task extensions for trip scheduling. Technical Report T2001:04, SICS, May 2001.
- [36] Tracy Larrabee and Yumi Tsuji. Evidence for a satisfiable threshold for random 3cnf formulas. In *In Proceedings AAAI Spring Symposium*, 1993.
- [37] Jimmy H. M. Lee, Ho-Fung Leung, and Hon-Wing Won. Performance of a comprehensive and efficient constraint library based on local search. In *Australian Joint Conference on Artificial Intelligence*, pages 191–202, 1998.
- [38] Kim Marriott and Peter J. Stuckey. *Programming with Constraints, An Introduction*. MIT Press, 1998.
- [39] Harald Meyer auf'm Hofe. Finding regions for local repair in partial constraint satisfaction. In *KI-98: Advances in Artificial Intelligence*, pages 57–68. Springer Verlag, 1998.

- [40] Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In *Principles and Practice of Constraint Programming*, pages 237–251, 1997.
- [41] Laurent Michel and Pascal Van Hentenryck. Localizer++: An open library for local search. Technical Report CS-01-02, Brown University, January 2001.
- [42] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 2002.
- [43] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1–3):161–205, 1992.
- [44] Paul Morris. The breakout method for escaping from local minima. In *National Conference on Artificial Intelligence*, pages 40–45, 1993.
- [45] Alexander Nareyek. Using global constraints for local search. In *Proc. DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization*, pages 1–18, 1998.
- [46] Rémi onasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *NATURE: Nature*, 400, 1999.
- [47] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.
- [48] Claude Le Pape. Constraint propagation in planning and scheduling. Technical report, Robotics Laboratory, Department of Computer Science, Stanford University, Palo Alto, CA, 1991.
- [49] Bruce D. Parrello and Waldo C. Kabat. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.

- [50] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. *Lecture Notes in Computer Science*, 2239:451–??, 2001.
- [51] Dionisios Pothos and Barry Richards. An empirical study of min-conflict hill climbing and weak commitment search. In Cassis, editor, *Proceedings of CP-95 Workshop: Studying and Solving Really Hard Problems*, pages 140–146, 1995.
- [52] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.
- [53] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on AI (AAAI/IAAI'96)*, volume 1, pages 209–215, Portland, August 1996.
- [54] Jean-Charles Regin and Jean-Francois Puget. A filtering algorithm for global sequencing constraints. In *Principles and Practice of Constraint Programming*, pages 32–46, 1997.
- [55] E. Thomas Richards and Barry Richards. Nogood Learning for Constraint Satisfaction. In *Proceedings Second International Conference on Constraint Programming Workshop on Constraint Programming Applications*, 1996.
- [56] Norman Sadeh. *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1991.
- [57] Norman Sadeh, Shinichi Otsuka, and Robert 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*, Chambéry, France, Aug 1993.
- [58] Hani El Sakkout and Mark 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.

- [59] Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc. of the 15th International Joint Conf. on Artificial Intelligence (IJCAI-96)*, pages 1254–1259, Nagoya, Japan, 1997. Morgan Kaufmann.
- [60] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *AAAI/IAAI*, pages 297–302, 2000.
- [61] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *IJCAI*, pages 334–341, 2001.
- [62] Bart Selman and Henry Kautz. Domain-independent extension to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI-93, 13th International Joint Conference on Artificial Intelligence*, pages 290–295, Sidney, AU, 1993.
- [63] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, pages 337–343. AAAI Press, 1994.
- [64] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.
- [65] Yi Shang and Benjamin W. Wah. A discrete lagrangian based global search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–99, 1998.
- [66] Stephen Smith. *Opis: A methodology and architecture for reactive scheduling*, 1994.
- [67] Stephen Smith. Reactive scheduling systems. In D.E. Brown and W.T. Scherer, editors, *Intelligent Scheduling Systems*. Kluwer Press, 1995.
- [68] Stephen Smith, N. Keng, and K. Kempf. Exploiting local flexibility during execution of pre-computed schedules, 1990.

- [69] Rok Susic and Jun Gu. 3,000,000 queens in less than one minute. In *SIGART Bulletin*, volume 2,2, pages 22–24, April 1991.
- [70] Thomas Stützle. Tabu search and iterated local search for constraint satisfaction problems.
- [71] Neven Tomov. Hill-climbing heuristics for solving constraint satisfaction problems, 1994.
- [72] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1994.
- [73] Joachim Paul Walser. *Integer Optimization by Local Search*, volume 1637 of *Lecture Notes in Artificial Intelligence*. Springer, 1999.
- [74] Zhe Wu and Benjamin W. Wah. Solving hard satisfiability problems: A unified algorithm based on discrete lagrange multipliers. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 210–217. IEEE, November 1999.
- [75] Zhe Wu and Benjamin W. Wah. Trap escaping strategies in discrete lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *AAAI/IAAI*, pages 673–678, 1999.
- [76] Zhe Wu and Benjamin W. Wah. An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems. In *AAAI/IAAI*, pages 310–315, 2000.
- [77] Daniel Yellin and Robert E. Strom. INC: a language for incremental computations. *SIGPLAN Not.*, 23(7):115–124, 1988.
- [78] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *AAAI, Vol. 1*, pages 313–318, 1994.

