Consistency and the Quantified Constraint Satisfaction Problem

A thesis submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

DOCTOR OF PHILOSOPHY

by

Peter Nightingale

School of Computer Science

University of St Andrews

August 2007

Abstract

Constraint satisfaction is a very well studied and fundamental artificial intelligence technique. Various forms of knowledge can be represented with constraints, and reasoning techniques from disparate fields can be encapsulated within constraint reasoning algorithms. However, problems involving uncertainty, or which have an adversarial nature (for example, games), are difficult to express and solve in the classical constraint satisfaction problem. This thesis is concerned with an extension to the classical problem: the Quantified Constraint Satisfaction Problem (QCSP). QCSP has recently attracted interest. In QCSP, quantifiers are allowed, facilitating the expression of uncertainty.

I examine whether QCSP is a useful formalism. This divides into two questions: whether QCSP can be solved efficiently; and whether realistic problems can be represented in QCSP. In attempting to answer these questions, the main contributions of this thesis are the following:

- the definition of two new notions of consistency;
- four new constraint propagation algorithms (with eight variants in total), along with empirical evaluations;
- two novel schemes to implement the pure value rule, which is able to simplify QCSP instances;
- a new optimization algorithm for QCSP;
- the integration of these algorithms and techniques into a solver named Queso;
- and the modelling of the Connect 4 game, and of faulty job shop scheduling, in QCSP.

These are set in context by a thorough review of the QCSP literature.

Acknowledgements

I would like to thank all members of the CPPod and APES research groups for their input. In particular, I would like to thank my supervisor Ian Gent and second supervisor Ian Miguel for all their help and support. I would also like to thank all those I have collaborated with, in particular Ian Gent, Kostas Stergiou and Andrew Rowley. I would like to thank my parents for all their support over the last three years. This work was supported by an EPSRC doctoral training grant.

Contents

Chapter 1. Introduction 11

1.1. Solving The Classical Constraint Satisfaction Problem 12

1.2. Limitations of CSP 16

1.3. The Quantified Constraint Satisfaction Problem 17

1.4. Contributions of this thesis 19

1.5. Thesis outline 21

Chapter 2. Literature review 25

2.1. Constraint Programming 26

2.2. QBF 38

2.3. QCSP 41

2.4. Other formalisms 61

2.5. Summary 64

Chapter 3. A search framework for QCSP 65

3.1. Introduction 65

3.2. Definitions 66

3.3. Search algorithms 83

3.4. Managing domains and the constraint queue 91

3.5. The pure value rule 97

3.6. Backjumping and learning 102

3.7. Queso solver implementation 103

3.8. Encodings 106

3.9. Summary 116

Chapter 4. Strong consistency for arbitrary constraints 117
4.1. Introduction 117
4.2. Motivating examples 117
4.3. An algorithm for enforcing SQGAC 120
4.4. A general schema for enforcing WQGAC 124
4.5. Space and time complexities 134
4.6. Empirical evaluation 136
4.7. Summary 155

Chapter 5. Logical constraints 157
5.1. Introduction 157
5.2. A propagation algorithm for reified disjunction 162
5.3. Empirical evaluation 173
5.4. Summary 180

Chapter 6. Bounds consistency and the sum constraint 181
6.1. Introduction 181
6.2. Definition of bounds consistency 182
6.3. Propagation rules 185
6.4. Implementation 192
6.5. Summary 195

Chapter 7. Application of QCSP to factory scheduling 197
7.1. Introduction 197
7.2. Modelling Faulty Job Shop Scheduling in QCSP 203
7.3. Empirical evaluation 213
7.4. Summary 218

Chapter 8. Conclusion 221
8.1. Summary 221
8.2. Future work 224

8.3. Final words 225

Bibliography 229

CHAPTER 1

Introduction

Constraints are a natural way to represent many different computational problems. This thesis is about extending classical constraint reasoning with quantifiers. I will first introduce the classical constraint satisfaction problem, then move on to the extended problem.

A constraint is simply a relation over a set of variables. Constraints often take the form of equalities, inequalities and logical relations. To illustrate, figure 1 shows a simple puzzle, where two six-digit numbers (DONALD and GERALD) are added together to form another six-digit number (ROBERT). Each letter A, B, D, E, G, L, N, O, R and T represents a distinct digit $0 \ldots 9$. The puzzle can be represented with the expressions below, given by Bessière and Régin [**13**].

$$\begin{aligned} 100000 \times \mathrm{D} + 10000 \times \mathrm{O} + 1000 \times \mathrm{N} + 100 \times \mathrm{A} + 10 \times \mathrm{L} + \mathrm{D} \\ +100000 \times \mathrm{G} + 10000 \times \mathrm{E} + 1000 \times \mathrm{R} + 100 \times \mathrm{A} + 10 \times \mathrm{L} + \mathrm{D} \\ = 100000 \times \mathrm{R} + 10000 \times \mathrm{O} + 1000 \times \mathrm{B} + 100 \times \mathrm{E} + 10 \times \mathrm{R} + \mathrm{T} \\ \text{and allDifferent}(\mathrm{A, B, D, E, G, L, N, O, R, T}) \end{aligned}$$

This representation of the puzzle illustrates the main concepts of constraint programming. A, B, D, E, G, L, N, O, R and T are variables, each with initial domain $0 \ldots 9$. There are two constraints, one representing the sum and the other representing that the variables each take a different value. A solution is a function mapping each variable to a value in its initial domain, such that all constraints are satisfied. The solution to this puzzle is A=4, B=3, D=5, E=9, G=1, L=8, N=6, O=2, R=7, T=0.

Many different kinds of information, from many areas of artificial intelligence, can be represented with constraints. The following are simple examples: one variable is less than another;

```
  D O N A L D
 + G E R A L D
 -------------
 = R O B E R T
```

FIGURE 1. Alphametic problem

a set of variables must take distinct values; task A must be scheduled before task B; two objects may not occupy the same space. It is this flexibility which allows constraints to be applied to many theoretical, industrial and mathematical problems.

Constraints are *declarative* — the statement of the problem and the algorithms used to solve it are separated. This is an attractive feature of constraints, since it can reduce the human effort required to solve a problem. Various general purpose and specialized algorithms exist for solving systems of constraints. A great variety of problems can be expressed with constraints. The following list of subject areas was taken from CSPLib [**63**]:

- Scheduling (e.g. job shop scheduling [**77**]),
- Design, configuration and diagnosis (e.g. template design [**80**]),
- Bin packing and partitioning (e.g. social golfer problem [**59**]),
- Frequency assignment (e.g. the golomb ruler problem [**89**]),
- Combinatorial mathematics (e.g. balanced incomplete block design [**42**]),
- Games and puzzles (e.g. maximum density still life [**90**]),
- Bioinformatics (e.g. discovering protein shapes [**67**]).

1.1. Solving The Classical Constraint Satisfaction Problem

The classical constraint satisfaction problem (CSP) has a finite set of variables, each with a finite domain, and a set of constraints over those variables. A solution to an instance of CSP is an assignment to each variable, such that all constraints are simultaneously *satisfied* — that is, they are all true under the assignment. Solvers typically find one or all solutions, or prove there are no solutions. The decision problem ('does there exist a solution?') is NP-complete [**6**], therefore there is no known polynomial-time procedure to find a solution.

Constraint programming includes a great variety of domain specific and general techniques for solving systems of constraints. This section is necessarily restricted to the most common techniques, and only those used on classical CSP as described above. Since CSP is NP-complete, most algorithms are based on a search which potentially explores an exponential number of nodes. Since the applications of CSP are difficult combinatorial problems, efficiency is very important, and this has led to a great deal of effort to reduce the number of search nodes.

The most common technique is to interleave splitting and propagation. Splitting is the basic operation of search, and propagation simplifies the CSP instance. Apt views the solution process as the repeated transformation of the CSP until a solution state is reached [**6**]. In this view, both splitting and propagation are transformations, where propagation simplifies the CSP by removing values which cannot take part in any solution. A splitting operation transforms a CSP instance into two or more simpler CSP instances, and by recursive application of splitting any CSP can be solved.

There are a variety of ways of employing splitting and propagation [**6**]. Avoiding details, and assuming all constraints involve just two variables, I sketch three methods here.

Forward Checking: Reasoning is performed on individual constraints from instantiated variables to uninstantiated variables, reducing their domains. If a variable domain becomes empty, the simplified instance is false.

Partial Lookahead: This is a stronger version of forward checking, which additionally performs directional consistency: for all pairs of variables x_i and $x_{j>i}$, if a value for x_i is incompatible with all remaining values of x_j it is removed. This is repeated to exhaustion.

Maintaining Arc-Consistency: This is stronger than partial lookahead since it performs *arc-consistency* reasoning to exhaustion on all individual constraints. Informally, a domain value is arc-inconsistent iff it is incompatible with all remaining values of some other variable.

Maintaining Arc-Consistency (MAC) can be trivially generalized to constraints over more than two variables. MAC is used by commercial constraint programming toolkits such as ILOG Solver

[**1**] and Eclipse [**3**], as well as research systems such as Minion [**50**], Gecode [**88**] and Choco [**68**]. It is typically the most robust and efficient of these three methods, and has enjoyed the most attention.

In MAC, each constraint has an associated definition of *consistency*, for example bounds consistency for numerical constraints. If the constraint is consistent, then no reasoning is required on it unless it becomes inconsistent by the modification of the domain of one of its variables. At this point, a propagation algorithm would be executed on the constraint, which returns the constraint to consistency by removing the necessary values from variable domains.

A small example of MAC. Consider the CSP instance below, representing the pigeonhole problem for three variables. All three variables must take a distinct value, and there are two values in total, so the CSP instance is unsatisfiable.

$$A, B, C \in \{1, 2\} \; : \; A \neq B, \; A \neq C, \; B \neq C$$

Propagation is typically applied to individual constraints. Each type of constraint has a propagation algorithm and these are applied iteratively. In this case, it is not possible to remove any values from the variable domains by reasoning on any constraint individually. Therefore we search by setting the variable A to 1. This is referred to as splitting for $A = 1$.

$$A = 1, \; B, C \in \{1, 2\} \; : \; A \neq B, \; A \neq C, \; B \neq C$$

At this point, propagating $A \neq B$ and $A \neq C$ would remove value 1 from the domains of B and C respectively. Propagating $B \neq C$ would remove value 2 from B or C — either way, the domain of a variable is empty and this indicates the CSP instance has no solution.

The most recent search decision did not lead to a solution, so it is reversed and A is set to 2.

$$A = 2, \; B, C \in \{1, 2\} \; : \; A \neq B, \; A \neq C, \; B \neq C$$

Again, propagation empties the domain of some variable, and the CSP instance is unsatisfiable. There are no remaining values to try for variable A, and there were no search decisions made before searching on A, so the original CSP instance is unsatisfiable. This simple example illustrates the use of depth-first backtracking search with MAC.

Constraint solvers. Figure 2 is a simple representation of how many constraint solvers work. The search element is typically depth-first chronological backtracking by default, although a solver will often allow different search algorithms to be programmed. When searching, a variable and value must be selected. This can be done statically or with a dynamic heuristic.

After each search operation, the relevant constraints are propagated. The key to the success of constraint programming is efficient and effective propagation. The diagram represents a constraint queue, containing constraints which may be inconsistent, because some variable domain has changed. Whenever a variable domain is changed, either by a search operation or by propagation, the relevant constraints are added to the queue for propagation. The algorithm iteratively propagates constraints until the queue is empty, which indicates that all constraints are consistent.

The queue system can be seen as a collaborative solver, where each constraint propagator contributes to simplifying the problem, and the whole is greater than the sum of its parts. Common propagators include:

- allDifferent for various different consistency definitions [**71**,**83**];
- bounds consistency propagators for $a + b = c$, $ab = c$ and other numerical constraints (explained by Marriott and Stuckey [**76**]), which update only the upper and lower bounds of variables;
- generalized arc consistency propagators for logical constraints $a \vee b \Leftrightarrow c$, $\neg a \Leftrightarrow b$, $a \wedge b \Leftrightarrow c$, found in common solvers and described by Apt [**1**,**3**,**6**,**50**,**68**];
- generalized arc consistency propagators for arbitrary constraints of any arity [**13**,**14**,**78**].

Heuristics are important to reduce search effort, because the application of a suitable heuristic can reduce the number of nodes hugely, compared to a static ordering. Russell and Norvig cover three heuristics for CSP:

- *most-constrained-variable*, which selects the variable with the lowest number of remaining values, picking a value with a static order,

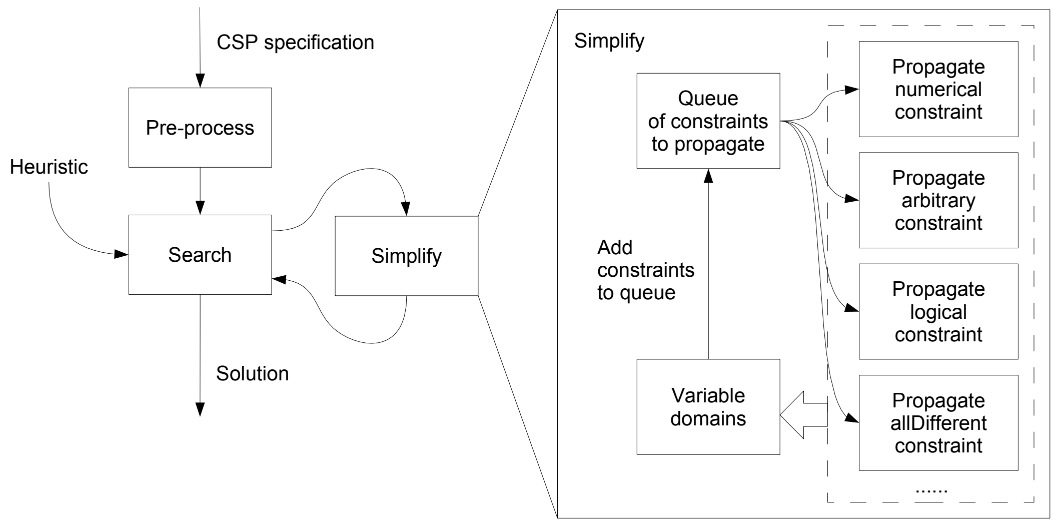

CSP specification
Pre-process
Heuristic
Search
Simplify
Solution
Simplify
Queue of constraints to propagate
Add constraints to queue
Variable domains
Propagate numerical constraint
Propagate arbitrary constraint
Propagate logical constraint
Propagate allDifferent constraint
......

FIGURE 2. Overview of a MAC constraint solver

- *most-constraining-variable* which selects the variable involved in the largest number of constraints on other unassigned variables, and
- *least-constraining-value* which chooses a value which rules out the smallest number of values in variables connected to the current variable by constraints. This is applied after selecting a variable by some means [**86**].

Many problems have *symmetries* inherent in them, for example in a vehicle routing problem, two vehicles with the same starting point and same capabilities can be swapped in any solution, yielding another solution [**32**, **48**, **91**]. These can be exploited in various ways and there are a number of different definitions of symmetry, and related concepts such as conditional symmetry (which emerges if some condition is satisfied [**49**]).

1.2. Limitations of CSP

Many useful problems can be expressed as a CSP. However, some real life problems are not fully specified at solution-time, or even part way through executing the solution. There is often ample time for computation before action is required, but while executing the solution there is often no time to re-solve or re-optimize according to the actual environment. These problems are typically not representable compactly as a CSP.

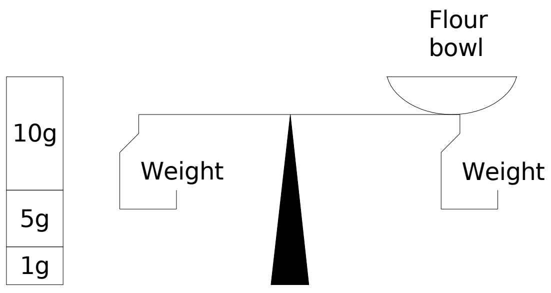


FIGURE 3. Baker's puzzle

For example, factory scheduling with uncertainty in the durations of tasks or with possible faults can be approached by constructing a single robust schedule, or by generating multiple schedules or schedule fragments for different scenarios [**33**]. The first approach could be represented as a constraints optimization problem, but the second approach would require a more expressive language. In CSP, each schedule (fragment) would have to be represented explicitly which could be very costly in space if there are a large number of scenarios.

Such problems can be modelled compactly in the Quantified Constraint Satisfaction Problem (QCSP). QCSP is a challenging PSPACE-complete problem [**23**], which is interesting theoretically and also relevant to solving PSPACE problems arising in AI, such as reasoning with uncertainty, model checking and adversarial games.

1.3. The Quantified Constraint Satisfaction Problem

The classical CSP has been extended in many ways, to encompass problems which cannot be efficiently represented in the classical problem. In this thesis I explore one such generalization. The Quantified Constraint Satisfaction Problem (QCSP) extends CSP by allowing the quantification of each variable with the existential ($\exists$) or universal ($\forall$) quantifiers.

Consider the baker's puzzle: a baker needs to purchase four weights of different sizes in the range $1 \dots 40$, such that it is possible to weigh out all integral quantities of flour in the range $1 \dots 40$, using a balance (figure 3). To weigh out a quantity of flour, each weight can be placed on either side of the balance, or not used.

This puzzle can be written in first-order logic as follows. The w variables represent the masses of the four weights. f represents the mass of the flour, and each integral value from 1 to 40 must be covered. The c variables represent how a weight is used. $c_1 = 0$ if w_1 is not used on the balance. with 1 representing that it is on the balance, opposite the flour.

$$\begin{aligned}
\exists w_1 \exists w_2 \exists w_3 \exists w_4\ w_1, w_2, w_3, w_4 \in \{1, 2, \ldots, 40\} \\
\wedge [\forall f\ f \in \{1, 2, \ldots, 40\} \\
\Rightarrow [\exists c_1 \exists c_2 \exists c_3 \exists c_4\ c_1, c_2, c_3, c_4 \in \{-1, 0, 1\} \\
\wedge c_1 w_1 + c_2 w_2 + c_3 w_3 + c_4 w_4 = f]]
\end{aligned}$$

QCSP allows quantifiers in prenex form, but not infinite domains. Each variable has an associated initial domain. The puzzle can be expressed in QCSP as follows.

$$\begin{aligned}
\exists w_1, w_2, w_3, w_4 \in \{1, 2, \ldots, 40\}, \forall f \in \{1, 2, \ldots, 40\}, \exists c_1, c_2, c_3, c_4 \in \{-1, 0, 1\}\ : \\
c_1 w_1 + c_2 w_2 + c_3 w_3 + c_4 w_4 = f
\end{aligned}$$

The puzzle could be expressed in CSP by duplicating the c variables, and the constraint, for each value of f. Even for this tiny example, the CSP representation would be approximately 40 times larger than the QCSP representation. It is suspected, but not proven, that NP $\subsetneq$ PSPACE. If this inclusion is true, then any encoding from QCSP to CSP could not be polynomial in size. The solution (which is unique up to symmetry) to the puzzle is $\{1, 3, 9, 27\}$, which are all powers of three.

1.3.1. QCSP and uncertainty. The QCSP can be used to model problems containing uncertainty, in the form of the universal variables which have a finite domain but whose value is unknown at solution time. Therefore a QCSP solver finds solutions suitable for each value of these variables.

Kenyon and Sellmann [**65**] give an example. Consider a delivery company that, every night, solves a vehicle routing problem to schedule the deliveries for the next day in a road network with some unreliable links. They tackle this problem by generating a set of optimized solutions such that when the uncertainty is resolved (the morning when the trucks are loaded), one of the solutions in the set is near optimal for the actual problem. This example makes two assumptions: that a road link is busy but never impassable (i.e. a solution cannot become infeasible because of an impassable road) and that road conditions can be reasonably predicted at the beginning of the day. Using a QCSP model, both these assumptions can be removed at the cost of higher computational complexity.

Let L be the loading configuration of the trucks, R_t be a routing of all trucks during time interval t, and D_t be data about the roads during time interval t. If we have two time intervals, 0 and 1, the QCSP instance could be $\forall D_0 \exists L, R_0 \forall D_1 \exists R_1 \;\; : \;\;$ feasible(D_0, R_0, D_1, R_1), which reads: 'for all D_0, there exists L and R_0, such that for all D_1 there exists R_1 such that the routing is feasible'. At the start of the second time period, the company would communicate to its drivers their route for the second half of the day, based on up-to-date road information. Solving with a branch and bound procedure would allow optimization of the route. The solver would produce a plan which branches for all values of D_0, and later for all values of D_1 hence there are (at most) $|D_0| \times |D_1|$ solutions for R_1. The modeller would have to be careful not to include unlikely contingencies in D_t (e.g. more than one road impassable) because this would cause excessive branching. Online repair could be used for those unlikely contingencies. This scheme has the advantage that the second stage (R_1) can be optimized just as effectively as the first, with up-to-date road information.

1.4. Contributions of this thesis

The contributions of this thesis can be divided into two areas. The first is a significant development of algorithms to solve QCSP. These include four novel constraint propagation algorithms (or eight variants of algorithms in total), procedures to perform search and optimization, and also novel procedures to apply the pure value rule — a form of local reasoning which is able to reduce the domains of universal variables. These algorithms compare favourably to existing methods both analytically and experimentally, in some cases being orders of magnitude faster.

The second area is modelling problems in QCSP. There are currently no models of realistic problems in the literature [**12**]. The major contribution here is a model of job shop scheduling, where machines may require periods of servicing with a certain probability. This model is specifically designed for the algorithms described herein, to ensure that it can be solved efficiently. Connect 4 and Noughts and Crosses are also modelled, each in two ways. All the models exhibit a novel feature, which allows the pure value rule to prune universal variables.

Specifically, the individual contributions are as follows.

- The enhanced log encoding of binary QCSP into QBF, which significantly outperforms the previous best encoding.
- The hidden variable encoding of arbitrary QCSP into binary QCSP.
- The definition of the non-binary pure value rule which is able to prune universal variables.
- Two general schemes for applying the pure value rule, both of which are shown to be effective in later experiments.
- The SQGAC-propagate algorithm which enforces a strong consistency (SQGAC [**21**]) on quantified constraints.
- The definition of the WQGAC consistency notion.
- The development of the WQGAC-Schema algorithm from GAC-Schema [**13**].
- Adaptation of the positive and predicate instantiations of GAC-Schema for WQGAC-Schema, allowing the algorithm to enforce WQGAC on constraints expressed as a list of tuples (positive), or constraints expressed compactly with a program (predicate).
- The development of the Next-Difference list as a much more efficient alternative to the positive instantiation. This is also applicable in the CSP context, and is effective there [**51**].

- SQGAC-propagate, and WQGAC-Schema with two variants of Next-Difference list and the positive and predicate instantiations, are experimentally compared amongst themselves and with QBF and binary QCSP solvers.
- A linear-time propagation algorithm for logic (reified disjunction) constraints, which enforces SQGAC. This is instantiated to handle two different types of constraint.
- The reified disjunction is experimentally compared with the algorithms above using Connect 4. It is also compared with the work of Bordeaux [**19**] and found to be more effective in both cases.
- The definition of Qbounds($\mathbb{R}$) consistency, an adaptation of bounds($\mathbb{R}$) consistency in CSP.
- A propagation algorithm for long weighted sum constraints which enforces Qbounds($\mathbb{R}$) consistency. This is compared against SQGAC-propagate and found to be much more efficient.
- Models of Connect 4 and Noughts and Crosses, which are used in experiments to compare propagation algorithms. These models take advantage of the pure value rule in a novel way to prune cheating moves.
- A model of a job shop scheduling problem with probabilistic machine faults. All possible worlds within a probability bound are scheduled. This is a proof of concept for applying QCSP to real scheduling problems.

1.5. Thesis outline

The aim of this thesis is to investigate the usefulness of QCSP as a formalism for reasoning with uncertainty. This breaks down into two main questions: can QCSP be solved efficiently, and can problems containing uncertainty be modelled effectively in QCSP? To address the first question, I extend the existing work on QCSP, developing new algorithms and methods. These are evaluated using games and random QCSP instances. The second question is whether problems containing uncertainty can be modelled effectively. To address this, I have modelled a factory scheduling problem, along with Connect 4[1] and noughts and crosses. Lessons learned from this are explored in the relevant chapters.

The primary motivation for investigating QCSP is that it is a natural generalization of CSP. CSP has been applied very successfully to many useful problems, and QCSP extends it in such a way that the powerful propagators of CSP are still applicable to constraints which contain no universal variables. Therefore all the power of CSP is available in a system which also supports reasoning with uncertainty.

This thesis has a particular emphasis on constraint propagation, as a promising way of attacking QCSP instances. Chapters 4, 5 and 6 present propagation algorithms for various constraints. These are developed to take advantage of the quantifier prefix and therefore to do stronger reasoning than their CSP counterparts when applied to constraints with universal variables.

Another form of simplification, the pure value rule, is implemented using constraint propagators. Two schemes to achieve this are developed in chapter 3. The pure value rule is able to remove values from universal variables, and it has an important part in solving the Connect 4 and factory scheduling examples.

The structure of the thesis is outlined below.

Chapter 2	Review of the most relevant literature.
Chapter 3	Development of the underlying definitions related to QCSP, with search algorithms directly based on those definitions, and the pure value rule which also simplifies the instance based on local reasoning.
Chapter 4	Presents propagation algorithms for arbitrary constraints of any arity, and evaluates them experimentally, using games and random instances.
Chapter 5	Presents propagation algorithms for logical constraints, and evaluates them experimentally with the Connect 4 game.
Chapter 6	Presents a definition of bounds consistency in QCSP, along with a propagation algorithm for the sum constraint.

[1]Connect 4 has been suggested as a grand challenge for QBF by Toby Walsh at the SAT-2003 conference, in his talk ‘Challenges for SAT and QBF’ [**100**]. Gent and Rowley have developed an encoding of Connect 4 in QBF [**46**].

Chapter 7	Develops a QCSP model for a factory scheduling problem with faulty machines, using a probability bounding approach to exclude highly improbable fault scenarios.
Chapter 8	Conclusions and future work.

CHAPTER 2

Literature review

The work in this thesis depends on, and is inspired by, various publications in the areas of constraint programming, and quantified Boolean formulae, as well as the limited literature about QCSP. I will also cover other extensions to constraint programming which are related to QCSP, to set this thesis in context.

Figure 4 summarizes the relationship among six combinatorial problems. There are three problems where all variables are existentially quantified, namely SAT, binary CSP and constraint programming (CSP), which are all NP-complete, and their quantified PSPACE-complete equivalents which are called Quantified Boolean Formulae (QBF), binary QCSP and (non-binary) QCSP respectively. QBF and binary QCSP are both restricted cases of QCSP.

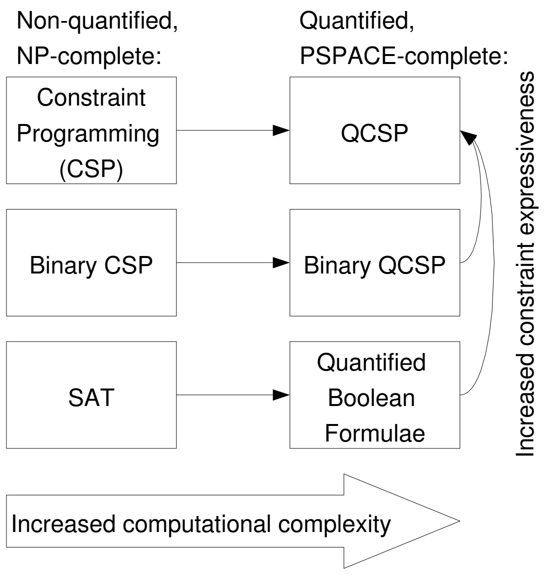

Non-quantified, NP-complete:
Quantified, PSPACE-complete:
Constraint Programming (CSP)
QCSP
Binary CSP
Binary QCSP
SAT
Quantified Boolean Formulae
Increased constraint expressiveness
Increased computational complexity

FIGURE 4. Six combinatorial problems

This literature review is divided into three sections. Firstly, I review a small amount of the large body of work on constraint programming in section 2.1. In section 2.2, some key techniques for QBF are reviewed. In section 2.3, I will look at the existing work on QCSP with both binary and non-binary constraints. Each of these three areas is broadly divided into search and propagation.

2.1. Constraint Programming

Apt provides neat definitions of the central concepts in classical constraint programming [**6**]. I adapt them slightly to match the usual notation for QCSP. I also restrict the domains to be finite subsets of the integers, without loss of generality. In this thesis, Finite Constraint Satisfaction Problem (CSP) refers to the problem defined below, and constraint programming to the whole process of representing a problem in CSP then solving it using domain specific or general means. This definition of CSP deliberately excludes any problems with infinite domains.

DEFINITION 2.1.1. Finite Constraint Satisfaction Problem (CSP)

A CSP is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$. It consists of n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$ and n initial domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ where $D_i \subsetneq \mathbb{Z}$, $|D_i| < \infty$ is the finite set of all potential values of x_i, and a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints.

To understand the semantics of the CSP, it is necessary to define constraints.

DEFINITION 2.1.2. CSP Constraint

Within CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a constraint $C_k \in \mathcal{C}$ consists of a sequence of $m > 0$ variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_m} \rangle$ with respective domains $\mathcal{D}_k = \langle D_{k_1}, \ldots, D_{k_m} \rangle$ s.t. $\mathcal{X}_k$ is a subsequence of $\mathcal{X}$ and $\mathcal{D}_k$ is a subsequence of $\mathcal{D}$. C_k has an associated set $C_k^S \subseteq D_{k_1} \times \cdots \times D_{k_m}$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_k$.

A solution to a CSP is an assignment to all variables in the CSP, such that every constraint is *satisfied*: under the assignment, $\forall k : \langle x_{k_1}, \ldots, x_{k_m} \rangle \in C_k^S$. The problem of deciding whether a CSP has a solution (the decision problem) is NP-complete [**38**].

In the context of QCSP, the equivalent concepts to these are developed in detail in chapter 3. CSP is also re-defined as a subset of QCSP.

2.1.1. Search. Search involves repeatedly transforming the CSP instance into two or more CSP instances, and solving these recursively. The original instance is satisfiable iff one or more of the transformed instances are satisfiable. The transformation is known as branching or splitting, and usually involves partitioning the domain of a variable. The variable to branch on is decided by a static ordering or some dynamic heuristic like *most-constrained-variable* (smallest domain first). Van Beek identifies three popular branching strategies [**85**]. He also observes that these are identical if the domains are binary.

(1) Enumeration. The variable x is instantiated to each value in its domain in turn. For each value a, an instance is created with additional constraint $x = a$, and it is solved recursively. The order in which the values are used can be decided by a dynamic heuristic.
(2) Binary branching. The variable x is instantiated to some value a in its domain. Two instances are created with the additional constraints $x = a$ and $x \neq a$ respectively. The value a can be chosen by a heuristic. Usually, the $x = a$ instance is solved first, then $x \neq a$.
(3) Domain splitting. The domain of x is partitioned into two sets A and B, and two instances are generated with the additional constraints $x \in A$ and $x \in B$ respectively.

It is clear that enumeration can be simulated using binary branching. However, the converse is not true, and Hwang and Mitchell give a class of problems where enumeration takes exponentially more steps than binary branching [**64**]. A domain splitting strategy is likely to be problem-specific.

Figure 5 illustrates the three branching strategies. Each dot is a CSP instance and the arrows labelled with a constraint are transformations to create a new instance. When branching is applied recursively, a search tree is built. For example, for a very simple CSP, a binary branching search tree is shown in figure 6. At each node of this tree, the constraints are checked. If any is violated, the node is labelled false, and it is not branched further. If all variables are instantiated and no constraint is violated, the node is labelled true. This CSP has two solutions, $\langle 3, 1, 2\rangle$ and $\langle 3, 2, 1\rangle$.

Typically the search tree is explored depth-first. In the case of binary branching, the $=$ branch is usually explored before the $\neq$ branch. In figure 6 the nodes are numbered according to the order

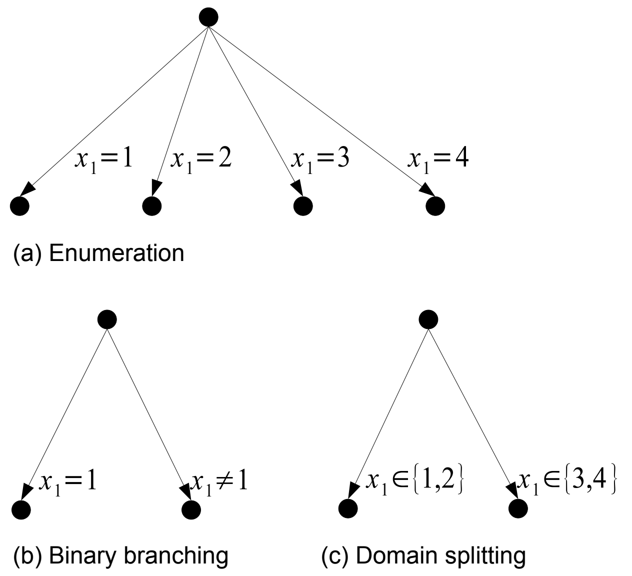


FIGURE 5. Three branching strategies for variable x_1 with domain $D_1 = \{1, 2, 3, 4\}$

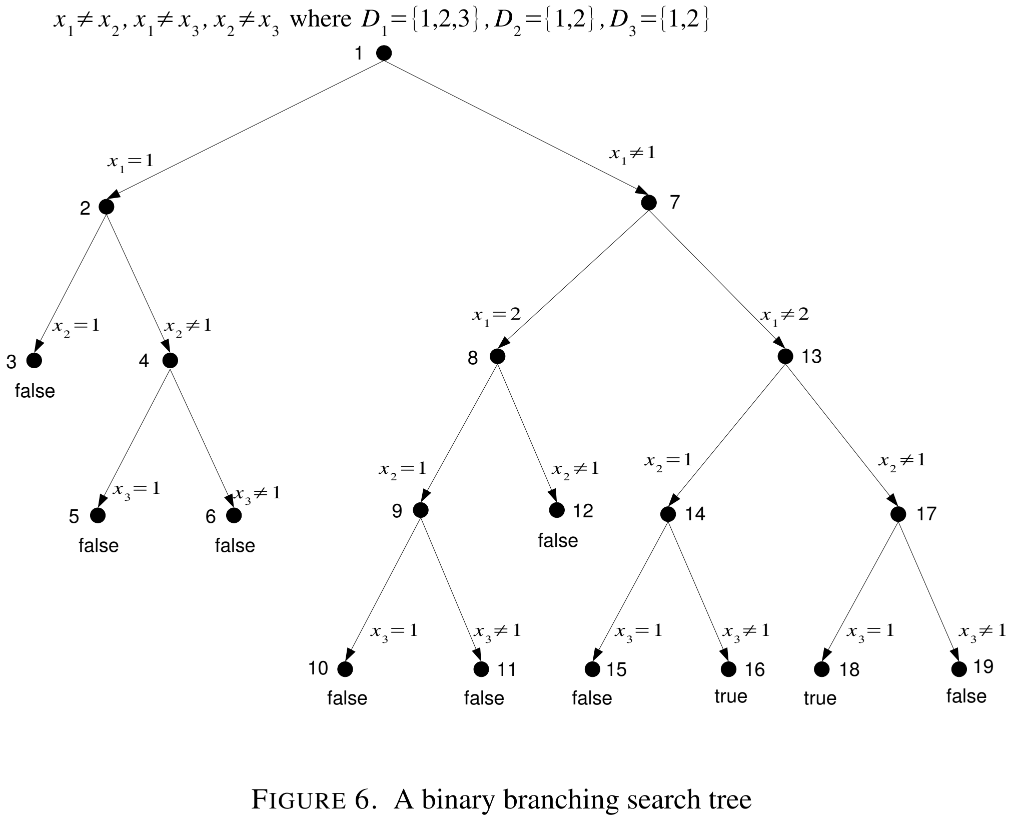


FIGURE 6. A binary branching search tree

the search algorithm would explore them. If we were searching for the first solution, the search would stop at node 16.

The search tree explored by such an algorithm could be very large. Indeed the tree shown in figure 6 is very large for such a simple CSP instance. The typical solution to this problem is to simplify the instance at each node, through the application of consistency. This can reduce the size of the search tree dramatically. Consistency is discussed in section 2.1.2.

Several more sophisticated search algorithms have been proposed. One of the most important is backjumping (first informally proposed by Stallman and Sussman [**92**]). Backjumping is applied at a node where all sub-nodes are false leaves (e.g. node 4 in figure 6). It potentially backtracks further than chronological backtracking, thus saving search effort. This is done by reasoning about which search decisions contributed to the contradiction. Conflict backjumping (CBJ, first proposed by Prosser [**81**]) is a generalization of backjumping which is able to backjump from any failed node in the search tree. CBJ is of particular interest because it has been adapted to QBF and binary QCSP. I omit to describe it here because the adapted algorithms (described in sections 2.2.3 and 2.3.4.1) are more relevant to this thesis.

2.1.2. Consistency and propagation. In order to solve large-scale instances of CSP with search, strong simplification is required at each node of the search tree. This is usually supplied by *propagation*, which performs reasoning on the constraints to remove values from variable domains that cannot take part in any solution[1]. If some domain is emptied, then the CSP instance is false. In this way, propagation can simplify a CSP instance, sometimes determining that it is false.

Here I will define consistency on individual constraints. A constraint is consistent or inconsistent, and it is made consistent by constraint propagation. Each constraint may have a different propagation algorithm, enforcing a different definition of consistency. Solvers such as ILOG Solver [**1**], Eclipse [**3**], Minion [**50**] and Gecode [**88**] support this heterogeneous approach to consistency.

[1]Some propagation algorithms add constraints to the instance, instead of or in addition to removing values. However this approach is not widely used.

Not all the forms of consistency in the literature are defined on individual constraints. Some are defined over several constraints (e.g. path consistency [**6**]) or the entire instance (e.g. singleton consistencies [**37**]). However, in the majority of the recent literature consistency notions (and propagation algorithms) are defined on individual constraints.

2.1.2.1. *Arc consistency.* Arc consistency (AC) is the oldest definition of consistency, with the first algorithms to enforce it given by Waltz [**101**] and Gaschnig [**45**]. The notion of arc consistency was first formally defined by Mackworth [**72**].

AC is defined for binary constraints. A constraint C_k is AC iff all values $a \in D_{k_1}$ are compatible with some value $b \in D_{k_2}$ ($\forall a : a \in D_{k_1} \Rightarrow [\exists b : b \in D_{k_2} \wedge \langle a, b\rangle \in C_k^S]$), and in the same way all values $b \in D_{k_2}$ are compatible with some value $a \in D_{k_1}$.

The propagation algorithms for AC are many and varied. Mamoulis and Stergiou [**74**] have generalized arc consistency for QCSP, adapting the optimal and simple AC-2001 algorithm by Bessière and Régin [**15**] to QCSP. This is covered in section 2.3.1.2.

The AC algorithms may be broadly divided into two sets: coarse-grained (such as AC-2001), where the removal of a value will cause all relevant constraints to be propagated completely, and fine-grained (such as AC-4) where the removed value is used to minimize the amount of work done [**16**]. This is important for this thesis since the following chapters will deal with both fine and coarse-grained algorithms.

2.1.2.2. *Generalized arc consistency.* Generalized arc consistency (GAC) is the generalization of arc consistency for non-binary constraints[2]. Historically, it was first defined by Mackworth [**73**]. It is defined neatly by Apt [**6**]. For some constraint C_k with set of allowed tuples of values $C_k^S \subseteq D_{k_1} \times \cdots \times D_{k_r}$, C_k is generalized arc consistent when each value a in the domain of each variable x_{k_i} in the constraint is contained in some tuple $t \in C_k^S$ ($t_i = a$). The tuple t is referred to as a *support* for $x_{k_i} \mapsto a$. When a value loses all its supports, it must be removed to maintain consistency. GAC is defined more formally in the following chapter, alongside analogous consistencies in QCSP. GAC propagation algorithms for arbitrary constraints take exponential time in the constraint arity.

[2]Generalized arc consistency refers to a generalization within CSP, not to be confused with the generalization of CSP to QCSP.

GAC4. Mohr and Masini proposed GAC4 [**78**], a generalization of the AC4 algorithm. AC4 is an optimal time algorithm for binary arc consistency. For each pair $x_{k_i} \mapsto a$, GAC4 maintains a doubly-linked list of pointers to supporting tuples. Suppose value a is removed from the domain of x_{k_i} by some other constraint. The list of supports for $x_{k_i} \mapsto a$ contains all tuples t which have become *invalid* (i.e. $t \notin D_{k_1} \times \cdots \times D_{k_r}$). For each t, it is removed from all lists it is present in. This can be done in $O(r)$ time, with careful implementation. Whenever a list becomes empty, its corresponding value has no remaining supports and must be removed.

One criticism of GAC4 is that a separate data structure is required for each constraint. In an instance with a set of identical constraints, the space overhead can be reduced by storing the tuples in a single static data structure. Also, doubly-linked lists containing pointers can be wasteful, since each element has three pointers in it.

GAC-Schema. Bessière and Régin proposed GAC-Schema [**13**], which addresses the criticisms of GAC4. GAC-Schema maintains mutable lists of supports much like GAC4, except that only a polynomial number of tuples are stored in the lists. When a list becomes empty, GAC-Schema calls a procedure to search for another supporting tuple. If this procedure does not return a valid supporting tuple, then the appropriate value is not supported and must be removed. This procedure (*seekNextSupport*) can be instantiated in different ways depending on the type of constraint.

GAC-Schema maintains three data structures, described below.

- $S_C(x_{k_i}, a)$ contains tuples τ that have been found to satisfy C_k and which include value a: $\tau_i = a$. Each tuple supports n values, so when a tuple is found, it is added to all n relevant lists in S_C. The lists are doubly-linked so that removals and restorations can be made in $O(1)$ time. Removals are made lazily, so at any time $S_C(x_{k_i}, a)$ may contain invalid tuples.
- $S(\tau)$ contains the set of pairs $x_{k_i} \mapsto a$ for which τ is the current support.
- $last_C(x_{k_i}, a)$ is the last tuple returned by *seekNextSupport* as a support for $x_{k_i} \mapsto a$; nil otherwise. This is used to allow *seekNextSupport* to continue searching at the point where it left off in the lexicographic ordering of tuples.

The algorithm is fine-grained and very simple in principle. It is called with a variable and value pair $x_{k_i} \mapsto a$ which has been removed from D_{k_i}. All tuples containing $x_{k_i} \mapsto a$ are now invalid. $S_C(x_{k_i}, a)$ contains all such tuples discovered so far. The algorithm iterates through $S_C(x_{k_i}, a)$, and for each tuple $\tau \in S_C(x_{k_i}, a)$ it removes τ from all $S_C(x_{k_j}, b)$ lists it is present in.

If τ is the current support for some value $x_{k_j} \mapsto b$, then $x_{k_j} \mapsto b$ may need a new current support. The list of such values is $S(\tau)$. The algorithm iterates for each element of $S(\tau)$, checking if the related S_C list contains a valid tuple. If not, a new tuple is sought by calling *seekNextSupport*. If a new tuple is found, it is added into each of the three data structures. The key point of the algorithm is that each tuple found by *seekNextSupport* is reused for r values. This is referred to as *multidirectionality* and is described in more detail in chapter 4, where GAC-Schema is adapted for quantified constraints.

The original paper contains two instantiations of *seekNextSupport*. The simpler one is for constraints expressed as a set of satisfying tuples. This addresses the problems with GAC4, since the set of satisfying tuples is not changed, it is simply searched, and can therefore be shared between constraints. Also, it is searched linearly so arrays or singly-linked lists can be used, saving space.

The more sophisticated instantiation of GAC-Schema uses a *predicate*, which is a procedure that takes a tuple and returns a Boolean value indicating whether the tuple satisfies the constraint. For example, if the constraint is allDifferent, and the tuple is $\langle 2, 3, 3, 4 \rangle$ then the predicate returns false. The *seekNextSupport* procedure explores the space of valid tuples in lexicographic order, jumping forward in the ordering to avoid discovering the same tuple twice. The predicate instantiation is also generalized to QCSP in chapter 4.

In a second paper, Bessière and Régin propose to instantiate *seekNextSupport* with a CSP sub-problem [**14**]. When a new satisfying valid tuple is required, a search is performed in the sub-problem. Constraint propagation in the sub-problem avoids exhaustive search.

Efficient tuple search. When the constraint is expressed as a set of satisfying tuples, the *seekNextSupport* procedure must search through the set to find a tuple which is valid w.r.t. current domains. The most basic algorithm, presented by Bessière and Régin [**13**], creates a list of allowed tuples for each literal $x_{k_i} \mapsto a$ before search. A pointer into each list is maintained, and updated whenever a new valid tuple is discovered. When *seekNextSupport*(x_{k_i},a) is called, it iterates from the pointer to the next valid tuple, thus avoiding repeated work down one branch of the search tree.

This simple algorithm can be very inefficient when there is a large set of satisfying tuples, because it iterates through all tuples regardless of whether they are valid with respect to current variable domains. More efficient alternatives have recently been developed by Lecoutre and Szymanek [**31**], Lhomme and Régin [**70**], and by Gent, Miguel and Nightingale [**51**]. These work by jumping forward in the tuple list.

Lecoutre and Szymanek give an algorithm *seekSupport-valid+allowed*, which uses binary search rather than simple iteration [**31**]. Lists for each literal are constructed as in the basic algorithm. All lists are sorted in lexicographic order (lex, $\prec_{lex}$). The algorithm first constructs the lex least valid tuple t, then performs a binary search for t in the list, finding t' which is the lex least tuple in the list s.t. $t \prec_{lex} t'$. If t' is valid, we are done. Otherwise, the lex least tuple t'' is constructed s.t. $t' \prec_{lex} t''$ and t'' is valid. The algorithm then repeats the binary search and proceeds from there.

Lhomme and Régin add extra information to the list of allowed tuples, enabling jumping forward over sequences of invalid tuples [**70**]. The data structure and algorithm are unfortunately complicated, and a comparative experimental study shows that simpler algorithms are preferable [**51**].

Gent, Miguel and Nightingale present two approaches: tries and Next-Difference lists [**51**]. Tries are tree data structures which are searched in a depth-first way, avoiding entering areas of the tree which represent invalid tuples. Next-Difference lists are described in chapter 4 section 4.4.1.3 as a contribution of this thesis. Experimental evaluation, comparing these and other approaches in the context of CSP, is provided [**51**]. This shows the benefits of both tries and Next-Difference lists.

GAC2001/3.1. Bessière et al. [**16**] develop a new algorithm for GAC, which is coarse-grained and therefore has a simpler interface to the core of a backtracking constraint solver. The algorithm itself is also simpler, but unfortunately takes $O(r^2d^r)$ (where r is the arity and d the domain size) to establish GAC at each node down a branch of the search tree, whereas GAC-Schema takes $O(rd^r)$ time. The difference is because GAC-Schema exploits multidirectionality. Therefore I make use of GAC-Schema later in this thesis.

Polynomial time generalized arc consistency algorithms. It is possible to enforce GAC on many classes of constraint in polynomial time. One popular polynomial time propagation algorithm is Régin's allDifferent propagator [**83**]. The allDifferent constraint simply expresses that a set of variables take distinct values in any solution. The propagator constructs a graph representing the variables and values in the constraint, then uses results from graph theory to compute new domains for all variables in the constraint.

It may be possible to adapt this algorithm (and others) to deal with quantifiers while retaining the polynomial time bound, but it is not clear how this could be done.

2.1.2.3. *Bounds consistency.* When dealing with numerical variables, the domains can be very large. This creates two problems: storing and updating the domains is inefficient, and enforcing GAC (or some other consistency which is concerned with every value in the domain) is inefficient. A popular way of solving these problems in CSP is to approximate the variable with an interval $[\underline{x_i},\ \overline{x_i}]$, and to enforce a form of consistency which narrows the interval but does not remove values in the middle of the domain. The standard term for this form of consistency is *bounds consistency* (BC).

Choi, Harvey, Lee and Stuckey [**30**] identify three commonly used but incompatible definitions of BC. They are described below, restated in the terminology used for this thesis.

(1) A constraint C_k is bounds($\mathbb{D}$) consistent iff for each variable x_{k_i} in C_k, for each bound $d_i \in \{\underline{x_{k_i}}, \overline{x_{k_i}}\}$, there exist integers d_j with $d_j \in D_{k_j}$, $j \neq i$ such that $\langle d_1, \ldots, d_r \rangle \in C_k^S$. That is, each bound must be supported by a solution to C_k, such that the solution contains only values which are in their respective domains.

(2) A constraint C_k is bounds($\mathbb{Z}$) consistent iff for each variable x_{k_i} in C_k, and for each bound $d_i \in \{\underline{x_{k_i}}, \overline{x_{k_i}}\}$, there exist integers $d_j \in \mathbb{Z}$, with $\underline{x_{k_j}} \leq d_j \leq \overline{x_{k_j}}$, $j \neq i$ such that

$\langle d_1, \ldots, d_r \rangle \in C_k^S$. That is, each bound must be supported by a solution to C_k, such that the solution contains only integers within their respective bounds.

(3) A constraint C_k is bounds($\mathbb{R}$) consistent iff for each variable x_{k_i} in C_k, and for each bound $d_i \in \{\underline{x_{k_i}}, \overline{x_{k_i}}\}$, there exist real numbers $d_j \in \mathbb{R}$, with $\underline{x_{k_j}} \leq d_j \leq \overline{x_{k_j}}$, $j \neq i$ such that $\langle d_1, \ldots, d_r \rangle \in C_k^S$. That is, each bound must be supported by a solution to C_k, such that the solution contains only real numbers within their respective bounds.

In the definition of bounds($\mathbb{R}$) consistency, the set C_k^S is used loosely, it is assumed in this case to contain all real number solutions of constraint C_k.

The three definitions are given in order, such that the following holds for any constraint. In words, bounds($\mathbb{D}$) is the strongest and bounds($\mathbb{R}$) is the weakest.

$$\begin{aligned} \text{bounds}(\mathbb{D}) &\Rightarrow \text{bounds}(\mathbb{Z}) \\ \text{bounds}(\mathbb{Z}) &\Rightarrow \text{bounds}(\mathbb{R}) \end{aligned}$$

For a simple constraint, deriving a bounds consistency propagation algorithm can be very simple, particularly bounds($\mathbb{R}$) consistency. Consider $x_1 + x_2 = x_3$ (the example used by Marriott and Stuckey [**76**]). Rearranging for x_1 gives a simple expression, and the maximum and minimum values of that expression can be straightforwardly derived. These are then applied to x_1, narrowing the bounds if necessary, as shown below.

$$\begin{aligned} x_1 &= x_3 - x_2 \\ x_1 &\leq \overline{x_3} - \underline{x_2} \\ x_1 &\geq \underline{x_3} - \overline{x_2} \end{aligned}$$

The case for x_2 is very similar. The new bounds for x_3 are shown below.

$$\begin{aligned} x_3 &\leq \overline{x_1} + \overline{x_2} \\ x_3 &\geq \underline{x_1} + \underline{x_2} \end{aligned}$$

Assuming the bounds are initially integral, applying the six inequalities to exhaustion gives us a bounds($\mathbb{Z}$) propagator. Each new bound corresponds to an integer solution to the constraint within the domains. However, the propagator does not achieve bounds($\mathbb{D}$) consistency in all cases. Consider the following domains: $x_1 \in \{0, 2\}$, $x_2 \in \{0, 2\}$ and $x_3 \in \{0, 1, 2, 3\}$. 3 is an integer solution but not a domain solution, and the propagator is unable to reduce the upper bound of x_3.

CSP toolkits often provide bounds consistency propagators for *primitive* constraints of the form $x_1 + x_2 + x_3 + \ldots = 0$ and similar, and use a decomposition method to rewrite more complex expressions using only primitive constraints. This is discussed in section 2.3.3.1 below. For example, in Eclipse [**3**], using the FD (finite domain propagators) library, the expression $x_1x_2x_3 + x_4 + x_5 = x_6$ is decomposed into the following three primitives.

$$\begin{aligned} x_1 \times x_2 &= \tau_1 \\ \tau_1 \times x_3 &= \tau_2 \\ \tau_2 + x_4 + x_5 - x_6 &= 0 \end{aligned}$$

Bordeaux and Monfroy studied bounds consistency for QCSP. This work is reviewed in section 2.3.3 below.

2.1.3. Difficulty of a CSP instance. CSP instances of the same size can vary considerably in difficulty, due to their structure. Since CSP is NP-complete, the worst-case execution time for any known algorithm is exponential in the size of the instance. However, considering only the size gives an incomplete picture of how difficult instances are likely to be. Many instances can be solved much faster than the worst-case analysis suggests.

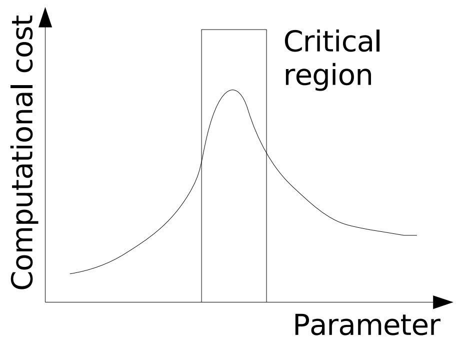


FIGURE 7. Phase transition and the critical region

Cheeseman, Kanefsky and Taylor [**28**] introduced the concept of *phase transition* in NP-complete problems. They proposed that some aspect of a problem can be described by an *order parameter*, and varying this parameter while fixing other parameters exhibits a phase transition. At some critical value for the order parameter, the computational cost of the instances peaks. Away from the critical value, computational cost falls off. If computational cost is plotted against the order parameter, a phase diagram is obtained. A typical phase diagram is sketched in figure 7. The critical value lies within the critical region. Some property of the problem instances, such as satisfiability, changes abruptly within the critical region.

For random binary CSP, the order parameter could be the constraint tightness (i.e. proportion of disallowed tuples per constraint). In this case, to the left of the critical region, almost all instances will be satisfiable. To the right of the region, almost all will be unsatisfiable. When the tightness is very low, it is very easy to find a solution by searching. When it is very high, consistency is very effective, and unsatisfiability can be proven with very little search. Hence the computational cost is low for the extreme values of constraint tightness. In the critical region, search tends to explore many dead-end paths before either finding a solution or proving unsatisfiability.

In chapter 4, phase transitions are seen for experiments with random QCSP instances, when varying the number of constraints in the instance. In some cases the usual phase transition behaviour is apparent.

2.2. QBF

Quantified Boolean Formulae (QBF) solvers typically operate on conjunctive normal form (CNF), which is a conjunction of disjunctions of literals (e.g. $(l_1 \vee l_2 \vee l_3) \wedge (l_2 \vee l_4)$) where each literal l_i is a positive or negative instance of some Boolean variable x_i ($l_i = x_i$ or $l_i = \neg x_i$). Each variable is quantified in a prefix. For example, the formula

$$\exists x_1 \forall x_2 \exists x_3 \ : \ (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

is false, since no value of x_1 extends to a solution for both values of x_2. The disjunctions are called clauses, and can be considered as constraints since they must all be satisfied for the formula to be satisfied. Hence QBF can be considered a subset of QCSP, with a restricted form of constraint. QBF is PSPACE-complete [**24**].

There are two main approaches to QBF. Search-based solvers, which interleave search with reasoning [**47**], and resolution solvers, which eliminate variables one by one until the formula is trivial [**18**]. There are also hybrids of these two approaches [**10**]. Resolution solvers tend to use a large amount of memory (exponential in the size of the instance). The majority of QBF solvers are search-based, generalizing the Davis Putnam Logemann Loveland (DPLL) procedure, which is the work of Davis and Putnam including unit propagation and the pure literal rule [**35**], refined by Davis, Logemann and Loveland who introduced the branching rule [**34**].

2.2.1. Unit propagation. Search-based QBF solvers perform unit propagation on the clauses, which is a form of quantified constraint propagation. Quantified unit propagation is a generalization of unit propagation in SAT [**35**], proposed by Cadoli et al. [**24**] for their solver Evaluate. It works as follows.

(1) Clauses are simplified as literals are assigned. If the assigned literal l_i satisfies the clause, the clause is removed. Otherwise l_i is removed from the clause.

(2) If a clause is composed entirely of universal literals, the clause is false.

(3) If a clause is of the form $\exists x_1 \forall x_2 \dots x_n \ : \ l_1 \vee l_2 \vee \cdots \vee l_n$ (an existential literal with 0 or more universal literals quantified later) then x_1 is set to satisfy the clause. This

is because the value of x_1 must extend to a solution for all combinations of values of $x_2 \dots x_n$, including the combination which does not satisfy the clause, therefore l_1 must satisfy the clause. If $l_1 = x_1$, then x_1 is set *true*, and if $l_1 = \neg x_1$ then x_1 is set *false*.

If there are no universal literals, then this rule is equivalent to unit propagation in SAT. If there is a universal literal quantified before the existential literal, for example $\forall x_1 \exists x_2 \; : \; l_1 \vee l_2$, then l_2 cannot be set because the universal may satisfy the clause, leaving the existential free. If there are more than one existential literals, none can be set because we do not know which one will satisfy the clause.

QBF solvers are designed for fast search, with lightweight constraint propagation, whereas the overall goal of this thesis is to evaluate an approach with more powerful propagation. Any new logic constraint should be at least as powerful as quantified unit propagation. Therefore the propagation algorithm I develop in chapter 5 is exactly as powerful as quantified unit propagation when it is applied to a clause, but can be applied to a larger set of logical constraints.

2.2.2. Pure literal rule. Cadoli et al. [**24**] also generalized the pure literal rule originally written by Davis and Putnam [**35**] (and called the *affirmative-negative rule* by them). If some literal l_i is present in the formula but $\neg l_i$ is not, then l_i is pure. It is proposed to instantiate l_i if x_i is existential, and $\neg l_i$ if x_i is universal. This rule has also been adapted to binary QCSP (described in section 2.3.1.3 below).

Informally, if some positive literal $l_i = x_i$ is not contained in any clause, and x_i is universal, then x_i can be assigned to true. This is because setting it to true satisfies no clauses. In terms of the game analogy, the universal player intends to falsify the formula, therefore it sets x_i to true. If x_i is existential, then x_i would be set to false, because this satisfies all clauses containing $\neg x_i$ without any other effects on the formula.

2.2.3. Search. Straightforward backtracking search is used by Evaluate [**24**]. This search procedure is similar to enumeration in CSP (section 2.1.1), where each variable x_i branches for values true and false, and the QBF instance is simplified at each node before branching. The simplifications are quantified unit propagation, pure literal rule, and simple rules to detect trivial truth

Algorithm 1 A simple QBF search algorithm

```
procedure QBFSearch($\phi$: QBF instance): Boolean
$\sigma$ ←Simplify($\phi$)
if $\sigma$ =true: return true
if $\sigma$ =false: return false
if $\sigma$ is of the form $\forall x_1 \ldots$:
    $t_1$ ←QBFSearch($\sigma[x_1 =$ true$]$) {Set $x_1$ to true and recurse}
    if $t_1$ =false: return false
    $t_2$ ←QBFSearch($\sigma[x_1 =$ false$]$) {Set $x_1$ to false and recurse}
    return $t_2$
else: {$\sigma$ is of the form $\exists x_1 \ldots$}
    $t_1$ ←QBFSearch($\sigma[x_1 =$ true$]$) {Set $x_1$ to true and recurse}
    if $t_1$ =true: return true
    $t_2$ ←QBFSearch($\sigma[x_1 =$ false$]$) {Set $x_1$ to false and recurse}
    return $t_2$
```

and falsity. There are two important differences compared to CSP enumeration: the variable order must respect the quantifier sequence (although two adjacent variables with the same quantifier can be transposed) and when branching on a universal variable, *both* branches must be satisfiable.

The search procedure is summarized in algorithm 1. The *Simplify* procedure is assumed to (at a minimum) simplify ϕ to true or false if all variables are instantiated, and to remove quantifiers for instantiated variables. The notation $\sigma[x_i = a]$ creates a new QBF instance where x_i has only the value a in its domain.

More sophisticated search algorithms have been proposed. The clausal form of the constraints eases the development of efficient backjumping and learning algorithms. These are summarized below.

- Conflict Backjumping (CBJ), proposed by Giunchiglia et al. [**56**, **57**], exploits information available at a search dead-end. At a dead-end, a series of search decisions has led to a contradiction. CBJ computes the minimal set of decisions which contributed to the contradiction (the *reason*), and backtracks directly (backjumps) to the most recent decision in that set. (Universal literals are excluded from the reason [**56**].) Therefore CBJ typically backtracks further than naive search from a dead-end, reducing the number of nodes in the search tree.

- Solution Backjumping (SBJ), which exploits solutions found during search in a similar way to CBJ. At a search node where all clauses are satisfied, a series of search decisions has led to the satisfaction of the clauses. SBJ computes the minimal set of decisions which contributed to the satisfaction. This set is also referred to as the reason, and existential literals are excluded. SBJ backjumps to the most recent decision in the reason.
- Conflict Learning (CL) and Solution Learning (SL), introduced by Giunchiglia et al. [**58**], extend CBJ and SBJ respectively. For CL, the reason that is computed with CBJ is converted into a clause: if the reason contains literal l_i, the clause contains $\neg l_i$. Therefore, unit propagation of the clause ensures that the search never explores search nodes where all the literals in the reason are true. This avoids re-discovering the same contradiction.

  SL uses the reason computed by SBJ to construct a conjunction C of literals: if the reason contains literal l_i, the conjunction contains l_i. Thereafter, if C is satisfied, then all the literals in the reason are true at the current search node, and the search can backtrack.

  Recent QBF solvers with SL (such as QuBE [**58**]) perform a variant of unit propagation on C: if all but one of its literals are true, leaving l_i, then we know that instantiating l_i leads to a solution, therefore we can instantiate $\neg l_i$ even though this involves pruning a universal variable.

The algorithms CBJ and SBJ have been adapted to binary QCSP. This work is reviewed in section 2.3.4.1 below.

2.3. QCSP

Allowing quantifiers significantly changes the character of CSP. QCSP is PSPACE-complete, whereas CSP is NP-complete (Börner et al. [**23**]), and different algorithms are required to prove satisfiability or unsatisfiability. The scope of this thesis is to investigate the usefulness of, and algorithms for solving, the finite quantified constraint satisfaction problem (QCSP). Therefore handling variables with infinite domains is outside the scope of this thesis. A real-valued variable (i.e. a variable which takes any real value between two initial bounds) can be approximated to some degree of accuracy using a finite-domain variable. Since all finite domains can be represented by a set of integers, all domains throughout this thesis are a finite subset of the integers, $\mathbb{Z}$. I define the finite quantified constraint satisfaction problem below.

DEFINITION 2.3.1. Finite Quantified Constraint Satisfaction Problem

A QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ is defined as a set of n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, a set of domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ where $D_i \subsetneq \mathbb{Z}$, $|D_i| < \infty$ is the finite set of all potential values of x_i, a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints, and a quantifier sequence $\mathcal{Q} = Q_1 x_1, \ldots, Q_n x_n$ where each Q_i is a quantifier, $\exists$ (existential, 'there exists') or $\forall$ (universal, 'for all').

Following the literature, I use n for the number of variables, e for the number of constraints, and for instances where all variables have the same initial domain, d for the cardinality of the domain. The terms *inner* and *outer variables* are also used in the literature. With respect to variable x_i, an outer variable is a variable x_j such that $j < i$ (i.e. a variable which is quantified before x_i in $\mathcal{Q}$). Similarly an inner variable is a variable x_j where $j > i$.

In the following sections I review the literature on QCSP.

2.3.1. Consistency and other properties for QCSP. In the literature, QCSP has been adequately defined, along with the notion of winning strategy which allows the solution of QCSP. This section is concerned with definitions of consistency and other properties related to local reasoning.

2.3.1.1. *Theoretical framework for non-binary QCSP.* The key paper in this area neatly defines QCSP, consistency and several other properties in terms of winning strategies (Bordeaux, Cadoli and Mancini [**21**]). Three of these properties are used in this thesis. These three properties are described informally below, and the other properties are briefly sketched.

First, the paper defines the notions of *solution*, *strategy*, *winning strategy*, *scenario* and *outcome*. These are formally defined in the following chapter, but I sketch them here. QCSP can be understood as an adversarial game, where the existential and universal quantifiers are the players. The players assign their own variables in quantification order. The aim of the existential player is to satisfy all constraints, and the aim of the universal player is to break at least one constraint.

- Solution. A tuple $\langle a_1, \ldots, a_n \rangle$ of values for each variable such that if $x_1 = a_1,\ x_2 = a_2, \ldots$ then all constraints are satisfied.
- Strategy. A family of functions which specify the values of each existential variable based on the values of universal variables which precede it in the quantifier sequence.
- Scenario of a strategy S. A tuple $\langle a_1, \ldots, a_n \rangle$ of values s.t. the value for each existential variable is defined by S based on preceding values for universal variables.
- Winning strategy. A strategy S s.t. all scenarios of S are also solutions.
- Outcome. A solution which is also a scenario of some winning strategy.

All of these concepts are used within this thesis, although the terminology is slightly changed. These are defined formally in chapter 3 section 3.2.1. These concepts are illustrated in the paper using a small QCSP instance with a tree diagram. Figure 8 is based on figure 1 from Bordeaux et al. [**21**], and it is described there as follows.

> "Illustration of the notions of solution, winning strategy, scenario and outcome on the QCSP represented by the logical formula $\exists x_1 \in [2, 3] \forall x_2 \in [3, 4] \exists x_3 \in [3, 6].x_1 + x_2 \leq x_3$. *And* and *or* labels on the nodes correspond to universal and existential quantifiers, respectively. The solutions are all triples $\langle x_1, x_2, x_3 \rangle$ *s.t.* $x_1 + x_2 \leq x_3$. The only two winning strategies assign x_1 to 2: one (s_1) assigns x_3 to 6 while the 2nd one (s_2) assigns it to x_2+2 (note that each strategy is constrained to choose one unique branch for each existential node). The scenarios of s_1 and s_2 are therefore those indicated, while the set of outcomes of the QCSP is the union of the scenarios of s_1 and s_2 (also shown in bold line)."

The figure illustrates that solutions are not necessarily included in any winning strategy, and that such solutions should not influence which values are defined as consistent. From inspection of the diagram, it is clear that a definition of consistency should regard $x_1 \mapsto 3$ as inconsistent, along with $x_3 \mapsto 3$ and $x_3 \mapsto 4$, since these three values do not take part in any winning strategy.

As their second contribution, Bordeaux et al. define various properties which allow a QCSP instance to be simplified. The definitions are given in terms of a QCSP instance, but their application

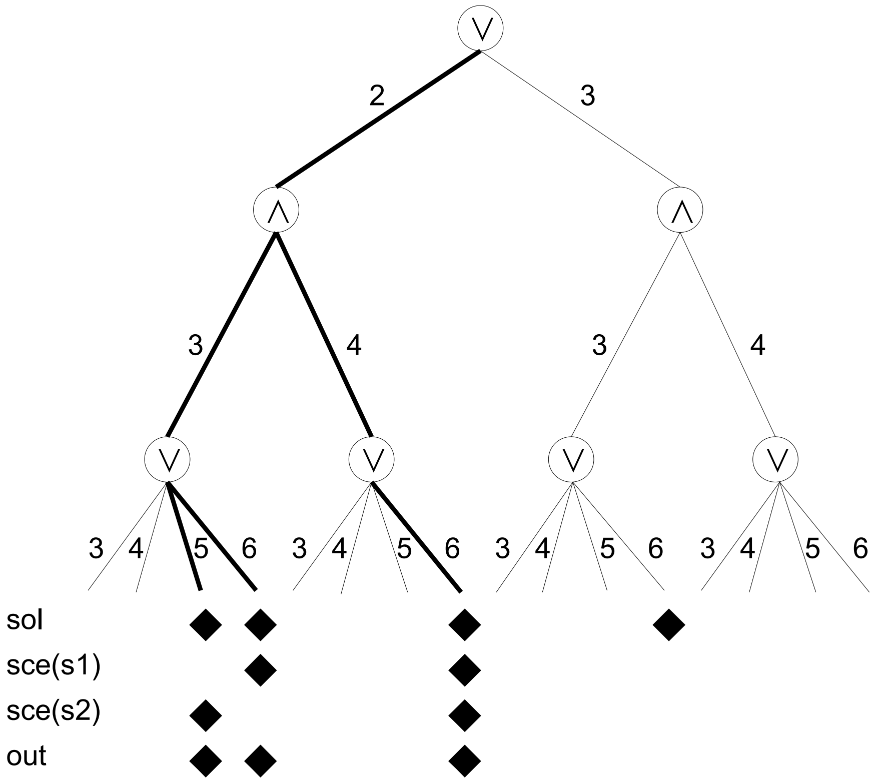


FIGURE 8. Illustration of scenarios and outcomes for QCSP instance $\exists x_1 \in \{2,3\} \forall x_2 \in \{3,4\} \exists x_3 \in \{3,4,5,6\} : x_1 + x_2 \leq x_3$

is usually to a single constraint. The three most relevant properties are listed below. The definition of consistency is inconveniently named *inconsistent*, but I refer to it as SQGAC to distinguish it from other forms of consistency in QCSP.

- *inconsistent*(x_{k_i}, a) iff there is no outcome t such that $t_i = a$.
- *d-fixable*(x_{k_i}, a) iff for all outcomes t, there exists an outcome t' which is identical except that $t'_i = a$ (deep fixability).
- *s-fixable*(x_{k_i}, a) iff for all outcomes t, there exists an outcome t' where $t'_i = a$ and for each $j < i$, $t'_j = t_j$ (shallow fixability).

D-fixable and s-fixable are defined precisely in chapter 3 section 3.2.6. Other properties include implied values, substitutable values (i.e. value b may be substituted for value a in any outcome), removable values, pairs of interchangeable values, determined and irrelevant values, and dependent variables (where the value of the dependent variable is a function of the values of a set of other variables). Substitutable, removable, interchangeable and irrelevant all have deep and shallow variants, where the deep property is a special case of the shallow property.

Bordeaux et al. also show some relationships between the various properties, and state various propositions about them. In particular, all properties are related to either removability or fixability. It is then proven that removable values can be removed from the domain of an existential variable without affecting the existence of a winning strategy for the QCSP instance. Also, if an existential variable has a fixable value, then all other values in its domain may be removed without affecting the existence of a winning strategy.

Bordeaux et al. do not consider the possibility of removable or fixable values of universal variables [**21**]. In chapter 3 I prove that instantiating a d-fixable value is sound for existential variables, and that for universal variables the d-fixable value can be removed from the domain.

2.3.1.2. *Consistency for binary QCSP.* In CSP, one of the first forms of consistency to be proposed was arc consistency, discussed in section 2.1.2.1.

Prior to the work of Bordeaux Cadoli and Mancini [**21**], Mamoulis and Stergiou extended arc-consistency for binary constraints from CSP to QCSP, embedding it in search in the form of forward checking or MAC [**74**].

There are four quantifier subsequences for a binary constraint: $\forall x_{k_1}, x_{k_2}$; $\forall x_{k_1}, \exists x_{k_2}$; $\exists x_{k_1}, \forall x_{k_2}$ and $\exists x_{k_1}, x_{k_2}$. For each one, Mamoulis and Stergiou define quantified binary arc consistency. This definition generalizes the classical definition in CSP [**38**].

$\forall x_{k_1}, x_{k_2}$: If the constraint has any unsatisfying tuples, it is not QAC. In other words it is QAC iff each value $a \in D_{k_1}$ is compatible with all values $b \in D_{k_2}$.

$\forall x_{k_1}, \exists x_{k_2}$: The constraint is QAC iff all values $a \in D_{k_1}$ are compatible with some value $b \in D_{k_2}$ ($\exists b : \langle a, b\rangle \in C_k^S$), and all values $b \in D_{k_2}$ are compatible with some value $a \in D_{k_1}$.

$\exists x_{k_1}, \forall x_{k_2}$: The constraint is QAC iff each value $a \in D_{k_1}$ is compatible with all values $b \in D_{k_2}$.

$\exists x_{k_1}, x_{k_2}$: This is the case for standard binary CSP. The constraint is QAC iff all values $a \in D_{k_1}$ are compatible with some value $b \in D_{k_2}$ ($\exists b : \langle a, b\rangle \in C_k^S$), and all values $b \in D_{k_2}$ are compatible with some value $a \in D_{k_1}$.

For the four types of constraint, rules are given to propagate them.

$\forall x_{k_1}, x_{k_2}$: If C_k is inconsistent, then the QCSP instance simplifies to false.

$\forall x_{k_1}, \exists x_{k_2}$: If there exists a value $a \in D_{k_1}$ which is not compatible with some value $b \in D_{k_2}$, then the QCSP instance is false. All values $b \in D_{k_2}$ which are not compatible with some value $a \in D_{k_1}$ are removed from D_{k_2}.

$\exists x_{k_1}, \forall x_{k_2}$: Any value $a \in D_{k_1}$ which is not compatible with *all* values $b \in D_{k_2}$ is removed.

$\exists x_{k_1}, x_{k_2}$: Any value $a \in D_{k_1}$ which is not compatible with some value $b \in D_{k_2}$ ($\exists b : \langle a, b \rangle \in C_k^S$) is removed from D_{k_1}. Similarly, any value $b \in D_{k_2}$ which is not compatible with some value $a \in D_{k_1}$ is removed from D_{k_2}.

If the domain of an existential variable is emptied, then the QCSP instance is false. Mamoulis and Stergiou observe that for the types $\forall x_{k_1}, x_{k_2}$ and $\exists x_{k_1}, \forall x_{k_2}$, the propagation rules can be applied once before search, and then these constraints can be deleted, because these constraints will remain arc-consistent throughout search.

The propagation rules are embedded in search as follows. For Maintaining Arc-Consistency (MAC), all constraints are propagated to exhaustion at each node in the search tree. The propagation algorithm for MAC is closely based on AC-2001 [**15**]. For Forward Checking (FC), when the search procedure instantiates a variable, only the constraints containing that variable are propagated.

I prove in chapter 3 that binary quantified arc consistency is equivalent to the inconsistency property defined by Bordeaux et al. [**21**].

2.3.1.3. *Pure value property for binary QCSP.* Gent, Nightingale and Stergiou [**54**] defined the pure value property for binary QCSP. This work was inspired by the pure literal rule in QBF solvers (reviewed in section 2.2.2).

Bacchus and Walsh define the same property in CSP for their constraint algebra [**7**]. They name a pure value *valid*, and observe that the set of inconsistent values of some constraint $\mathcal{E}$ is equal to the set of valid values of $\neg\mathcal{E}$. This is a useful result when combining constraints with conjunction, disjunction and negation symbols. However, they do not use pure values directly to prune the domains.

By encoding binary QCSP into QBF (repeating the experiments of an earlier paper by Gent, Nightingale and Rowley [**52**], which is reviewed below), and investigating the effect of the pure literal rule on the encoded instances, we discovered that the pure literal rule improved the performance of the QBF solver by several orders of magnitude. This is the motivation for adapting the rule to QCSP.

The binary pure value property is defined as follows.

DEFINITION 2.3.2. Binary pure value property in a binary QCSP $\mathcal{P}$

A value $a \in D_i$ is a pure value iff $\forall x_j \in \mathcal{X}$ where $j \neq i$ and $\forall b \in D_j$, the assignments $x_i \mapsto a$ and $x_j \mapsto b$ are compatible in any constraints C_k where $\mathcal{X}_k = \langle x_i, x_j \rangle$ or $\mathcal{X}_k = \langle x_j, x_i \rangle$.

If some value is found to be pure, it is also d-fixable (by inspection of the definitions) but d-fixability does not imply purity.

The pure value property can be used to reduce domains of both existential and universal variables, as in QBF. The algorithm used by Gent, Nightingale and Stergiou in the solver QCSP-Solve [**54**] is naive: to check value $a \in D_i$, the algorithm iterates for each constraint containing x_i and some other variable x_j, and iterates for each value $b \in D_j$, checking compatibility.

Since this is joint work, led by Kostas Stergiou, I do not claim the binary pure value property as a contribution of this thesis. However, in chapter 3 I re-define the pure value rule for non-binary QCSP, and prove that this implies d-fixability.

2.3.1.4. *Dual consistency.* Another approach to the problem of pruning universal variables is to apply consistency in a *dual* problem. This was proposed by Bordeaux et al. [**22**]. It is not related to the dual transformation of a non-binary CSP into a binary CSP, but the dual problem represents the negation of the original (*primal*) problem $\mathcal{P}$. Propagation in the dual problem can yield useful information.

Ideally the dual problem would be automatically generated from the primal problem. I believe it would be unreasonable to expect the user to provide a dual, and verify that it is the negation of the primal. I first provide a method for transforming any QCSP into a dual.

Transformation of arbitrary QCSP into a dual. If $\mathcal{P}$ is the primal problem and $\mathcal{P}'$ is the dual, a dual problem can be derived as follows. First I just write down the negation of $\mathcal{P}$.

$$\mathcal{P}' = \neg(Q_1x_1, Q_2x_2, \ldots, Q_nx_n \ : \ \bigwedge \mathcal{C})$$

Now the negation symbol can be pushed inside the quantifier prefix, because $(\neg\forall x : C) \equiv (\exists x : \neg C)$ and $(\neg\exists x : C) \equiv (\forall x : \neg C)$. By applying these two identities repeatedly, we have the following. The quantifier Q'_1 is the opposite quantifier to Q_1.

$$\mathcal{P}' = Q'_1x_1, Q'_2x_2, \ldots, Q'_nx_n \ : \ \neg(\bigwedge \mathcal{C})$$

By De Morgans rule, we can turn the conjunction into a disjunction.

$$\mathcal{P}' = Q'_1x_1, Q'_2x_2, \ldots, Q'_nx_n \ : \ \neg C_1 \vee \neg C_2 \vee \cdots \vee \neg C_e$$

For local constraint reasoning, a conjunction of constraints is required, so the disjunction is reformulated by introducing an additional Boolean variable b_k for each term, and *reifying* the constraint C_k, forming constraint C_k^r, by adding b_k to the scope. Suppose $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$, then the new scope would be $\mathcal{X}_k^r = \langle x_{k_1}, \ldots, x_{k_r}, b_k \rangle$. The set C_k^{Sr} for the new constraint is constructed from C_k^S so that b_k is constrained to be 1 if C_k would be satisfied, and 0 otherwise. For all tuples $t \in C_k^S$, the tuple $\langle t_1, \ldots, t_r, 1 \rangle$ is in C_k^{Sr}. For all other tuples $t \in (D_{k_1} \times \cdots \times D_{k_r}) \backslash C_k^S$, the tuple $\langle t_1, \ldots, t_r, 0 \rangle$ is in C_k^{Sr}. The reformulated problem is shown below.

(1) $$\mathcal{P}' = Q'_1x_1, Q'_2x_2, \ldots, Q'_nx_n \exists b_1, \ldots, b_e \ : \ C_1^r \wedge \cdots \wedge C_e^r \wedge (\neg b_1 \vee \neg b_2 \vee \cdots \vee \neg b_e)$$

This is a general form of the dual problem, which only requires that each constraint is reifiable. There must be efficient procedures to test the entailment and disentailment ($C_k^S = \emptyset$) of each constraint in the problem.

Consistency in the dual. The domains of variables $x_1, \ldots, x_n$ can be synchronized between the primal and dual problems when performing search, and local consistency can be performed on the dual problem. The dual problem is unsatisfiable iff the primal problem is satisfiable. If local consistency is performed on the dual problem, it can determine that the dual is unsatisfiable and therefore the primal is satisfiable. If the primal is satisfiable, the search procedure can backtrack immediately.

Local consistency can also prune a value $x_i \mapsto a$ of an existential variable in the dual, and $x_i \mapsto a$ can be pruned in the primal as well. Since the quantifiers are opposite in the dual problem, this is a way of pruning universal variables in the primal problem. In terms of the primal problem, the intuition is that setting $x_i \mapsto a$ would cause every constraint to be trivially true, therefore there is no need to branch for $x_i \mapsto a$.

In the transformation described above (equation 1), suppose SQGAC is enforced for all constraints in the dual problem, including the extra disjunction constraint. All the b variables except one b_i must be set to true before b_i is set to false by the disjunction constraint. This is equivalent to constraints $\forall k \neq i : C_k$ in the primal being trivially true ($C_k^S = D_{k_1} \times \cdots \times D_{k_r}$). Following this, the constraint C_i^r can potentially perform some pruning which will carry through to the primal problem. All but one of the constraints in the primal must be trivially true before any useful computation occurs in the dual problem. Therefore reasoning on the dual problem during search will only perform useful computation towards the leaves of the search tree, when sufficient variables have been instantiated to satisfy all but one of the constraints.

In specific cases, it may be possible to construct the dual problem differently (and more effectively). One such case is described below. For arbitrary QCSPs however, I do not know of a better transformation than the one given above. Therefore I do not pursue the dual problem approach any further.

As an example where dual is effective, suppose we originally wanted to solve $\mathcal{P}'$ (equation 1), rather than $\mathcal{P}$. Then a useful dual problem would be $\mathcal{P}$, because it is a negation of $\mathcal{P}'$. The primal and dual problems have been swapped. Now the propagation in the primal problem is highly ineffective, and in the dual it is highly effective. This illustrates an attractive symmetry: solving some problem $\mathcal{P}$ is approximately as difficult as solving $\neg\mathcal{P}$ when using this approach.

2.3.2. Adapting CSP propagation algorithms to QCSP. Benedetti, Lallouet and Vautard [**11**] propose to re-use existing CSP propagation algorithms in QCSP, adapting them in one of four ways. For constraint C_k with propagation algorithm *Prop* and variables $x_{k_1} \dots x_{k_r}$ (in quantification order), the four adaptations are as follows.

Existential analysis: *Prop* is applied, ignoring quantification. *Prop* may prune existential variables as usual. If a universal variable is pruned, then the constraint is considered to be unsatisfiable.

Functional domain analysis: C_k is functional with respect to variable x_{k_i} iff the constraint is a function mapping the rest of its variables to x_{k_i}. For example, $x_{k_1} + x_{k_2} = x_{k_3}$ is functional w.r.t. all its variables, since instantiating any two variables leaves exactly one value for the third variable.

If C_k is functional w.r.t. x_{k_i}, and x_{k_i} is universal, with inner existential variables $X = \{x_{k_{j>i}}, x_{k_{l>j}}, \dots\}$, each value of x_{k_i} must map to a distinct combination of values for X. Intuitively this is because universal and outer variables cannot be freely set *after* setting the value of x_{k_i}, and because of the functional property no combination of values for X can be used for two values of x_{k_i}. (Unfortunately, Benedetti et al. omit the proof of correctness of this rule [**11**].) If there are too few combinations of values for X, then C_k is not functionally consistent for x_{k_i}. Therefore if the product of domain sizes of X is less than the domain size of x_{k_i}, C_k is not functionally consistent.

There is also a property for injective functional constraints. If C_k is injectively functional w.r.t. x_{k_i}, and some inner universal variable $x_{k_{j>i}}$ has domain size greater than one, then C_k is inconsistent because no single value of x_{k_i} will be compatible with all values of x_{k_j}.

Look-ahead analysis: An enumeration step is applied to a variable of the constraint. For each value, some form of propagation is applied (possibly existential analysis), and the final domains are obtained by intersection if the variable is universal, and union otherwise. Benedetti et al. also identify a special case for convex constraints, where it is sufficient to use only the upper and lower bounds.

Unfortunately I believe there is an error: when performing look-ahead analysis on a universal variable x_{k_i}, the computed domains for variable x_{k_j} where $j \neq i$ are combined by intersection. Where $j > i$, union should be used. Consider the QCSP $\forall x_1 \in \{0,1\} \exists x_2 \in \{0,1\} : x_1 = x_2$. The unique winning strategy for this instance contains both values for x_2, and both values of x_2 are consistent. Look-ahead analysis on for the values of x_1 gives domains $\{0\}$ and $\{1\}$ for x_2. Combining these by intersection gives $D_2 = \emptyset$, but union gives $D_2 = \{0, 1\}$.

Dual analysis: The dual of a QCSP instance is discussed in section 2.3.1.4. Benedetti et al. propose to construct the dual of a single constraint, and apply some form of consistency to it (possibly existential analysis). The values which are pruned in the dual can then be ignored in the primal. The analysis applied to the primal constraint can then be stronger (e.g. existential analysis) or more efficient (e.g. look-ahead analysis).

Existential and functional analysis are implemented in the solver Qecode, derived from Gecode [**88**]. Experiments are provided comparing Qecode to QCSP-Solve on random binary QCSP. The instances are not suitable to evaluate functional analysis. The brief experiments show that Qecode can be competitive with QCSP-Solve on some instances.

Ratschan proposed a way of generalising CSP algorithms to solve first-order constraint satisfaction over the reals [**82**]. This language admits quantifiers and various numerical constraints including equality, inequality, multiplication, addition, sine, cosine and tangent. The language is undecidable, but Ratschan claims that in many cases, useful results can be derived by a constraint solver. This work is related to finite QCSP because constraint propagators over the reals are also sound over the integers, therefore this work could be a source of propagation algorithms.

The work concentrates on interval reasoning (where each variable domain is approximated by an interval $[\underline{x_i}, \overline{x_i}]$). Ratschan defines *first-order C-consistency* (FO^C-consistency), where C-consistency can be any form of interval consistency. Rules are given to use a narrowing operator (a propagation algorithm for C-consistency) for an unquantified constraint on a constraint with existential and universal quantifiers, enforcing FO^C-consistency.

However, FO^C-consistency is weak on constraints over universal variables, as the example in the paper demonstrates (figure 2). The handling of quantifiers in the definition is as follows: 'if ϕ is of the form $Qy \in I' : \phi'$, where Q is a quantifier, then $FOC^C(\phi, B)$ iff $FOC^C(\phi', B\frac{I}{y})$',

Constraint type	Primitive	Given by Bordeaux [19]
Logical	$x_1 \vee x_2 \Leftrightarrow x_3$	yes, x_3 quantified last
	$x_1 \wedge x_2 \Leftrightarrow x_3$	no, but $\vee$ can be used
	$x_1 \Leftrightarrow \neg x_2$	yes
Numerical	$x_1 + x_2 = x_3$	yes, x_3 quantified last
	$-x_1 = x_2$	yes
	$c \times x_1 = x_2$	yes, x_2 quantified last
	$x_1 \leq x_2, x_1 < x_2, x_1 = x_2, x_1 \neq x_2$	yes
	$(x_1 \leq x_2) \Leftrightarrow x_3, (x_1 = x_2) \Leftrightarrow x_3$	yes, x_3 existentially quantified last

TABLE 1. Primitive constraints for logic and arithmetic

where ϕ is a quantified constraint. The notation $B\frac{I}{y}$ simply adds the interval of the quantified variable, y, to the set of intervals B. There is no special handling of the universal quantifier in the definition. For this reason, I do not consider FO^C-consistency in the remainder of this thesis.

2.3.3. Numerical and logic constraints in QCSP. Lucas Bordeaux in his Ph.D. work (first published with Eric Monfroy [**22**] then later, independently and more comprehensively [**19**]) designed a general framework for propagating numerical and logical constraints in QCSP. In this framework long expressions are decomposed into ternary and binary primitive constraints, which makes the framework extendable (with new primitive constraints) and propagation tractable. Table 1 shows the propagators given by Bordeaux [**19**] (in the form of a set of rules). In some cases, rules are given for all possible quantifier sequences, but in other cases one variable must be quantified at the end. This is noted in the table.

For the numerical primitive constraints, the propagators enforce a form of bounds consistency which takes advantage of quantification. Therefore it is significantly stronger in many cases than any of the CSP bounds consistency notions defined by Choi et al. [**30**]. However Bordeaux et al. do not define a quantified bounds consistency notion, and do not prove either the correctness or completeness of their propagation rules.

2.3.3.1. *Decomposition.* It is necessary to break down complex expressions into the primitive constraints in table 1. First an expression is put into prenex form using equivalence rules such as $\neg(\forall x\ A) \equiv \exists x\ (\neg A)$ and $\neg(\exists x\ A) \equiv \forall x\ (\neg A)$. Second, the expression is broken down by introducing existential variables. If we have a predicate symbol p with k arguments, and a functional symbol f with t arguments, the following formula:

$$p(x_1, \ldots, f(y_1, \ldots, y_t), \ldots, x_k)$$

is equivalent to the following:

$$\exists z\,(z = f(y_1, \ldots, y_t) \wedge p(x_1, \ldots, z, \ldots, x_k))$$

where z may be Boolean or numerical [**19**]. For Boolean expressions, f is one of the following three functions, corresponding to the three primitives.

(1) $f(y_1,\ y_2) = y_1 \wedge y_2$
(2) $f(y_1,\ y_2) = y_1 \vee y_2$
(3) $f(y_1) = \neg y_1$

Every connector ($\wedge$, $\vee$, $\neg$) in the original expression is replaced with a single primitive constraint and an existential variable. For example, consider the quantifier sequence and expression below.

$$\exists x_1 \forall x_2 \exists x_3, x_4\ :\ \neg(x_3 \wedge \neg x_1) \vee (\neg x_2 \wedge x_4)$$

The subexpressions $\neg x_1$ and $\neg x_2$ can both be rewritten by using function 3 twice:

$$\exists x_1 \forall x_2 \exists x_3, x_4, x_5, x_6\ :\ (\neg x_1 \Leftrightarrow x_5) \wedge (\neg x_2 \Leftrightarrow x_6) \wedge \neg(x_3 \wedge x_5) \vee (x_6 \wedge x_4)$$

Now the conjunctions can be rewritten by using function 1 twice:

$$\exists x_1 \forall x_2 \exists x_3, x_4, x_5, x_6, x_7, x_8\ :$$

$$(x_3 \wedge x_5 \Leftrightarrow x_7) \wedge (x_6 \wedge x_4 \Leftrightarrow x_8) \wedge (\neg x_1 \Leftrightarrow x_5) \wedge (\neg x_2 \Leftrightarrow x_6) \wedge (\neg x_7 \vee x_8)$$

Function 3 is applied to $\neg x_7$:

$$\exists x_1 \forall x_2 \exists x_3, x_4, x_5, x_6, x_7, x_8, x_9 \ :$$
$$(\neg x_7 \Leftrightarrow x_9) \wedge (x_3 \wedge x_5 \Leftrightarrow x_7) \wedge (x_6 \wedge x_4 \Leftrightarrow x_8)$$
$$\wedge (\neg x_1 \Leftrightarrow x_5) \wedge (\neg x_2 \Leftrightarrow x_6) \wedge (x_9 \vee x_8)$$

Finally function 2 is applied:

$$\exists x_1 \forall x_2 \exists x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10} \ :$$
$$(x_9 \vee x_8 \Leftrightarrow x_{10}) \wedge (\neg x_7 \Leftrightarrow x_9) \wedge (x_3 \wedge x_5 \Leftrightarrow x_7)$$
$$\wedge (x_6 \wedge x_4 \Leftrightarrow x_8) \wedge (\neg x_1 \Leftrightarrow x_5) \wedge (\neg x_2 \Leftrightarrow x_6) \wedge x_{10}$$

Now there are six primitive constraints and six additional variables, representing the six connectors in the original expression. The variable x_{10} can be simplified to 1 and then deleted at this point.

2.3.3.2. *Advantages and disadvantages of decomposition.* The main advantage of decomposition is that new primitives can be introduced, expanding the input language. Unfortunately the decomposition yields a large number of constraints and additional variables, which could cause efficiency problems.

Table 2 shows the propagation rules for the constraint $\neg x \Leftrightarrow y$, for each of four quantifier sequences. The other four sequences, with x and y transposed, are equivalent. Even for this very simple constraint, there are a large number of rules and they are derived by hand. $x_1 + x_2 = x_3$ has many more propagation rules. This shows another difficulty with the approach.

A third problem with this approach is that interaction between variables can be lost when they are not contained in the same primitive constraint, so enforcing consistency on the primitive constraints is potentially much weaker than enforcing the same definition of consistency on the

constraint	condition		action
$\forall x \forall y : \neg x = y$	$0 \in D_x, 0 \in D_y$	$\Rightarrow$	false
	$1 \in D_x, 1 \in D_y$	$\Rightarrow$	false
$\forall x \exists y : \neg x = y$	$0 \in D_x, 1 \notin D_y$	$\Rightarrow$	false
	$1 \in D_x, 0 \notin D_y$	$\Rightarrow$	false
	$0 \notin D_x$	$\Rightarrow$	$y = 0$
	$1 \notin D_x$	$\Rightarrow$	$y = 1$
$\exists x \forall y : \neg x = y$	$1 \in D_y$	$\Rightarrow$	$x = 0$
	$0 \in D_y$	$\Rightarrow$	$x = 1$
$\exists x \exists y : \neg x = y$	$0 \notin D_y$	$\Rightarrow$	$x = 0$
	$1 \notin D_y$	$\Rightarrow$	$x = 1$
	$0 \notin D_x$	$\Rightarrow$	$y = 0$
	$1 \notin D_x$	$\Rightarrow$	$y = 1$

TABLE 2. Propagation rules for the $x_1 \Leftrightarrow \neg x_2$ primitive

original expression. Examples where this is the case are given in chapter 5 section 5.1.1 and chapter 6 section 6.1.1, for logical and numerical constraints respectively.

2.3.4. Search for QCSP. The literature contains two main approaches to search, both in the context of binary QCSP. Top-down search instantiates variables in quantification order (as implemented in QCSP-Solve). This is very similar to the search for QBF described in section 2.2.3. It is possible that top-down search may explore two identical subtrees, therefore bottom-up search was developed by Verger and Bessière [**97**]. This approach attempts to factor out identical subtrees, and explore them only once. As a third approach, binary QCSP can be encoded into QBF and solved with a QBF solver.

The three main approaches to binary QCSP are elaborated in the following three subsections. QCSP-Solve and its relatives use a search-based approach similar to search for QBF. Blocksolve attempts to reduce the size of the search tree by factoring out common branches. Encodings into QBF exploit efficient QBF solvers.

2.3.4.1. *Top-down search with QCSP-Solve.* Simple top-down QCSP search (algorithm 2, which is very similar to QBF search, algorithm 1) branches on variables in quantifier order. It is assumed that the *Simplify* procedure is able to simplify the QCSP instance to true or false when all variables have been instantiated, and that it can simplify away all instantiated variables. In practice, *Simplify* would do much more than this.

Algorithm 2 Simple top-down QCSP search algorithm

```
procedure TopdownSearch($\phi$: QCSP instance): Boolean
$\sigma$ ←Simplify($\phi$)
if $\sigma$ =true: return true
if $\sigma$ =false: return false
if $\sigma$ is of the form $\forall x_1 \ldots$:
    for all values $a \in D_1$:
        $t_1$ ←TopdownSearch($\sigma[x_1 = a]$) {Set $x_1$ to $a$ and recurse}
        if $t_1$ =false: return false
    return true
else: {$\sigma$ is of the form $\exists x_1 \ldots$}
    for all values $a \in D_1$:
        $t_1$ ←TopdownSearch($\sigma[x_1 = a]$) {Set $x_1$ to $a$ and recurse}
        if $t_1$ =true: return true
    return false
```

In QCSP-Solve [**54**], simple top-down search is augmented with preprocessing, forward checking of the binary constraints, application of the pure value rule, conflict directed backjumping (CBJ) and solution directed pruning (SDP). QCSP-Solve does not exactly follow the structure of algorithm 2 because of various complications. It is presented iteratively.

As a preprocessing step, binary arc-consistency is enforced to exhaustion (as described in section 2.3.1.2) and the binary pure value rule is applied to exhaustion (section 2.3.1.3). Binary quantified arc-consistency completely solves all constraints with quantifiers $\forall x_1 \forall x_2$ and $\exists x_1 \forall x_2$, therefore these types of constraint are deleted during preprocessing.

The QCSP-Solve algorithm performs the following reasoning at each search node.

- Forward checking is performed for the last assignment $x_i \mapsto a$. If this empties the domain of some variable, then the value a cannot extend to a solution.
- The pure value rule is applied to the current variable before branching on it. This potentially reduces the number of values to explore.
- For universal variables, before branching, each value is instantiated in turn and forward checking is performed. If any value leads to a contradiction (i.e. an empty domain) then the current assignments cannot lead to a solution.

These three forms of local reasoning could be integrated with algorithm 2, by means of the *Simplify* function. However, the other two features of QCSP-Solve, CBJ and SDP, go beyond the recursive structure of algorithm 2, because they both perform backjumping, which is non-chronological backtracking.

When the search reaches a dead end, CBJ is applied. CBJ constructs a *conflict set* for each variable x_i, which consists of the outer variables $x_{j<i}$ whose assignment $x_j \mapsto b$ has directly or indirectly contributed to the removal of some value in the domain D_i. CBJ is applied in two circumstances:

(1) When branching on existential variable x_i, and x_i has only one value a left in its domain D_i: if forward checking on a empties some other domain $D_{l>i}$, then the conflict set of x_i is used to backjump. The search procedure backtracks to the innermost variable in the conflict set.

(2) When branching on universal variable x_i, if forward checking on *any* value $a \in D_i$ empties some other domain $D_{l>i}$, then the conflict set of x_i is used to backjump, in the same way as for existential variables.

A possible execution of CBJ is illustrated in figure 9, where the solid arrows represent flow of control in the algorithm. This does not illustrate CBJ on a particular problem instance, but is just a general illustration. Without backjumping, after exploring $x_3 = 3$, the algorithm would have returned to x_2 and potentially explored other values in D_2.

If all variables are assigned and all constraints are satisfied, then we have a solution $\tau = \langle a_1, a_2, \dots, a_n \rangle$ and SDP is applied. SDP is inspired by SBJ for QBF. For the innermost universal variable x_i, the solution is adapted for each value $a \in D_i$ by changing τ_i to a and checking the constraints. If the constraints are all satisfied by the modified solution, value a can be ignored, since it has a solution. Thus values can be pruned from D_i. If D_i is not emptied, SDP backtracks directly to x_i and continues searching using the remaining values. If D_i is emptied, then SDP moves on to the next universal (in order of decreasing index i) and repeats the same process. If the domains of all universal variables are emptied by SDP, then the algorithm halts, having found a winning strategy.

QCSP-Solve is evaluated experimentally using random QCSP instances. Unfortunately the version of QCSP-Solve evaluated in the paper [**54**] contains a bug with the pure value rule. The

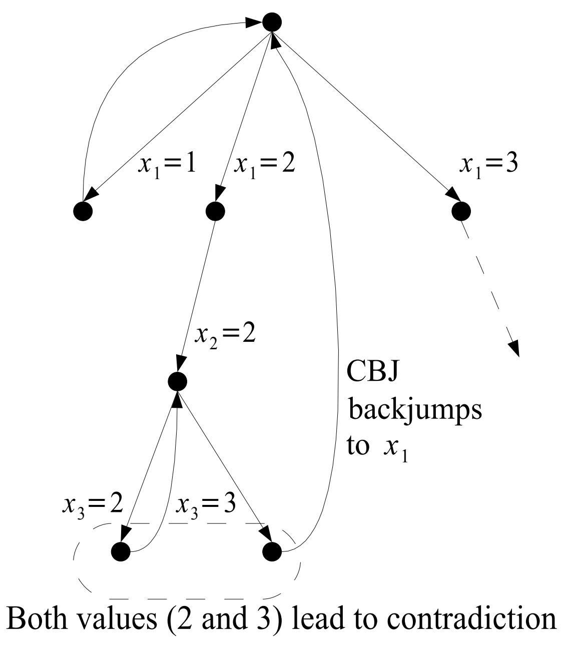


FIGURE 9. Conflict Backjumping in QCSP-Solve

algorithm presented there will remove the final value of a universal variable and immediately backtrack, assuming that the simplified instance is true. It is incorrect to remove the final value, since the pure value rule is a subsumption rule. Consider the following instance, which would trigger the bug: $\forall x_1 \in \{1,2\}\exists x_2 \in \{1,2\} : x_2 \neq x_2$. All values of x_1 are immediately pure, but the instance is not satisfiable. The experiments reported in this thesis are not subject to this bug.

2.3.4.2. *Top-down search and repair.* Stergiou later proposed to use repair-based methods in QCSP, with a general framework (RB-Schema) which incorporates propagation and repair-based methods [**93**]. Essentially this expands on the idea of SDP, by adapting (*repairing*) the solution to be compatible with different assignments to the universal variables. RB-Schema has two parameters: Propagate, the algorithm to propagate search assignments, and Repair, the algorithm which repairs a solution τ, after a change to τ_i, by changing values $\tau_{i+1}\dots\tau_n$ such that all constraints are satisfied. If Repair is not complete, RB-Schema is also not complete.

RB-Schema behaves like simple top-down search until the first solution is found. After that, the algorithm uses a combination of top-down search (using Propagate) and repair. The Repair algorithm is called when the value of a universal variable is changed, to adapt the last solution for the new value, and if this fails RB-Schema backtracks like top-down search.

Any repair-based technique subsumes SDP, however CBJ has not been embedded in RB-Schema. The paper presents some experiments on random binary QCSP instances [**93**], showing that RB-Schema can outperform QCSP-Solve.

2.3.4.3. *Bottom-up search with Blocksolve.* Verger and Bessière propose to search over the innermost quantified variables first, with their algorithm Blocksolve [**97**]. A block is a maximal substring of variables in the quantifier sequence that have the same quantifier. The variables are partitioned into blocks, and the blocks are paired into levels, where a universal block followed by an existential block forms one level.

The levels and the blocks are processed in reverse quantifier order, finding solutions to the innermost block first. Currently Blocksolve is restricted to binary constraints, and the innermost block is assumed to be existential.

Simple top-down search (algorithm 2) can explore two identical subtrees of the search tree. As a simple example, for a QCSP instance with variables $x_1 \dots x_n$, if $x_1 \dots x_m$ are not connected by any constraint to $x_{m+1} \dots x_n$, then whenever algorithm 2 reaches variable x_{m+1} it will explore a subtree which is identical in every case. The aim of Blocksolve is to avoid exploring identical subtrees of the search tree, by factoring them out.

For each existential block, starting with the innermost, Blocksolve uses a CSP search algorithm combined with some level of consistency (at least forward checking on the binary constraints). This is used to instantiate (if possible) the variables of the existential block, without causing empty domains for variables in other blocks. These instantiations are matched with compatible tuples (compatible w.r.t. the constraints) of the values of outer variables. This allows one instantiation of the current block to be used with many different instantiations of outer blocks, thus avoiding exploring identical subtrees.

Figure 10 shows a complete tree of possible assignments for a small example, with the two levels marked on the diagram. The simple top-down search algorithm could explore one of the trees marked with thick black lines, by first setting x_1 to 2. Blocksolve in contrast would first attempt to

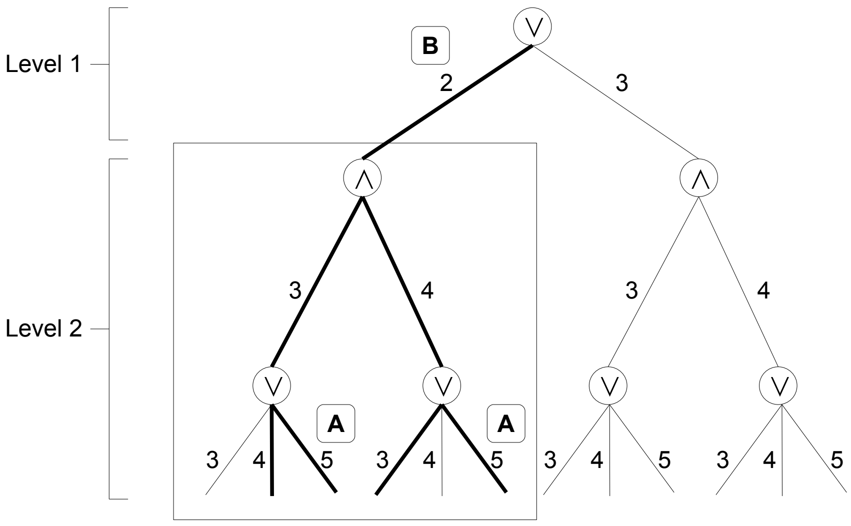


FIGURE 10. Illustration of the operation of Blocksolve on instance $\exists x_1 \in \{2,3\} \forall x_2 \in \{3,4\} \exists x_3 \in \{3,4,5\} : x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3$

instantiate all existential variables in the innermost level (only x_3 in this case). Blocksolve's value ordering heuristic favours values which are compatible with a large number of combinations of values of the universals in level 2. In this case, the instantiation $x_3 \mapsto 5$ would be made, labelled with an A in the figure. This instantiation is compatible with both values of x_2. Blocksolve has constructed the subtree marked with a box in the figure.

Following this, Blocksolve would move up to level 1 and attempt to instantiate the existential variable at this level. The instantiation $x_1 \mapsto 2$ is made (labelled B in the figure), and the algorithm terminates at this point. However, if it had not been possible to instantiate the existentials at level 1, then Blocksolve would have returned to level 2 to find another subtree, guided by the failure at level 1.

This example has only one existential variable at each level, so it does not illustrate search at each level. This search would probably take up a substantial part of the running time on a large instance.

The experimental evaluation shows Blocksolve is often faster than QCSP-Solve for soluble random instances. On the other hand, the experiments in this thesis (in chapter 4 section 4.6.1.2) show that there are instances where Blocksolve performs several orders of magnitude worse than

QCSP-Solve. The probable reason for this is that the innermost block of existentials contains a large number of variables with large domains (as an artifact of the hidden variable encoding), and that search in this innermost block is taking a large amount of time.

Currently Blocksolve is restricted to binary constraints, and it is not clear how it might be extended for non-binary constraints. If this limitation can be overcome, then a huge array of propagation algorithms from CSP could be applied.

2.3.5. Restricted forms of QCSP. There is a small body of work on restricted forms of QCSP. Fargier proposed mixed CSP [**39**], where a set of *parameters* represents the state of the world, and the CSP is solved repeatedly for all (or a large subset of) the possible worlds. If the parameters are $\mathcal{P}$ and decision variables $\mathcal{X}$, the quantifier sequence of the equivalent QCSP would be $\forall\mathcal{P}\exists\mathcal{X}$.

Gervet and Yorke-Smith define uncertain CSP [**102**], which is very similar to mixed CSP, and propose various *closures* including the most robust individual solution, and the minimal set of solutions which covers all possible worlds. This work is very interesting but the algorithms bear little resemblance to those for unrestricted QCSP.

Chen and Dalmau [**29**] and Börner et al. [**23**] identify tractable subsets of QCSP. Chen and Dalmau give an algorithm to enforce k-consistency [**29**], however the algorithm they propose is designed to give tractability results, with no reference to optimality or embedding in a search procedure, therefore I will not consider this algorithm to be competitive with the algorithms I develop in this thesis.

2.4. Other formalisms

QCSP is not the only extension of CSP intended to address uncertainty. I briefly review three other formalisms.

2.4.1. Stochastic CSP. Walsh introduced Stochastic CSP and gave some basic algorithms [**99**], and Tarim, Manandhar and Walsh proposed an encoding of Stochastic CSP into CSP [**75**, **96**]. Balafoutis and Stergiou correct and extend the work of Walsh, introducing the stochastic equivalent of generalized arc consistency and integrating it within search [**8**], generalizing the GAC2001/3.1 algorithm.

Stochastic CSP allows decision variables (which the solver must find values for) and stochastic variables (which follow some probability distribution), and the constraints must all be satisfied with some probability ϕ. The variables have an ordering, similar to a quantifier sequence. A *policy* (similar to a winning strategy) must assign the decision variables based on the values of previous decision and stochastic variables. Stochastic CSP may be PSPACE-complete in general, and if $\phi = 1$ then stochastic CSP is equivalent to QCSP. The framework is extended with an optimization function.

Interesting problems can be modelled in stochastic CSP. Walsh gives a production planning example, where customer orders are satisfied with probability 0.8, given stochastic demand for the product [**99**]. Tarim et al. give further examples from portfolio management, farming yield management and production/inventory control [**75**, **96**].

2.4.2. Strategic CSP. Bessière and Verger identify a difficulty in modelling with QCSP [**17**]. When modelling an adversarial game in either QCSP or QBF, the moves of one player are represented with existential variables, and the other with universal variables. The variables are quantified in chronological order. The difficulty is that the valid moves often depend on previous moves. For the existential move variables, the invalid moves can be pruned by consistency. Unfortunately, with universal variables, if a value is inconsistent then the entire QCSP instance is false, so some other device must be used to deal with invalid moves.

Gent and Rowley encode Connect-4 into QBF using *indicator variables*, which indicate when an invalid move has been made [**46**]. When a universal variable is set to an invalid value, unit propagation simplifies the formula to true. The invalid value is not pruned, but the search procedure branches for every value of each universal variable. The encoding is also very complex. Ansótegui, Gomes and Selman [**5**] also explore this modelling difficulty, proposing to avoid it by altering the search algorithm of QBF solvers to avoid invalid moves. The invalid values of universal variables are not pruned, therefore no further pruning can occur subsequently.

Bessière and Verger address this modelling issue by defining Strategic CSP [**17**]. This new formalism has a new type of variable similar to a universal variable, but which can be pruned by certain constraints. These are quantified with existential variables, and there is also a set of *state* variables which are not quantified in the conventional way. (The state variables are handled separately from those in the quantifier sequence.) The state variables are intended to represent the board state of a game, or the state of the environment. The rules of a game are expressed by constraints over the state variables. The quantified variables are connected with the state variables by channelling constraints. Connect-4 is modelled tersely in Strategic CSP, and solved effectively (by top-down search and consistency), demonstrating its applicability.

However, Strategic CSP has some unusual properties. For example, where A is a state variable and E is existential, the constraints $A = E \wedge E = 1$ are not equivalent to $A = 1 \wedge E = 1$. The framework may need to be re-defined in a cleaner way to avoid this problem.

2.4.3. QCSP+. Benedetti et al. [**12**] perceive the same difficulty with modelling as Bessière and Verger [**17**], and propose QCSP+ to address it. This is a generalization of QCSP. Written as first-order logic, a quantifier in QCSP+ has one of the following two forms.

$$\exists X_1 : L_1 \wedge [\ldots]$$

$$\forall X_2 : L_2 \Rightarrow [\ldots]$$

In these expressions, X_1 and X_2 are sets of variables, and L_1 and L_2 are instances of CSP. The ellipsis represents the rest of the QCSP+ instance, including other quantifiers if necessary. QCSP+ also has free (unquantified) variables, and a set of constraints.

The CSPs L_1 and L_2 allow to encode conditions on the quantifiers (e.g. rules of a game) which rule out some combinations of values. Each CSP instance can contain variables from the quantifier it is attached to, from quantifiers to the left, and from the set of free variables. Therefore, the approved combinations of values of universal variables can depend on the values of any variables quantified to the left. This solves the modelling difficulty, and allows the re-use of CSP constraint propagators.

QCSP+ can be straightforwardly encoded into QCSP. However, in QCSP a top-down search solver may branch for each value of a universal variable, regardless of whether it satisfies the conditions. This difficulty can be solved by making use of the pure value rule, as it is used in the models of Connect 4 and faulty job shop scheduling presented later in this thesis. Therefore I see no advantage of QCSP+ compared to QCSP with the pure value rule.

Benedetti et al. do not conclusively show that QCSP+ can be solved efficiently: the solver uses conventional CSP propagation algorithms rather than the stronger quantified variants [**12**]. However, experimental results are promising.

2.5. Summary

This chapter reviewed the most relevant literature from the fields of constraint programming and QBF. This was followed by detailed examination of the QCSP literature, and some discussion of other, similar formalisms. In each of the three fields of constraint programming, QBF and QCSP, the definitions and algorithms related to local reasoning (particularly consistency) and search are covered in most detail, since they are most relevant to this thesis.

CHAPTER 3

A search framework for QCSP

3.1. Introduction

The aim of this thesis is to explore whether QCSP is a useful formalism, and whether QCSP can be effectively solved using a combination of consistency and search, with a particular emphasis on consistency. The point of this chapter is to provide the theoretical background for the following chapters about consistency, and to provide the algorithmic framework that the consistency algorithms fit into. This second part includes search algorithms and an algorithm to enforce the pure value rule, using consistency.

The theoretical background (section 3.2) includes definitions of finite QCSP, local consistency in general and specific definitions of consistency, as well as the pure value rule and various other notions which are used in the following chapters. In this section, the definition of WQGAC (a consistency notion) and of the pure value rule (as a form of local reasoning) are novel.

Two search algorithms are presented in section 3.3, one to decide satisfiability of QCSP and the other to perform optimization. Both allow a winning strategy to be extracted, represented by a tree. For the second algorithm, each branch of the winning strategy is optimal. The second algorithm is new, since there are no QCSP optimization algorithms in the literature.

The search algorithms interface with a constraint queue for the purpose of maintaining local consistency (section 3.4). The pure value rule is another form of local reasoning which is able to prune values from universal variables. Two novel schemes to enforce this are described in section 3.5.

The various algorithms are implemented in a solver called Queso. I give an overview of the structure and implementation decisions of Queso in section 3.7.

Finally, in section 3.8, two encodings from binary QCSP to QBF are discussed. The second of these is a novel improvement of the first.

3.2. Definitions

The finite quantified constraint satisfaction problem is defined in chapter 2, definition 2.3.1. It is reproduced here for convenience.

DEFINITION 3.2.1. Finite Quantified Constraint Satisfaction Problem

A QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ is defined as a set of n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, a set of domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ where $D_i \subsetneq \mathbb{Z}$, $|D_i| < \infty$ is the finite set of all potential values of x_i, a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints, and a quantifier sequence $\mathcal{Q} = \langle Q_1 x_1, \ldots, Q_n x_n \rangle$ where each Q_i is a quantifier, $\exists$ (existential, 'there exists') or $\forall$ (universal, 'for all').

I use n for the number of variables, e for the number of constraints, and for QCSPs where all variables have the same domain, d for the cardinality of the domain. r is used for the arity of a constraint.

Before defining the semantics of a QCSP, it is necessary to define constraints. I use *subsequence* in the sense where $\langle 1, 3 \rangle$ is a subsequence of $\langle 1, 2, 3, 4 \rangle$.

DEFINITION 3.2.2. Constraint

Within QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$, a constraint $C_k \in \mathcal{C}$ consists of a sequence of $r > 0$ variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ with respective domains $\mathcal{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle$ s.t. $\mathcal{X}_k$ is a subsequence of $\mathcal{X}$, $\mathcal{D}_k$ is a subsequence of $\mathcal{D}$, and each variable x_{k_i} and domain D_{k_i} matches a variable x_j and domain D_j in $\mathcal{P}$. C_k has an associated set $C_k^S \subseteq D_{k_1} \times \cdots \times D_{k_r}$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_k$.

The variables $\mathcal{X}_k$ are called the *scope* of the constraint. C_k^S may be represented implicitly, for example by an algebraic expression. Notice that the set C_k^S depends on the domains, hence in a QCSP solver which simplifies the problem $\mathcal{P}$ by removing values from the domains to form a smaller problem $\mathcal{P}'$, C_k^S in $\mathcal{P}'$ is reduced accordingly.

Simplifying a QCSP by reducing the size of the domains is key to the definition of QCSP semantics, and also to the algorithms for solving QCSP as we will see later in this chapter. This simplification is defined here. A variable x_i is called *instantiated* or *set* if $|D_i| = 1$.

Let the initial problem be $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ with domains $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ and the simplified problem be $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}', \mathcal{C}', \mathcal{Q} \rangle$ with domains $\mathcal{D}' = \langle D'_1 \subseteq D_1, \ldots, D'_n \subseteq D_n \rangle$ and constraints $C'_k \in \mathcal{C}'$. Now $\forall k \, : \, C_k^{S'} = C_k^S \cap (D'_{k_1} \times D'_{k_2} \times \cdots \times D'_{k_m})$. Only the domains and sets C_k^S are different between $\mathcal{P}$ and $\mathcal{P}'$. When the simplification involves setting a variable x_i to value $a \in D_i$, I use the following notation: $\mathcal{P}' = \mathcal{P}[D_i = \{a\}]$.

In order to define the semantics of QCSP, function firstx($\mathcal{P}$) gives the first uninstantiated variable, or $\perp$ if no such variable exists.

$$\text{if } \forall i \, : \, |D_i| = 1 \text{ then firstx}(\mathcal{P}) = \perp$$
$$\text{otherwise firstx}(\mathcal{P}) = x_i \text{ s.t. } |D_i| > 1 \text{ and } \nexists j < i : |D_j| > 1$$

When firstx($\mathcal{P}$) $= \perp$, all domains D_i are unit and the set $D_{k_1} \times D_{k_2} \times \cdots \times D_{k_n}$ consists of a single tuple, therefore for all constraints $\forall k \, : \, \left|C_k^S\right| \in \{0, 1\}$. Following Apt [**6**], if $\left|C_k^S\right| = 0$ the constraint is *failed* and if $\left|C_k^S\right| = 1$ the constraint is *solved*.

Informally, a QCSP instance is *satisfiable* if there is some way of setting the existential variables such that all constraints are solved, whatever values the universal variables take. This is analogous to the concept of satisfiability in traditional CSP. The following definition formalises this concept. The definition has a base case where all variables are set, and a recursive case which is subdivided for existential and universal variables.

DEFINITION 3.2.3. Semantics of the QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$

- In the case where firstx($\mathcal{P}$) $= \perp$: If all constraints $C_k \in \mathcal{C}$ are solved, the problem is satisfiable. If any constraint is failed, $\mathcal{P}$ is unsatisfiable.
- Otherwise, let firstx($\mathcal{P}$) $= x_i$. If $(\exists x_i) \in \mathcal{Q}$ then $\mathcal{P}$ is satisfiable iff there exists a value $a \in D_i$ such that the simplified problem $\mathcal{P}[D_i = \{a\}]$ is satisfiable. If $(\forall x_i) \in \mathcal{Q}$ then $\mathcal{P}$ is satisfiable iff for all values $a \in D_i$ the simplified problem $\mathcal{P}[D_i = \{a\}]$ is satisfiable.

The finite constraint satisfaction problem (CSP) has been extensively studied, and is a subset of QCSP where all variables are existentially quantified. It is defined below.

DEFINITION 3.2.4. Finite Constraint Satisfaction Problem

A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ with n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, is defined to be a QCSP $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ where $\mathcal{Q} = \exists x_1, \exists x_2, \ldots, \exists x_n$.

CSPs are most commonly solved by interleaving constraint propagation and search. Work on QCSPs has mainly followed the same path so far [**22**, **54**, **74**], and I take the same approach here.

3.2.1. Solutions, strategies and scenarios. All three concepts defined in this subsection are to do with solving QCSPs. A *solution* to a QCSP is an assignment to all variables such that the constraints are satisfied. This is a linear structure, and the existence of a solution does not prove that the QCSP is satisfiable under definition 3.2.3. The second object is a *strategy*, which specifies how existential variables should be set with reference to the values of universal variables. Finally the *winning strategy* is defined. The existence of a winning strategy does prove that a QCSP is satisfiable. This is proven in theorem 3.2.9 below. Unless stated otherwise, the definitions closely follow those of Bordeaux et al. [**21**].

An intuition for understanding quantifier alternation is to consider existential and universal as two players who are interacting. The aim of the existential player is to satisfy the set of constraints $\mathcal{C}$, by assigning values to existential variables. The aim of the universal player is to cause at least one constraint to be unsatisfied. The variables are assigned in quantification order (as definition 3.2.3 suggests), however two adjacent variables with the same quantifier may be transposed. If the existential player can win the game whatever moves (assignments) the universal player makes, then the QCSP is satisfiable.

A certificate proving the satisfiability of the QCSP is called a winning strategy. It describes the moves the existential player must make to win, given the moves the universal player makes in response. In the literature there are two ways of formalising a winning strategy. The form favoured here is that of Bordeaux, Cadoli and Mancini [**21**], who define a family of total functions which map universal assignments to later existential assignments. (The other form is the solution tree, discussed in section 3.2.2.) First I will define solutions, then strategies.

A solution to the QCSP $\mathcal{P}$ is a tuple t of values for all variables, such that $t_i \in D_i$, and in the simplified QCSP $\mathcal{P}'$ with domains $D_i = \{t_i\}$, all the constraints in $\mathcal{C}'$ are solved. The set of all solutions of $\mathcal{P}$ is called $\text{sol}^{\mathcal{P}}$.

DEFINITION 3.2.5. Solution

A tuple t is a solution to QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ iff $|t| = n$ and $\forall i \; : \; t_i \in D_i$ and in the simplified QCSP $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}', \mathcal{C}', \mathcal{Q} \rangle$ where $\mathcal{D}' = \langle \{t_1\}, \{t_2\}, \ldots, \{t_n\} \rangle$, all constraints C'_k are solved.

To define a strategy requires tuples of outer universal assignments. The whole set of outer universal assignments for variable x_i is denoted OUA_i. The universal variables outside (i.e. earlier in the quantifier prefix than) x_i are $U_i = \{x_j \in \mathcal{X} | Q_j = \forall \; \wedge \; j \leq i\}$. Given $U_i = \{x_j, x_l, \ldots\}$, then $\text{OUA}_i = \{\langle v_j, v_l, \ldots \rangle | v_j \in D_j \; \wedge \; v_l \in D_l \; \wedge \cdots \}$. That is, all tuples constructed from the initial domains of the outer universal variables.

DEFINITION 3.2.6. Strategy

A strategy S is a set of functions $S = \{s_i | Q_i = \exists\}$ of the following type: for each x_i which is existential, function s_i associates to each tuple $t \in \text{OUA}_i$ a value in D_i. Therefore S specifies which value should be assigned to every existential variable depending on the values assigned to the preceding universal variables.

Note that if some variable x_i has no outer universal variables, there is just one member of OUA_i, the empty tuple, which is mapped to a single value from D_i.

Bordeaux, Cadoli and Mancini continue by defining a scenario. This is a tuple of values which form a complete assignment to variables $x_1 \ldots x_n$. The scenario is affected by both the strategy and the values assigned to universal variables. To define the set of scenarios, we need to extract the values of outer universal variables from a scenario t. Given $U_i = \{x_j, x_l, \ldots\}$, $\text{OUA}_i(t) = \langle t_j, t_l, \ldots \rangle$.

DEFINITION 3.2.7. Scenario

The set of scenarios of a strategy S for a QCSP $\mathcal{P}$, denoted sce(S), is the set of all tuples t which are such that, for each $i \in 1 \dots n$, we have:

$$t_i \in D_i$$

and

$$\text{if } Q_i = \exists \text{ then } t_i = s_i(\text{OUA}_i(t))$$

This definition allows the values of existential variables to be consistent with the strategy and the values of outer universal variables. There is no restriction to the values of universal variables. In the degenerate case where all variables are universal, sce$(S) = D_1 \times \cdots \times D_n$.

If all the scenarios of a strategy satisfy the constraints, then the strategy is a *winning strategy*.

DEFINITION 3.2.8. Winning strategy

A strategy S is a winning strategy for the QCSP $\mathcal{P}$ iff sce$(S) \subseteq$ sol$^{\mathcal{P}}$.

Winning strategies are particularly interesting, since the existence of such a strategy proves the solubility of the QCSP. A winning strategy is also useful in itself, because it encodes a solution to the original problem. For example, if the QCSP represented a contingent planning problem, a winning strategy would represent a contingent plan.

THEOREM 3.2.9. *QCSP instance $\mathcal{P}$ is satisfiable under definition 3.2.3 if and only if there exists a winning strategy S for $\mathcal{P}$.*

PROOF. The proof follows the recursive structure of definition 3.2.3.

Let firstx$(\mathcal{P}) = x_i$. If $Q_i = \exists$ then function $s_i \in S$ specifies the value a for x_i as a function of the values of outer universals $x_{j<i}$. Now according to definition 3.2.3 the simplified problem $\mathcal{P}[D_i = \{a\}]$ must be satisfiable so we recurse (n.b. the definition of scenario requires that x_i is instantiated according to function s_i). If $Q_i = \forall$ then for all values $a \in D_i$ the simplified

problem $\mathcal{P}[D_i = \{a\}]$ must be satisfiable, so we recurse for all values $a \in D_i$ (n.b. the definition of scenario allows a universal variable to take any value in its domain).

In the base case where firstx($\mathcal{P}$) $= \perp$, all variables have been instantiated in accordance with S. Where the instantiations are $x_1 \mapsto a_1, \dots, x_n \mapsto a_n$, $\tau = \langle a_1, \dots, a_n \rangle$. By the definition of scenario, $\tau \in \text{sce}(S)$. By the definition of winning strategy, $\text{sce}(S) \subseteq \text{sol}^{\mathcal{P}}$, therefore $\tau \in \text{sol}^{\mathcal{P}}$ and all constraints are solved, as required. $\square$

We denote by WIN$^{\mathcal{P}}$ the set of winning strategies of $\mathcal{P}$. WIN$^{\mathcal{P}}$ is an important set for computing consistency. The set of scenarios sce(WIN$^{\mathcal{P}}$) (which is denoted *outcomes* by Bordeaux et al. [**21**]) can be used to define consistency in the following way: the set of supported assignments is exactly the set of assignments contained in scenarios in sce(WIN$^{\mathcal{P}}$). The set of supported assignments is $\{x_i \mapsto a | t \in \text{sce}(\text{WIN}^{\mathcal{P}}) \ \wedge \ t_i = a\}$. I denote this consistency *Strong Quantified GAC* (SQGAC) and it is equivalent to the definition of inconsistency by Bordeaux et al. [**21**]. It is formalised in the definition of SQGAC (definition 3.2.13).

3.2.2. Solution trees. A winning strategy may be alternatively represented as a tree. This is the approach taken by Verger and Bessière, who define such a tree as follows.

DEFINITION 3.2.10. Solution tree of QCSP instance $\mathcal{P}$.

A solution tree for $\mathcal{P}$ is a tree such that:

- the root node r has no label,
- every node s at distance i ($1 \leq i \leq n$) from the root r is labelled by an instantiation $x_i \mapsto v$ where $v \in D_i$,
- for every node s at depth i ($0 \leq i \leq n-1$), the number of successors of s in the tree is $|D_{i+1}|$ if x_{i+1} is a universal variable or 1 if x_{i+1} is an existential variable. When x_{i+1} is universal, every value w in D_{i+1} appears in the label of one of the successors of s,
- for any leaf, the instantiation on $x_1, \dots, x_n$ defined by the labels of nodes from r to the leaf satisfies all constraints in $\mathcal{C}$ [**97**].

The definition is quoted almost exactly, except for some notation. Henceforth I will refer to a tree $T = \langle V, E, r, L \rangle$, where r is the root, V is the set of vertices, E is the set of edges and L is the labelling (a function $L : V \setminus \{r\} \to \{x_i \mapsto a | a \in D_i\}$).

Verger and Bessière clearly intended the solution tree to be a representation of a winning strategy, equivalent to the other representation using a family of functions. Indeed there is a bijection between solution trees T and and winning strategies S for any QCSP instance $\mathcal{P}$. I show this by giving an isomorphism between a solution tree T and the equivalent winning strategy S.

The isomorphism connects every node in the tree which is labelled with an existential variable, to a mapping in one of the functions in S, and vice versa. This is illustrated in figure 11, where there are five nodes labelled with existential variables, and five equivalent mappings in the functions of the winning strategy. In the figure, for x_4 there are four paths from r to some node labelled with x_4. These four paths correspond to four possible combinations of inner universal assignments. The isomorphism depends on this correspondence.

The tree T branches for each value of universal variables, and for exactly one value of existential variables. Therefore at any level i of the tree, where all nodes are labelled with x_i, the number of nodes at that level is the product $\prod \{|D_j| \, | 1 \leq j \leq i, \ Q_j = \forall\}$. For each possible combination of values of outer universal variables $x_{j \leq i}$, there is a path from r to a node at level i. Where x_i is existential, this exactly matches the definition of the set OUA_i, which contains all tuples of outer universal assignments. Each element $t \in \text{OUA}_i$ corresponds to a path from r to a node labelled with x_i, and vice versa.

For all existential variables $x_i \in \mathcal{X}$, the winning strategy contains a total function $s_i \in S$ of type $s_i : \text{OUA}_i \to D_i$. Each mapping in s_i, $s_i(t) = a_i$, corresponds to a node in T which is labelled $x_i \mapsto a_i$ and vice versa.

Finally, the corner case where all variables are universal. S is the empty set, and the set of scenarios is $\text{sce}(S) = D_1 \times D_2 \times \cdots \times D_n$. Since S is a winning strategy, $\text{sce}(S) \subseteq \text{sol}^{\mathcal{P}}$. In T it is required by the definition that each path from r to a leaf node corresponds to a solution in $\text{sol}^{\mathcal{P}}$. Since T branches for all values of each variable, this is identical to the requirement for S.

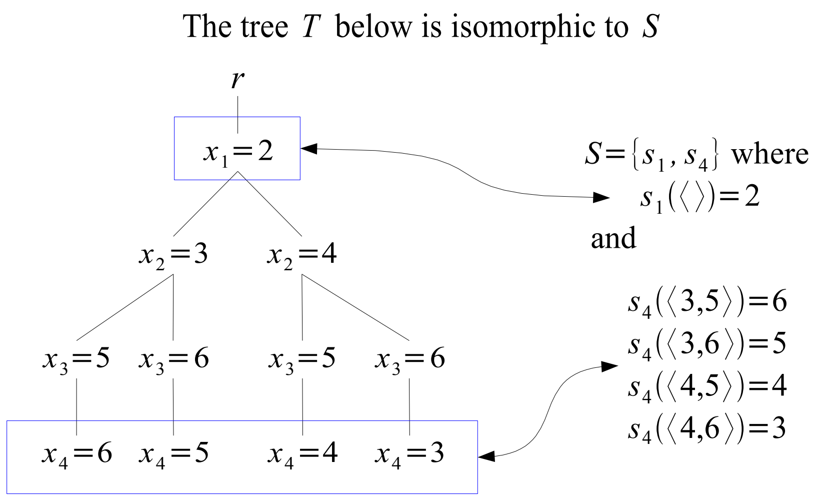

The tree T below is isomorphic to S
r
$x_1=2$
$x_2=3$
$x_2=4$
$x_3=5$
$x_3=6$
$x_3=5$
$x_3=6$
$x_4=6$
$x_4=5$
$x_4=4$
$x_4=3$
$S=\{s_1, s_4\}$ where
$s_1(\langle\rangle)=2$
and
$s_4(\langle 3,5\rangle)=6$
$s_4(\langle 3,6\rangle)=5$
$s_4(\langle 4,5\rangle)=4$
$s_4(\langle 4,6\rangle)=3$

FIGURE 11. For a QCSP instance $\exists x_1 \in \{1,2\}\forall x_2 \in \{3,4\}\forall x_3 \in \{5,6\}\exists x_4 \in \{3,4,5,6\} : \mathcal{C}$, example of mapping between a solution tree and a winning strategy.

There is a close relationship between T and the scenarios sce(S). Each leaf node in T corresponds to a scenario in sce(S). The scenario can be constructed from the path from r to the leaf node: the sequence of labels $x_1 \mapsto a_1, x_2 \mapsto a_2, \ldots, x_n \mapsto a_n$ corresponds to the scenario $\langle a_1, a_2, \ldots, a_n\rangle$.

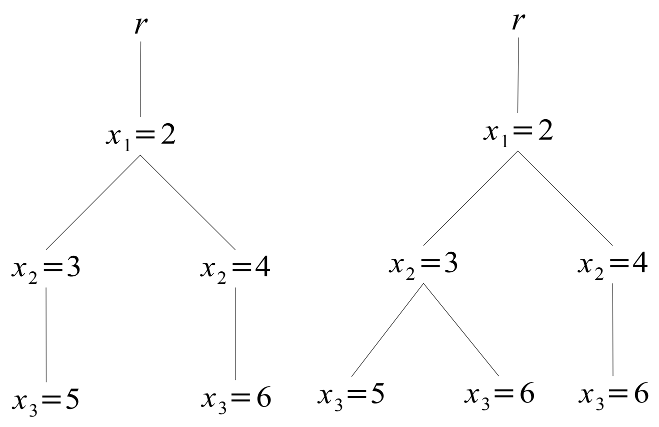
To illustrate solution trees, consider the following QCSP [**21**].

$$\exists x_1 \in \{2,3\}\ \forall x_2 \in \{3,4\}\ \exists x_3 \in \{3,4,5,6\} :\ x_1 + x_2 \leq x_3$$

There is a winning strategy $S = \{s_1,\ s_3\}$ such that $s_1(\langle\rangle) = 2$, $s_3(\langle 3\rangle) = 5$, $s_3(\langle 4\rangle) = 6$. S is represented by the tree shown in figure 12(a). Just the labels are shown for all vertices except r.

More than one winning strategy can be represented in a single tree. For example there is a second winning strategy for the problem above: $S' = \{s'_1,\ s'_3\}$ such that $s'_1(\langle\rangle) = 2$, $s'_3(\langle 3\rangle) = 6$, $s'_3(\langle 4\rangle) = 6$. The tree representing both S and S' is shown in figure 12(b).

In this second example, two winning strategies are represented using one tree. The tree corresponds to a set of three scenarios, $\langle 2,3,5\rangle$, $\langle 2,3,6\rangle$ and $\langle 2,4,6\rangle$, which is the union of the sets of scenarios of the two winning strategies.



(a) Single winning strategy (b) Two winning strategies

FIGURE 12. Strategy trees

3.2.3. Multiple winning strategy trees. The idea of representing multiple winning strategies in one tree finds a use in the SQGAC propagation algorithm in chapter 4. In that context, a multiple winning strategy tree containing *all* winning strategies is maintained. If any value is not contained in the tree, it is inconsistent.

The multiple winning strategy tree simply represents a larger set of scenarios. The main difference is that nodes labelled with existential variables can have siblings, therefore multiple values for existential variables can be represented. It is defined below.

DEFINITION 3.2.11. Multiple winning strategy tree (MWST) for a set of winning strategies $\mathcal{S} = \{S_1, \ldots, S_n\}$ of QCSP instance $\mathcal{P}$.

An MWST T for $\mathcal{P}$ is a tree such that:

- the root node r has no label,
- every node s at distance i ($1 \le i \le n$) from the root r is labelled by an instantiation $x_i \mapsto v$ where $v \in D_i$,
- for every node s at depth i ($0 \le i \le n-1$), the number of successors of s in the tree is $|D_{i+1}|$ if x_{i+1} is a universal variable or 1 *or more* if x_{i+1} is an existential variable. When x_{i+1} is universal, every value w in D_{i+1} appears in the label of one of the successors of s, and when x_{i+1} is existential, the labels of the successors of s must be distinct.

- for any leaf, the instantiation on $x_1, \ldots, x_n$ defined by the labels of nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.
- The set of winning strategies $\mathcal{S} = \{S_1, \ldots, S_n\}$ has a combined set of scenarios $\text{sce}(\mathcal{S}) = \text{sce}(S_1) \cup \cdots \cup \text{sce}(S_n)$. There is a bijection between leaf nodes w in T and scenarios $\text{sce}(\mathcal{S})$ as follows: the path from r to w in T traverses the sequence of vertices labelled $x_1 \mapsto a_1, x_2 \mapsto a_2, \ldots, x_n \mapsto a_n$, corresponding to the scenario $\langle a_1, a_2, \ldots, a_n \rangle \in \text{sce}(\mathcal{S})$.

Note that for all nodes s in an MWST T, the labels of children of s are distinct. Therefore, for any two distinct tuples τ, τ' in $\text{sce}(\mathcal{S})$ (represented by two paths in T), the lowest index i where $\tau_i \neq \tau'_i$ must also be the point where the two paths in T diverge. The point of divergence is unique. Because of this, T is unique in representing $\text{sce}(\mathcal{S})$.

An MWST can be thought of as a trie [**40**], and $\text{sce}(\mathcal{S})$ as the set of words stored in the trie.

The definition above leads naturally to a consistency definition when $\mathcal{S}$ includes all winning strategies for the QCSP $\mathcal{P}$ ($\mathcal{S} = \text{WIN}^{\mathcal{P}}$). If some assignment $x_i \mapsto a$ is contained in the tree (i.e. $\exists v \in V : F(v) = x_i \mapsto a$) then it is also contained in a tuple in $\text{sce}(\text{WIN}^{\mathcal{P}})$, and $x_i \mapsto a$ is consistent, otherwise not. This is formally defined later.

3.2.4. Local consistency. To enforce consistency on the whole problem $\mathcal{P}$ including a conjunction of constraints is intractable in general [**21**], therefore Bordeaux et al. define *local consistency* (consistency on individual constraints) which is tractable in general if the arity of the constraints is bounded above by a constant. Local consistency is also tractable for particular classes of constraints of any arity (for example, logic constraints and sum, which will be explored in later chapters).

Local consistency is consistency in the *local* problem $\mathcal{P}_k = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}_k = \{C_k\}, \mathcal{Q} \rangle$ containing just the constraint C_k. $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{Q}$ are identical to their counterparts in the *global* problem $\mathcal{P}$. If an assignment is inconsistent in $\mathcal{P}_k$ then it must be inconsistent in $\mathcal{P}$, since the semantics of $\mathcal{P}$ require that all constraints are satisfied. This allows us to detect inconsistent values in $\mathcal{P}$ tractably. I call $\mathcal{P}$ *locally consistent* iff *all* the local problems $\mathcal{P}_k$ are consistent. This definition of $\mathcal{P}_k$ is taken from Bordeaux et al. [**21**], so theorems in their work apply without alteration here.

Unfortunately, defining $\mathcal{P}_k$ like this means that the winning strategies of $\mathcal{P}_k$ cover too many variables. Consider a second reduced problem $\mathcal{P}'_k = \langle \mathcal{X}_k, \langle D_{k_1}, \ldots, D_{k_r} \rangle, \{C_k\}, \langle Q_{k_1}, \ldots, Q_{k_r} \rangle \rangle$. There is a simple relationship between the two: every scenario of a winning strategy of $\mathcal{P}'_k$, $t' \in \text{sce}(\text{WIN}^{\mathcal{P}'_k})$ is a subsequence of some $t \in \text{sce}(\text{WIN}^{\mathcal{P}_k})$ (where $\forall i \in \{1 \ldots r\} : t'_i = t_{k_i}$), and every scenario of a winning strategy $t \in \text{sce}(\text{WIN}^{\mathcal{P}_k})$ is a supersequence of some $t' \in \text{sce}(\text{WIN}^{\mathcal{P}'_k})$.

By any definition of consistency, $\mathcal{P}'_k$ is consistent iff $\mathcal{P}_k$ is consistent, because the extra variables are irrelevant to the constraint. From here on, by an abuse of notation, I refer to the winning strategies WIN^{C_k} of constraint C_k, meaning the winning strategies of the local problem $\mathcal{P}'_k$.

3.2.5. Definitions of consistency. I say two consistencies, consistency A and consistency B, are *equivalent* iff (in the given context) a constraint is A iff it is also B. For example, consistencies for QCSP are sometimes equivalent to consistencies for CSP when there are no universally quantified variables in the scope of the constraint.

Consistencies are defined on constraints, so that constraints are either consistent or inconsistent. To make an inconsistent constraint C_k consistent according to some definition, a set of values is removed from the domains of $\mathcal{P}$ such that C_k is consistent. This set will be unique under any definition of consistency in this thesis. This is dealt with more carefully at the end of the section.

An algorithm which does this is a *propagation algorithm*. If the set of values to be removed contains a value of a universal variable, then the constraint cannot be made consistent, and $\mathcal{P}$ simplifies to false. This can be seen from the definition of QCSP semantics (definition 3.2.3).

DEFINITION 3.2.12. Generalized Arc-Consistency (GAC)

A constraint C_k is Generalized Arc-Consistent (GAC) iff for each variable $x_i \in \mathcal{X}_k$ and value $a \in D_i$ there exists a tuple $t \in C_k^S$ where $t_i = a$.

Note that the definition of C_k^S is the set of allowed tuples within the current domains. The definition of GAC clearly ignores the quantification of the variables. However, in QCSP there is an additional rule that values may not be removed from the domains of universal variables by consistency (by the definition of QCSP semantics). If such a removal is required to establish

GAC, then the constraint cannot be made GAC and the QCSP instance is false. This means that establishing GAC can have a greater effect in QCSP than in CSP.

DEFINITION 3.2.13. Strong Quantified GAC (SQGAC)

M is a multiple winning strategy tree representing all winning strategies WIN$^{\mathcal{P}_k}$ for constraint C_k. C_k is SQGAC iff for each variable $x_{k_i} \in \mathcal{X}_k$ and value $a \in D_{k_i}$, a vertex labelled $x_{k_i} \mapsto a$ is contained in M (i.e. $\exists v \in V : F(v) = x_{k_i} \mapsto a$).

SQGAC is defined based on the multiple winning strategy tree containing all winning strategies for constraint C. Since there is a bijection between the MWST and the set of scenarios sce(WIN$^{\mathcal{P}_k}$), SQGAC is equivalent to the consistency defined by Bordeaux et al. [**21**] (reviewed in chapter 2 section 2.3.1.1). In their definition, a value $x_{k_i} \mapsto a$ is inconsistent iff $\nexists t \in$ sce(WIN$^{\mathcal{P}_k}$) $: t_i = a$. The values that are inconsistent by their definition are exactly the values that would have to be removed for the constraint C_k to be SQGAC in my definition. This is because the MWST containing all winning strategies is a compressed form of the scenarios sce(WIN$^{\mathcal{P}_k}$). The only difference is that their definition operates on values whereas mine operates on the constraint. Bordeaux et al. simply named their property *inconsistency*. I rename it SQGAC to distinguish it from the other definitions of consistency in this thesis.

Weak Quantified GAC (WQGAC) relaxes the condition that a support must be part of a valid winning strategy. For a constraint to be WQGAC, for each variable and value there must exist a set of tuples containing the value, with one tuple for every combination of inner universal values. This is stronger than GAC, but weaker than SQGAC because the tuples do not necessarily constitute a subset of scenarios of a winning strategy. This definition is new to the best of my knowledge.

DEFINITION 3.2.14. Weak Quantified GAC (WQGAC)

A constraint C_k is weak quantified GAC (WQGAC) iff for each variable $x_{k_i} \in \mathcal{X}_k$ and value $a \in D_{k_i}$, with inner universal variables $U_{k_i} = \{x_{k_j} | j > i \wedge Q_{k_j} = \forall\}$, each function p such that $\forall x_{k_j} \in U_{k_i} : p(x_{k_j}) \mapsto b \in D_{k_j}$ has a matching tuple $t \in C_k^S$ s.t. $t_i = a$ and $\forall x_{k_j} \in U_{k_i} : t_j = p(x_{k_j})$

The function p maps inner universal variables to values in their domain. For $x_{k_i} \mapsto a$ to be supported, there must be a supporting tuple for every p, therefore every combination of values of inner universal variables. This can be a much stronger consistency than GAC, and in experiments it appears to be almost as strong as SQGAC. WQGAC and SQGAC are not equivalent: they are separated by an example in section 4.2, on page 120.

When there are no universal variables in the scope of a constraint C_k, SQGAC, WQGAC and GAC are equivalent for that constraint. This is because $C_k^S = \text{sce}(\text{WIN}^{\mathcal{P}_k})$: there is a bijection between satisfying tuples and winning strategies for the constraint.

In the case of binary constraints, Mamoulis and Stergiou [**74**] defined binary arc-consistency (referred to as QAC). This is reviewed in chapter 2 section 2.3.1.2, and the definition is given there. I argue here that both SQGAC and WQGAC applied to binary constraints are equivalent to QAC.

There are four quantifier sequences for a binary constraint C_k: $\forall x_{k_1}, x_{k_2}$; $\forall x_{k_1}, \exists x_{k_2}$; $\exists x_{k_1}, \forall x_{k_2}$ and $\exists x_{k_1}, x_{k_2}$. For each one, I argue that QAC is equivalent to both SQGAC and WQGAC.

$\forall x_{k_1}, x_{k_2}$: C_k is QAC iff $C_k^S = D_{k_1} \times D_{k_2}$. For C_k to be SQGAC, the degenerate strategy S when all variables are universal, has $\text{sce}(S) = D_{k_1} \times D_{k_2}$. For S to be a winning strategy, $\text{sce}(S) \subseteq C_k^S$, therefore $C_k^S = D_{k_1} \times D_{k_2}$. For C_k to be WQGAC, each value a_1 of x_{k_1} must have a tuple $\langle a_1, a_2 \rangle \in C_k^S$ for all values a_2 of x_{k_2}, so again $C_k^S = D_{k_1} \times D_{k_2}$.

$\forall x_{k_1}, \exists x_{k_2}$: C_k is QAC iff (1) all values $a_1 \in D_{k_1}$ have support for some value $a_2 \in D_{k_2}$ ($\forall a_1 \in D_{k_1} \exists a_2 : \langle a_1, a_2 \rangle \in C_k^S$), and (2) all values $a_2 \in D_{k_2}$ have support for some value $a_1 \in D_{k_1}$. For SQGAC, it is possible to construct a winning strategy S iff (1) is true, and S can be adapted to include any value in D_{k_2} iff (2) is true. Therefore C_k is SQGAC iff it is QAC. Identically, the constraint is WQGAC (and GAC) iff each assignment $x_{k_i} \mapsto a_i$ where $a_i \in D_{k_i}$ is contained in some satisfying tuple $t \in C_k^S$.

$\exists x_{k_1}, \forall x_{k_2}$: The constraint is QAC and WQGAC iff each value $a_1 \in D_{k_1}$ has a supporting tuple in C_k^S for all values $a_2 \in D_{k_2}$. For SQGAC, each winning strategy S with value $a_1 \in D_{k_1}$, has scenarios $\text{sce}(S) = \{\langle a_1, a_2 \rangle | a_2 \in D_{k_2}\}$ which must be in C_k^S.

The constraint is SQGAC iff there exists a winning strategy containing each value $a_1 \in D_{k_1}$, and constructing these winning strategies requires the same set of tuples sce(WIN^{C_k}) that QAC and WQGAC require.

$\exists x_{k_1}, x_{k_2}$: This is the case for standard binary CSP. QAC, GAC, SQGAC and WQGAC are equivalent on a constraint of this form, because for each assignment $x_{k_i} \mapsto a_i$ where $a_i \in D_{k_i}$, all four definitions require that a tuple $t \in C_k^S$ contains $x_{k_i} \mapsto a_i$ for the constraint to be consistent, $\exists t \in C_k^S : t_i = a_i$.

In these definitions, the constraint C_k is said to be consistent or inconsistent. If C_k is inconsistent, consistency may be *established* on C_k. This is a transformation on $\mathcal{P}$, forming a new QCSP instance $\mathcal{P}'$ where $\forall i : D'_i \subseteq D_i$, and WIN$^{\mathcal{P}}$ = WIN$^{\mathcal{P}'}$: the simplification does not remove any values which take part in a winning strategy in WIN$^{\mathcal{P}}$. When WIN$^{\mathcal{P}} = \emptyset$, the transformation may empty some domain, or may remove a value from the domain of a universal variable. In these cases, $\mathcal{P}'$ is further simplified to false.

The second of these two requirements (that winning strategies are preserved) is intractable to compute. Therefore the transformation is defined only on the constraint, with the proviso that the exact set of values (and no more) are removed from each domain such that C_k is consistent in $\mathcal{P}'$. More precisely, $\mathcal{P}'$ is unique and is such that $\sum_{i=1}^{n} |D'_i|$ is maximised and C_k is consistent.

If it is not always possible to establish consistency while maintaining the property WIN$^{\mathcal{P}}$ = WIN$^{\mathcal{P}'}$, then the definition of consistency must be incorrect: values which take part in a winning strategy are removed.

During the process of solving a QCSP, consistency is established on C_k using a propagation algorithm. This algorithm establishes consistency by removing (*pruning*) a set of values, referred to as *inconsistent values*. If the algorithm is *complete*, then it always removes enough values to make C_k consistent, and if it is *sound* then it removes no more values than required to make C_k consistent. Therefore if it is sound and complete, it performs exactly the transformation of establishing consistency.

3.2.6. The pure value rule. The previous work on the pure value rule is reviewed in chapter 2 section 2.3.1.3, and the related concepts of d-fixability and s-fixability in section 2.3.1.1.

In CSP, the similar concept of a *fixable* value was introduced by Bordeaux et al. [**20**]. If assignment $x_i \mapsto a$ is fixable, for all solutions to the CSP there is another solution which is identical except that x_i takes value a. This is a global condition, but Bordeaux et al. observe that it is sound to compute the condition locally (for each constraint x_i is in), giving a sufficient (but not necessary) condition for fixability of a value, which can be computed in polynomial time.

The same authors defined fixability for QCSP in two different ways: d-fixability and s-fixability [**21**] (standing for deep and shallow fixability, where shallow is the more widely applicable property). d- and s-fixability are defined as follows.

DEFINITION 3.2.15. d-fixability of $x_i \mapsto a$ in the QCSP instance $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$

$$\text{d-fixable}(x_i, a, \mathcal{P}) \equiv \forall t \in \text{sce}(\text{WIN}^{\mathcal{P}})\ :\ \exists t' \in \text{sce}(\text{WIN}^{\mathcal{P}}) \wedge t'_i = a \wedge \forall j \neq i, t'_j = t_j$$

In words, for all scenarios t of winning strategies of $\mathcal{P}$, there exists another scenario t' in the same set which is identical except that $t'_i = a$. Therefore, for any winning strategy, $x_i \mapsto a$ can take the place of any other value for x_i. If x_i is universal, a can be subsumed with any other value. If x_i is existential, all other values can be subsumed by a.

DEFINITION 3.2.16. s-fixability of $x_i \mapsto a$ in the QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$

$$\text{s-fixable}(x_i, a, \mathcal{P}) \equiv \forall t \in \text{sce}(\text{WIN}^{\mathcal{P}})\ :\ \exists t' \in \text{sce}(\text{WIN}^{\mathcal{P}}) \wedge t'_i = a \wedge \forall j < i, t'_j = t_j$$

This definition is identical to the one above except that only the first part (t_j where $j < i$) of the scenarios t and t' must match. This is a more widely applicable property (d-fixable$(x_i, a, \mathcal{P}) \Rightarrow$ s-fixable$(x_i, a, \mathcal{P})$).

For both these definitions, Bordeaux et al. have proven that if the condition holds locally for all constraints containing x_i, then it holds globally. To do this, they defined a local problem containing only one constraint C_k: $\mathcal{P}_k = \langle \mathcal{X}, \mathcal{D}, \{C_k\}, \mathcal{Q} \rangle$ but with the same variables, domains and quantifier sequence as the global problem $\mathcal{P}$. Bordeaux et al. do not prove that if some value is fixable, it can be removed or instantiated, therefore I will argue it here for d-fixability. I will argue it separately for universal and existential variables. In the context of QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ and for variable x_i, value a is d-fixable. I also assume that there exists some other value $b \neq a$ where $a, b \in D_i$.

- x_i is universal. The set of scenarios of winning strategies, containing $x_i \mapsto b$ is called B: $B = \{t \mid t \in \text{sce}(\text{WIN}^{\mathcal{P}}) \wedge t_i = b\}$. The same set for $x_i \mapsto a$ is called A: $A = \{t \mid t \in \text{sce}(\text{WIN}^{\mathcal{P}}) \wedge t_i = a\}$. By the definition of d-fixability, for each element of B there is an element of A which is equal except at index i. In the definition of QCSP semantics (definition 3.2.3), when all variables $x_{j<i}$ are instantiated (when firstx($\mathcal{P}$) $= x_i$), both simplified problems $\mathcal{A} = \mathcal{P}[x_i \mapsto a]$ and $\mathcal{B} = \mathcal{P}[x_i \mapsto b]$ (and those for other values in D_i) must be satisfiable for $\mathcal{P}$ to be satisfiable. From the sets A and B, it can be seen that if $\mathcal{B}$ is satisfiable, then $\mathcal{A}$ is also. Therefore the conjunction $\mathcal{A} \wedge \mathcal{B}$ can be simplified to $\mathcal{B}$, and to remove value a from D_i in $\mathcal{P}$ is a satisfiability-preserving simplification.
- x_i is existential. The argument proceeds the same way, except that the semantics of QCSP require (when firstx($\mathcal{P}$) $= x_i$) $\mathcal{P}[D_i = \{a\}] \vee \mathcal{P}[D_i = \{b\}] \vee \ldots$ Therefore it is sound to prune $x_i \mapsto b$ since if $\mathcal{P}[D_i = \{b\}]$ is satisfiable then so is $\mathcal{P}[D_i = \{a\}]$. Note that this means *all* values $b \neq a$ can be pruned.

I will call domain removals *pruning*, whether the reason for the removal is to establish consistency, or to exploit d-fixability. With the proof above and the fact that these conditions can be computed locally, we have a scheme that is similar to the pure literal rule in QBF, for pruning both universal and existential variables.

Next I define purity in the context of non-binary QCSP. Purity is not a new concept, having been invented for binary QCSP [**54**]. The concept extends trivially to QCSP. Purity is sufficient but not necessary for d-fixability, which I will prove below. Purity is defined over an arbitrary QCSP instance $\mathcal{P}$, but the definition is used later on instances $\mathcal{P}_k$ containing only the constraint C_k. The property is quite useless applied to the entire instance $\mathcal{P}$, but when it is applied to $\mathcal{P}_k$ it can be used to prove that values are d-fixable in $\mathcal{P}$.

DEFINITION 3.2.17. Purity of $x_i \rightarrow a$ in the QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$

$$\text{pure}(x_i, a, \mathcal{P}) \equiv (D_1 \times \cdots \times D_{i-1} \times \{a\} \times D_{i+1} \times \cdots \times D_n) \subseteq \text{sol}^{\mathcal{P}}$$

In words, if all possible solutions including value $x_i \mapsto a$ are indeed solutions to $\mathcal{P}$, then $x_i \mapsto a$ is a pure value. The definition of purity is equivalent to the *valid values* property of Bacchus and Walsh [**7**]. It is simply restated here in a more precise way, and following the terminology of Gent et al. [**54**] rather than Bacchus and Walsh.

I will now prove that purity is a sufficient condition for d-fixability.

THEOREM 3.2.18. *pure*$(x_i, a, \mathcal{P}) \Rightarrow$ *d-fixable*$(x_i, a, \mathcal{P})$

PROOF. Where x_i is existential, for any winning strategy $S \in \text{WIN}^{\mathcal{P}}$, another strategy S' can be constructed by substituting $x_i \mapsto a$ into S (in the notation of the family of functions, this means $\forall j \neq i : s'_j = s_j$ and $\forall t : s'_i(t) = a$). This new strategy is a winning strategy because (by the definition of purity) its scenarios are all in $\text{sol}^{\mathcal{P}}$. Therefore for any tuple $t \in \text{sce}(S)$, there is a second tuple $t' \in \text{sce}(S')$ such that $t'_i = a \wedge \forall j \neq i : t'_j = t_j$. Since this is true for all winning strategies, this meets the definition of d-fixability.

Where x_i is universal, for any winning strategy $S \in \text{WIN}^{\mathcal{P}}$, other winning strategies S' can be constructed by observing that, when $x_i \mapsto a$, the values of inner existential variables $x_{j>i}$ are only restricted by their domains D_j, because all valid tuples containing $x_i \mapsto a$ are in $\text{sol}^{\mathcal{P}}$. Therefore for any tuple $t \in \text{sce}(S)$, there exists some winning strategy S' where $t' \in \text{sce}(S')$ and $t'_i = a \wedge \forall j \neq i : t'_j = t_j$. □

Since local d-fixability for each constraint implies global d-fixability [**21**], if $x_i \mapsto a$ is pure for all constraints $C_k \in \mathcal{C}$ where $x_i \in \mathcal{X}_k$, then $x_i \mapsto a$ is d-fixable in $\mathcal{P}$. Therefore, local purity can be used for pruning universal and existential variables.

The potential of the pure value rule was identified by Gent, Nightingale and Rowley when working on encodings of binary QCSP to QBF [**52**] (section 3.8). We observed that the pure literal rule was very effective on the encoded problems.

Notice that the pure value rule is a form of dual, somewhat similar to that proposed by Bordeaux and Monfroy [**22**], but without the difficulty (discussed in chapter 2 section 2.3.1.4) that all but one of the constraints of the primal problem must be trivially true before any useful computation occurs in the dual problem.

From the definitions, the pure value rule is considerably weaker than d- or s-fixability, so why would it be used? The advantage of the pure value rule is that local purity can be detected cheaply by re-using propagation algorithms, whereas (to the best of my knowledge) d- and s-fixability would require new algorithms to be developed. Also, the pure value rule seems to be sufficient for solving real problems, such as Connect 4 and contingent job-shop scheduling, which I model and solve in the following chapters. The algorithms to enforce the pure value rule are given in section 3.5.

3.3. Search algorithms

Both the algorithms presented in this section have some relation to the corresponding algorithms for ordinary CSP, which are reviewed in chapter 2 section 2.1.1. The search algorithm presented below is very similar conceptually to the QCSP algorithms presented by Gent, Nightingale and Stergiou [**54**] (reviewed in chapter 2 section 2.3.4.1) even though the presentation is very different, so I do not claim novelty for this algorithm.

The terms *branching* and *backtracking* are used in the same sense as in the literature review. More formally, to branch on value a of variable x_i, the problem is simplified by reducing the domain of x_i: $D_i \leftarrow \{a\}$, as in definition 3.2.3, then attempting to solve the simplified problem. Normally the first step in solving the simplified problem is to simplify it further using local reasoning algorithms. This is combined with *backtracking*. For example, after branching for value a and performing $D_i \leftarrow \{a\}$, it is usually necessary to restore the state of the problem to before $D_i \leftarrow \{a\}$ was performed, so that it is possible to branch for other values.

Also, the notation $\overline{x_i}$ for the greatest value in D_i, and $\underline{x_i}$ for the least value in D_i is used. This assumes a total ordering on the elements of D_i. $D_i \subset \mathbb{Z}$ so I use the $<$ ordering.

3.3.1. Search without optimization.

The *search* procedure (algorithm 3) is recursive, closely following the definition of QCSP semantics (definition 3.2.3), and also following search for QBF (chapter 2 section 2.2.3). It is called with $i = 1$ and returns whether the QCSP is satisfiable. It has the following basic structure:

(1) The consequences of domain removals are propagated (with procedure *propagate*). If this procedure infers there can be no winning strategy, then false is returned.

(2) If all variables have been instantiated ($i > n$ where n is the number of variables) then we have reached a satisfying scenario.

(3) There is no need to search over variables with only one value remaining, so jump to the first variable with more than one value in its domain.

(4) The procedure recurses for each value of the current variable x_i:

  (a) If x_i is universal, and one of the recursive calls returns false, then we return false, otherwise true. This matches the universal part of the definition of QCSP semantics.

  (b) If x_i is existential, and one of the recursive calls returns true, then we can return true, otherwise false. This matches the existential part of the definition.

The domains and any other important state are managed by *addBacktrackLevel* and *backtrack*. The procedure *addBacktrackLevel* pushes a record on a stack for the purpose of backtracking. The procedure *backtrack* pops a record from the top of the stack and returns the domains D_i to their state when the record was made. Also, some constraints have internal state which must be restored at this point.

Definition 3.2.3 would suggest an algorithm that recursively constructs a complete scenario t and then tests it against the constraints. This is modified for efficiency by simplifying the problem (by calling *propagate*) at each step of the recursion.

Contract with *propagate*: *Propagate* returns false iff the problem simplifies to false, otherwise it returns true whether or not the problem was simplified. When all variables are instantiated (all domains are of size one), *propagate* is able to decide the problem (i.e. *propagate* returns true iff the problem is satisfiable). To simplify the problem, *propagate* removes values from the domains. I assume here that these removals are sound.

Whenever *propagate* returns false, the procedure returns false and backtracks. At the leaves of the recursion, the search procedure matches the definition closely because *propagate* can decide

Algorithm 3 Search for finite QCSP

```
(1)  procedure search($i$: variable index): Boolean
(2)  if $\neg$propagate():
(3)      return false
(4)  if $i > n$: {base case}
(5)      return true
(6)  while $D_i = \{a\}$: {One value left in domain}
(7)      $i \leftarrow i + 1$
(8)      if $i > n$ then: return true
(9)  {recursive cases}
(10) if $Q_i = \forall$: {variable $x_i$ is universal}
(11)     while true:
(12)         $a \leftarrow$pickValue($x_i$) {choose a value heuristically}
(13)         addBacktrackLevel()
(14)         $D_i \leftarrow \{a\}$
(15)         $b \leftarrow$search($i + 1$)
(16)         backtrack()
(17)         if $\neg b$ then: return false
(18)         else:
(19)             $D_i \leftarrow D_i \setminus \{a\}$
(20)             if $D_i = \emptyset$ then: return true
(21)             if $\neg$propagate() then: return false
(22) else: {variable $x_i$ is existential}
(23)     while true:
(24)         $a \leftarrow$pickValue($x_i$) {choose a value heuristically}
(25)         addBacktrackLevel()
(26)         $D_i \leftarrow \{a\}$
(27)         $b \leftarrow$search($i + 1$)
(28)         backtrack()
(29)         if $b$ then: return true
(30)         else:
(31)             $D_i \leftarrow D_i \setminus \{a\}$
(32)             if $D_i = \emptyset$ then: return false
(33)             if $\neg$propagate() then: return false
```

the fully instantiated problem. Therefore, assuming that *propagate* performs only sound simplifications, *search* is correct.

Space and time complexity. The *search* algorithm implicitly stores the domain of each variable, and for backtracking purposes it is possible to use a stack of domain removals, which are restored on backtracking. The maximum height of the stack is nd, and each entry takes $\log n + \log d$

space. Current domains can be represented as an array of Booleans, with size nd. Overall, this gives $O(nd)$ (ignoring logs) plus the space complexity of the constraint propagation algorithms.

The algorithm can explore n^d leaf nodes in the worst case. It is not clear what the complexity of performing propagation is. The algorithm will perform propagation $O(n^d)$ times.

3.3.2. Search with optimization. The optimization algorithm is based on branch and bound [**6**]. Briefly, solving a CSP with branch and bound involves finding a solution, scoring it, then modifying the CSP such that any further solutions are better, then continuing search. This is repeated until the CSP is no longer satisfiable. The last solution found is in the set of highest scoring solutions.

To the best of my knowledge, this is the first optimization algorithm for QCSP. It is based on the intuition that a strategy represents a branching plan, and the appropriate branches will be selected at execution time. Only one scenario of the strategy will be used when the plan is executed, so an optimal strategy is one made up of optimal scenarios.

The optimization algorithm maximizes the value of an existential variable x_{opt}. Ordinarily, x_{opt} would be quantified in the innermost block, and linked to some other variables with constraints representing the optimization function. I assume that $x_{opt} \in \mathbb{Z}$, but it would be trivial to generalize this.

Procedure *branchBound* (algorithm 4) is very similar to algorithm 3. The main difference is that when recursing for values of an existential variable, all values are tried, and the one with the highest score is taken. Similarly to traditional branch and bound, the domain of x_{opt} is modified based on the highest score seen so far at this level of the recursion. The process that happens globally in traditional branch and bound happens locally here, when branching on an existential variable.

When branching on a universal variable, the procedure is very similar to search, however in the case where all values return a score (rather than false), the score for each value is stored and the minimum is returned.

Algorithm 4 Branch and bound for finite QCSP

```
(1)  procedure branchBound($i$: variable index, $x_{opt}$: variable): $\mathbb{R} \cup \{\text{false}\}$
(2)  if ¬propagate() then: return false
(3)  if $i > n$: {base case}
(4)      return $\overline{x_{opt}}$ {return the upper bound of $x_{opt}$}
(5)  while $D_i = \{a\}$: {One value left in domain}
(6)      $i \leftarrow i + 1$
(7)      if $i > n$ then: return $\overline{x_{opt}}$
(8)  {recursive cases}
(9)  if $Q_i = \forall$: {variable $x_i$ is universal}
(10)     while true:
(11)         $a \leftarrow$pickValue($x_i$) {choose a value heuristically}
(12)         addBacktrackLevel()
(13)         $D_i \leftarrow \{a\}$
(14)         $b_a \leftarrow$branchBound($i + 1$, $x_{opt}$) {record score for value $a$}
(15)         backtrack()
(16)         if $b_a =$ false then: return false
(17)         else:
(18)             $D_i \leftarrow D_i \setminus \{a\}$
(19)             if $D_i = \emptyset$ then: return min($b_*$)
(20)             if ¬propagate() then: return false
(21) else: {variable $x_i$ is existential}
(22)     $m \leftarrow$ false {$m \in \mathbb{R} \cup$ false}
(23)     while true:
(24)         $a \leftarrow$pickValue($x_i$) {choose a value heuristically}
(25)         addBacktrackLevel()
(26)         $D_i \leftarrow \{a\}$
(27)         $b \leftarrow$branchBound($i + 1$, $x_{opt}$)
(28)         backtrack()
(29)         if $b \neq$ false and [$m =$ false or $b > m$]:
(30)             $D_{opt} \leftarrow D_{opt} \setminus \{c|c \leq \lfloor b \rfloor\}$ {raise the lower bound of $x_{opt}$ to $\lceil b \rceil$}
(31)             $m \leftarrow b$
(32)         $D_i \leftarrow D_i \setminus \{a\}$
(33)         if $D_i = \emptyset$ then: return $m$
(34)         if ¬propagate() then: return $m$
```

The only other difference between *branchBound* and *search* is the base case, where *branchBound* returns the maximum value of x_{opt}. If $i > n$ then all variables are instantiated so the maximum value is also the minimum.

3.3.2.1. *Optimizing for the average case.* In many cases, optimizing for the worst-case is too pessimistic. It assumes that the situation is adversarial. In other words, that QCSP is like a game (between an existential player and universal player [**46**]) where your opponent (the universal player) plays to reduce your utility as much as possible. In planning with uncertainty, the source of uncertainty may be passive (for example, weather conditions), so it would be more appropriate to optimize a weighted average over the values of a universal variable. The values are weighted according to their probabilities. The weighted mean average of the score x_{opt} is assumed to be meaningful. (In Stevens's taxonomy of scales of measurement [**95**], the score should be a ratio scale, meaning that for example 100 is twice as good as 50. This is required for the weighted average to be meaningful.) It is assumed that probability distributions for each universal variable are independent.

As a planning example, consider the problem of delivering some packages by road and dirt track. We can choose one of two vehicles, a car and a 4×4 truck. We must be able to tolerate both types of weather $w \in \{\text{sun, rain}\}$. The route R accomplishes goals C, with score x_{opt}. This is expressed in the QCSP below. There is a weather forecast predicting a 0.3 probability of sun and 0.7 probability of rain. When it is raining, the car is much slower than the 4×4 on the tracks. The car performs slightly better on roads. The scores for the two vehicles in both situations are in the table below.

$$\exists v \in \{\text{car}, 4 \times 4\}, \forall w \in \{\text{sun}, \text{rain}\} \exists R, x_{opt} \; : \; C$$

Scores	car	4×4
sun	10	8
rain	5	6

Unmodified *branchBound* would select the 4×4 because its minimum score is better, in line with simple maximin. Optimizing the unweighted mean score would select the car with a mean of 7.5. Optimizing the mean weighted with the probabilities from the weather forecast would select the 4×4 with a weighted mean of 3.3, over the car with a weighted mean of 3.25.

Modifying *branchBound* to use this scheme requires two changes: on line 19 the function min(b_*) is replaced with weighted-mean(i, b_*). The weighted-mean function uses static weights associated with variable x_i. The second change is to remove line 30. When using min, it is sound to restrict the search to areas with a better score than the one already found. However, with the weighted mean, a poor score for one value of a universal can be counteracted by a good score for another value, so it is no longer sound to restrict the value of x_{opt} in this way.

Tarim, Manandhar and Walsh [**75**,**96**] proposed an optimization algorithm for stochastic CSP, which is able to minimize (maximize) the expected value of some variable. However, the algorithm expands out the stochastic CSP into a CSP, thereby potentially using a large amount of space.

3.3.2.2. *Time behaviour.* The algorithm *branchBound* (for both minimum and average case optimization) takes $O(d^n)$ time, which is the same worst case time as the search algorithm. However, it will explore more nodes in the average case.

If the outermost block of variables are existential, then after finding the first winning strategy *branchBound* has an anytime behaviour: it can be interrupted and will provide a winning strategy that is locally optimized over existential variables other than those in the outermost block. However, to find this first solution is more expensive than using the *search* algorithm, because it optimizes for each existential variable not in the outermost block. The difference can be seen on line 29 of *search*, where the *search* algorithm returns as soon as it finds a value of an existential variable which extends to a solution. In contrast *branchBound* does not return, but continues to search for the optimal value of the variable.

3.3.3. An example of search. To illustrate how *search* and *branchBound* work on a simple example, consider the trivial game of noughts and crosses (tic-tac-toe), played on a 2×2 board with the aim of making a line of 2. The first player can clearly always win. The board spaces are numbered as shown below. The structure of the QCSP is also shown below. The variables $x_1 \dots x_3$ have domain sizes decreasing from 4 to 2, representing the free spaces on the board. The fourth move is omitted because there is only one space left on the board. x_{opt} is included if we are using *branchBound*. In figure 13, these values are represented using the number of the space on the board. A single constraint C_1 is used for both examples. For search, C_1 has scope $\{x_1, x_2, x_3\}$ and is satisfied iff player 1 wins the game. For *branchBound*, the scope of C_1 is $\{x_1, x_2, x_3, x_{opt}\}$ and C_1 is satisfied iff player 1 wins the game *and* x_{opt} equals the score according to the type of line player 1 has constructed: x_{opt} is 3 for a vertical line, 2 for a diagonal and 1 for a horizontal.

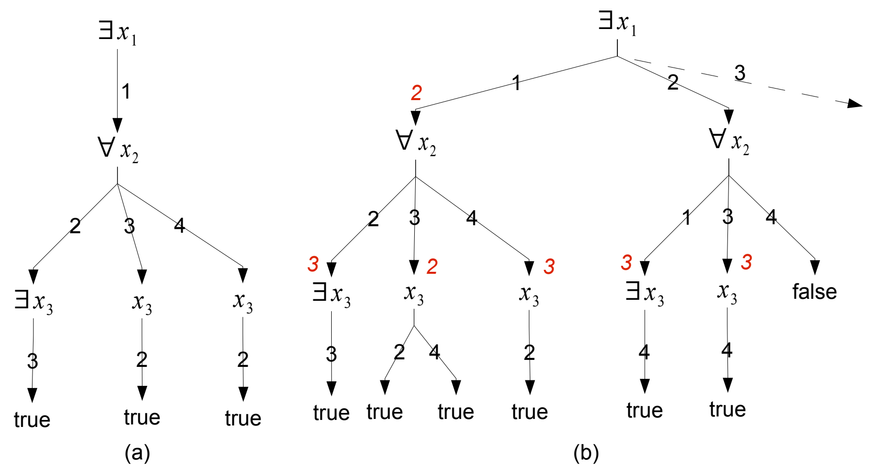


FIGURE 13. Search performed by (a) *search* and (b) *branchBound*

In both cases, SQGAC was applied to the constraint.

1	2
3	4

$$\exists x_1 \forall x_2 \exists x_3 (\exists x_{opt}) \ : \ C_1$$

For both methods, the values of variables are tried in number order. The second player can always prevent the first player from getting a vertical line (which has the highest score), but not also prevent the first player from getting a diagonal. Hence the highest score is 2, associated with value 1 of variable x_1. After exploring $x_1 \mapsto 1$, $\underline{x_{opt}} = 3$ hence the subtree beneath $x_1 \mapsto 2$ is considerably smaller because of pruning.

3.3.4. Recording a winning strategy. In many cases (such as planning with uncertainty) it is essential to record a winning strategy. Both algorithms presented above lend themselves to recording a winning strategy in a form similar to the solution tree (definition 3.2.10).

Consider the following rooted tree $T = \langle V, E, r, L \rangle$: the set of vertices V is the set of calls to *search* or *branchBound* (including recursive calls). The root node r is the initial call. Two vertices a and b have an edge between them if one instance of the procedure called the other. The vertices are labelled with a function $L : V \setminus \{r\} \rightarrow \{x_i \mapsto a | a \in D_i\}$ such that the child b is labelled with the assignment made just before b was called. For the *search* algorithm, when making a recursive call the following is executed: $D_i \leftarrow \{a\}$; $b \leftarrow$search($i + 1$). The child created by this call would be labelled $x_i \mapsto a$. The recursive call is almost identical for *branchBound*. We will call this tree T the search tree.

The record of a winning strategy is a subtree $T' = \langle V' \subseteq V, E' \subseteq E, r, L' \subseteq L\rangle$ of the search tree T. For the *search* algorithm, for each vertex in V, it is also in V' iff the recursive call that it represents returns true, and it is connected to r in T'. Hence as the *search* algorithm runs, it can create new vertices and edges as if building T, and also cut off any vertex v and all its descendents when a recursive call returns false. (In the Queso implementation, both these operations take approximately constant time.)

For *branchBound*, we only store the optimal strategy. Similarly to *search*, *branchBound* creates new vertices and edges as if building T, and cuts off any vertex (and its descendents) which does not satisfy the following rule. A vertex $v \in V$ is also in V' iff:

- the call that it represents does not return false, and
- where $L(v) = x_i \mapsto a$, if x_i is existential, and v returned a higher score than its siblings (or if equal, a is less than the label of the sibling), and
- v is connected to r in T'.

Finally, both algorithms use the pure value rule thus they do not necessarily branch for each value in the initial domain of a universal variable. Without this feature, T' would be a solution tree, but with it some vertices are omitted. It is straightforward to construct a third tree T'' which is a solution tree, but often T' is sufficient because we do not care about the missing areas of the tree. For example if the QCSP models a game, the pure values may be cheating moves. T' can also be much smaller than T''.

3.4. Managing domains and the constraint queue

The search algorithms above both call procedure *propagate*, which enforces local consistency. Section 3.2.4 defined the notion of local consistency. For the problem $\mathcal{P}$ to be locally consistent, all constraints $C_k \in \mathcal{C}$ must be consistent. The system Queso uses to achieve this is similar to most recent constraint solvers. Events (such as removal of a value from a domain) cause constraints to be 'woken up' because they may no longer be consistent. These constraints are added to a queue for propagation[1]. When the queue is empty, all constraints are consistent.

3.4.1. Constraint queue. I accommodate both the major types of constraint propagation algorithm: *fine-grained* and *coarse-grained* as discussed in the literature review. For a fine-grained algorithm, the declaration looks like the following.

procedure propagateConstraint(C_k: constraint, x_{k_i}: variable, a: value): Boolean

The propagator must be called for each value a removed from the domain D_{k_i} of some variable $x_{k_i} \in \mathcal{X}_k$.

For *propagateConstraint* to interact correctly with the queue, it must fulfil the following contract. Suppose C_k is consistent with domains $\mathcal{D}_1$, then values $V = \{x_{k_i} \mapsto a, \ldots\}$ are removed from the domains (possibly by other constraint propagators) to form $\mathcal{D}_2$. The algorithm *propagate-Constraint* is called for all values in V in any order. If the constraint cannot be made consistent, one of those calls will return false, otherwise the unique minimal set of values will be removed from the domains to construct $\mathcal{D}_3$, and C_k will be consistent with $\mathcal{D}_3$.

The queue for fine-grained constraints consists of two stacks (last-in-first-out) of records of the form $\langle C_k, x_i, a\rangle$. The two stacks HPropagateStack and LPropagateStack are for high and low priority constraints respectively: the high priority stack is emptied first. The exact behaviour is given in algorithm 5.

The interface for coarse-grained constraint propagators is simpler.

procedure propagateConstraint(C_k: constraint): Boolean

The queue for coarse-grained constraints is a single stack CoarsePropagateStack of constraints C_k, containing no duplicates. A constraint C_k is only added to the stack if it is not already there. An array of Boolean variables (named present) is used to keep track of which constraints are on the stack s.t. present$(C_k) \in \{\text{false}, \text{true}\}$.

[1]The phrase 'constraint queue' is used in the literature and I use it here without implying that it is a first-in-first-out queue.

Algorithm 5 processQueue procedure to establish consistency for all constraints

```
**procedure** processQueue(): Boolean
{HPropagateStack, LPropagateStack, CoarsePropagateStack, present are available as global variables}
$C \leftarrow$Pop()
**while** $C \neq$*nil*:
    {$C$ may be of the form $\langle C_k \rangle$ or $\langle C_k, x_i, a \rangle$. The appropriate *propagateConstraint* procedure is called in each case.}
    $b \leftarrow$propagateConstraint($C$)
    **if** $\neg b$: **return** false
    $C \leftarrow$Pop()
**return** true

**procedure** Pop(): $\langle \ldots \rangle$
{The Stack.pop procedure removes and returns the top item on Stack.}
$p \leftarrow$CoarsePropagateStack.pop()
**if** $p \neq$*nil*:
    present($p$) $\leftarrow$false
    **return** $p$
$p \leftarrow$HPropagateStack.pop()
**if** $p \neq$*nil*: **return** $p$
$p \leftarrow$ LPropagateStack.pop()
**if** $p \neq$*nil*: **return** $p$
**return** *nil*
```

The set of coarse-grained propagators can be further divided into one-pass and multi-pass algorithms. A one-pass propagator makes the constraint consistent (or returns false if this is not possible) when it is called. Therefore it is inefficient to have more than one reference to C_k on the queue.

Multi-pass propagators may need to be called several times before the constraint becomes consistent. If some constraint C_k with a multi-pass propagator is added to the queue, there are three cases:

(1) C_k is consistent. The propagator should be called once, and it does not change the domains.

(2) C_k is inconsistent, the propagator is called and it returns false.

(3) C_k is inconsistent, the propagator is called and it reduces the domains. Now C_k is not guaranteed to be consistent.

To behave correctly for the first case, the queue does not allow more than one reference to C_k, but C_k is re-queued if the propagator changes the domains. This takes care of case 3.

One-pass and multi-pass propagators are also known as idempotent and non-idempotent respectively. Most constraint programming systems support both ([**85**], chapter 14).

3.4.2. Wake up lists. Queso has a list of constraints wakeUp(x_i) for each variable. Constraints register themselves when they are initialized, and whenever x_i is pruned the constraints in wakeUp(x_i) are added to a global queue of constraints which may be inconsistent. All propagation algorithms use the *exclude* procedure (algorithm 6) to prune individual values, or *excludeUpper* and *excludeLower* to move the bounds. These procedures deal with queueing other constraints appropriately.

3.4.3. Exclude procedures. All three types of constraint propagator may perform some pruning, and then deduce that the constraint has no winning strategies, WIN$^{C_k} = \emptyset$ and return false. The propagator could deduce failure directly, or by pruning a variable. Pruning any value from a universal variable, or pruning the final value from an existential variable both imply failure. The *exclude* procedure (algorithm 6) for removing a value from some domain D_i returns a Boolean, which is false iff the variable is a universal or a is the last value.

Exclude can be called without a constraint C_k, using nil in its place. In this case, all constraints on the wakeUp lists, including the calling constraint, are queued. This is important for multi-pass constraints, which need to be re-queued whenever one of their variables is changed.

The *excludeUpper* procedure (algorithm 7) is very similar, with the only differences being the removal of all values above a from D_i. *ExcludeLower* is also provided. It is symmetric to *excludeUpper*.

Exclude and *excludeUpper* both run in time proportional to the size of wakeUp(x_i). It is not clear how this could be improved. In Minion [**50**], the equivalent procedures take constant time, but duplicates are allowed on CoarsePropagateStack and fine-grained constraints are implemented through the entirely separate watched literals mechanism.

Algorithm 6 exclude procedure to prune a value from a variable

```
procedure exclude($C_k$, $x_i$, $a$): Boolean
{Assumes all stacks and wake up lists are global variables}
if $Q_i = \forall$: return false
$D_i \leftarrow D_i \setminus \{a\}$
if $D_i = \emptyset$: return false
$W \leftarrow$wakeUp($x_i$)
$W \leftarrow W \setminus \{C_k\}$ {Do not re-queue $C_k$}
for $C_l \in W$:
    {queue according to type and priority}
    if $C_l$ is coarse-grained and present($C_l$)=false:
        CoarsePropagateStack.push($\langle C_l \rangle$)
        present($C_l$) $\leftarrow$true
    else:
        HPropagateStack.push($\langle C_l, x_i, a \rangle$) or LPropagateStack.push($\langle C_l, x_i, a \rangle$)
return true
```

Algorithm 7 excludeUpper procedure to reduce the upper bound of a variable

```
procedure excludeUpper($C_k$, $x_i$, $a$): Boolean
{Assumes all stacks and wake up lists are global variables}
if $Q_i = \forall$: return false
$D_i \leftarrow \{b | b \in D_i \wedge b \leq a\}$
if $D_i = \emptyset$: return false
$W \leftarrow$wakeUp($x_i$)
$W \leftarrow W \setminus C_k$ {Do not re-queue $C_k$}
for $C_l \in W$:
    {queue according to type and priority}
    if $C_l$ is coarse-grained and present($C_l$)=false:
        CoarsePropagateStack.push($\langle C_l \rangle$)
        present($C_l$) $\leftarrow$true
    else:
        HPropagateStack.push($\langle C_l, x_i, a \rangle$) or LPropagateStack.push($\langle C_l, x_i, a \rangle$)
return true
```

3.4.4. Fixed point behaviour. The various propagation algorithms should move the problem from an initial set of domains $\mathcal{D}_1$ to a set of smaller domains $\mathcal{D}_2$, regardless of the order in which they are called by processQueue, i.e. they should converge to a single fixed point. Assuming each propagation algorithm correctly implements its consistency, the definitions of consistency determine whether there is a single fixed point. Apt proves this condition for GAC [**6**]. I conjecture it is also true for SQGAC and WQGAC because they are similar in style to GAC: for all three definitions the constraint is consistent iff for each value $x_{k_i} \mapsto a$, there exists an object (a tuple, a set of tuples or a winning strategy) which somehow matches $x_{k_i} \mapsto a$ and is composed of other values s.t. if any of the other values are pruned, the object becomes invalid.

The single fixed point condition is not required for correctness, but if there are multiple fixed points, the number of nodes in the search tree could vary depending on which fixed point is reached at each node.

3.4.5. A small example of propagation. Suppose we have the following CSP.

$$\begin{aligned}
&\exists x_1, x_2, x_3 \in \{1,2,3\}\ : \\
&C_1 :\ \text{allDifferent}(x_1, x_2, x_3) \\
&C_2 :\ x_1 + x_2 + x_3 = 5 \\
&C_3 :\ x_1 < x_2
\end{aligned}$$

The consistency for all constraints is GAC. The allDifferent constraint is satisfied iff the variables take three different values. For simplicity, all propagators are one-pass and coarse-grained. The wake up lists are wakeUp$(x_1) = \langle C_1, C_2, C_3\rangle$, wakeUp$(x_2) = \langle C_1, C_2, C_3\rangle$, wakeUp$(x_3) = \langle C_1, C_2\rangle$. Initially, C_1 and C_2 are consistent, but C_3 is not. With initial queue CoarsePropagateStack$= \langle C_3\rangle$, propagation could proceed as follows. The exact order of propagation depends on the order that members of the wake up lists are added to the stack. The example shows that propagating the constraints to exhaustion with a queue can achieve much more than propagating each constraint once.

Propagator	Domains	CoarsePropagateStack
C_3	$D_1 = \{1,2\}, D_2 = \{2,3\}, D_3 = \{1,2,3\}$	$\langle C_1, C_2\rangle$
C_2	$D_1 = \{1,2\}, D_2 = \{2,3\}, D_3 = \{1,2\}$	$\langle C_1\rangle$
C_1	$D_1 = \{1,2\}, D_2 = \{3\}, D_3 = \{1,2\}$	$\langle C_2, C_3\rangle$
C_3	$D_1 = \{1,2\}, D_2 = \{3\}, D_3 = \{1,2\}$	$\langle C_2\rangle$
C_2	$D_1 = \{1\}, D_2 = \{3\}, D_3 = \{1\}$	$\langle C_1, C_3\rangle$
C_3	$D_1 = \{1\}, D_2 = \{3\}, D_3 = \{1\}$	$\langle C_1\rangle$
C_1	Failed	$\langle\rangle$

3.5. The pure value rule

In classical constraint programming, all variables can be pruned during search using propagation algorithms, which can make the search efficient. Values which cannot be extended to a solution are removed. Universal variables cannot be pruned in this way. In QCSP, if a value of a universal variable is inconsistent, then the QCSP is false and the search backtracks. However universal variables can be pruned using the pure value rule, defined in section 3.2.6. In this section I give two variants of a scheme for implementing the pure value rule. Both variants are experimentally evaluated in the following chapters.

The previous work on the pure value rule is in the context of binary QCSP [**54**]. The algorithm by Kostas Stergiou is given in the literature review (chapter 2 section 2.3.1.3). It iterates through C_k^S, which for non-binary constraints can be very large. Therefore I propose this new scheme, which uses propagation algorithms, therefore C_k^S need not be explicitly represented and searched. However, there must be a propagator for $\neg C$ in this scheme.

Consider constraint C_k with scope $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ and value $x_{k_i} \mapsto a$ where $a \in D_{k_i}$. A CSP $\mathcal{PV}_k$ is constructed with the variables $\mathcal{X}_k$ and their domains, and only one constraint, which is the negation of C_k.

$$\mathcal{PV}_k = \langle \mathcal{X}_k, \mathcal{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle, \{\neg C_k\} \rangle$$

The negated constraint $\neg C_k$ is constructed with $\neg C_k^S = (D_{k_1} \times \cdots \times D_{k_r}) \setminus C_k^S$. The negated constraint is solved iff C_k is failed. $x_{k_i} \mapsto a$ is pure w.r.t. C_k iff $x_{k_i} \mapsto a$ is inconsistent in $\mathcal{PV}_k$. This is because $x_{k_i} \mapsto a$ is consistent in $\neg C_k$ iff there exists a supporting tuple (by definition 3.2.12). If there is no supporting tuple in $\neg C_k$ then there must be a complete set of tuples containing $x_{k_i} \mapsto a$ in C_k (i.e. $D_{k_1} \times \cdots \times D_{k_{i-1}} \times \{a\} \times D_{k_{i+1}} \times \cdots \times D_{k_r} \subseteq C_k^S$), which is the definition of purity. A value $x_i \mapsto a$ must be pure for all constraints C_k where x_i is in the scope, $x_i \in \mathcal{X}_k$, for $x_i \mapsto a$ to be pure in $\mathcal{P}$.

3.5.1. Pure value rule for one variable. Figure 14 illustrates the proposed scheme to enforce the pure value rule on x_i only. The three boxed areas with a negated constraint in them are three

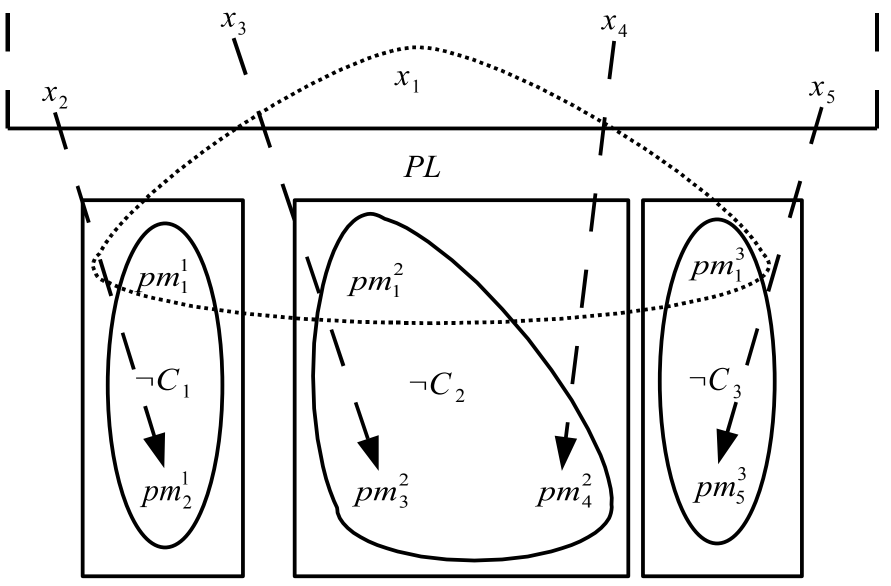


FIGURE 14. Implementation of pure value rule for one variable x_i

side CSPs $\mathcal{PV}_1$, $\mathcal{PV}_2$ and $\mathcal{PV}_3$. The area at the top represents the problem $\mathcal{P}$, with variables x_1, x_2, x_3, x_4, x_5. I have called the variables in the side problems pm. One side problem $\mathcal{PV}_k$ is constructed for each constraint C_k in $\mathcal{P}$, and it contains the single negated constraint $\neg C_k$. If C_k has scope $\mathcal{X}_k = \langle x_{k_1}, \dots, x_{k_r} \rangle$ then $\neg C_k$ has scope $\langle pm^k_{k_1}, \dots, pm^k_{k_r} \rangle$. (The superscript matches the constraint, and the subscript is the same as the subscript of the x_{k_i} variable in $\mathcal{X}_k$.)

The dashed arrows indicate that values that are pruned in $\mathcal{P}$ are also pruned from the pm variables, but not vice versa. The dotted area marked PL (for pure link) represents the algorithm for removing values of x_i when they are subsumed. The algorithm resembles a fine-grained constraint propagation algorithm, and it is called by the constraint queue as needed. It is given in algorithm 8. For value $x_i \mapsto a$, the main idea is to *watch* a pm_i variable which contains a. At least one such variable must exist if a is not pure. A watching algorithm maintains a reference to an object which has a specific property, only changing the reference when the watched object loses its property. In

Algorithm 8 Pure link algorithm

```
(1)  procedure pureLink($v$: variable, $a$: value): Boolean
(2)  {$x_i$ is the variable in $\mathcal{P}$, $pm_i^1, \ldots pm_i^m$ belong to side problems}
(3)  {$\forall a \in D_i \; : \; w_a$ is the superscript $1 \ldots m$ of a $pm_i$ variable which contains value $a$}
(4)  if $v = x_i$:
(5)      for $j \in \{1 \ldots m\}$:
(6)          exclude($nil, pm_i^j, a$)
(7)  else:
(8)      $v = pm_i^j$ {find the index $j$}
(9)      if $w_a = j$: {$pm_i^j$ no longer contains $a$, watch invalidated}
(10)         for $k$ in $j+1, \ldots, m, 1, \ldots, j-1$:
(11)             if $a \in D_{pm_i^k}$:
(12)                 $w_a \leftarrow k$
(13)                 return true
(14)         pure($PL_i, x_i, a$)
(15) return true
```

this case, the property is simply containing value a. The index of a pm_i variable is stored in w_a between calls to *pureLink*, and it is not backtracked when the search procedure backtracks.

If $pm_i^j \mapsto a$ is pruned and $w_a = j$, the algorithm searches (cyclically) for a new pm_i variable to watch. If none is found, $x_i \mapsto a$ is pure. The *pure* procedure that is called on line 14 deals with the quantification of x_i. If x_i is universal, and $\exists b \neq a \; : \; b \in D_i$, then a is subsumed by b and is removed. If x_i is existential, all other values $b \neq a$ are subsumed by a.

The *pureLink* algorithm calls *pure* when it discovers that some value is pure. The logic of *pure* is similar to *exclude* (algorithm 6). *Pure* for universal variables is given in algorithm 9. If there exists some value $b \neq a$, then a is subsumed by b. a is removed from D_i and constraints are queued appropriately. For an existential variable, the *pure* procedure (algorithm 10) removes all values other than a, and queues constraints appropriately.

Integration with the solver. To integrate this efficiently with the constraint queue and search procedure, I have blurred the distinction between the problem $\mathcal{P}$ and the CSP side problems $\mathcal{PV}$. The pm variables of the side problems are separate and are invisible to the search procedure but not to the queue. The constraints $\neg C_k$ in side problems are added to, and called from, the queue like any constraint in $\mathcal{P}$. A *pureLink* constraint is queued whenever any of its variables are changed, and is called like any fine-grained constraint.

Algorithm 9 pure procedure for universal variables

```
**procedure** pure($C_k$, $x_i$, $a$)
{Assumes all stacks and wake up lists are global variables}
**if** $\nexists b \neq a :\ b \in D_i$: **return**
$D_i \leftarrow D_i \setminus \{a\}$
$W \leftarrow$wakeUp($x_i$)
**if** $C_k$ is not multi-pass: $W \leftarrow W \setminus C_k$ {Do not re-queue $C_k$}
**for** $C_l \in W$:
    {queue according to type and priority}
    **if** $C_l$ is coarse-grained and $\neg$present($C_j$):
        CoarsePropagateStack.push($\langle C_l \rangle$)
        present($C_l$) $\leftarrow$true
    **else**:
        HPropagateStack.push($\langle C_l, x_i, a \rangle$) or LPropagateStack.push($\langle C_l, x_i, a \rangle$)
```

Algorithm 10 pure procedure for existential variables

```
**procedure** pure($C_k$, $x_i$, $a$)
{Assumes all stacks and wake up lists are global variables}
**if** $\nexists b \neq a :\ b \in D_i$ **then**: **return**
$W \leftarrow$wakeUp($x_i$)
**if** $C_k$ is not multi-pass: $W \leftarrow W \setminus C_k$ {Do not re-queue $C_k$}
**for** $C_l \in W$:
    {queue according to type and priority}
    **if** $C_l$ is coarse-grained and $\neg$present($C_l$):
        CoarsePropagateStack.push($\langle C_l \rangle$)
        present($C_l$) $\leftarrow$true
    **else**:
        **for all** $b \neq a :\ b \in D_i$:
            HPropagateStack.push($\langle C_l, x_i, b \rangle$) or LPropagateStack.push($\langle C_l, x_i, b \rangle$)
$D_i \leftarrow \{a\}$
```

A second fine-grained algorithm *pureCopy* is used to link x_i with all pm_i variables. This simply prunes values from all pm_i variables as they are pruned from x_i, but not vice versa. This is not a constraint, because it cannot be expressed as a set of satisfying tuples, but it interacts with the constraint queue in the same way as a constraint. It is only queued for removals from x_i. Instances of this algorithm are shown in figure 14 as dashed arrows.

The failure of a constraint in a side problem $\mathcal{PV}$ does not imply failure in $\mathcal{P}$. Therefore, if a constraint $\neg C_k$ from a side problem is queued in $\mathcal{P}$, the propagation algorithm must not return false. In the case where the $\neg C_k$ propagation algorithm would return false, the algorithm is altered to empty the domains of all variables in the scope of $\neg C_k$, then return true. (This is achieved with another procedure which calls the propagation algorithm, then checks its return value.) The *pureLink* and *pureCopy* algorithms also never return false.

If the domain of a pm variable becomes empty, this does not signify failure in $\mathcal{P}$ because the pm variables are not part of the original QCSP instance. Fortunately, the measures taken with $\neg C_k$ constraints are sufficient to isolate the pm variables from $\mathcal{P}$, so that empty pm variables do not cause failure in $\mathcal{P}$.

3.5.2. Pure value rule for all variables. If the pure value rule is required for all variables, the scheme can be optimized. For each constraint $C_k \in \mathcal{C}$, one side CSP $\mathcal{PV}_k$ is constructed with the single constraint $\neg C_k$ and a set of pm^k variables, as above. The same *pureLink* algorithm is used to link variables $pm_i^* = \{pm_i^k | 1 \leq k \leq e\}$ with x_i, with one instance of *pureLink* for each variable x_i. The *pureLink* algorithm keeps pm_i^* synchronized with x_i as well as detecting when some value of x_i is pure. The scheme is illustrated in figure 15, for a QCSP with four variables and four constraints. Each instance of *pureLink* is shown with a dotted line.

The implementation details are similar to the single-variable scheme. Negated constraints and instances of *pureLink* are called from the constraint queue of $\mathcal{P}$. Once again the propagation algorithms of the negated constraints are changed so that they do not return false, but empty the domains of all variables in their scope.

The time and space complexities of these schemes are not straightforward to derive, because they depend on the propagation algorithms used.

3.5.3. Propagation algorithms in these schemes. The schemes are both very general, because they work for different definitions of consistency, although if the consistency is not equivalent to GAC, then it only approximates the pure value rule. Any CSP propagation algorithm can be used for the negated constraint. It is also possible to decompose the negated constraint into multiple constraints when convenient. For example, if C_k is allDifferent(x_1,x_2,x_3), which is satisfied iff the variables take three different values, $\neg C_k$ could be written as

$$pm_1^k = pm_2^k \vee pm_1^k = pm_3^k \vee pm_2^k = pm_3^k$$

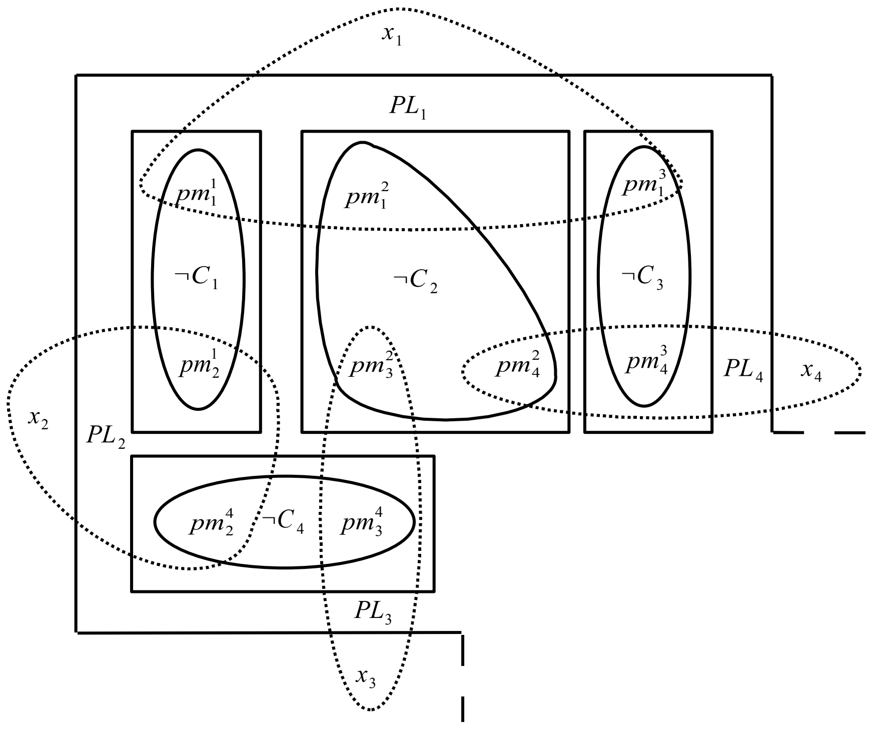


FIGURE 15. Implementation of pure value rule for all variables

(any pair are equal) then decomposed to four constraints,

$$(pm_1^k = pm_2^k) \Leftrightarrow b_1, (pm_1^k = pm_3^k) \Leftrightarrow b_2, (pm_2^k = pm_3^k) \Leftrightarrow b_3, b_1 \vee b_2 \vee b_3$$

The propagation algorithms in the next chapter process arbitrary constraints, specified by a set of tuples, so generating the negated constraint is as simple as taking the complement of the set. In chapter 5, a reified disjunction constraint is described, for expressions of the form $(\neg)x_{k_1} \vee (\neg)x_{k_2} \vee \cdots \Leftrightarrow (\neg)x_{k_i}$. To negate these constraints, the right hand side is negated.

3.6. Backjumping and learning

In the literature review (chapter 2 section 2.3.4.1) I mentioned the QCSP-Solve algorithm by Kostas Stergiou. This is a backtracking search algorithm for binary QCSP (i.e. QCSPs where all constraints contain two variables) with a form of consistency (forward checking), conflict directed backjumping and solution directed pruning. The search-based QBF solvers have features such as conflict learning and solution learning alongside their form of consistency (an extended unit propagation). All these techniques are potentially useful for general QCSP, but they are outside the scope of this thesis because I focus on consistency.

3.7. Queso solver implementation

The QCSP solver Queso was created from scratch for this thesis. Queso is implemented in an object-oriented style, in Java 1.5. The main class is called QCSP, with other classes for constraints (all derived from the class Constraint), variables and the constraint queue. Figure 16 shows a simplified UML diagram of Queso, to give an overview of the design. Most of the constraint subclasses have been omitted, and some complications due to the pure value rule have been removed.

The main design goal is flexibility for experimentation. A secondary goal is efficiency, which centres around how the domains of variables are represented. Using an object-oriented programming language helps with the first goal, because classes can be easily subclassed and their behaviour changed without changing their relationship with other classes.

All experiments were performed on a 3.06 GHz Pentium 4 PC with 1 GB of RAM, running linux 2.4, with the Java 1.5 runtime environment in server mode. Since the runtime environment optimizes the program as it runs (for the first few seconds), experiments are repeated two or three times to get a reliable CPU timing.

3.7.1. Representing variable domains. The variable domains are a finite subset of the integers. There are two variable representations in Queso: the Boolean array and the bounds representation. In all experiments, the Boolean array representation is used unless stated otherwise.

3.7.1.1. *Boolean array representation.* If the initial domain of x_i is D_i^0, it is desirable to be able to represent every subset of D_i^0. A Boolean array (called hash) is used, with one entry for every integer between $\min(D_i^0)$ to $\max(D_i^0)$, where hash(a) is true iff $a \in D_i$. The upper and lower bounds are also maintained and updated with every domain removal. This representation provides an $O(1)$ implementation of the most common operations: testing whether some value

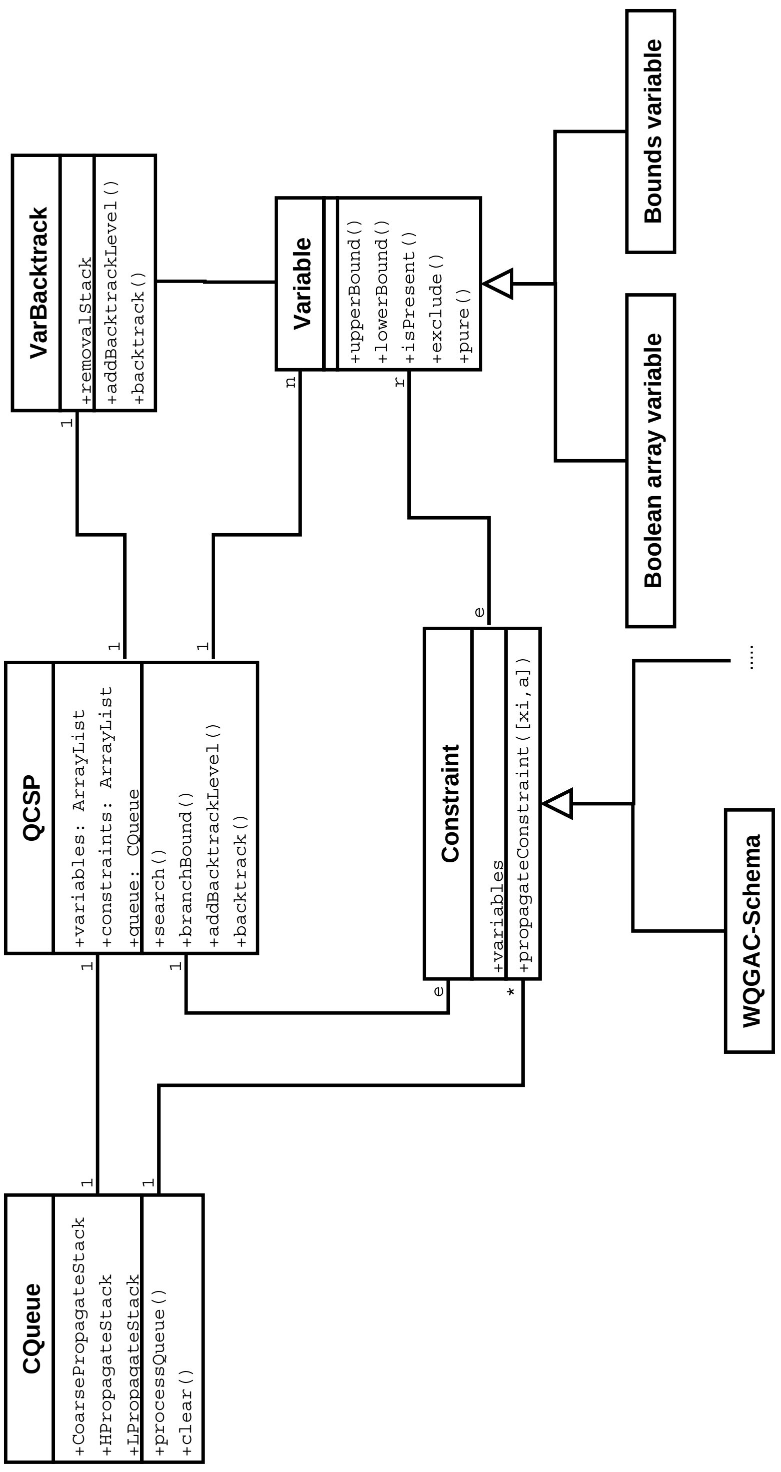


FIGURE 16. Simplified UML model of Queso

is in the domain (*isPresent*), finding the lower bound (*lowerBound*) and the upper bound (*upper-Bound*). Assuming that *isPresent* is inlined by the Java runtime, the cost should be the same as performing three additions and two pointer dereferences. The operations which remove values from the domain are unavoidably less efficient, since they iterate through the wakeUp lists.

This representation is used in all experiments unless stated otherwise.

3.7.1.2. *Bounds representation.* If the initial domain is very large, the bounds representation can be used. The upper and lower bounds are stored as integers whose size is bounded only by available memory. Clearly in this representation values can only be excluded from the top or bottom of the domain, so the *exclude* procedure is not available. Other procedures (*excludeLower*, *excludeUpper*, *isPresent*, *lowerBound*, *upperBound*) are available as variants which accept or return unlimited integers. Currently the bounds representation does not support pure values, and cannot be used for universal variables.

3.7.2. Backtracking. The two methods *addBacktrackLevel* and *backtrack* of class QCSP are called from the *search* and *branchBound* procedures. They call the methods of the same name in class VarBacktrack and on all constraints which have backtracking internal state.

The VarBacktrack class manages backtracking the domains of all the variables. Whenever some value $a \in D_i$ is removed, the index i and value a are pushed onto the removalStack. The *addBacktrackLevel* method pushes a marker onto the stack. The *backtrack* method pops elements off the stack and restores them to their respective domains, until it reaches a level marker or the stack is empty.

This backtracking mechanism minimizes the amount of memory that is restored when backtracking. Recent research has shown that it may be more efficient to arrange all the backtracking state into a single static block of memory and simply copy the whole block to save or restore the state of the solver [**50**]. Unfortunately this is difficult to do in Java because the language does not allow arbitrary copying of memory. Queso is not designed for the set of constraints to change during search, because I had no need to add constraints during search. The backtracking mechanism cannot currently undo a change to the set of constraints, or to the wakeUp lists, although this facility could be easily added.

3.8. Encodings

For the purpose of comparing Queso with other QCSP solvers and QBF solvers, various encodings are presented here.

3.8.1. Arbitrary QCSP to binary QCSP. The binary solver QCSP-Solve is quite sophisticated, so it is important to compare Queso with QCSP-Solve. In CSP binary arc-consistency (AC) combined with the hidden variable encoding [**84**] of an arbitrary non-binary constraint C_k simulates GAC on C_k. It is reasonable to ask whether QAC on the hidden variable encoding simulates SQGAC or WQGAC on C_k, in the presence of universal variables.

DEFINITION 3.8.1. Hidden variable encoding for QCSP $\mathcal{P}$ and constraint C_k

Given a constraint $C_k \in \mathcal{C}$ with set C_k^S of satisfying tuples and variables $\mathcal{X}_k = \langle x_{k_1}, \dots, x_{k_r} \rangle$, it is encoded as an additional existential variable $h \in C_k^S$ and r binary constraints $c_{k_1}(x_{k_1}, h)$ up to $c_{k_r}(x_{k_r}, h)$ such that $c_{k_i}(x_{k_i}, h)$ has a satisfying tuple $\langle a, t \rangle$ for each t, containing assignment $x_{k_i} \mapsto a$ and $h \mapsto t$ iff $t_i = a$. h is existentially quantified after each of $\mathcal{X}_k$.

This encoding is novel, but the difference from the CSP case is quite small.

If the encoded QCSP is QAC, h mirrors C_k^S because invalid tuples are removed by QAC on $c_{k_1} \dots c_{k_r}$. Also, if some assignment $x_{k_i} \mapsto a$ has no valid supporting tuple in $h = C_k^S$, then $x_i \mapsto a$ will be removed to maintain QAC on c_{k_i}. Assuming the hidden variable is never instantiated by the search procedure, this is equivalent to GAC on C_k. The proof lifts from that of Stergiou and Walsh [**94**].

THEOREM 3.8.2. *GAC on a constraint C_k and QAC on the hidden variable encoding of C_k are equivalent.*

PROOF. The proof is by lifting the property from CSP to QCSP. The constraints $c_{k_1} \dots c_{k_n}$ in the hidden variable encoding all have quantification pattern $\forall x_{k_i} \exists h$ or $\exists x_{k_i}, h$. For these types of binary constraint, when propagating from x_i to h, the propagation rule is the same as in AC. When propagating from h to x_{k_i}, the propagation rule for type $\exists x_{k_i}, h$ is the same as in AC. For type $\forall x_{k_i} \exists h$, the constraint c_{k_i} is inconsistent iff some value of x_{k_i} conflicts with all remaining values of h. Equivalently, enforcing GAC would attempt to remove the value of x_{k_i} which is not contained in any remaining tuple in C_k^S. Since x_{k_i} is universal, C_k would also be found inconsistent. $\square$

Therefore the hidden variable encoding for QCSP is quite weak. The obvious way of generalising it is to represent winning strategies in the domain of h. Now $c_{k_i}(x_{k_i}, h)$ has a conflict between assignment $x_{k_i} \mapsto a$ and $h \mapsto S$ iff $\forall t \in \text{sce}(S)\ :\ t_i \neq a$ ($x_{k_i} \mapsto a$ is not supported by the strategy S). Unfortunately the number of winning strategies can grow exponentially in the number of tuples in C_k^S, which itself can be exponential in the arity of the constraint. Therefore I think this approach would be impractical.

There may be some other encoding which is practical and which enforces some consistency greater than GAC on C_k, but it is not clear, therefore I will use the hidden variable encoding.

3.8.2. Encoding binary QCSP to QBF. In this section I first briefly elaborate on the difficulties in encoding QCSP into QBF. Then I present the previous best encoding of binary QCSP into QBF (the adapted log encoding) which was developed together with Ian Gent and and Andrew Rowley, and finally I introduce a new encoding (the enhanced log encoding) which was developed alone and which improves on the adapted log, both in simplicity and performance.

3.8.2.1. *The difficulty in encoding QCSP to QBF.* Gent, Nightingale and Rowley introduced a number of different ways to encode a QCSP instance into QBF [**52**]. To encode an existential QCSP variable to a set of QBF variables, some assignments to the QBF variables represent values in the original variable, and other assignments are ruled out by imposing clauses. For example, if an assignment indicates that the original QCSP variable has no values in its domain, the assignment is invalid and is ruled out with a clause. However this approach is not possible for a universal QCSP variable.

To see why, consider the QBF encoding as a game between existential and universal players, where the existential player aims to make the formula true and the universal player aims to make it false. The players set their own variables in quantification order. If clauses are imposed to rule out illegal assignments of universal variables, the universal player simply wins the game by unsatisfying the clauses. From the definition of QCSP semantics (definition 3.2.3), this is not the required behavior.

The encodings introduced in [**52**] were able to overcome this difficulty. However, the *global acceptability encoding* and the *local acceptability encoding* were very inefficient compared to direct QCSP algorithms. In contrast, the *adapted log encoding*, which I describe below, turned out to be very efficient.

3.8.2.2. *Adapted log encoding.* In order to deal with the difficulty described above, the adapted log encoding (by Gent, Nightingale and Rowley [**52**]) uses indicator variables (first described by Gent and Rowley [**46**]) to indicate when a universal assignment is not valid. There is one indicator variable z^{x_i} for each of the original universal QCSP variables x_i. z^{x_i} is existentially quantified in a final block at the end of the quantifier sequence. If z^{x_i} is set true, then all remaining clauses representing constraints (conflict clauses) are satisfied.

In SAT, it has often been noted that just three variables are needed to encode 8 values of a CSP variable, instead of the 8 in the direct encoding [**43, 98**], reducing the branching factor from 256 to 8. This is known as the log encoding. However, since only one value is allowed by at-least-one (ALO) and at-most-one (AMO) clauses, there is no real reduction, and the use of three variables in clauses for one CSP variable reduces the effect of propagation [**98**]. For QCSPs Gent et al. show that the log encoding can be very effective when applied to universal variables only.

Each variable in a QCSP is encoded to a set of variables in QBF, with these sets quantified in the same way and in the same order as in the QCSP. Additional existential variables are added to the end of the quantifier sequence. For an existential variable, each QBF variable represents one value. For a universal, each value is represented by a unique assignment of the QBF variables, and also each value is represented by an existential QBF variable quantified at the end. There are channelling clauses which maintain correspondence between these two representations. The existential QBF variables are used in the conflict clauses to avoid having a conjunction of literals, therefore to avoid distributing conjunction over disjunction.

The quantifier sequence for the encoding is summarized below.

- To encode some existential variable $x_i \in \{1 \dots d\}$:

$\exists b_1^{x_i}, \ldots, b_d^{x_i}$

- To encode some universal variable $x_j \in \{1 \ldots d\}$:

  $\forall w_1^{x_j}, \ldots, w_{\lceil \log_2 d \rceil}^{x_j}$
- Finally, for all universal variables $x_j \in \{1 \ldots d\}$ there is a set of existential variables representing each value:

  $\exists b_1^{x_j}, \ldots, b_d^{x_j}$
- Also there are indicator variables for each invalid assignment to $w_*^{x_j}$, and one overall indicator variable:

  $\exists i_*^{x_j}, z^{x_j}$

Suppose there is a universal variable x_j, where x_k is also universal and precedes x_j in the QCSP quantifier sequence, s.t. there is no other universal variable between them. The following clauses map assignments of $w_*^{x_j}$ to $b_*^{x_j}$ and $i_*^{x_j}$. The nature of these channelling clauses ensures that at least one of the latter variables is set to true. This is given as an example for $d = 5$, but the general form is easy to infer from the example.

- **Channelling clauses**

$$\begin{gathered}
z^{x_k} \vee b_1^{x_j} \vee w_3^{x_j} \vee w_2^{x_j} \vee w_1^{x_j} \\
z^{x_k} \vee b_2^{x_j} \vee w_3^{x_j} \vee w_2^{x_j} \vee \neg w_1^{x_j} \\
z^{x_k} \vee b_3^{x_j} \vee w_3^{x_j} \vee \neg w_2^{x_j} \vee w_1^{x_j} \\
z^{x_k} \vee b_4^{x_j} \vee w_3^{x_j} \vee \neg w_2^{x_j} \vee \neg w_1^{x_j} \\
z^{x_k} \vee b_5^{x_j} \vee \neg w_3^{x_j} \vee w_2^{x_j} \vee w_1^{x_j} \\
z^{x_k} \vee \left(i_6^{x_j} \iff (w_3^{x_j} \vee \neg w_2^{x_j} \vee w_1^{x_j})\right) \\
z^{x_k} \vee \left(i_7^{x_j} \iff (w_3^{x_j} \vee w_2^{x_j} \vee \neg w_1^{x_j})\right) \\
z^{x_k} \vee \left(i_8^{x_j} \iff (w_3^{x_j} \vee w_2^{x_j} \vee w_1^{x_j})\right)
\end{gathered}$$

The variables $i_*^{x_j}$ indicate when the assignment is invalid. The overall indicator variable is set from $i_*^{x_j}$ and also the previous overall indicator variable z^{x_k}. Hence they cascade forward.

- **Indicator collector clauses**

$$z^{x_j} \iff i_6^{x_j} \vee i_7^{x_j} \vee i_8^{x_j} \vee z^{x_k}$$

For each existential variable x_i, at least one of the QBF variables $b_*^{x_i}$ must be set to true. This is accomplished with an at-least-one clause.

- **ALO clause**

$$\bigvee_{a\in\{1\dots d\}} b_a^{x_i}$$

Constraints are represented as follows. Consider a constraint C_k between any variables x_i and x_j. A pair of values $\langle c, d\rangle$, where $c \in D_i$ and $d \in D_j$, that do not satisfy the constraint (i.e. $\langle c, d\rangle \notin C_k^S$) is represented with a single clause in the QBF. x_i precedes x_j in the variable sequence, and variable x_l is universally quantified and directly precedes x_j in the variable sequence. Note that x_l may be the same as x_i. The indicator variable for x_l is used, so that if a preceding universal variable is set in an invalid way, the conflict clause is satisfied. (When a universal is set invalidly, the remaining part of the QBF must be satisfiable.)

- **Conflict clauses**

$$\forall\langle c, d\rangle \notin C_k^S \;:\; z^{x_l} \vee \neg b_c^{x_i} \vee \neg b_d^{x_j}$$

For channelling and conflict clauses, if there is no preceding universal variable in the QCSP, the indicator variable is omitted. To illustrate the encoding, I encode the following simple QCSP.

- QCSP:

$$\forall x_1 \exists x_2 \;:\; x_1 \neq x_2 \text{ where } D_1 = D_2 = \{1, \dots, 5\}$$

- QBF quantifier sequence:

$$\forall w_3^{x_1}, w_2^{x_1}, w_1^{x_1}, \exists b_1^{x_2}, b_2^{x_2}, b_3^{x_2}, b_4^{x_2}, b_5^{x_2}, \exists b_1^{x_1}, b_2^{x_1}, b_3^{x_1}, b_4^{x_1}, b_5^{x_1}, \exists z^{x_1}, i_6^{x_1}, i_7^{x_1}, i_8^{x_1}$$

- Channelling clauses for x_1:

$$b_1^{x_1} \lor w_3^{x_1} \lor w_2^{x_1} \lor w_1^{x_1}$$
$$b_2^{x_1} \lor w_3^{x_1} \lor w_2^{x_1} \lor \neg w_1^{x_1}$$
$$b_3^{x_1} \lor w_3^{x_1} \lor \neg w_2^{x_1} \lor w_1^{x_1}$$
$$b_4^{x_1} \lor w_3^{x_1} \lor \neg w_2^{x_1} \lor \neg w_1^{x_1}$$
$$b_5^{x_1} \lor \neg w_3^{x_1} \lor w_2^{x_1} \lor w_1^{x_1}$$
$$\neg i_6^{x_1} \iff (\neg w_3^{x_1} \lor w_2^{x_1} \lor \neg w_1^{x_1})$$
$$\neg i_7^{x_1} \iff (\neg w_3^{x_1} \lor \neg w_2^{x_1} \lor w_1^{x_1})$$
$$\neg i_8^{x_1} \iff (\neg w_3^{x_1} \lor \neg w_2^{x_1} \lor \neg w_1^{x_1})$$

- Indicator collector clauses for x_1:

$$z^{x_1} \iff i_6^{x_1} \lor i_7^{x_1} \lor i_8^{x_1}$$

- At-least-one clause for x_2:

$$b_1^{x_2} \lor b_2^{x_2} \lor b_3^{x_2} \lor b_4^{x_2} \lor b_5^{x_2}$$

- Conflict clauses representing $x_1 \neq x_2$:

$$z^{x_1} \lor \neg b_1^{x_1} \lor \neg b_1^{x_2}$$
$$z^{x_1} \lor \neg b_2^{x_1} \lor \neg b_2^{x_2}$$
$$z^{x_1} \lor \neg b_3^{x_1} \lor \neg b_3^{x_2}$$
$$z^{x_1} \lor \neg b_4^{x_1} \lor \neg b_4^{x_2}$$
$$z^{x_1} \lor \neg b_5^{x_1} \lor \neg b_5^{x_2}$$

Unit propagation on the encoding is equivalent to forward checking on the QCSP, modulo differences in branching during search.

The subtlety of this encoding is that clauses which force equivalence between $b_a^{x_i}$ and the corresponding values of $w_*^{x_i}$ are omitted. So clauses such as $z^{x_1} \lor \neg b_1^{x_2} \lor \neg w_2^{x_2}$, where x_1 and x_2 are universal, are omitted. It might seem that this is erroneous, as it allows a universal to take two values, if $b_1^{x_2}$ and $b_2^{x_2}$ are both true. But there is no way that setting of the universals $w_*^{x_2}$ can *force* more than one $b_*^{x_2}$ to be true. On the other hand, consider the extreme case where all $b_*^{x_2}$ can be set true, as for example can happen if x_2 occurs in no constraints. This makes all log value clauses satisfied by $b_*^{x_2}$. The benefit of this is that the pure literal rule for existential variables does some work. If $b_a^{x_2}$ only occurs positively, it can be set to true. If this is the case for all values $a \in D_2$, the $w_*^{x_2}$ variables will be instantiated, greatly reducing the need for search. So, with sufficient care, the pure literal rule included in our QBF solver can do useful simplifications. This property carries over to the enhanced log encoding, described below.

3.8.2.3. *Enhanced log encoding.* The *enhanced log encoding* is a refinement of adapted log. Each universal variable x_i is encoded by $\lceil \log_2 d \rceil$ variables w^{x_i} which are universally quantified together. The order of variables is preserved in the quantifier sequence. I also introduce $b_1^{x_i} \dots b_d^{x_i}$ variables for each universal QCSP variable x_i, which are existentially quantified at the end of the quantifier sequence. These b^{x_i} variables are used in the conflict clauses, to avoid having several w^{x_i} literals in conjunction to represent a value, therefore to avoid distributing conjunction over disjunction. The $w_*^{x_i}$ variables are channelled to the b^{x_i} variables with a set of d clauses. For the following example $d = 5$.

- **Log value clauses**

$$b_1^{x_i} \vee w_3^{x_i} \vee w_2^{x_i} \vee w_1^{x_i}$$
$$b_2^{x_i} \vee w_3^{x_i} \vee w_2^{x_i} \vee \neg w_1^{x_i}$$
$$b_3^{x_i} \vee w_3^{x_i} \vee \neg w_2^{x_i}$$
$$b_4^{x_i} \vee \neg w_3^{x_i} \vee w_2^{x_i}$$
$$b_5^{x_i} \vee \neg w_3^{x_i} \vee \neg w_2^{x_i}$$

There are 8 possible assignments to the w^{x_i} variables, and 5 values, so for the values 3, 4 and 5 there are two w^{x_i} assignments mapped onto each, hence all 8 assignments are valid. In contrast to adapted log, no local acceptability variable (z^{x_k} above) is present, because no assignments to previous universal variables can be invalid.

To state this formally, I represent a QBF with the tuple $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}', \mathcal{Q}' \rangle$ where $\mathcal{Q}'$ is the quantifier sequence, $\mathcal{X}'$ is the set of Boolean variables and $\mathcal{C}'$ is the set of disjunctive clauses, to mirror the QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$.

The quantifier sequence $\mathcal{Q}$ for the QCSP is encoded as shown by the following two recursive rules. This also implies the set of variables in the encoding. $\mathcal{Q}' = \text{translate}(\mathcal{Q})$.

$$\text{translate}(\exists x_i \in \{1 \dots d\}, \mathcal{Q}_1) = \exists b_1^{x_i} \dots b_d^{x_i}, \text{translate}(\mathcal{Q}_1)$$
$$\text{translate}(\forall x_i \in \{1 \dots d\}, \mathcal{Q}_2) = \forall w_1^{x_i} \dots w_{\lceil \log_2 d \rceil}^{x_i}, \text{translate}(\mathcal{Q}_2), \exists b_1^{x_i} \dots b_d^{x_i}$$

Existential variable x_i in the QCSP is mapped to d existential variables $b_1^{x_i} \dots b_d^{x_i}$ representing the presence of each value in the domain, with one clause (known as at-least one, ALO): $b_1^{x_i} \vee b_2^{x_i} \vee \cdots \vee b_d^{x_i}$, which guarantees one or more of the variables will be set to true in a solution.

Universal variable x_i in the QCSP is mapped to $\lceil \log_2 d \rceil$ variables w^{x_i}. Every complete assignment A to variables w^{x_i} (of which there are $2^{\lceil \log_2 d \rceil}$) is mapped to a value $a \in D_i$. All values a map to one assignment, or two assignments with only one literal different. $2^{\lceil \log_2 d \rceil} - d$ values must map to two assignments. An assignment A is represented as a conjunction of literals (e.g. $w_1^{x_i} \wedge \neg w_2^{x_i}$). For some value $a \in D_i$ which maps to just one assignment A, the channelling clause is as follows.

$$\neg b_a^{x_i} \vee \neg A$$

The negated conjunction $\neg A$ is converted to a disjunction in the usual way. For some other value c which maps to two assignments A_1 and A_2, the channelling clause is given below.

$$\neg b_a^{x_i} \vee \neg (A_1 \wedge A_2)$$

The simplification of $\neg(A_1 \wedge A_2)$ ends with a disjunction of $\lceil \log_2 d \rceil - 1$ literals.

For a constraint C_k with $\mathcal{X}_k = \langle x_i, x_j \rangle$ and satisfying tuples C_k^S the conflict clauses are:

- **Conflict clauses** For all tuples $\langle c, d \rangle \notin C_k^S$,

$$(\neg b_c^{x_i} \vee \neg b_d^{x_j})$$

To illustrate the encoding, I use the same example as in the previous section.

- QCSP:

$$\forall x_1 \exists x_2 \ : \ x_1 \neq x_2 \text{ where } D_1 = D_2 = \{1, \dots, 5\}$$

- QBF quantifier sequence:

$$\forall w_3^{x_1}, w_2^{x_1}, w_1^{x_1}, \exists b_1^{x_2}, b_2^{x_2}, b_3^{x_2}, b_4^{x_2}, b_5^{x_2}, \exists b_1^{x_1}, b_2^{x_1}, b_3^{x_1}, b_4^{x_1}, b_5^{x_1}$$

- Channelling clauses for x_1:

$$b_1^{x_1} \vee w_3^{x_1} \vee w_2^{x_1} \vee w_1^{x_1}$$
$$b_2^{x_1} \vee w_3^{x_1} \vee w_2^{x_1} \vee \neg w_1^{x_1}$$
$$b_3^{x_1} \vee w_3^{x_1} \vee \neg w_2^{x_1}$$
$$b_4^{x_1} \vee \neg w_3^{x_1} \vee w_2^{x_1}$$
$$b_5^{x_1} \vee \neg w_3^{x_1} \vee \neg w_2^{x_1}$$

- At-least-one clause for x_2:

$$b_1^{x_2} \vee b_2^{x_2} \vee b_3^{x_2} \vee b_4^{x_2} \vee b_5^{x_2}$$

- Conflict clauses representing $x_1 \neq x_2$:

$$\neg b_1^{x_1} \vee \neg b_1^{x_2}$$
$$\neg b_2^{x_1} \vee \neg b_2^{x_2}$$
$$\neg b_3^{x_1} \vee \neg b_3^{x_2}$$
$$\neg b_4^{x_1} \vee \neg b_4^{x_2}$$
$$\neg b_5^{x_1} \vee \neg b_5^{x_2}$$

To prove the correctness of the encoding, I'll slightly abuse the notation of simplification by assuming the quantifier for an instantiated variable is removed, and that $x_i \mapsto a$ is equivalent to $D_i = \{a\}$.

THEOREM 3.8.3. *A QCSP is true if and only if the encoded QBF is true, for the enhanced log encoding.*

PROOF. The proof is recursive and closely follows the definition of QCSP semantics (definition 3.2.3). A QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ represents the logical formula $\phi = Q_1 x_1 \in D_1, \ldots, Q_n x_n \in D_n : (\mathcal{C})$ which is encoded as a QBF $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}', \mathcal{Q}' \rangle$ representing $\phi' = Q'_1 x'_1, \ldots, Q'_n x'_n : (\mathcal{C}')$. The encoding of the empty QCSP (containing no variables or constraints) is the empty QBF which is vacuously true.

Existential case:

- $\mathcal{Q}$ is of the form $\exists x_1 Q_2 x_2 \ldots$
- $\mathcal{Q}'$ is of the form $\exists b_1^{x_1} \ldots b_d^{x_1} Q_2 \ldots$
- $\mathcal{P}$ is satisfiable iff there exists some value $a \in D_1$ such that $\mathcal{P}[D_1 = \{a\}]$ is satisfiable.

- $\mathcal{P}'$ is satisfiable iff there exists an assignment $A = (\neg)b_1^{x_1} \wedge \cdots \wedge (\neg)b_d^{x_1}$ such that the ALO clause is satisfied and $\mathcal{P}'[A]$ is satisfiable.

The QCSP value a can be any value such that $b_a^{x_1} \mapsto 1$. If there is more than one a s.t. $b_a^{x_1} \mapsto 1$, then all these values can be extended to a winning strategy.

Universal case:

- $\mathcal{Q}$ is of the form $\forall x_1 Q_2 x_2 \ldots$
- $\mathcal{Q}'$ is of the form $\forall w_1^{x_1} \ldots w_{\lceil \log_2 d \rceil}^{x_1} Q_2 \ldots$
- $\mathcal{P}$ is satisfiable iff for all values $a \in D_1$, $\mathcal{P}[D_1 = \{a\}]$ is satisfiable
- $\mathcal{P}'$ is satisfiable iff for all assignments $A = (\neg)w_1^{x_1} \ldots (\neg)w_{\lceil \log_2 d \rceil}^{x_1}$ $\mathcal{P}'[A]$ is satisfiable.

Note that each value a is covered by some assignment A. $\square$

The problem with this encoding is that the QBF solver can search two equivalent subtrees in some cases. In the example above, when $w_3^{x_1} \mapsto$ true and $w_2^{x_1} \mapsto$ true, the solver can branch on $w_1^{x_1}$ which is not contained in any clause.

After setting $w_3^{x_1}$ and $w_2^{x_1}$, if either is set to true then the first two clauses above are satisfied and in the reduced set of clauses $w_1^{x_1}$ does not exist. Both $w_1^{x_1}$ and $\neg w_1^{x_1}$ are pure, so if the solver implements the pure literal rule then it will not branch on this variable. This solves the repeated subtree problem mentioned above, on the condition that $w_1^{x_1}$ is set last. Also, in common with the adapted log encoding, the channelling works only from w^{x_i} to b^{x_i} variables, so only positive b^{x_i} literals are included in the clause set above, therefore the pure literal rule can detect cases where the $b_a^{x_i}$ is involved in no conflicts. In some circumstances, this can also lead to the elimination of w^{x_i} variables. For example, if $b_4^{x_1}$ and $b_5^{x_1}$ become pure, then $w_3^{x_1}$ becomes pure as well and the search is reduced accordingly.

There is no reason that the adapted log encoding might be more efficient than enhanced log on any instance. Experiments show the enhanced log, combined with the search-based solver CSBJ [**47**], to be much more efficient on a range of random instances [**53**].

3.8.3. Summary of empirical results. In [**53**], we describe a random binary QCSP generator. We discuss the issue of flaws and ensure that the generator controls the probability of flaws. The

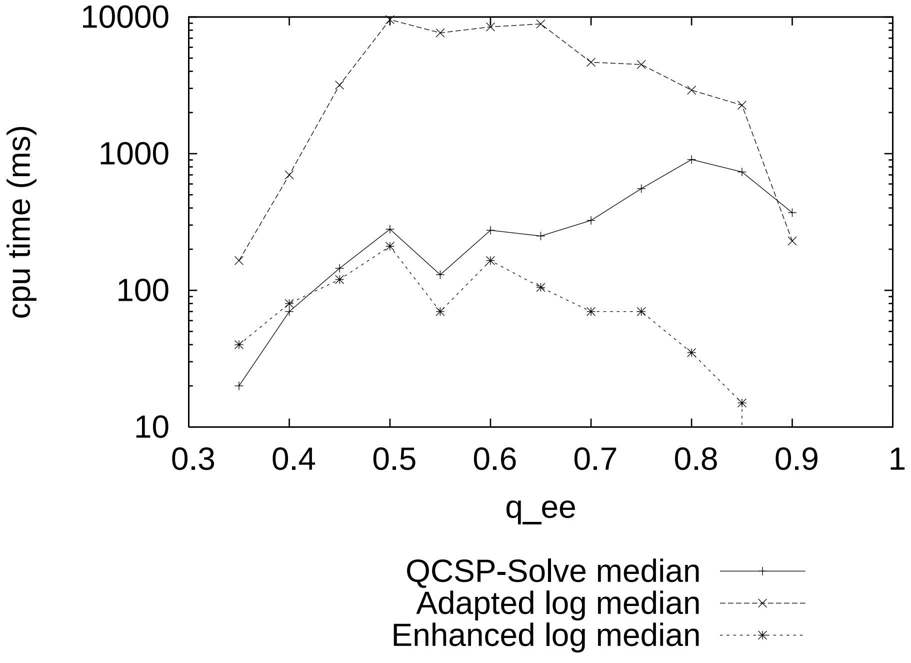


FIGURE 17. Comparison of the enhanced log encoding with adapted log and QCSP-Solve

adapted and enhanced log encodings are compared using random binary QCSP instances, along with QCSP-Solve. CSBJ was used to solve the encoded instances. Figure 17 shows the results of the experiment, where q_{ee} is a constraint looseness parameter. The graph plots the median of CPU time over 100 instances. The enhanced log encoding significantly outperforms adapted log, and also competes well with QCSP-Solve, particularly when the constraints are loose.

3.9. Summary

This chapter has provided the context for the following chapters about propagation algorithms, including the theoretical background and search and pure value algorithms, as well as some material used for experiments, and some other contributions.

CHAPTER 4

Strong consistency for arbitrary constraints

4.1. Introduction

The practice in constraint programming of expressing arbitrary constraints as a list of satisfying tuples has been around for some time, and many non-binary constraint propagation algorithms exist. Some of the literature about arbitrary non-binary constraints in CSP is reviewed in chapter 2 section 2.1.2.2. These constraints are actively used [**49**, **90**]. This is the main motivation for developing similar techniques in QCSP. A secondary motivation is that any constraint with reasonable parameters (i.e. such that the propagation algorithm performs well) can be expressed, and propagated strongly using the algorithms developed in this chapter. This eases the problem of modelling in QCSP, by increasing the expressiveness of the language relative to the existing work in the field.

This chapter addresses the need for propagation of arbitrary constraints in QCSP, by developing two new propagation algorithms. The algorithms presented here are the first practical algorithms for arbitrary constraints in QCSP, therefore no direct comparison with other work is possible. They are compared using encodings to QBF solvers and binary QCSP solvers. The new algorithms are also applied to two games.

The existing literature on consistency algorithms for QCSP is reviewed in chapter 2 section 2.3.1.

4.2. Motivating examples

This section presents various examples where propagation of arbitrary non-binary constraints is stronger (i.e. able to make more removals) than the scheme proposed by Bordeaux and Monfroy, or the binary arc-consistency proposed by Mamoulis and Stergiou combined with the hidden variable encoding.

Consider the expression below. This expression evaluates to false, because of the case where $x_1 = x_2 = x_3 = true$.

(2) $$\forall x_1, x_2, x_3 \ : \ \neg x_1 \vee \neg x_2 \vee \neg x_3$$

In Bordeaux and Monfroy's scheme, this is broken down into 5 primitive constraints, with 4 additional variables, as follows.

(3) $$\forall x_1, x_2, x_3 \exists t_1, t_2, t_3, t_4 \ : \ t_1 = \neg x_1,\ t_2 = \neg x_2,\ t_3 = \neg x_3,\ t_4 = t_1 \vee t_2,\ t_3 \vee t_4 = true$$

The disjunction primitives are now operating on existentially quantified variables, so they are consistent. (Other decompositions are possible, but the same property holds.) Applying the hidden variable encoding to expression (2) yields the following decomposition:

(4) $$\forall x_1, x_2, x_3 \exists h \in \{1..7\} \ : \ r_1(x_1, h) \wedge r_2(x_2, h) \wedge r_3(x_3, h)$$

In this expression the domain of h and the relations r_1, r_2 and r_3 are defined according to definition 3.8.1 in chapter 3. No propagation is possible on the binary constraints r_1, r_2 and r_3, because they are already consistent according to the definition of QAC.

In contrast, applying SQGAC or WQGAC directly to expression (2) detects inconsistency. Therefore the algorithms in this chapter could detect inconsistency, but their worst-case running time is exponential in the arity. The next chapter (chapter 5) describes an algorithm specifically for reified disjunction constraints of any arity, which maintains SQGAC and can therefore detect inconsistency of expression (2). It has a linear worst-case running time, so it is preferable for this particular example.

Constructing a similar example with arithmetic is trivial. Consider the expression below. This is also false (because $x_1 = 4,\ x_2 = 3$ cannot be extended to a satisfying tuple), and not WQGAC or SQGAC. However it is GAC, so the hidden variable encoding is ineffective.

$$\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\} \; : \; x_1 - x_2 + x_3 = 3 \tag{5}$$

In Bordeaux and Monfroy's scheme, this can be decomposed in three ways. These are shown below, after the domain of t_2 is reduced.

$$\begin{gathered}\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\}, \exists t_1 \in \{-3,-4\}, \exists t_2 \in \{-1,0\} \; : \\ t_1 = -x_2, t_2 = x_1 + t_1, t_2 + x_3 = 3\end{gathered} \tag{6}$$

$$\begin{gathered}\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\}, \exists t_1 \in \{-3,-4\}, \exists t_2 \in \{-1,0\} \; : \\ t_1 = -x_2, t_2 = x_3 + t_1, t_2 + x_1 = 3\end{gathered} \tag{7}$$

$$\begin{gathered}\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\}, \exists t_1 \in \{-3,-4\}, \exists t_2 \in \{6,7\} \; : \\ t_1 = -x_2, t_2 = x_1 + x_3, t_2 + t_1 = 3\end{gathered} \tag{8}$$

The addition primitives are consistent in all of these decompositions, so the scheme does not detect inconsistency. Applying SQGAC or WQGAC directly to (5) detects inconsistency, therefore the algorithms in this chapter are useful on this example. In chapter 6 I give an algorithm for long sum constraints which has a linear worst-case running time, and which can also detect inconsistency on this example.

The final example C_k is expressed over four Boolean variables $\forall x_1, \exists x_2, \forall x_3, \exists x_4$. The set C_k^S of five tuples $t_{1..5}$ is given below.

	$\forall x_1$	$\exists x_2$	$\forall x_3$	$\exists x_4$
$t_1 =$	0	0	0	1
$t_2 =$	0	1	1	0
$t_3 =$	1	0	1	0
$t_4 =$	1	0	1	1
$t_5 =$	1	1	0	1

If this constraint can be reformulated as a logic constraint of the form $\bigvee l_i \iff l_j$ (where $l_i = x_i$ or $l_i = \neg x_i$, and l_j can be x_j, $\neg x_j$, or a constant 0 or 1), then SQGAC can be enforced in linear time by the algorithm described in chapter 5. I will briefly show this is not possible. If l_j is not a constant (i.e. it represents a variable, negated or otherwise), the number of satisfying tuples (over four variables) is 8. If $l_j = 1$, only one tuple is disallowed, so it is 15. If $l_j = 0$, only one tuple is satisfying. None of these matches the five satisfying tuples we have here, so C_k cannot be expressed as a single logic constraint. It could be expressed with several logic constraints, but in this form it is not possible to enforce SQGAC using the algorithm in chapter 5.

For the original constraint to be WQGAC, for the assignment $x_1 \mapsto 0$, a tuple is required for each value of x_3. t_1 and t_2 meet the requirement. However, the value for x_2 is different, so these two tuples could not form part of the same winning strategy for the constraint. Assignment $x_1 \mapsto 1$ is similar. There does not exist a value of x_2 such that all values of x_3 can be extended to a satisfying assignment. GAC (or QAC on the hidden variable encoding) and WQGAC can make no inferences, but SQGAC determines falsity.

It is not clear if C_k could be expressed as a single sum constraint. Assuming it cannot, the algorithm to enforce SQGAC for arbitrary constraints (section 4.3) is the most suitable in this thesis (and most suitable in general, to the best of my knowledge) for a constraint such as this.

4.3. An algorithm for enforcing SQGAC

This section and the next section describe the primary contribution of this chapter: algorithms for enforcing consistency on arbitrary non-binary constraints. In this section, the *SQGAC-propagate* algorithm is described.

Algorithm 11 high level description of SQGAC-propagate

```
procedure SQGAC-propagate($x_i$, $a$): Boolean
Consider MWST $T = \langle V, E, r, L \rangle$:
    (1) Remove all vertices labelled $x_{k_i} \mapsto a$ from $V$ and all edges including a vertex labelled
        $x_{k_i} \mapsto a$ from $E$, and remove all subtrees which become disconnected from the root, $r$.
    (2) Repeat the following to exhaustion:
        (a) Remove all vertices $b$ with no children if $b$ is not labelled with the final variable
            $x_{k_r}$.
        (b) For all universal variables $\forall x_{k_j}$ in the scope of $C_k$, a vertex $b$ labelled $x_{k_j} \mapsto v_j$
            must have siblings for all other values in $D_j$ (by the definition of MWST, definition
            3.2.11 in chapter 3). If not, $b$ is removed.
    (3) Any assignment no longer contained in the tree is pruned. If any domain is emptied or
        universal is pruned, return false, otherwise true.
```

First I describe the algorithm at a very high level. For some constraint C_k and subproblem $\mathcal{P}_k$, a multiple winning strategy tree (MWST) containing all winning strategies is maintained. If some assignment $x_{k_i} \mapsto v_i$ is not contained in the tree, it is inconsistent (by the definition of SQGAC, definition 3.2.13). When an assignment is removed (perhaps by some other constraint), the tree must be updated as shown in algorithm 11. When an assignment is removed, some winning strategies are no longer valid, so the set sce$(\mathcal{S})$ changes, and the corresponding MWST changes. On backtracking, the tree is restored.

Beginning with tree T, if algorithm 11 is called for all domain removals, then no nodes labelled $x_{k_i} \mapsto a$ where $a \notin D_{k_i}$ remain in the tree, by rule 1. I refer to the derived tree as T'. Any value not in T' is pruned from the relevant domain. Therefore it is necessary to prove that the transformation from T to T' is correct.

THEOREM 4.3.1. *Soundness and completeness of SQGAC-propagate to enforce SQGAC (or return false if that is impossible) when called for all domain removals.*

PROOF. I assume that T is an MWST before the procedure is applied. I prove that rules 2 (a) and 2 (b) are sound, and therefore that no unsound removals were made from T to form T'. This is followed by a proof that T' is indeed an MWST, therefore the transformation is sound and complete.

Rule 2 (a) is sound since a vertex a with no descendents, where a is not labelled with variable x_{k_r}, cannot be part of an MWST, because all leaf nodes are labelled x_{k_r} in such a tree. For rule 2 (b), any MWST has the appropriate siblings, hence a cannot be part of such a tree, therefore rule 2 (b) is also sound.

The proof of completeness appeals to the fact that an MWST represents one or more solution trees (definition 3.2.10 in chapter 3), which are in turn isomorphic to winning strategies. To extract a solution tree from an MWST, wherever a node has more than one child, and the children are labelled with x_{k_i} where $Q_{k_i} = \exists$, select one child and delete all others (and their descendents). This forms ST. ST is a solution tree: each node in ST is labelled correctly; each node has the correct number of descendents; each path from r to a leaf node in ST represents a scenario of a winning strategy of $\mathcal{P}_k$ (because ST is a subtree of T'). Therefore ST is isomorphic to some winning strategy of $\mathcal{P}_k$.

For all leaf nodes u in T', it is possible to construct a solution tree ST which contains u, simply by not making a choice that excludes u when constructing ST. ST is isomorphic to a winning strategy, therefore u corresponds to a scenario of a winning strategy, as required by the definition of MWST. □

4.3.1. Implementation. First the tree data structure is defined. Initially, a vertex (other than r) has the following data associated with it:

- Label (variable and value) var and val (equivalent to $x_{var} \mapsto val$)
- Reference to parent, p
- References to $|D_{i+1}|$ children, if x_{k_i} is not the final variable: $c_{j \in D_{i+1}}$
- The number of children, nc
- Two references to vertices with the same label, *left* and *right*.

The root r has only the references to children. There is a double-linked list header $list(x_{k_i} \mapsto a)$ for each label $x_{k_i} \mapsto a$. The list contains all vertices labelled $x_{k_i} \mapsto a$. $list(x_{k_i} \mapsto a).right$ is the reference to a vertex. The purpose of $list(x_{k_i} \mapsto a)$ is to detect when all the relevant vertices have been removed, allowing the algorithm to remove a from the domain of x_{k_i}. It also allows the algorithm to locate all vertices with a particular label.

Algorithm 12 removeVertex($vertex$: ver)

```
procedure removeVertex(vertex: $ver$, in out list: removeList):
$ver.p.c_{val} \leftarrow nil$ {disconnect from the parent}
$ver.p.nc \leftarrow ver.p.nc - 1$
if $ver.right = nil$ and $ver.left = list(x_{ver.var} \mapsto ver.val)$: {$ver$ is the last vertex with $ver.var$ and $ver.val$}
    Add $x_{ver.var} \mapsto ver.val$ to removeList {therefore add to removeList}
if $ver.right \neq nil$: {disconnect $ver$ from the list}
    $ver.right.left \leftarrow ver.left$
if $ver.left \neq nil$:
    $ver.left.right \leftarrow ver.right$
for all children $ver.c_j$: {remove all children as well}
    removeVertex($ver.c_j$, removeList)
```

Algorithm 13 restoreVertex(vertex: ver)

```
procedure restoreVertex(vertex: $ver$):
$ver.p.c_{ver.val} \leftarrow ver$ {reattach to parent}
$ver.p.nc \leftarrow ver.p.nc + 1$
if $ver.right \neq nil$: {reconnect $ver$ to the list}
    $ver.right.left \leftarrow ver$
if $ver.left \neq nil$:
    $ver.left.right \leftarrow ver$
```

All the algorithms focus on a single constraint C_k, with no reference to any variables not contained in C_k. Therefore to simplify the presentation slightly I refer to x_{k_i} as x_i from here.

Algorithm 12 is used by the propagation algorithm to remove a vertex from the tree. When removing a vertex, all its children become disconnected from the root so they must also be removed. The procedure *removeVertex* also checks if the vertex is the final one in its list, and if so schedules the appropriate value $ver.val$ for pruning. When backtracking, *restoreVertex* (algorithm 13) is called for all vertices that were removed. The method of removing and restoring elements in a double-linked list is by Hitotumatu and Noshita [**62**], and popularized by Knuth [**66**].

Algorithm 14 shows a more concrete version of the *SQGAC-propagate* algorithm. For each vertex labelled $x_i \mapsto a$, it is removed. However, to implement rules 2 (a) and 2 (b) from algorithm 11, before removing a vertex, the algorithm checks if its parent also needs to be removed. For rule 2 (a), this occurs if the parent has no other children. For 2 (b), if the vertex to be removed represents some value in the domain of a universal variable, then all its siblings need to be removed, and therefore by 2 (a) its parent as well. It is sufficient to remove the parent. Hence the vertex to

Algorithm 14 SQGAC-propagate(x_i, a)

```
procedure SQGAC-propagate($x_i$, $a$): Boolean
removeList← $\emptyset$
for all $ver \in list(x_i \mapsto a)$ {iterate through the list}
    while $ver.p.nc = 1$ or [$\forall(ver.var)$ and $ver.val \in D_{ver.var}$]:
        if $ver.p.nc = 1$: {this vertex has no siblings, 2 (a)}
            $ver \leftarrow ver.p$
        if $\forall(ver.var)$ and $ver.val \in D_{ver.var}$: {universal, 2 (b)}
            $ver \leftarrow ver.p$
    removeVertex($ver$, removeList)
for all $x_j \mapsto b \in$ removeList:
    if not exclude($x_j$, $b$): return false
return true
```

be removed iteratively climbs the tree. This is done to exhaustion, then the appropriate vertex is removed by calling *removeVertex*.

Once all vertices in the list have been processed, removeList contains a list of domain removals. These are performed using the *exclude* procedure (which is assumed to return false if there is a domain wipe out, or a universal variable is pruned).

4.4. A general schema for enforcing WQGAC

This section describes the proposed WQGAC-Schema algorithm, derived from GAC-Schema [**13**], a successful framework for GAC by Bessière and Régin. In this section most attention will be given to the differences between WQGAC-Schema and GAC-Schema. On constraints with no universal variables, the behaviour of WQGAC-Schema is identical to GAC-Schema.

The key feature of GAC-Schema is multidirectionality, defined below.

DEFINITION 4.4.1. Multidirectionality

(1) The algorithm never looks for a support for a partial assignment p on a constraint C_k when a tuple supporting p has already been found, and
(2) it never checks whether a tuple is a support for a value when it has already been checked for another value [**14**].

Algorithm 15 procedure establishWQGAC

```
**procedure** establishWQGAC(): Boolean
$S_C = \emptyset, S = \emptyset, last_C = \emptyset$
**for** each variable $x_i$:
    **for** each value $a \in D_i$:
        **if not** findSupport($x_i$, $\{\langle x_i, a\rangle\}$):
            **if not** exclude($x_i$, $a$): **return** false
**return** true
```

The main change to GAC-Schema is to replace the notion of support to match the definition of WQGAC: that a value of some variable must be supported for all sequences of values of inner universal variables. The modified data structures S_C, S and $last_C$ are described below.

- $S_C(p)$ contains tuples that have been found to satisfy C and which include the partial assignment p. Each tuple supports n partial assignments, so when a tuple is found, it is added to all n relevant sets in S_C. The current support τ for p is included, and is removed when it is invalidated. Domain removals may invalidate other tuples $\lambda \neq \tau$ contained in S_C, but λ may not be removed immediately, so when searching for a new current support for p, $S_C(p)$ may contain invalid tuples.
- $S(\tau)$ contains the set of partial assignments for which τ is the current support.
- $last_C(p)$ is the last tuple returned by *seekNextSupport* as a support for the partial assignment p; nil otherwise. This is used to allow *seekNextSupport* to continue searching at the point where it left off in the lexicographic ordering of tuples.

Point (1) of multidirectionality is taken into account with the S_C data structure. The *seekInferable-Support* procedure (algorithm 18) ensures that a new support is not sought if one is already stored in S_C. Point (2) is dealt with by the individual constraint representations described in section 4.4.1.

Initialization. To initialize the above data structures, the required supports must be found for all pairs $\langle x_i, a\rangle$, and recorded appropriately (or the value a must be removed from D_i). This is achieved with *establishWQGAC* (algorithm 15), which calls *findSupport* for each pair $\langle x_i, a\rangle$ (algorithm 16). If *findSupport* cannot find all supports for $\langle x_i, a\rangle$, *establishWQGAC* calls *exclude*(x_i, a), which returns false if the removal falsifies the QCSP (possibly by domain wipeout

Algorithm 16 procedure findSupport

```
procedure findSupport($x_i$: variable, $p$: partial assignment): Boolean
if $i = r$: {base case: if we have reached the last variable}
    if $S_C(p) \neq \emptyset$:
        return true {already supported}
    $\tau$=seekNextSupport($p$, $nil$)
    if $\tau = nil$: return false
    $last_C(p) = \tau$
    $Q$=findSupportedPA($\tau$)
    for $q$ in $Q$:
        if $S_C(q) = nil$:
            add $q$ to $S(\tau)$ {i.e. $\tau$ is the first support}
        add $\tau$ to $S_C(q)$
    return true
else: {recursive case}
    if $\forall(x_{i+1})$:
        for value $v \in D_{i+1}$:
            $p = p \cup \{\langle x_{i+1}, v\rangle\}$ {add $x_{i+1} = v$ to the partial assignment}
            if not findSupport($x_{i+1}$, $p$): return false
        return true
    else:
        return findSupport($x_{i+1}$, $p$)
```

of an existential or pruning a universal). The Boolean returned by *establishWQGAC* represents whether the constraint is WQGAC at the point of return.

The procedure *findSupport* is recursive. The first parameter is a variable x_i, which is incremented to x_{i+1} for the recursive calls. A partial assignment p is recursively built up, for all combinations of values of inner ($x_{j>i}$) universal variables. For universal variables, a recursive call is made for each value in the current domain, hence all possible sequences are built.

When the last variable in the constraint is reached, if p is not already supported a support τ is sought. If found, the *findSupportedPA*(τ) procedure is called which returns the set Q of all n partial assignments that τ supports. For each variable x_i, τ supports the partial assignment q including x_i and all inner universal values: $q = \{\langle x_i, \tau_i\rangle\} \cup \bigcup_{j>i \wedge \forall(x_j)} \{\langle x_j, \tau_j\rangle\}$. τ is then added to $S_C(q)$. If τ is the first support for q, q is added to $S(\tau)$.

Propagation. After initialization, removing an element from a domain may result in one or more supports becoming invalid. The procedure *WQGAC-propagate*(x_i, a) (algorithm 17) replaces the invalid supports if possible, otherwise prunes the unsupported values. The procedure

Algorithm 17 procedure WQGAC-propagate

```
procedure WQGAC-propagate($x_i$: variable, $a$: value): Boolean
$P$ =generatePA($x_i$, $a$)
for each partial assignment $p \in P$:
    for each tuple $\tau \in S_C(p)$
        $\chi$ =findSupportedPA($\tau$)
        for each partial assignment $q \in \chi$: remove $\tau$ from $S_C(q)$
        for each partial assignment $u \in S(\tau)$:
            if $u$ valid given current domains:
                $\sigma$ =seekInferableSupport($u$)
                if $\sigma \neq nil$:
                    add $u$ in $S(\sigma)$
                else:
                    $\sigma$=seekNextSupport($u$, $last_C(u)$)
                    if $\sigma \neq nil$:
                        add $u$ in $S(\sigma)$
                        $last_C(u) = \sigma$
                        $\alpha$ =findSupportedPA($\sigma$)
                        for each partial assignment $s \in \alpha$:
                            add $\sigma$ in $S_C(s)$
                    else:
                        $(x_j, b)$=outermost literal of $r$
                        if not exclude($x_j, b$): return false
return true
```

generatePA generates the set of partial assignments containing (x_i, a) for all possible sequences of inner universal assignments. More precisely, where $U_i = \{x_{k_j} \in \mathcal{X}_k | j > i,\ Q_{k_j} = \forall\}$ (U_i is the set of universals quantified after x_{k_i}), generatePA returns the set below.

$$PA_i = \Big\{\{x_{k_i} \mapsto \tau_i,\ x_{k_j} \mapsto v_j,\ x_{k_l} \mapsto v_l, \ldots\} | U_i = \{x_{k_j}, x_{k_l}, \ldots\},\ v_j \in D^0_{k_j},\ v_l \in D^0_{k_l}, \ldots\Big\}$$

Note that the original domains D^0 are used. This means the algorithm still behaves correctly when universal values have been pruned.

PA_i is the set whose supports are invalidated by the removal. For each of these partial assignments p, $S_C(p)$ contains all the tuples τ containing p which were previously supporting something and are now invalid. τ is removed from S_C, then all the partial assignments u which are currently supported by τ are processed: if u is still valid, a new support is required. $S_C(u)$ may contain

Algorithm 18 procedure seekInferableSupport

```
**procedure** seekInferableSupport($p$: partial assignment): tuple
**for** $\sigma \in S_C(p)$:
    **if** $\exists k\ \sigma_k \notin D_k$: remove $\sigma$ from $S_C(p)$
    **else**: **return** $\sigma$
**return** $nil$
```

another valid support: this would be discovered by *seekInferableSupport*. If not, *seekNextSupport* is called to find the next support in lexicographic order. If one is found, it is added to the relevant sets in S_C and S, and $last_C$ is updated. If no support for u is found, the appropriate value is pruned.

The procedure *seekInferableSupport* (algorithm 18) searches the set $S_C(p)$ to find a valid support for p which was found earlier to support some other partial assignment. It clears invalid tuples from the set as they are found. This satisfies part 1 of the the definition of multidirectionality (definition 4.4.1).

These procedures make up the general WQGAC-Schema. The procedure *seekNextSupport*, called in *findSupport* and *WQGAC-propagate*, is instantiated differently to deal with different types of constraint.

4.4.1. How to deal with specific constraint representations.

WQGAC-Schema can be instantiated to deal with predicates (arbitrary expressions) and with lists of allowed tuples.

4.4.1.1. *Predicates.* The constraint is defined by an arbitrary expression. The user provides a black box function $f_{C_k}(\tau)$, which returns *true* iff the tuple τ satisfies the constraint, *false* otherwise. This is used in the *seekNextSupport* procedure shown in algorithm 19. *seekNextSupport*(p: partial assignment, τ: tuple) returns the smallest (in lexicographic order) tuple greater than τ which is checked to be allowed by C_k. The only change from the GAC-Schema version in [**13**] is that the variable y and value b have been replaced everywhere with p.

The procedure *seekCandidateTuple* (algorithm 20) uses the $last_C$ data structure to jump forward, skipping tuples which have already been checked and found not to satisfy C_k. This satisfies part 2 of multidirectionality (definition 4.4.1). Together with the other part of the definition, a

Algorithm 19 procedure seekNextSupport for the predicate instantiation

```
**procedure** seekNextSupportPredicate($p$: partial assignment, $\tau$: tuple): tuple
**if** $\tau \neq nil$:
    $(\tau, dummy) \leftarrow$ nextTuple($p,\ \tau,\ |\tau|$)
**else**:
    {generate the lex-least tuple containing $p$, valid w.r.t. current domains}
    $\tau \leftarrow$ smallest valid tuple containing $p$
$\tau \leftarrow$ seekCandidateTuple($p,\ \tau,\ 1$)
**while** $\tau \neq nil$:
    **if** $f_{C_k}(\tau)$:
        **return** $\tau$
    **else**:
        $(\tau, k) \leftarrow$ nextTuple($p,\ \tau,\ |\mathcal{X}_k|$)
        $\tau \leftarrow$ seekCandidateTuple($p,\ \tau,\ k$)
**return** $nil$
```

tuple will not be checked (i.e. passed to f_{C_k} for evaluation) more than once, therefore limiting the total number of checks to d^r.

In GAC-Schema, a *candidate* is a valid tuple which has not been checked. Therefore, we want to find the smallest (in lexicographic order) candidate. In WQGAC-Schema+predicate this is approximated: the procedure *seekCandidateTuple*(p, τ, i) returns a tuple which is less than or equal to the smallest candidate. The procedure *seekCandidateTuple*(p, τ, i) returns a tuple greater than or equal to τ, assuming τ is valid and includes p and the prefix $\tau_{1..i-1}$ is a possible prefix for a candidate.

In GAC-Schema+predicate, for each iteration of the main while loop in *seekCandidateTuple*, one tuple γ is retrieved from $last_C$ and used to jump forward, with the intuition that the algorithm has already checked all the relevant tuples $\tau \leq_{lex} \gamma$. Here, a lower bound on γ is used. For all the partial assignments p which assign x_{k_i} to τ_i, and x_{k_i} is the outermost variable assigned by p, the tuple is retrieved from $last_C$. The lower bound λ is the least of these tuples (line 8). The reason is that the algorithm must have checked all tuples containing $x_{k_i} \mapsto \tau_i$ which are less than or equal to λ. Apart from this change, the algorithm is very similar to the original presented by Bessière and Régin [**13**].

If the difference between λ and τ occurs before k, *nextTuple*$(p,\ \tau,\ k)$ is called which ensures that the prefix $\tau_{1..k}$ is increased while respecting p. Otherwise, *nextTuple*$(p,\ \tau,\ |\tau|)$ is called to get

Algorithm 20 procedure seekCandidateTuple

```
(1)  procedure seekCandidateTuple($p$: partial assignment, $\tau$: tuple, $i$: index): (tuple, index)
(2)  while $\tau \neq nil$ and $i \leq |\tau|$:
(3)      $U_i \leftarrow \{x_{k_j} \in \mathcal{X}_k | j > i,\ Q_{k_j} = \forall\}$ {$U_i$ is the set of universals quantified after $x_{k_i}$}
(4)      $UA_i \leftarrow \left\{x_{k_j} \mapsto v_j,\ x_{k_l} \mapsto v_l, \ldots | U_i = \{x_{k_j}, x_{k_l}, \ldots\},\ v_j \in D^0_{k_j},\ v_l \in D^0_{k_l}, \ldots\right\}$
(5)      $PA_i \leftarrow \{\{x_{k_i} \mapsto \tau_i\} \cup A | A \in UA_i\}$
(6)      {$PA_i$ is the set of all partial assignments which assign $x_{k_i}$ to $\tau_i$, }
(7)      {and assign each variable in $U_i$ some value from its original domain $D^0_k$.}
(8)      $\lambda \leftarrow$ minlex($\{last_C(q) | q \in PA_i\}$)
(9)      {$\lambda$ is minimum under lexicographic order, where $nil$ is least}
(10)     if $\lambda \neq nil$:
(11)         $split \leftarrow 1$
(12)         while $\tau_{split} = \lambda_{split}$: $split \leftarrow split + 1$
(13)         if $split > |\tau|$ or $\tau_{split} < \lambda_{split}$:
(14)             if $split < i$:
(15)                 $(\tau, i') \leftarrow$ nextTuple($p$, $\tau$, $i$)
(16)                 $i \leftarrow i' - 1$
(17)             else:
(18)                 $(\tau, i') \leftarrow$ nextTuple($p$, $\tau$, $|\tau|$)
(19)                 $i \leftarrow \min(i,\ i' - 1)$
(20)     $i \leftarrow i + 1$
(21) return $\tau$
```

the valid tuple following λ. i decreases to the smallest index where the value of τ has changed. Bessière and Régin give a sketch proof which shows that *seekCandidateTuple* cannot miss any candidates when jumping forwards with the calls to *nextTuple* [**13**]. This applies here unchanged, apart from the substitution of p for the variable and value y, b.

The procedure *nextTuple*(p, τ, i) finds the next valid tuple $\tau' >_{lex} \tau$ where τ' includes p and has the property $\tau'_{1..i} \neq \tau_{1..i}$. The returned i' is the position of the first difference between τ' and τ: $\tau'_{1..i'-1} = \tau_{1..i'-1}$ and $\tau'_{i'} \neq \tau_{i'}$.

4.4.1.2. *Positive constraints.* Here the set of satisfying tuples (C^S_k) is given explicitly. Again, this is generalized from the algorithm given by Bessière and Régin [**13**], with the data structure from Mohr and Masini [**78**]. In practice this will only be practical for tight constraints, but it can sometimes outperform the predicate instantiation despite being much less sophisticated. The set C_S is sorted by partial assignment, to match the requirements of supporting a value. For each pair $\langle x_i, a\rangle$, the tuples matching $\langle x_i, a\rangle$ are divided into each possible sequence of inner universal

Algorithm 21 procedure seekNextSupport for the positive instantiation

```
**procedure** seekNextSupportPositive($p$: partial assignment, $dummy$): tuple
$i \leftarrow CS(p)$ {current support}
$l \leftarrow$ tupleLists($p$) {retrieve the appropriate list to search}
**while** $i \leq$ length($l$):
    $\sigma \leftarrow l(i)$
    **if** $\forall j : \sigma_j \in D_{k_j}$:
        $CS(p) \leftarrow i$
        **return** $\sigma$
    $i \leftarrow i + 1$
**return** $nil$ {no valid tuple found}
```

assignments. (This does not increase the asymptotic space consumption because each tuple of length n has n references to it, both in GAC-Schema+positive and here.) The *seekNextSupport* procedure is given in algorithm 21. This instantiation is referred to as WQGAC-Schema+positive.

4.4.1.3. *Next-Difference lists.* Next-Difference lists are very similar to the positive instantiation. The change is that the algorithm is able to jump forward in the list of satisfying tuples, by jumping tuples which are known to be invalid. The existing work on this is reviewed in chapter 2 section 2.1.2.2. The data structure and algorithm are novel to the best of my knowledge. I invented them for use in CSP [**51**] and then modified the algorithm slightly to work for QCSP.

Each item in the list is a record containing the tuple t, and a precomputed array of list indices called ND. $ND(x_{k_i})$ is the index of the next tuple which contains a *different* value for variable x_{k_i}. Therefore, if the current tuple contains value a for variable x_{k_i}, then $ND(x_{k_i})$ has the index of the next tuple to contain $b \neq a$ at position i. If value $x_{k_i} \mapsto a$ has been pruned, it is sound to skip to the next tuple not containing $x_{k_i} \mapsto a$.

To illustrate, consider Figure 18, which shows the Next-Difference list corresponding to a ternary constraint with scope $\langle x, y, z \rangle$, and four allowed tuples. If value 0 is pruned from variable x, when searching for a support for $z \mapsto 0$, it is possible to jump from tuple 1 to tuple 3 in one step.

The procedure for searching this data structure is given in algorithm 22. The new algorithm can be used with one list containing all tuples, or with lists containing supporting tuples for each variable and value (x_{k_i}, a). The flag OneList, used on line 3 of algorithm 22 determines if one

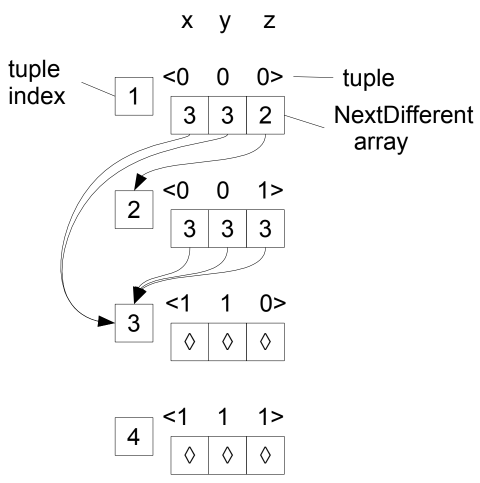


FIGURE 18. Next-Difference list for tuples $\{\langle 0,0,0\rangle, \langle 0,0,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\}$

Algorithm 22 procedure seekNextSupport for Next-Difference lists

```
(1)  procedure seekNextSupportND($p$: partial assignment, $dummy$): tuple
(2)  $i \leftarrow CS(p)$ {current support}
(3)  if OneList: $l$ is the global tuple list
(4)  else: $l \leftarrow$ tupleLists($p$)
(5)  for $pass \in \{1, 2\}$ {search from $CS(p)$ to end (pass 1) then beginning to $CS(p)$ (pass 2)}
(6)      while $i \leq$ length($l$) and $(pass = 2) \Rightarrow i < CS(p)$:
(7)          $t \leftarrow l(i).t$ {retrieve the current tuple}
(8)          $j \leftarrow 1$ {$j$ indexes into a tuple}
(9)          while $j \leq r$ and $t_j \in D_{k_j}$ and $(x_{k_j} \mapsto a) \in p \Rightarrow (t_j = a)$:
(10)             $j \leftarrow j + 1$ {increment $j$ until $t_j$ is not in the domain,}
(11)                 {or $t_j$ does not match the partial assignment}
(12)         if $j = r + 1$:
(13)             $CS(p) \leftarrow i$
(14)             return $t$
(15)         else:
(16)             $i \leftarrow l(i).ND(j)$ {jump to the next tuple with value $j$ different}
(17)     $i \leftarrow 1$ {restart for pass 2}
(18) return $nil$ {no tuple found}
```

list is used. The lists are sorted in lexicographic order, with the leftmost value in the tuple as the most significant. Therefore it is likely that finding the leftmost invalid value would allow to jump forward the furthest, so we iterate from the left when checking the validity of the tuple. Towards the end of the list, *ND*(j) is likely to contain $\diamond$ indicating that there is no subsequent tuple with a different value for $t(j)$. $\diamond$ is considered greater than length(l).

There are three differences between Next-Difference and the naive positive algorithm. Firstly the CS data structure is not backtracked when the search backtracks, therefore *seekNextSupportND* must wrap around when it reaches the end of the list. This avoids backtracking an integer for every partial assignment. Secondly *seekNextSupportND* checks if a tuple matches the partial assignment, as well as checking if it is valid. This allows the use of a single list for all partial assignments. Thirdly *seekNextSupportND* jumps forward on line 16 — if this line were replaced with $i \leftarrow i + 1$, and the algorithm adapted to make one pass, it would be the same as *seekNextSupportPositive*.

In the degenerate case where the first tuple examined is valid there is no significant extra overhead. *seekNextSupportND* (with a list for each partial assignment) makes two passes of the tuple list, therefore it could perform worse than *seekNextSupportPositive*, but it has potential to perform much better. This instantiation with one list is referred to as WQGAC-Schema+NDOnelist and with multiple lists as WQGAC-Schema+NDlists.

4.4.1.4. *Other instantiations.* If the list of disallowed tuples is supplied, Bessière and Régin give an efficient method based on hashing which uses the predicate instantiation and can be used without any modification [**13**]. They also instantiate GAC-Schema to process a conjunction of constraints (a subproblem) [**14**], which gives the same capabilities as the predicate instantiation but with greater efficiency — the subproblem instantiation uses CSP propagation algorithms to prune the inner search space when searching for a support. Adapting this remains for future work.

4.4.2. Implementation notes. It is vital that the core data structures S_C, S and $last_C$ are implemented efficiently, along with the elt data structure in the positive instantiation.

S_C and S contain lists (of tuples and partial assignments respectively) which must be backtracked. It must be efficient to remove and restore elements from the list, in any position. A doubly-linked list is used so that objects can be removed and restored in constant time by doing two pointer operations, with the dancing links method [**66**], the same method used for the lists in *SQGAC-propagate*. For backtracking, the index into S_C or S is kept along with the list element, and when backtracking the elements are restored or removed in reverse order to the original operations. For S, the list elements cannot be restored using the dancing links method because the list pointers can be overwritten before backtracking, but the order of the lists in S is not important so elements can be inserted at the head.

S_C and $last_C$ map a partial assignment to some other object. An obvious choice would be a hash table. Hash tables have approximately constant time access, but their performance is not very satisfactory in practice in this case. This appears to be caused by the overhead of executing the hash function. The trick used here is to attach a unique number to each partial assignment object. S_C and $last_C$ (and the elt structure in the positive instantiation) are represented as arrays, indexed by the number. An informal experiment was performed with Noughts and Crosses model A (described in section 4.6.2.1) with all constraints processed by WQGAC-Schema+predicate. The experiment compares a hash table implementation of S_C, S and $last_C$ with a hash table for S and the proposed arrays for S_C and $last_C$. The experiment showed a 24% reduction in run time for the entire search process (which is dominated by the run time of WQGAC-Schema).

S maps tuples to lists of partial assignments, and is implemented with a hash table. The trick of assigning unique numbers to tuples would not be suitable here because the array (size $O(d^r)$) would be too large in some cases. It is not clear if the implementation of S could be improved.

4.5. Space and time complexities

For SQGAC, the multiple strategy tree occupies $O(d^r)$ space, and when embedded in a search procedure, $O(d^r)$ nodes can be removed from the tree when descending one branch of the search tree. Because of the lists denoted $list(x_i \mapsto a)$ detecting values which have lost support can be done in constant time. Also, the lists make it trivial to find the vertices which need to be removed when *SQGAC-propagate* is called, so the dominant cost is removing vertices. Therefore the time complexity for one branch of the search tree is $O(d^r)$.

The space and time properties of WQGAC-Schema+predicate are compared against GAC-Schema+predicate. Let C_k have arity r and contain variables with domain size d. GAC-Schema requires $O(r^2d)$ space, with the greatest cost being the S_C data structure. WQGAC-Schema maintains more supports: for a constraint with $u \leq r$ universal variables, WQGAC-Schema maintains support for $O(rd^{u+1})$ partial assignments compared to rd values for GAC-Schema. Therefore there are potentially $O(rd^{u+1})$ tuples stored. Since each tuple supports r partial assignments, S_C contains r references to it, giving a space requirement of $O(r^2d^{u+1})$. However, there can be no more than d^r tuples, so if $d^r < rd^{u+1}$ then the space requirement is $O(rd^r)$ for S_C. In the other instantiations where there are lists of tuples, these occupy $O(rd^r)$ space, except the NDlists instantiation which occupies $O(r^2d^r)$.

To obtain an upper bound for the time, consider some tuple τ. Because of multidirectionality, τ will only be processed once. If $f_C(\tau)$ =*false*, the cost of processing τ is the same as the cost of running f_C which I assume will be $O(r)$. If $f_C(\tau)$ =*true* then a reference to τ is added to r sets in S_C and to one set in S and one entry in $last_C$. τ can be processed up to r times by the *seekInferableSupport* procedure, which verifies τ against current domains, taking r time. When τ is invalidated by a domain removal, it is removed from r sets in S_C (taking constant time per removal). For d^r tuples, this gives an upper bound of $O(r^2d^r)$. However, since falsified tuples are removed from S_C by propagate, *seekInferableSupport* is likely to find the first tuple in the list is a valid one, so the cost is close to rd^r.

The other cost of enforcing multidirectionality is in *seekNextSupport* and *seekCandidateTuple*. For each variable x_i and each value a, the space of assignments to other variables (size d^{r-1}) is divided up among the partial assignments supporting (x_i, a), but the whole space is covered. I assume no jumping forward is possible. Finding the lexicographically next tuple takes constant time on average[1], so the total time taken is $O(rd \times d^{r-1}) = O(rd^r)$.

The same line of reasoning can be followed for GAC-Schema+predicate. Bessière and Régin claim an upper bound of $O(d^r)$, presumably assuming some $O(r)$ operations to be constant time for any reasonable r.

In summary, *SQGAC-propagate* takes $O(d^r)$ time and space. WQGAC-Schema takes $O(rd^r)$ time and for positive and NDOnelist instantiations, $O(rd^r)$ space to store the tuples. For the

[1]Average symbol changes required to increment a tuple: $\alpha = \sum_{i=1}^{r}(d-1)i/d^i$ (sum of number of symbol changes times their probability). As $r \rightarrow \infty$, $\alpha \rightarrow 1 + 1/(d-1)$. As $d \rightarrow \infty$, $\alpha \rightarrow 1$.

NDlists instantiation, $O(r^2d^r)$ space is required for the Next-Difference lists. For the predicate instantiation the space requirement is lowest: $O(r^2d^{u+1})$.

4.6. Empirical evaluation

First I use random QCSP instances to investigate whether the proposed algorithms are competitive with QBF and binary ($r = 2$) QCSP solvers. Then I model two games, Noughts and Crosses (tic-tac-toe) and Connect 4, which have a variety of constraints

4.6.1. Random problems. In this section I compare the new algorithms introduced in this chapter against QBF solvers and binary QCSP solvers. All these experiments use random QCSPs so I will first describe the model I used to generate these QCSP instances.

The generator takes a tuple of parameters $\langle \mathcal{Q}, d, e, p_2, r\rangle$ where the quantifier sequence $\mathcal{Q}$ covers all variables, so that $|\mathcal{Q}| = n$. d is the size of each domain, e is the number of constraints, r is the arity of the constraints, and p_2 is the proportion of satisfying tuples a constraint has. The number of satisfying tuples is p_2d^r, and these are chosen with uniform probability for each constraint independently. The meaning of each parameter is similar to those for model B random binary CSPs [**2**]. Constraint scopes are chosen with uniform probability from among the variables. There is no check for connectedness of the constraint hypergraph, or for consistency. The generation model is likely to be susceptible to flaws. However, with the parameters I use below, a phase transition emerges therefore flaws are not apparent.

4.6.1.1. *Comparing SQGAC and WQGAC with QBF.*

Hypothesis. When the constraint arity is high, SQGAC and WQGAC are stronger in practice than QBF techniques.

Method. I generate random QCSP problems with parameters $d = 2$, arity $r \in \{8, 9, 10\}$, constraint tightness $p_2 \in \{0.2, 0.5, 0.8\}$. If $p_2 = 0.2$, $n = 30$ otherwise $n = 25$. If $p_2 = 0.2$ then $e \in \{1 \dots 20\}$ otherwise $e \in \{1 \dots 30\}$. In the quantifier sequence, every 6th quantifier is universal and all others are existential (starting $\exists x_1, x_2, x_3, x_4, x_5 \forall x_6 \exists x_7, \dots$). This is chosen to be similar to some structured instances (e.g. those in section 4.6.2 below) where there are many more existentials and the universals are equally spaced. The instances are quite small because some of the techniques do not scale well. For each parameter set, a suite of 10 QCSP instances were generated. The figures reported below are all sums over the suite.

QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ is encoded to the QBF $\mathcal{P}' = \langle \mathcal{X}, \mathcal{C}', \mathcal{Q} \rangle$ where $\mathcal{C}'$ is the conjunction of clauses representing the constraints. The variables and quantifier sequence is the same in the QBF and the QCSP because $d = 2$. Each constraint $C_k \in \mathcal{C}$ is encoded as a set of clauses representing each non-satisfying tuple $t \notin C_k^S$. Each clause $c \in \mathcal{C}'$ directly forbids one non-satisfying tuple. For all i, if $t_i = 1$, then $\neg x_{k_i}$ is in c, otherwise x_{k_i} is in c.

For solving the QCSPs directly, I used Queso with the *search* algorithm (chapter 3, algorithm 3), without the pure value rule. Therefore the only simplification at each search node is by consistency. The variable and value ordering is static. The following propagation algorithms were used: SQGAC, WQGAC-Schema+predicate, WQGAC-Schema+positive, WQGAC-Schema+NDlists, WQGAC-Schema+NDOnelist. For QBF, I chose a variety of solvers which are freely available, and which are competitive according to recent competitions in the QBF community. In each case I used the most recent publicly available version at the time of writing.

- CSBJ, a DPLL-based solver with conflict and solution backjumping [**47**].
- Quberel, another DPLL-based solver with relevance-bounded conflict learning [**56**].
- Quantor 2.11, a resolution-based solver [**18**].
- Skizzo 0.8.2, a solver which performs both search and resolution on a skolemized form of the QBF [**10**].

In all cases, the CPU times reported are for a Pentium 4 machine running at 3.06GHz with 1GB of RAM. Queso was executed with the Sun Java 1.5 runtime, in server mode and each experiment was executed twice. This gives the Java virtual machine the opportunity to perform optimizations on the program during the first run. Although some attention was paid to efficiency in the implementation of SQGAC and WQGAC-Schema, this was not the main concern and the CPU times could be improved. For SQGAC and WQGAC-Schema, time to set up data structures is included in the reported time. For the QBF solvers, the time to read the file and set up data structures is included.

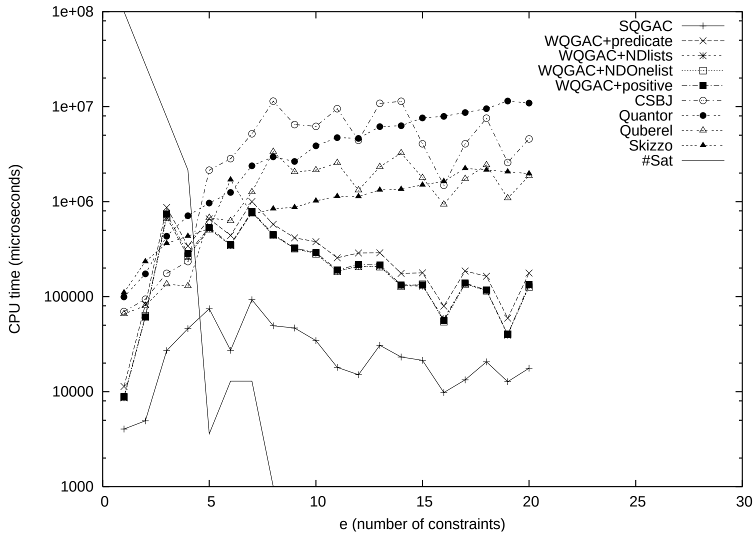


FIGURE 19. Run times for $p_2 = 0.2$, $r = 8$ for various solvers and values of e

Results. For instances with constraint looseness $p_2 = 0.2$, where $n = 30$, figure 19 shows the sums of runtimes for $r = 8$ and $e \in \{1 \ldots 20\}$. Even with these very small problems, the QBF solvers are starting to show poor scaling behaviour as the number of constraints is increased. The satisfiability phase transition is around $e = 5$, and this coincides with the difficulty peak for SQGAC and WQGAC-Schema. To the right of the phase transition, the QBF solvers perform between 10 times and 1000 times worse than SQGAC. Where e is 16, 17 and 19, all instances in the suite are solved without search by SQGAC, but not by WQGAC. The results for $r = 9$ are similar.

The data for $r = 10$ are plotted in figure 20. Again, there is a region where the SQGAC and WQGAC compare very well with the QBF solvers. Where $e > 12$, Skizzo exceeds the 1GB memory limit. Quantor and Skizzo in particular scale badly as the number of constraints is increased. This suggests that they are less able to deal with large numbers of clauses than Quberel. Quberel performs better than CSBJ, probably because Quberel has conflict learning.

Where $p_2 = 0.5$ and $r = 8$ (figure 21) the picture is not so clear. SQGAC still performs best overall with a margin of one to two orders of magnitude compared to the QBF solvers, but the WQGAC algorithms do not perform so well. With $p_2 = 0.5$ and $r = 10$ (figure 22) it is significantly clearer that the QBF solvers are scaling badly as e is increased.

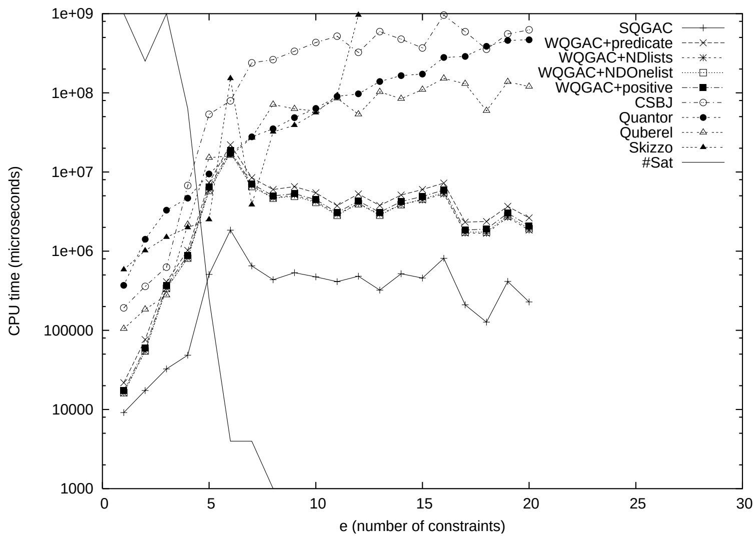


FIGURE 20. Run times for $p_2 = 0.2$, $r = 10$ for various solvers and values of e

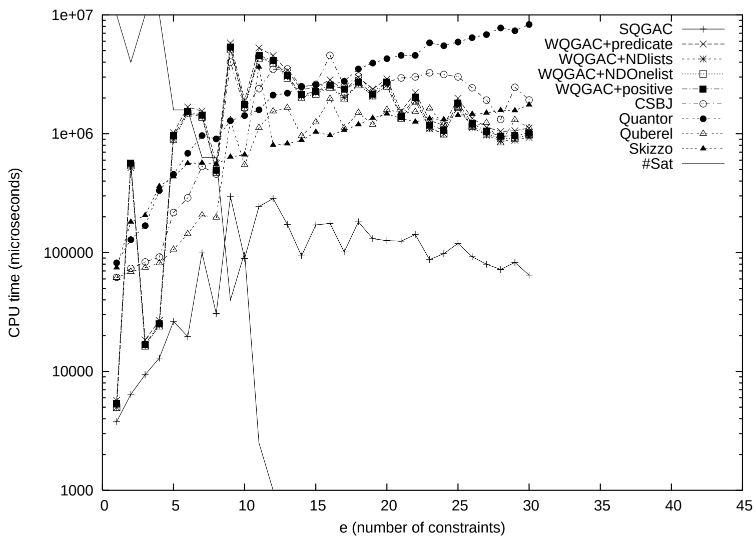


FIGURE 21. Run times for $p_2 = 0.5$, $r = 8$ for various solvers and values of e

When the constraints are loose ($p_2 = 0.8$), the phase transition shifts significantly. Figures 23 and 24 show results which are not favourable for SQGAC and WQGAC. When there are many satisfying tuples, SQGAC and WQGAC are not able to do much pruning, and they are likely to take more time because the tree or list of tuples is larger. This suggests that (if the hypothesis is true) the arity is not high enough to see the advantage of SQGAC or WQGAC.

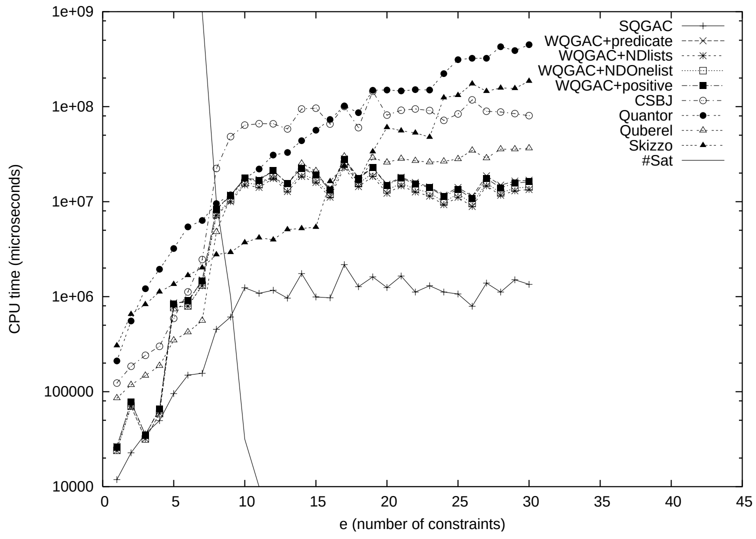


FIGURE 22. Run times for $p_2 = 0.5$, $r = 10$ for various solvers and values of e

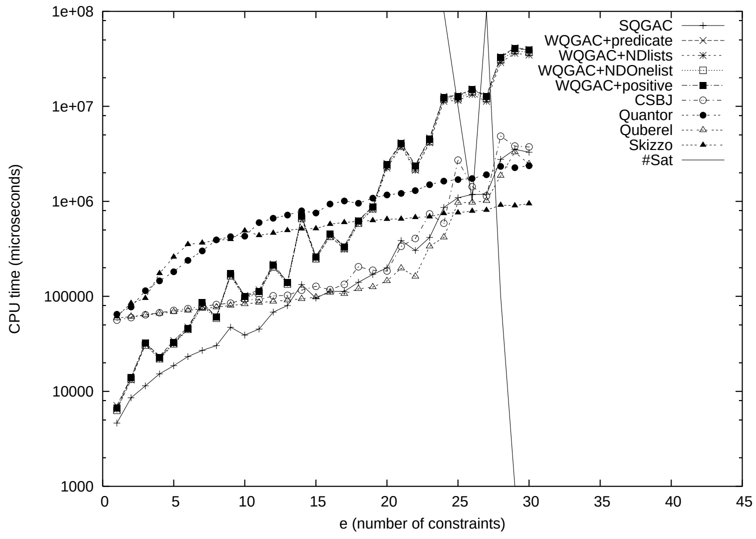


FIGURE 23. Run times for $p_2 = 0.8$, $r = 8$ for various solvers and values of e

At arity 10, for the hardest instances, SQGAC is the best QCSP method and Skizzo is the best QBF solver. Therefore I ran these two methods for $r = 11$ and $r = 12$, and these are plotted together in figure 25. For both $r = 11$ and $r = 12$, SQGAC is significantly faster for the range $e \in \{1 \ldots 30\}$, which supports the hypothesis. I have no explanation for the spikes in the Skizzo $r = 11$ line. They are not due to memory paging, because Skizzo is using less than 1MB of

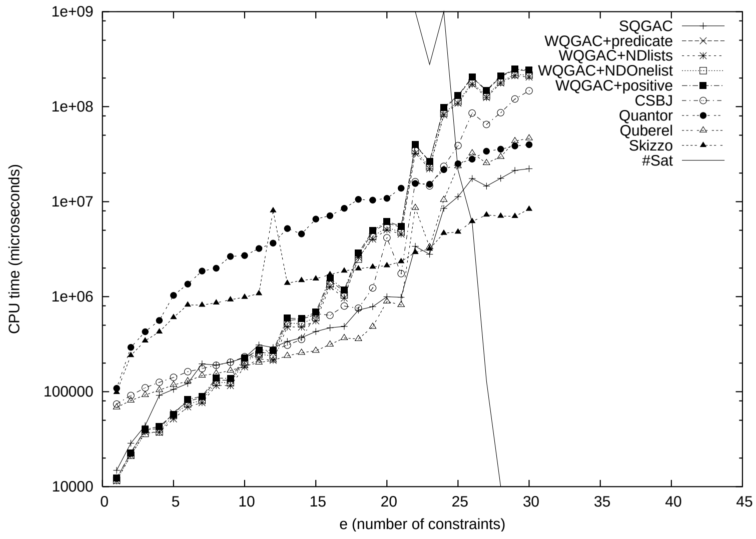


FIGURE 24. Run times for $p_2 = 0.8$, $r = 10$ for various solvers and values of e

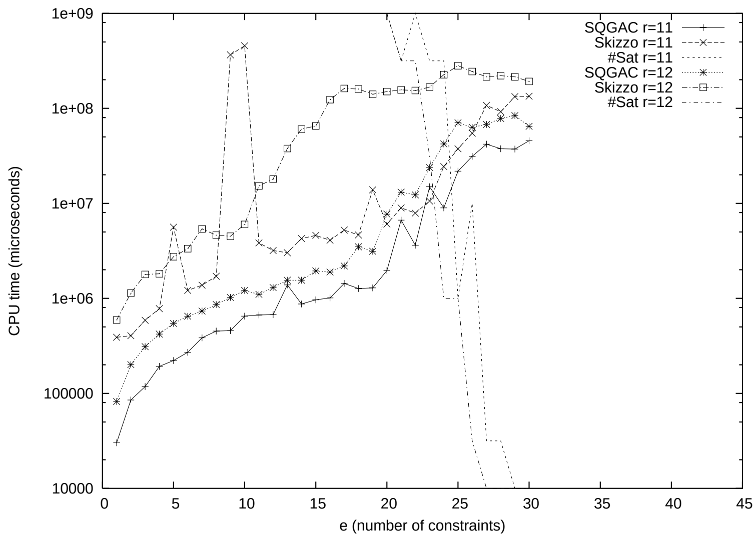


FIGURE 25. Run times for $p_2 = 0.8$, $r = 11$ and $r = 12$ for SQGAC and Skizzo

memory on these instances. Skizzo uses a mixture of resolution rules combined with search, with heuristics to select the algorithm used at each step. These spikes could therefore indicate thrashing or a failure of the heuristics.

To give an overview of the data (300 suites of 10 instances), figure 26 shows the time taken by the QBF solvers as a ratio to the time taken by SQGAC. This is plotted against the time taken

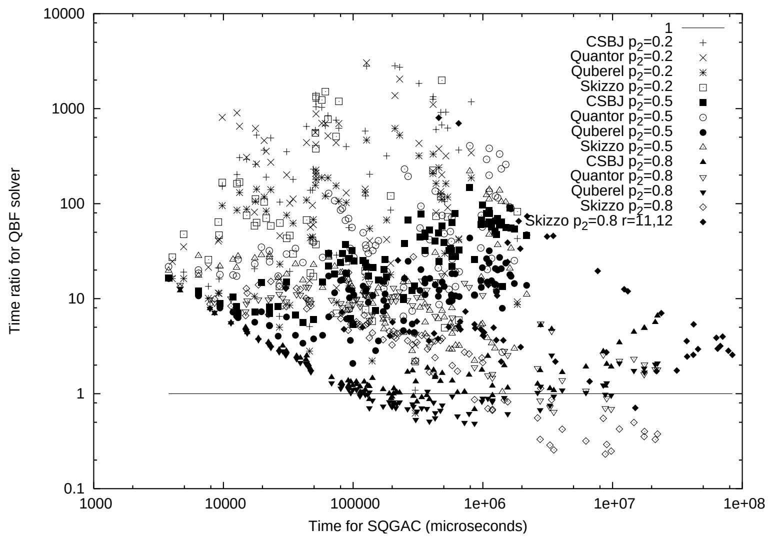


FIGURE 26. Ratio of times for QBF solvers to SQGAC for all suites

by SQGAC. For most suites, this is plotted for all four QBF solvers. However for the extra data where $p_2 = 0.8$ and $r = \{11, 12\}$, the experiment was only run for Skizzo. The graph shows that most suites can be solved faster with SQGAC than with any of the four QBF solvers. Nearly all the suites which cannot are where $p_2 = 0.8$ and $r \leq 10$.

Conclusion. The results partly support the hypothesis, showing the *SQGAC-propagate* algorithm to be faster than the QBF solvers on a range of random QCSPs. This is not the case for WQGAC-Schema though.

4.6.1.2. *Comparing SQGAC and WQGAC with binary QCSP solvers.* The literature about the binary QCSP solvers QCSP-Solve and Blocksolve is reviewed in chapter 2 sections 2.3.4.1 and 2.3.4.3. In this section I compare SQGAC and WQGAC against the solver QCSP-Solve, with the hidden variable encoding into binary QCSP. I also briefly discuss Blocksolve.

Hypothesis. When the constraint arity is high, SQGAC and WQGAC are stronger in practice than binary QCSP techniques.

Method. The random QCSP model described above is used again here. Similarly to the previous experiment, these are solved directly using search and nothing but consistency at each node. The hidden variable encoding (chapter 3 section 3.8.1) is used to encode the problem into binary QCSP, resulting in a QCSP instance with er binary constraints and $10 + e$ variables.

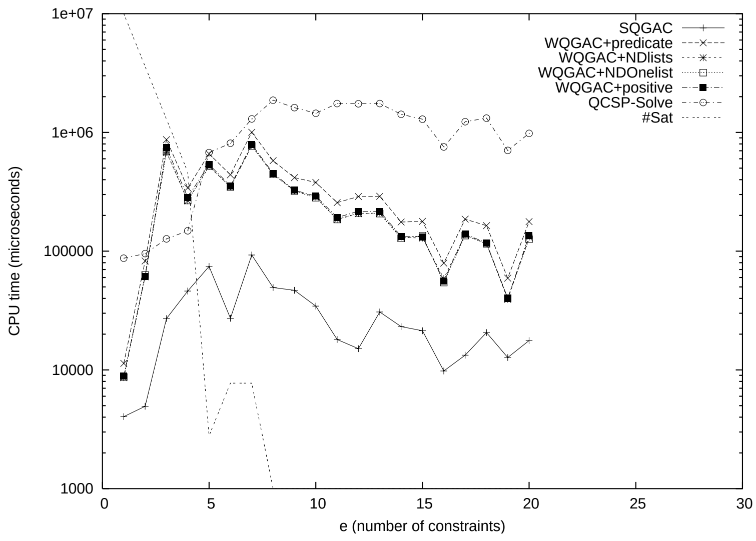


FIGURE 27. Run times for $p_2 = 0.2$, $r = 8$ for various solvers and values of e

For one set of instances I use the same parameters as above ($d = 2$, $p_2 \in \{0.2, 0.5, 0.8\}$, $r \in \{8, 9, 10\}$, if $p_2 = 0.2$ then $e \in \{1 \ldots 20\}$ otherwise $e \in \{1 \ldots 30\}$ and the same quantifier sequence). To generate a second set I use the parameters $d = 6$, $n = 10$ and $r = 5$, with various e and the same quantifier sequence (i.e. $\exists x_1, x_2, x_3, x_4, x_5 \forall x_6 \exists x_7, x_8, x_9, x_{10}$).

The same machine was used as for the previous experiment, and Queso was run in the same way. For the binary QCSP solver, the time to read the file and set up data structures is included in the reported time.

Results. Figures 27, 28, 29, 30, 31 and 32 show the results for $d = 2$, where $p_2 \in \{0.2, 0.5, 0.8\}$, $r \in \{8, 10\}$ for various e. There is no point on any of these graphs where QCSP-Solve is competitive with SQGAC, although it is sometimes competitive with the WQGAC-Schema algorithms. SQGAC is typically an order of magnitude faster, despite the features of QCSP-Solve such as conflict backjumping and solution directed pruning. The results for $r = 9$, $p_2 \in \{0.2, 0.5, 0.8\}$ are very similar, with no suite where QCSP-Solve outperforms SQGAC.

Similarly, for the other set of instances where $d = 6$ and $r = 5$, there is no suite where QCSP-Solve outperforms SQGAC. Figures 33, 34 and 35 show the data for $p_2 = 0.2$, 0.5 and 0.8 respectively, and in each case the range of e was chosen to cover the phase transition. Typically SQGAC is an order of magnitude faster.

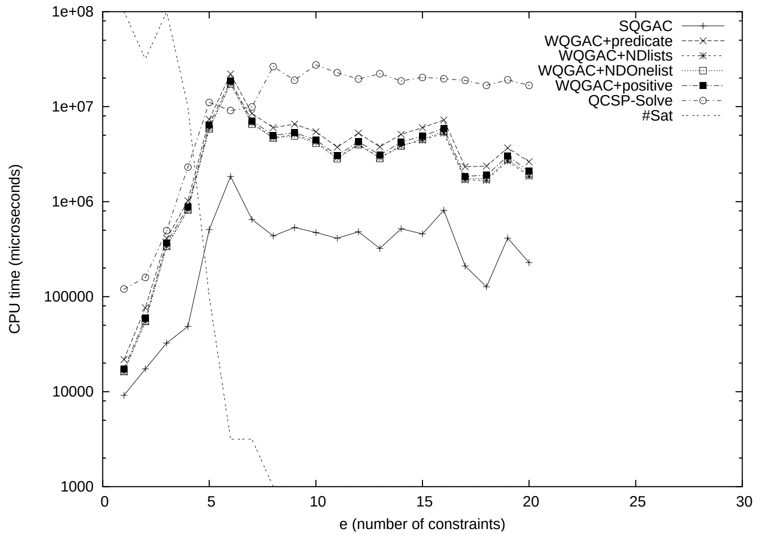


FIGURE 28. Run times for $p_2 = 0.2$, $r = 10$ for various solvers and values of e

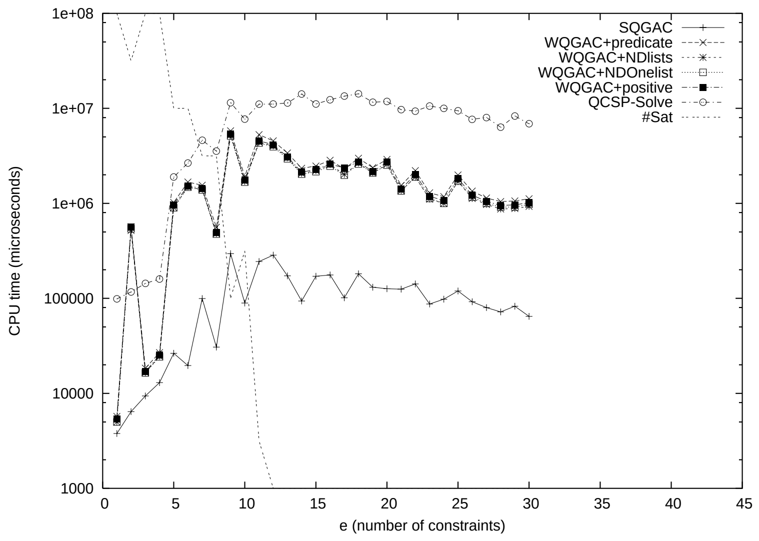


FIGURE 29. Run times for $p_2 = 0.5$, $r = 8$ for various solvers and values of e

Conclusion. For all suites, SQGAC is significantly faster than QCSP-Solve. The various instantiations of WQGAC-Schema are faster than QCSP-Solve for most suites, but there are a few suites (particularly where $d = 2$ and $p_2 = 0.2$) where this is not the case.

Blocksolve. Verger and Bessière's Blocksolve [**97**] is reported to be faster on some random binary instances. The Blocksolve implementation was kindly provided to me by Guillaume Verger.

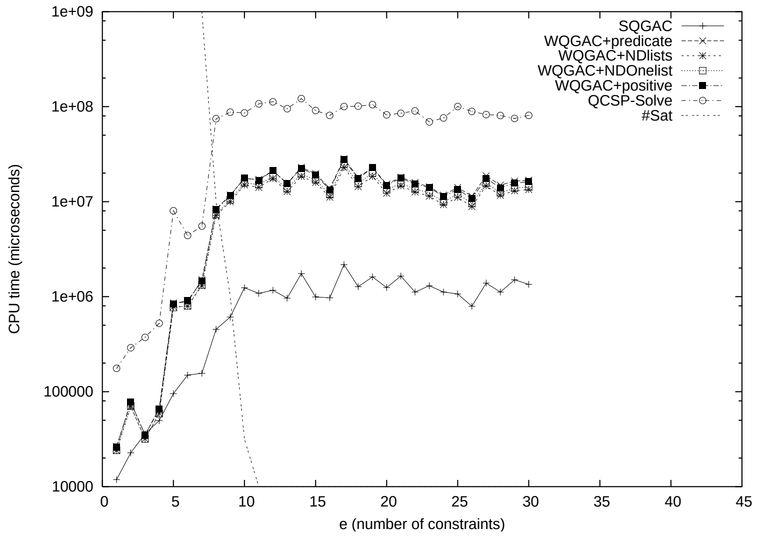


FIGURE 30. Run times for $p_2 = 0.5$, $r = 10$ for various solvers and values of e

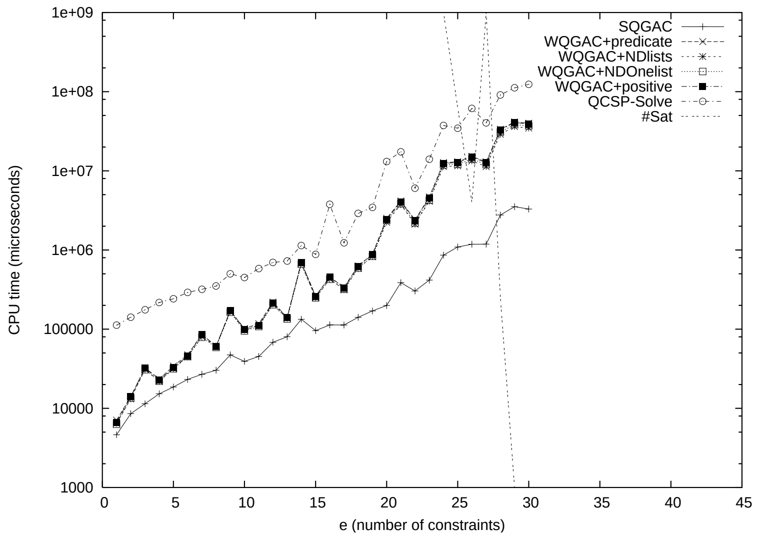


FIGURE 31. Run times for $p_2 = 0.8$, $r = 8$ for various solvers and values of e

Unfortunately it proved to be considerably slower than QCSP-Solve on all the instances I used, and it does not always report the same truth value as Queso and QCSP-Solve. Since Queso and QCSP-Solve report the same value for all instances, I assume that the Blocksolve algorithm or implementation is incorrect[2].

[2]I supplied bug reports to Guillaume Verger several times, and he did provide improved versions of Blocksolve in response, but he was not able to provide a completely bug free version.

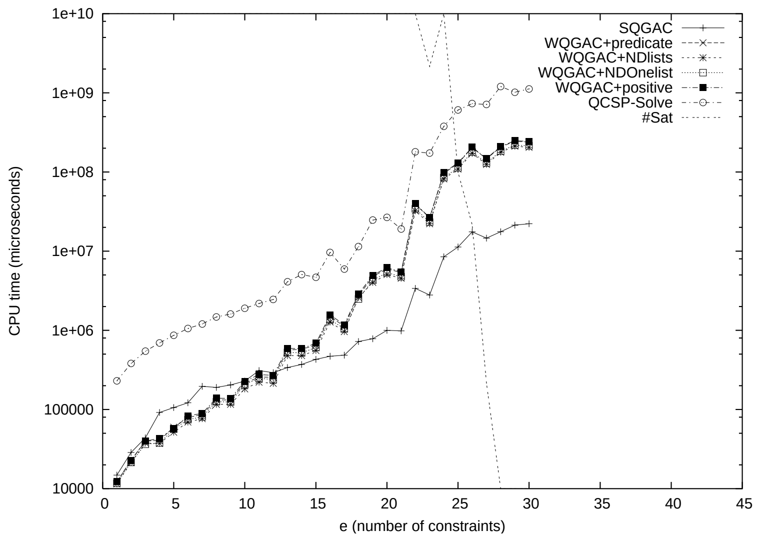


FIGURE 32. Run times for $p_2 = 0.8$, $r = 10$ for various solvers and values of e

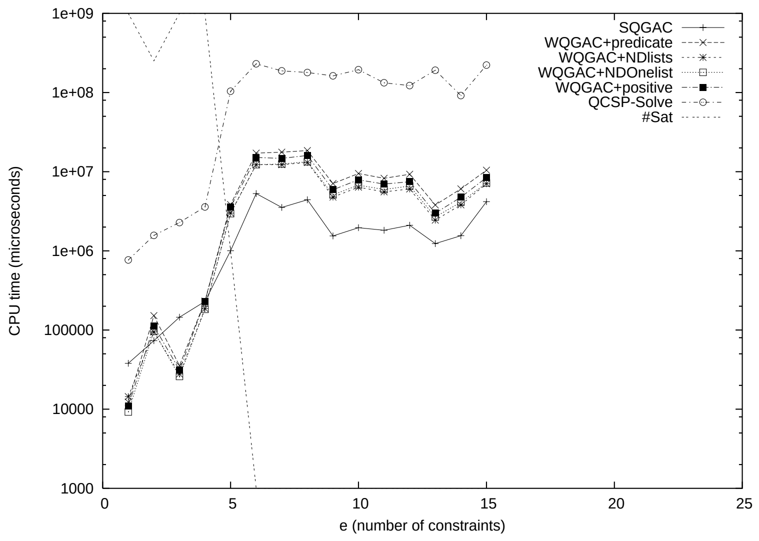


FIGURE 33. Run times for $d = 6$, $p_2 = 0.2$, $r = 5$ for QCSP-Solve

Figure 36 shows CPU times for SQGAC and Blocksolve where $d = 2$, $p_2 = 0.2$ and $r = 8$. QCSP-Solve never takes more than 2s on these suites, whereas Blocksolve takes over 1000s for some suites. Figure 37 shows the same data for arity 10. After solving the first four suites with Blocksolve, the experiment was halted because the suite for $e = 4$ took over 2 hours, and Blocksolve was reporting an unexpected truth value for over 10% of these instances.

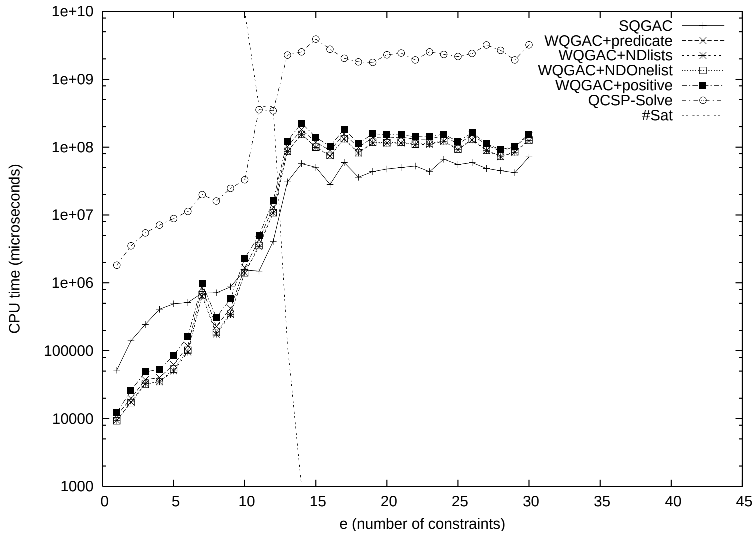


FIGURE 34. Run times for $d = 6$, $p_2 = 0.5$, $r = 5$ for QCSP-Solve

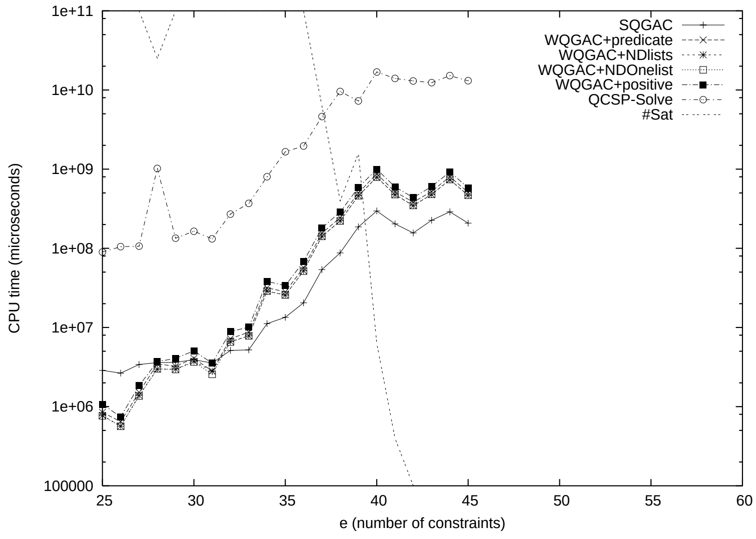


FIGURE 35. Run times for $d = 6$, $p_2 = 0.8$, $r = 5$ for QCSP-Solve

4.6.2. Structured problems. Two games were chosen for their simplicity and ability to display the comparative strengths and weaknesses of the algorithms. The same machine was used as in the previous two experiments, and Queso was run in the same way. When counting search nodes, only the nodes where branching occurs were counted, therefore leaf nodes were excluded.

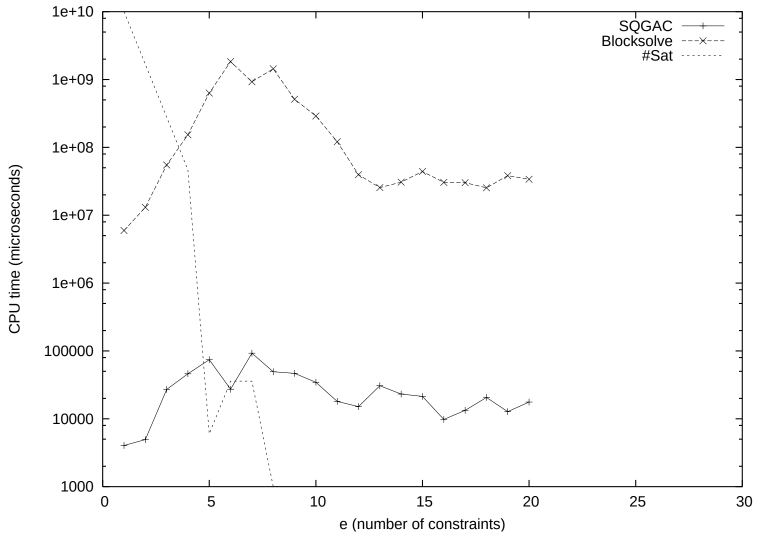


FIGURE 36. Run times for $d = 2$, $p_2 = 0.2$, $r = 8$ for Blocksolve and SQGAC

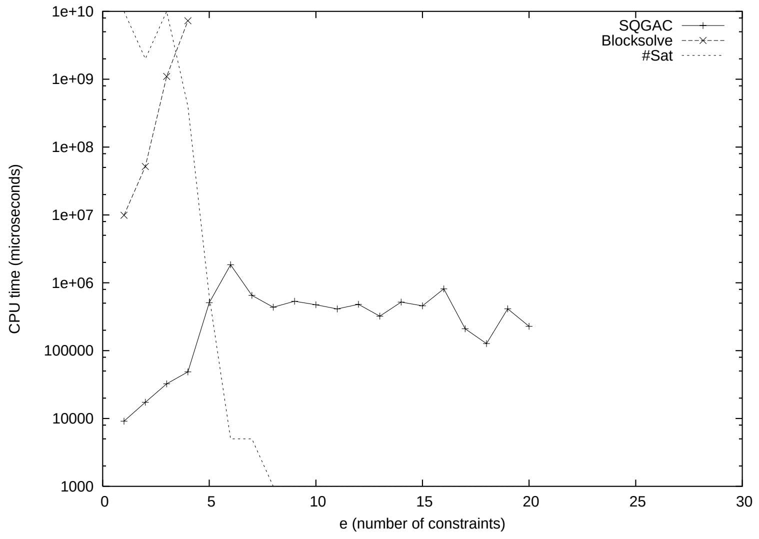


FIGURE 37. Run times for $d = 2$, $p_2 = 0.2$, $r = 10$ for Blocksolve and SQGAC

4.6.2.1. *Noughts and Crosses.* Noughts and Crosses (tic-tac-toe) is played on a 3×3 board. The aim is to make a line of three counters, including diagonal lines. The two players take turns to place a counter on any free slot. The first player is crosses ($\times$), followed by noughts ($\circ$). The aim is to find if crosses can win however noughts defends, therefore the constraints are all satisfied if crosses wins the game, and some constraint is unsatisfied if crosses cheats or noughts wins or draws. The case where noughts cheats is discussed after describing the constraints. To test the benefit of high-arity constraints, I compare two models.

Model A. The game is modelled with 9 move variables m^i each with domain size 9. The moves are represented by $i \in \{1 \dots 9\}$. The state of the board is represented with 9 variables per move, $b^i_{pos} \in \{\times, \circ, nil\}$ where $pos \in \{1 \dots 9\}$. $w^i \in \{\times, \circ, nil\}$ indicates the winner at move i, and Boolean variables xl^i and ol^i indicate if a player has a line at move i. For $\circ$ moves, there is an additional sm^i variable with domain size 9 (the shadow move variable). It is used with the shadow constraint to ensure that the pure value rule is applied to the move variable m^i at the appropriate time. The quantifier sequence is $Q^i m^i \exists sm^i, b^i_{1..9}, w^i, (x \text{ or } o)l^i$ where $Q^i = \exists$ for $\times$ moves (where sm^i is also omitted) and $Q^i = \forall$ for $\circ$ moves. For $\times$ moves, there are 11 constraints:

- $\forall pos$: mapmove1$(b^{i-1}_{pos}, m^i, b^i_{pos})$ which expresses that if $b^{i-1}_{pos} \neq nil$ then $m^i \neq pos$ and $b^i_{pos} = b^{i-1}_{pos}$, and also that $(m^i = pos) \Rightarrow (b^i_{pos} = \times)$. mapmove1 is satisfied iff both these conditions hold.
- findline1$(b^i_{1..9}, xl^i)$ where xl^i is 1 iff player $\times$ has a line on the board $b^i_{1..9}$, otherwise $xl^i = 0$. This constraint is present for $i \in \{5, 7, 9\}$ because these are the moves where $\times$ can win the game. For other values of i, $xl^i = 0$.
- wins1(w^{i-1}, xl^i, w^i) which constrains w^i according to whether a player has already won at move $i - 1$, and whether there is a line at move i ($xl^i = 1$). If $w^{i-1} \neq nil$ then $w^i = w^{i-1}$, else if $xl^i = 1$ then $w^i = \times$, otherwise $w^i = nil$.

For $\circ$ moves, we have:

- $\forall pos$: shadow$(b^{i-1}_{pos}, m^i, sm^i)$ which expresses $(b^{i-1}_{pos} = nil) \Rightarrow ((m^i = pos) \Rightarrow (sm^i = pos))$. In words, if a board position is free, the value representing that position is mapped from m^i to sm^i. This ensures that when a board position is occupied, the corresponding m^i value is pure, and is therefore pruned by the pure value rule.
- $\forall pos$: mapmove2$(b^{i-1}_{pos}, sm^i, b^i_{pos})$ which expresses that if $b^{i-1}_{pos} \neq nil$ then $sm^i \neq pos$ and $b^i_{pos} = b^{i-1}_{pos}$, and also that $(sm^i = pos) \Rightarrow (b^i_{pos} = \circ)$.

- findline2$(b^i_{1..9}, ol^i)$ which is very similar to findline1. This constraint is present for $i \in \{6, 8\}$, and otherwise $ol^i = 0$.
- wins2(w^{i-1}, ol^i, w^i) is the $\circ$ equivalent of wins1.

Additionally, $w^{1..4} = nil$ and $w^5 \neq \circ$. For $\circ$ moves, for values a of m^i where the corresponding board position is already filled (i.e. cheating moves), a is not contained in any unsatisfying tuples of the shadow constraints. Such values are dynamically pruned by the pure value rule (defined in chapter 3 section 3.2.6).

Model B. This is very similar, with the final three moves represented with a single constraint, over variables $w^6, b^6_{1..9}, m^7, m^8$. All other variables and constraints for these moves are eliminated. The variable m^9 is removed, because only one move is possible at this stage of the game. The initial domain of m^7 is now $\{0, 1, 2\}$ which maps onto the remaining three spaces on the board, wherever those spaces are located. Similarly, for m^8 the initial domain is $\{0, 1\}$. The constraint is satisfied iff crosses wins the game, either before move 7 (i.e. $w^6 = \times$), or $w^6 = nil$ and crosses wins the game at move 7 or 9.

Empirical comparison of the models. For both models, the pure value rule is applied to all variables using the scheme presented in chapter 3 section 3.5. The implementation of the pure value rule for all variables requires that a negated version of each constraint is also present in the problem. Although somewhat inefficient, this ensures a good mix of tight and loose constraints. There is no symmetry breaking. The reasoning at each node of the search is applying propagation and the pure value rule to exhaustion.

The *search* algorithm presented in the previous chapter is used, with a static value order which is shown below.

1	2	3
4	5	6
7	8	9

These models are designed to take advantage of the pure value rule to prune the universal variables. In a similar way, efficient models of problems in CSP are designed to take advantage of propagation algorithms to minimize the number of search nodes. Exploiting the pure value rule

Model A	Nodes	Search (s)	Setup (s)
WQGAC, predicate	4107	27.126	0.030
WQGAC, positive	4107	80.036	0.016
WQGAC, NDlist	4107	27.561	0.156
WQGAC, NDOnelist	4107	26.035	0.014
SQGAC	4107	10.788	0.479
Model B	Nodes	Search (s)	Setup (s)
WQGAC, predicate	999	5.818	0.378
WQGAC, positive	999	87.543	0.094
WQGAC, NDlist	999	6.540	0.254
WQGAC, NDOnelist	999	5.151	0.583
SQGAC	961	8.978	0.470

TABLE 3. Noughts and Crosses results

(or some other rule for pruning universal variables) seems to be crucial to modelling problems in QCSP.

In Noughts and Crosses, the second player can always force a draw. Therefore both models are unsatisfiable. Table 3 shows results for the two models: for model A, all constraints are processed using the specified algorithm. For model B, all constraints are processed using SQGAC, apart from the long constraint (and its negation which is required for the pure value rule), which is processed using the specified algorithm. The results show SQGAC to be faster on the shorter constraints, and WQGAC-Schema+NDOnelist to be most effective on the longer constraint in model B. In model A, the constraint with the largest space of tuples is findline, with $3^9 \times 2 = 39366$ assignments, half of which (19683) are satisfying tuples. In model B, the long constraint has $3^{11} \times 2 = 354294$ assignments, 160958 of which are satisfying tuples (just under half). The negated long constraint has 193336 satisfying tuples. The node counts for model B show that SQGAC is slightly stronger in practice on the long constraint, but this does not affect the outcome, that WQGAC-Schema+predicate, +NDlist and +NDOnelist are more effective.

For both models, the ranking of the four instantiations of WQGAC-Schema is the same: NDOnelist is most effective, followed by predicate then NDlist then the naive positive instantiation. Also, the performance of the three instantiations which perform intelligent jumping forward is quite similar, and much better than positive.

NDlist has to search fewer tuples than NDOnelist, and the search loop is identical in the implementation. However, NDOnelist performs better than NDlist for Noughts and Crosses, and for 3 out of 4 instances of Connect 4, presented below. I suspect this is because the tuple list is much smaller for NDOnelist, and it is shared for all partial instantiations, therefore its cache behaviour is better.

The CPU time taken per node is poor (> 2.5ms), but the number of nodes explored does show the potential of propagation algorithms for arbitrary constraints. Compared to model A, model B shows a fourfold decrease in the number of nodes, because of strong propagation on the large constraint in model B. By comparison, simply searching over the move variables (with the same value order, and without cheating moves) and backtracking when one or other player wins (or they draw) explores 16159 nodes.

4.6.2.2. *Connect 4.* Connect 4 is usually played on a board with 7 columns and 6 rows. The aim is to form a line (diagonally, vertically or horizontally) of four counters. Counters can only be placed in the lowermost empty position in each unfilled column. The model given here can be used for any number of rows and columns, and the aim is to find if the first player (red) can win however the second player (black) plays. The model has two parameters, row and col for the numbers of rows and columns. $row \geq 4$ and $col \geq 4$. This problem has also been attacked with QBF [**46**], however the encoding is flawed (it does not forbid placing counters in full columns [**4**]).

The game is modelled with the following variables, given in quantification order for a single move i. This sequence is then repeated $row \times col$ times for $i \in \{1 \dots row \times col\}$.

- If i is even, move variable $\forall u^i \in \{1 \dots col\}$.
- Move variable $\exists m^i \in \{1 \dots col\}$. If i is even, this variable shadows the one above.

- $row \times col$ variables $\exists b^i_{r,c}$ where $r \in \{1 \ldots row\}$ and $c \in \{1 \ldots col\}$, each with domain $\{red, black, nil\}$ representing the state of the board after the move. $b^i_{1,c}$ represents the piece at the base of column c.
- col variables $\exists h^i_c$, representing the height of column c after move i.
- $\exists gamestate^i$ with domain $\{red, black, nil\}$ representing the winner at move i.
- Boolean (0,1) variable $\exists line^i$ representing the presence of a line at move i.
- Boolean variable $\exists l^i_z$ indicating the presence of a line in each row, column or diagonal (numbered z) on the board.

The constraints for a single move $i \in \{1 \ldots row \times col\}$ are given below, numbered for comparison with another Connect 4 model in chapter 5.

(1) If i is even, for each column c, shadow($gamestate^{i-1}$, h^{i-1}_c, u^i, m^i) is satisfied iff $gamestate^{i-1} \neq nil$ or $h^{i-1}_c = row$ (i.e. column full) or $u^i = c \Rightarrow m^i = c$.

(2) If i is even, a black counter is placed on the board: for all columns c, constraint blackmove ($line^{i-1}$, h^{i-1}_c, m^i, $b^i_{1,c} \ldots b^i_{rows,c}$), arity $row + 3$, is satisfied iff $line^{i-1} = 1$ or $m^i = c \Rightarrow b^i_{h^{i-1}_c,c} = black \wedge b^i_{r>h^{i-1}_c,c} = nil$ (i.e. a black piece is placed at the appropriate height and all spaces above are nil) and $m^i \neq c \Rightarrow b^i_{r \geq h^{i-1}_c,c} = nil$. Note that a full column c implies $m^i \neq c$.

If i is odd, a similar constraint is posted for all columns c, redmove ($line^{i-1}$, h^{i-1}_c, m^i, $b^i_{1,c} \ldots b^i_{row,c}$), which is the same as the one above but placing a red counter.

(3) Map pieces from the board at $i-1$ to the board at move i: for all r, c, boardstate($b^{i-1}_{r,c}$, $b^i_{r,c}$) is satisfied iff $b^{i-1}_{r,c} = red \Rightarrow b^i_{r,c} = red$ and $b^{i-1}_{r,c} = black \Rightarrow b^i_{r,c} = black$.

(4) Link height and board state: for each column c, linkheight($b^i_{1,c} \ldots b^i_{row,c}$, h^i_c) is satisfied iff h^i_c equals the number of pieces in the column.

(5) Detect lines: for each column c and some unique number z, findline($b^i_{1,c} \ldots b^i_{row,c}$, l^i_z) is satisfied iff l^i_z represents the presence of a sequence of 4 pieces the same (of either colour) in $b^i_{1,c} \ldots b^i_{row,c}$. Similar constraints are posted for all rows and diagonals.

(6) Connect the main line variable $line^i$ with the l^i_z variables, where $y = \max(z) = row + col + 2(row-3) + 2(col-3) - 2$: $l^{i-1} \vee l^i_1 \vee l^i_2 \vee \cdots \vee l^i_y \Leftrightarrow line^i$. This is decomposed into

y ternary constraints (with additional existential variables t which are quantified with l^i_z) of the form $l^{i-1} \wedge l^i_1 \Leftrightarrow t_1,\ t_1 \vee l^i_2 \Leftrightarrow t_2, \ldots,\ t_{y-1} \vee l^i_y \Leftrightarrow line^i$.

(7) If i is odd, gamestatered$(gamestate^{i-1},\ line^i,\ gamestate^i)$ is satisfied if and only if
$gamestate^{i-1} = red \Rightarrow gamestate^i = red$ and
$gamestate^{i-1} = black \Rightarrow gamestate^i = black$ and
$gamestate^{i-1} = nil \wedge line^i \Rightarrow gamestate^i = red$ and
$gamestate^{i-1} = nil \wedge \neg line^i \Rightarrow gamestate^i = nil$.

If i is even, gamestateblack$(gamestate^{i-1},\ line^i,\ gamestate^i)$ is posted with the same meaning as above, except that $gamestate^{i-1} = nil \wedge line^i \Rightarrow gamestate^i = black$.

Some variables are referred to with indices which are out of range, for example $gamestate^0$. These are set as follows. $gamestate^0 = nil$, $l^0 = nil$, $h^0_c = 0$. The red player must win, so $gamestate^{row \times col} = red$. For the first move, symmetry is broken by removing the leftmost (lower) $\lfloor \frac{col}{2} \rfloor$ values, because they are equivalent to the higher values.

To attack this problem, the pure value rule is applied to universal variables only, using the scheme in chapter 3 section 3.5. Search is performed in quantification order, which in practice means that only move variables are searched over, and all others are set by propagation. The value order for search is determined by a heuristic: maximize the number of lines of 3 that the player has, breaking ties in favour of the leftmost move. The same heuristic is used for both players.

The constraints present in this model of Connect 4 are typically smaller than the largest constraint in model A for Noughts and Crosses. In model A, the constraint with the largest space of assignments is findline, with $3^9 \times 2 = 39366$ possible assignments to its variables. For Connect 4, the constraints with the largest space of assignments (for 5×5 board) are redmove and blackmove, with $2 \times 6 \times 5 \times 3^5 = 14580$ assignments.

The results are shown in table 4. For all four board sizes, the QCSP is unsatisfiable. SQGAC is significantly faster than WQGAC-Schema for all board sizes. NDlist and NDOnelist perform better than the other instantiations of WQGAC-Schema. Interestingly, the consistencies SQGAC

Board size		Nodes and time (s)			
col	*row*	Method	Nodes	Setup time	Search time
4	4	SQGAC	1692	0.395	5.812
		WQGAC, predicate	1692	0.211	13.916
		WQGAC, positive	1692	0.044	15.090
		WQGAC, NDlist	1692	0.375	12.808
		WQGAC, NDOnelist	1692	0.034	12.608
4	5	SQGAC	7744	0.794	41.104
		WQGAC, predicate	7744	0.363	92.343
		WQGAC, positive	7744	0.061	101.605
		WQGAC, NDlist	7744	0.716	78.002
		WQGAC, NDOnelist	7744	0.044	77.460
5	4	SQGAC	47712	0.324	243.709
		WQGAC, predicate	47712	0.185	519.686
		WQGAC, positive	47712	0.121	580.938
		WQGAC, NDlist	47712	0.047	493.246
		WQGAC, NDOnelist	47712	0.047	493.990
5	5	SQGAC	1027266	1.221	6204.549
		WQGAC, predicate	1027266	0.822	13955.849
		WQGAC, positive	1027266	0.079	17437.103
		WQGAC, NDlist	1027266	1.062	13225.556
		WQGAC, NDOnelist	1027266	0.519	13217.466

TABLE 4. Connect 4 results

and WQGAC appear to be equivalent on this problem, because the node counts are the same for all methods. The time taken per node is poor ($>$ 3ms per node in all cases).

4.7. Summary

Generalized arc-consistency has been well studied and is very important in CSP. To my knowledge, this is the first time practical algorithms have been developed for arbitrary constraints in QCSP. The work in this chapter is the foundational work on arbitrary constraints for QCSP. I have developed two new algorithms, one of which is instantiated in four different ways. These are compared on various problems.

The empirical results show the strengths of the two algorithms WQGAC-Schema and SQGAC. For most situations, the simpler SQGAC is preferable, but for the long constraint in model B of Noughts and Crosses, WQGAC-Schema+(predicate, NDlist, NDOnelist) is considerably more efficient.

For both the games, the time taken at each node is poor. The algorithms for logic and sum constraints in chapters 5 and 6 address the need for greater efficiency at each node, since they run in linear time and have no backtracking state.

CHAPTER 5

Logical constraints

5.1. Introduction

Logical constraints, such as conjunction, disjunction and implication (for example, $(x_1 = 5) \vee (x_2 = 3) \Rightarrow (x_3 \neq 7)$), are commonplace in classical constraint programming. Constraints like these have received some attention in QCSP as well. The main motivation is efficiency. The algorithms for arbitrary non-binary constraints presented in chapter 4 can process logical constraints, but it is possible to improve performance significantly by developing a specialized constraint propagation algorithm.

Existing CSP solvers offer facilities for logical constraints such as the example above. In QCSP, Bordeaux and Monfroy have developed various ternary primitive constraints for logical constraints, and a method of decomposing a complex expression into their primitives [**19**, **22**]. This is discussed in chapter 2 section 2.3.3. QBF solvers also deal with quantified disjunction constraints (chapter 2 section 2.2.1). Any new approach should compare well (in efficiency, expressiveness or strength of consistency) to both these items of work.

In this chapter I present a new algorithm to process reified disjunction constraints (e.g. $\neg x_1 \vee x_2 \vee \neg x_4 \Leftrightarrow \neg x_3$). This form of constraint is sufficient for a wide variety of logical expressions. The algorithm is instantiated in two different ways, and I show that it maintains SQGAC and executes in linear time, with no backtracking state. The time complexity is the same as for Bordeaux and Monfroy's existing work, and the level of consistency is stronger. The new constraint can also enforce the same level of consistency as unit propagation on a QBF clause, but it is more expressive.

Finally, the efficiency of the new primitive is evaluated with some experiments on Connect 4. These show its effectiveness against the approach of Bordeaux and Monfroy, and against the

SQGAC and WQGAC propagation algorithms in chapter 4. A further experiment shows that the pure value rule is very useful with Connect 4.

5.1.1. Motivating examples. This section contains some examples demonstrating that ternary primitives and decomposition is weak. The algorithms given later solve all the difficulties identified here.

Consider expression (9), which can break down into the two constraints shown in (10), according to the decomposition rules. In this example, enforcing SQGAC on expression (9) determines falsity, because there is no assignment for x_1, x_2 and x_3 which is compatible with all values of x_4. In the decomposition, enforcing SQGAC on both constraints is able to determine $x_3 \neq 1$ but it is not able to determine falsity. Returning to the intuitive understanding, some of the interaction between x_4 and the set of outer variables x_1, x_2 and x_3 has been lost. The other two minimum decompositions (by first factoring out $x_1 \vee x_3$ or $x_2 \vee x_3$ rather than $x_1 \vee x_2$) have the same weakness because x_1, x_2 and x_3 are symmetric. Also, the primitive $(x_3 \vee x_5 \Leftrightarrow x_4)$ in equation (10) is not allowed in Bordeaux and Monfroy's scheme, because x_4 is not quantified last.

$$\exists x_1, x_2, x_3 \forall x_4 \ : \ x_1 \vee x_2 \vee x_3 \Leftrightarrow x_4 \tag{9}$$

$$\exists x_1, x_2, x_3 \forall x_4 \exists x_5 \ : \ (x_1 \vee x_2 \Leftrightarrow x_5) \wedge (x_3 \vee x_5 \Leftrightarrow x_4) \tag{10}$$

As a second example, consider expression (11). Enforcing SQGAC on this expression determines falsity. The expression breaks down into two constraints shown in (12). Enforcing SQGAC on the constraints does not remove any values or determine falsity. In this particular case, if the primitive constraint could handle negation of its variables, then the problem would be avoided.

$$\exists x_1, x_2 \forall x_3 \ : \ x_1 \vee x_2 \Leftrightarrow \neg x_3 \tag{11}$$

$$\exists x_1, x_2 \forall x_3 \exists x_4 \;:\; (x_1 \vee x_2 \Leftrightarrow x_4) \wedge (\neg x_3 \Leftrightarrow x_4) \tag{12}$$

If the QCSP instance includes operators on integers such as $=$ or $\neq$ with Boolean connectors, a further problem arises. Consider expression (13). Applying SQGAC directly to this expression determines falsity because $x_1 \mapsto 5,\ x_2 \mapsto 3$ conflicts with $x_3 \mapsto 7$. Bordeaux et al. provide a primitive for integer equality: $(x_1 = x_2) \Leftrightarrow x_3$ where x_1 and x_2 are integers and x_3 is Boolean. Decomposing expression (13) yields expression (14). Enforcing SQGAC on each of the conjuncts of expression (14) does nothing.

$$\begin{gathered} \forall x_1 \in \{1 \ldots 10\},\ \forall x_2 \in \{1 \ldots 10\},\ \forall x_3 \in \{1 \ldots 10\} \;: \\ (x_1 \neq 5) \vee (x_2 \neq 3) \vee (x_3 \neq 7) \end{gathered} \tag{13}$$

$$\begin{gathered} \forall x_1 \in \{1 \ldots 10\},\ \forall x_2 \in \{1 \ldots 10\},\ \forall x_3 \in \{1 \ldots 10\}, \exists x_4, x_5, x_6, x_7, x_8, x_9, x_{10} \;: \\ (x_1 = 5 \Leftrightarrow x_4) \wedge (x_2 = 3 \Leftrightarrow x_5) \wedge (x_3 = 7 \Leftrightarrow x_6) \wedge (x_7 \Leftrightarrow \neg x_4) \\ \wedge (x_8 \Leftrightarrow \neg x_5) \wedge (x_9 \Leftrightarrow \neg x_6) \wedge (x_7 \vee x_8 \Leftrightarrow x_{10}) \wedge (x_{10} \vee x_9 \Leftrightarrow 1) \end{gathered} \tag{14}$$

5.1.2. Proposed logical primitive constraints. To solve all of the difficulties illustrated above, I propose two variants of a reified disjunction constraint. The simpler one (Boolean reified disjunction) acts on Boolean variables and can handle negation of any of its variables. The other variant acts on integer variables. An algorithm to enforce SQGAC in linear time (time $O(n)$ where n is the number of variables) for either form is given in section 5.2.

A *Boolean literal* is a positive or negated instance of a variable, represented by l_i for some variable x_i. Boolean reified disjunction is shown in formula 15 below.

(15) $$Q_1x_1 \dots Q_rx_r : \; l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_r \Leftrightarrow l_i$$

The other variant is a reified disjunction of comparisons $x_i = v_i$ or $x_i \neq v_i$, where x_i is an integer variable and v_i is a constant. I will call $x_i = v_i$ and $x_i \neq v_i$ *integer literals*. Integer reified disjunction is shown in formula 16, where $(=, \neq)$ represents either $=$ or $\neq$. If $\forall i : v_i = 1$ and the variables are all Boolean, then this primitive simulates the other. For a negated literal, the operator is $\neq$ and for a positive literal, the operator is $=$.

(16) $Q_1x_1 \dots Q_rx_r$:

$$x_1(=,\neq)v_1 \vee \dots \vee x_{i-1}(=,\neq)v_{i-1} \vee x_{i+1}(=,\neq)v_{i+1} \vee \dots \vee x_r(=,\neq)v_r \Leftrightarrow x_i(=,\neq)v_i$$

By using the De Morgan law[1], the same primitives can be used for reified conjunction with the same level of consistency, for both Boolean and integer literals. Therefore there is no need to develop a separate algorithm for reified conjunction. For example, $x_1 \wedge x_2 \wedge \neg x_3 \Leftrightarrow x_4$ is transformed to $\neg x_1 \vee \neg x_2 \vee x_3 \Leftrightarrow \neg x_4$.

A straightforward disjunction $l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_n$ can be represented as $l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_n \Leftrightarrow 1$. Straightforward conjunctions $l_1 \wedge \dots \wedge l_{i-1} \wedge l_{i+1} \wedge \dots \wedge l_n$ can be represented as $l_1 \wedge \dots \wedge l_{i-1} \wedge l_{i+1} \wedge \dots \wedge l_n \Leftrightarrow 1$, which is transformed to $\neg l_1 \vee \dots \vee \neg l_{i-1} \vee \neg l_{i+1} \vee \dots \vee \neg l_n \Leftrightarrow 0$ by the De Morgan law. (Disjunction and conjunction of integer literals can be handled in exactly the same way.)

The Boolean reified disjunction primitive can be used in place of $x_1 \Leftrightarrow \neg x_2$, by having a single disjunct l_1 on the left hand side. However the need for the $x_1 \Leftrightarrow \neg x_2$ primitive has been mostly removed by handling negation within the reified disjunction primitive.

To represent an expression with mixed $\wedge$ and $\vee$ connectors, where the expression cannot be rearranged into one which matches the primitive, additional existential variables may need to be introduced, and more than one primitive may be needed. For some logical expression $\mathcal{E}$,

[1]De Morgan's law in propositional logic: $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ where P and Q are arbitrary propositions.

decomposing for Bordeaux and Monfroy's primitives gives a set of constraints C and a set of additional existential variables V. Decomposing for reified disjunction gives a set of constraints C' and a set of additional existential variables V'. For all valid $\mathcal{E}$, $|C| \geq |C'|$ and $|V| \geq |V'|$ if C and C' are the shortest possible decompositions. Apart from the higher level of consistency, this has advantages for the implementation of a QCSP solver: it is potentially more efficient to handle fewer variables and fewer constraints.

As an example, formula (13) in the previous section is decomposed to 8 primitive constraints and 7 additional variables under the Bordeaux et al. scheme. With the reified disjunction, it can be expressed with one constraint, as shown in in formula (17). Enforcing SQGAC on this constraint determines failure.

$$\begin{gathered}\forall x_1 \in \{1 \ldots 10\},\ \forall x_2 \in \{1 \ldots 10\},\ \forall x_3 \in \{1 \ldots 10\}\ : \\ (x_1 \neq 5) \vee (x_2 \neq 3) \vee (x_3 \neq 7) \Leftrightarrow 1\end{gathered} \tag{17}$$

Also, equations (9) and (11) can be expressed with one reified disjunction.

Not all expressions $\mathcal{E}$ can be represented with a single reified disjunction constraint, so some expressions must be decomposed. However, since these constraints are much more expressive and therefore easier to use, I leave it to the modeller to decompose expressions manually, rather than provide an automated decomposition.

5.1.3. Other possible primitives. Boolean disjunction contains two connectors: $\vee$ and $\neg$. It may be possible to add other connectors to create a more powerful primitive, without compromising the linear time SQGAC propagation. However, if a primitive represents a computationally difficult problem, then to achieve SQGAC is also computationally difficult. This is because deciding the consistency of some value $x_{k_1} \mapsto a$ with respect to a constraint C_k is the same as deciding the satisfiability of the local problem $\mathcal{P}_k$ with x_{k_1} instantiated to a.

For example in a primitive representing general quantified boolean formulae (QBF), a PSPACE-complete problem, achieving SQGAC is PSPACE-hard. Conjunctive normal form (CNF) QBF is also known to be PSPACE-complete. Therefore, for a CNF primitive, which is a conjunction of disjunctions of literals $\bigwedge \bigvee l_j \Leftrightarrow l_i$, for example $(x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_3 \vee x_6) \Leftrightarrow x_4$, it would be PSPACE-hard to achieve SQGAC if l_i is set to 1.

Another possibility would be to use disjunctive normal form (DNF), which is a disjunction of conjunctions of literals, $\bigvee \bigwedge l_j \Leftrightarrow l_i$ for example $(x_1 \wedge \neg x_5) \vee (x_2 \wedge \neg x_3 \wedge x_6) \Leftrightarrow x_4$. This is a dual problem to CNF QBF, in the sense that proving the falsity of a DNF QBF is as difficult as proving the truth of a CNF QBF. Therefore if l_i is set to 0, it would be PSPACE-hard to achieve SQGAC over the other variables.

To summarize, propagating a general expression with any nesting of $\wedge$, $\vee$ and $\neg$ is PSPACE-hard, and so are reified CNF and DNF expressions, which are expressions containing $\wedge$, $\vee$ and $\neg$ but with the nesting restricted to two levels: specifically $\vee$ nested within $\wedge$ or vice versa. If such strong propagation is required, the general SQGAC algorithm given in chapter 4 can be used, or one of the WQGAC algorithms in the same chapter.

5.2. A propagation algorithm for reified disjunction

5.2.1. Introduction. In this section I give a coarse-grained, one pass propagation algorithm to achieve SQGAC on the constraints given in formulas (15) and (16), with an argument of correctness for each case. The algorithm works on literals $(\neg)x_i$ or $x_i(=,\neq)v_i$, where the literals take values 0 or 1 representing the truth of the literal, depending on the domain of x_i. The useful information about a literal is its index i, quantification Q_i, and whether or not it is fixed to either 0 or 1.

If a literal containing x_i is assigned to 0 or 1, the quantification of the variable becomes irrelevant in the context of the reified disjunction constraint C_k. The appropriate value can be substituted for the literal and the constraint simplified to one which does not contain x_i. Although the algorithm does not dynamically simplify the constraint during search, it does use this fact.

Both the algorithm and the proof that it establishes SQGAC are split into four independent parts. These are based on the value and quantification of l_i, since the value of this literal determines the truth or falsity of the disjunction. Each of the four subsections 5.2.4 to 5.2.7 contain an algorithm alongside its soundness and completeness argument. The four parts are outlined below.

Disjunction false: The literal $l_i = 0$ therefore the disjunction must be false. This is the simplest case.

Disjunction true: The literal $l_i = 1$ therefore the disjunction is true. This case is equivalent to a QBF clause.

x_i **is universal:** l_i does not have a set value and x_i is universally quantified. In this case, the sets of literals $\{l_1 \dots l_{i-1}\}$ and $\{l_{i+1} \dots l_r\}$ must be treated differently because of their quantification relative to l_i.

x_i **is existential:** l_i is not set and x_i is existential. In this case, the disjunction is examined to see if l_i must be set to 0 or 1.

Note that in the definition of constraint (definition 3.2.2 in chapter 3) it is not allowed to have more than one instance of a variable in a constraint. This fact is required for the proof that the algorithm establishes SQGAC. If an existential variable is repeated, the algorithm will be sound but not complete. For example, establishing SQGAC on $\exists x_1, x_2 : x_1 \vee \neg x_1 \Leftrightarrow x_2$ would set x_2 to 1, but the reified disjunction algorithm is unable to do so. If a universal variable is repeated, the algorithm is unsound. For example, applied to $\forall x_1 : x_1 \vee \neg x_1 \Leftrightarrow 1$ the algorithm would return false, when the constraint is a tautology. (Queso checks for repeated universals.)

5.2.2. Handling literals. Literals can only take two values 0 and 1, however many values are in the domain of the variable. Therefore if a literal is not fixed to 0 or 1, then the only information needed is the quantifier Q_i and i. Since i is a constant, this leaves four states that a literal can be in: 0, 1, unassigned (universal), and unassigned (existential). The procedure *literalState* abstracts away the quantification and negation or comparison, returning one of the set $\{0, 1, \forall, \exists\}$ when called for some variable x_i. Notice that (with literals of the form $x_j(=, \neq)v_j$) the literal may take the value 0 or 1 even when x_j is not instantiated. For example, if the literal is $x_j = 4$ and $D_j = \{1, 2, 5\}$, then the literal takes value 0. This is handled correctly by *literalState* (algorithm

Algorithm 23 propagateOr

```
**procedure** propagateOr(): Boolean
{Achieve consistency for the constraint $Q_1x_1 \dots Q_rx_r : l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_r \Leftrightarrow l_i$ or $Q_1x_1 \dots Q_rx_r$ :
$x_1(=,\neq)v_1 \vee \dots \vee x_{i-1}(=,\neq)v_{i-1} \vee x_{i+1}(=,\neq)v_{i+1} \vee \dots \vee x_r(=,\neq)v_r \Leftrightarrow x_i(=,\neq)v_i$}
$a \leftarrow$literalState($x_i$)
**if** $a = 0$ **then**: **return** disjunctionFalse()
**if** $a = 1$ **then**: **return** disjunctionTrue()
**if** $a = \forall$ **then**: **return** xiUniversal()
**if** $a = \exists$ **then**: **return** xiExistential()
```

30). To handle the two different types of literal, two versions of *literalState* are given in section 5.2.9. This gives a simple interface between the variables in the problem and the propagation algorithm.

The procedure *removeValue* is used to ‘prune’ a literal: this involves removing one or more values from the domain of x_i to force the literal to be either 0 or 1. The value to be removed (0 or 1) is passed in as a parameter. Again, to handle the two different types of literal, there are two versions of *removeValue* described in section 5.2.9.

5.2.3. The central procedure. The algorithm stores no state, and therefore nothing needs to be backtracked. The procedure *propagateOr* (algorithm 23) does a case-split on the literal l_i, since the four cases require significantly different propagation. In the following sections, I give the algorithm for each of the four cases and argue its correctness. I refer to the definition of SQGAC (definition 3.2.13 in chapter 3). The algorithms return false when it would be necessary to empty a domain or prune a universal to make the constraint consistent. This is a minor simplification, since in this case the simplified problem $\mathcal{P}$ would be false. They return true when the constraint is consistent.

For each of the four cases, the time taken is $O(r)$, disregarding the cost of waking up other constraints.

5.2.4. Case disjunctionFalse. Procedure *disjunctionFalse* (algorithm 24) simply sets all the literals in the disjunction to 0. This is linear time for Boolean literals. The constraint propagator

Algorithm 24 disjunctionFalse

```
procedure disjunctionFalse(): Boolean
{Achieve consistency for the simplified constraint
$Q_1x_1 \dots Q_rx_r : l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_r \Leftrightarrow 0$}
{This is done by setting each literal to 0}
for each $l_j$ where $j \neq i$:
    if not removeValue($j$, 1) then: return false
return true
```

will not be called again in the current branch of the search, because all its variables have been instantiated.

LEMMA 5.2.1. disjunctionFalse *is sound and complete*

PROOF. There are two cases, corresponding to the two return statements in the algorithm. In the first case, if there is a literal in the set $\{l_1 \dots l_{i-1}, l_{i+1} \dots l_r\}$ which cannot be set to 0 (i.e. it is 1 or universal) then there can be no winning strategy for C_k since there is a contradiction. Establishing SQGAC would empty all domains, therefore algorithm 24 returns false. In the second case, all literals in $\{l_1 \dots l_{i-1}, l_{i+1} \dots l_r\}$ can be set to 0. There is exactly one winning strategy S for C_k with exactly one scenario $\tau \in \text{sce}(S)$, which sets all literals to 0. Therefore algorithm 24 establishes SQGAC before returning true. □

The proof that *disjunctionFalse* establishes SQGAC for integer reified disjunction is very similar, but there may be more than one winning strategy in the second case. All scenarios of all winning strategies set all literals to value 0.

5.2.5. Case disjunctionTrue. Procedure *disjunctionTrue* (algorithm 25) is called when the disjunction must be satisfied. The disjunction (when applied to Boolean literals) is identical to a QBF clause [**24**]. The unit propagation algorithm, either with backtracking lists, or with watched literals (by Gent et al. [**47**]) takes $O(r)$ amortized down a branch of the search. The algorithm presented here is simpler and takes $O(r)$ time each time it is called. The main reason for this decision is that nearly all of the reified disjunction constraints used for Connect 4 (section 5.3) and for job shop scheduling (chapter 7) are short (with typically 5 or fewer literals in the disjunction).

Algorithm 25 disjunctionTrue

```
(1)  procedure disjunctionTrue(): Boolean
(2)  {Achieve consistency for the reduced constraint $Q_1x_1 \dots Q_rx_r : l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_r$}
(3)  $ui \leftarrow nil$ {store indices of a universal and existential literal respectively.}
(4)  $ei \leftarrow nil$
(5)  for $j \leftarrow 1..i-1,\ i+1..r$ in ascending order:
(6)      $a \leftarrow$literalState($j$)
(7)      if $a = 1$ then: return true {The disjunction is satisfied}
(8)      if $a = \forall$ and $ui = nil$ then: $ui \leftarrow j$
(9)      if $a = \exists$ then:
(10)         if $ei \neq nil$ or $ui \neq nil$ then: return true {If there is an existential and an outer existential or universal, therefore no work can be done}
(11)         $ei \leftarrow j$
(12) {$ui$ contains the outermost universal literal, if one exists}
(13) {$ei$ contains the outermost existential literal, if one exists}
(14) if $ei = nil$ then: return false {No way to satisfy the disjunction}
(15) return removeValue($ei$, 0) {Just one existential literal, with no outer universal, so set it to 1}
```

The decision was made on the intuition that watched literals would be of no benefit [**47**] and the overhead of backtracking lists should be avoided.

The strategy to prove that algorithm 25 is sound and complete is to observe that it terminates, and there are four exit points on lines 7, 10, 14 and 15. For each one I show that C_k is SQGAC (if true is returned) or that establishing SQGAC would empty a domain or prune a universal (if false is returned).

In algorithm 25 the for loop (lines 5-11) collects some information, and returns if no removals can be made. The leftmost universal and existential literals are stored in ui and ei respectively. The stopping conditions where no removals can be made are listed below:

(1) There is a 1 in the disjunction so the constraint is satisfied (line 7), or

(2) there are two or more existential literals, therefore it is not clear which literal will end up set to 1 (line 10), or

(3) there are one or more universals with an inner existential, $\dots \forall x_j \dots \exists x_m \dots :\ l_j \vee \dots \vee l_m$, in which case we cannot set x_k because x_m may satisfy the disjunction (line 10).

These are all the conditions which cause the procedure to return true without performing any pruning. I will show that in each of these cases the constraint C_k is SQGAC.

LEMMA 5.2.2. *If* disjunctionTrue *returns true without pruning, then C_k is SQGAC*

PROOF. (Stopping condition 1) The disjunction is satisfied, therefore C_k^S contains every valid tuple: $C_k^S = \{D_{k_1} \times D_{k_2} \times \cdots \times D_{k_r}\}$. Therefore for each value $x_{k_i} \mapsto a$ it is trivial to construct a winning strategy containing $x_{k_i} \mapsto a$, because all strategies are winning strategies. Therefore C_k is SQGAC.

(Stopping condition 2) For any literal l_j and existential literal $l_{m \neq j}$ in the disjunction, both truth values of l_j are supported by a winning strategy containing $l_m = 1$. Similarly to stopping condition 1, it is trivial to construct such a winning strategy because the disjunction is satisfied by l_m. Therefore all values of all literals are supported and C_k is SQGAC.

(Stopping condition 3) The existential literal is l_m. Construct a strategy S such that when all outer universals $l_{j<m} = 0$ then $l_m = 1$, and otherwise $l_m = 0$, and any other existential literals take the same value as l_m. S is a winning strategy which supports both values of l_m and both values of any other literal $l_{p \neq m}$. □

If none of the three stopping conditions apply, then the algorithm either returns false, or prunes a domain then returns true. Cadoli et al. give consistency rules for QBF clauses, and prove their soundness [**25**], but not their completeness. I will show that algorithm 25 implements the rules, and that they are complete.

(1) If any literal is 1 then the disjunction simplifies to 1.
(2) Any literal which is 0 can be removed.
(3) If the disjunction is empty, then it is false.
(4) If all literals in the disjunction are universal, then the disjunction simplifies to 0 (lemma 2.1 of [**25**]). The disjunction cannot be a tautology because no variable is allowed to occur more than once.

(5) For a disjunction of the form $\exists x_1, \forall x_2, x_3, \ldots : l_1 \vee l_2 \vee l_3 \vee \cdots$ (where there exists a single existential literal with a set of universal literals quantified inside), l_1 is instantiated to 1 (lemma 2.6 of [**25**]).

The first two of these rules correspond directly to the algorithm — it ignores literals which are set to 0 and if it finds a 1 it returns true (line 7). To implement rules 3 and 4, if the algorithm finds no existential or 1 literals, it returns false on line 14. Cadoli et al. have proven that rules 3 and 4 are sound.

Rule 5 corresponds to the final return statement on line 15. After completing the loop, ei contains the index of the outermost existential literal, if there is one. If there were two existential literals, the procedure would already have returned true (line 10). Also if there were a universal with an existential quantified inside, the procedure would have returned true (line 10). Therefore any universal literals are quantified inside l_{ei}. Literal l_{ei} is set to 1 on line 15, and *removeValue* returns true therefore *disjunctionTrue* returns true on line 15. Again, Cadoli et al. have proven that rule 5 is sound.

LEMMA 5.2.3. disjunctionTrue *is sound and complete*

PROOF. Between the rules of Cadoli et al. and lemma 5.2.2, it is proven that at all four exit points of the algorithm, either the constraint is SQGAC (and true is returned) or it cannot be made SQGAC (and false is returned). The one instance of pruning is sound [**24**], and the algorithm terminates. Therefore it is sound and complete. □

5.2.6. Case xiUniversal. Procedure *xiUniversal* (algorithm 26) is called when $Q_i = \forall$ and l_i is not set to 0 or 1. To prove it is sound and complete, I show at each exit point (of three) that the constraint is SQGAC or cannot be made SQGAC, and that the call to *removeValue* is sound.

The literals l_j where $j < i$ must be set to 0 because value 1 for l_j conflicts with $l_i = 0$. If a literal cannot be set to 0 (i.e. it is universal or 1) then $\mathcal{P}_k$ has no winning strategy, so to make C_k SQGAC would falsify $\mathcal{P}$. Therefore the algorithm either sets the literal to 0 or returns false on line 4.

Algorithm 26 xiUniversal

```
(1)  procedure xiUniversal(): Boolean
(2)  {The outer set of literals $l_j$ where $j < i$ must evaluate to 0, because 1 is not consistent
     with $l_i = 0$}
(3)  for $j \leftarrow 1 \ldots i-1$:
(4)      if not removeValue($j$, 1) then: return false {Set $l_j = 0$}
(5)  {The inner set of literals $l_j$ where $j > i$ must be free i.e. no universals, no 1's and at
     least one existential.}
(6)  $e \leftarrow 0$ {Whether an existential has been found}
(7)  for $j = i+1 \ldots r$:
(8)      $a$=literalState($j$)
(9)      if $a = 1$ or $a = \forall$ then: return false
(10)     if $a = \exists$ then: $e \leftarrow 1$
(11) return $e$
```

The truth of the second part of the disjunction $\bigvee l_j$ where $j > i$ must match l_i in any winning strategy, for both values of l_i. If any literal $l_j = 1$, then the disjunction evaluates to 1, which is not compatible with $l_i = 0$. Also, if any literal l_j is universal, then $l_j = 1$ is not compatible with $l_i = 0$ and there can be no winning strategy for $\mathcal{P}_k$. Therefore it is sound to return false on line 9.

The only case remaining is to find at least one existential literal $l_{j>i}$. If such a literal exists it is straightforward to construct a winning strategy S, where all $l_{j>i}$ are 0 when $l_i = 0$, and $l_j = 1$ when $l_i = 1$. To implement this, the algorithm (lines 6-10) checks if there is an existential literal. After completing the loop, it returns true if there is an existential literal and 0 otherwise. Returning true corresponds to the existence of S, and returning false corresponds to its non-existence, therefore the algorithm is sound.

LEMMA 5.2.4. xiUniversal *is sound and complete*

PROOF. By the case analysis above, any domain removals are sound, and C_k is SQGAC whenever true is returned. Also, C_k cannot be made SQGAC whenever false is returned. The algorithm clearly terminates, therefore it is sound and complete. □

5.2.7. Case xiExistential. Procedure *xiExistential* (algorithm 27) checks for the conditions that would lead to pruning l_i:

(1) Some literal $l_{j \neq i}$ set to 1, which implies that $l_i = 1$, or

Algorithm 27 xiExistential

```
(1)  procedure xiExistential(): Boolean
(2)  {$x_i$ may need to be pruned}
(3)  {Check for the two conditions which would lead to pruning $x_i$}
(4)  allFalse←true
(5)  for $j = 0 \dots i-1,\ i+1 \dots r$:
(6)      $a =$ literalState($j$)
(7)      if $a = 1$ then: return removeValue($i$, 0)
(8)      if ($a = \forall$ and $j > i$) then:
(9)          if not removeValue($i$, 0): return false {Set $l_i = 1$ or return false}
(10)         return disjunctionTrue() {Now $l_i = 1$, call the appropriate procedure to propagate the consequences}
(11)     if $a = \exists$ or ($a = \forall$ and $j < i$) then: allFalse←false
(12) if allFalse then: return removeValue($i$, 1)
(13) return true
```

(2) some inner literal $l_{j>i}$ is universal, which implies that $l_i = 1$, which must be propagated further by calling *disjunctionTrue*, or

(3) all literals $l_{j \neq i}$ are 0, so $l_i = 0$.

I will argue that these three rules are sound and complete, and implemented by algorithm 27.

Firstly, if there is some literal $l_{j \neq i} = 1$ then there is no winning strategy containing $l_i = 0$, so rule 1 is sound. Similarly, if there is a literal $l_{j>i}$ which is universal, then $l_i = 0$ is not compatible with both values of l_j, therefore there is no winning strategy containing $l_i = 0$, therefore rule 2 is sound. Also, if all literals $l_{j \neq i} = 0$ then there is no winning strategy where $l_i = 1$, so rule 3 is sound.

If no literal $l_{j \neq i}$ is universal, and none of the three rules apply, then there must be some literal $l_{j \neq i}$ which is existential. In this case two strategies can be constructed. S_1 has (for all $j \neq i$ where l_j is existential) $l_j = 0$ and $l_i = 0$, and S_2 has $l_j = 1$ and $l_i = 1$. Both S_1 and S_2 are winning strategies, and together they support all values for all variables, so C_k is SQGAC. Therefore the rules are sound and complete if there are no universal variables.

As always, it is more tricky with universal variables. Assuming rules 1 and 3 do not apply, if there is some universal literal $l_{j<i}$ and no universal literal $l_{j>i}$ then we have that $\dots \forall x_j \dots \exists x_i \dots : \dots \vee l_j \vee \dots \Leftrightarrow l_i$ and we can construct a winning strategy S_1 which supports $l_i = 0$ when $l_j = 0$, and supports $l_i = 1$ when $l_j = 1$. Other existential literals are set to 0 in S_1. We can construct a second winning strategy S_2 where all existential literals (including l_i) are set to 1. Therefore all values are supported and C_k is SQGAC. Therefore the rules are sound and complete when there are universal literals $l_{j<i}$.

Now I will briefly argue that algorithm 27 exactly implements rules 1, 2 and 3. Line 7 implements rule 1, since the algorithm iterates through all literals $l_{j\neq i}$. Lines 8-10 implement rule 2, and return the outcome of *disjunctionTrue*. For rule 3, the Boolean variable allFalse represents whether all literals seen so far are 0. It is maintainted on line 11. If the algorithm examines all literals and all are 0, then the rule fires (line 12). Finally, the algorithm returns true if no rule fires.

LEMMA 5.2.5. xiExistential *is sound and complete*

PROOF. The three rules are both sound and complete, and the algorithm exactly implements them, therefore the algorithm must be sound and complete. □

5.2.8. Proof of soundness and completeness. The central algorithm *propagateOr* (algorithm 23) just finds the state of literal l_i, then calls one of the four algorithms above. Therefore the proof follows from the lemmas for each case.

THEOREM 5.2.6. propagateOr *is sound and complete*

PROOF. By lemmas 5.2.1, 5.2.3, 5.2.4 and 5.2.5, each of the four cases of *propagateOr* is sound and complete, therefore *propagateOr* is sound and complete. □

5.2.9. Instantiating for different types of literals. Two procedures (*literalState* and *removeValue*) were left undefined in the algorithm above, because they depend on the type of literal. They are given here for Boolean literals and integer literals.

5.2.9.1. *Boolean literals.* For literals of the form $(\neg)x_i$, the two procedures are given in algorithms 28 and 29.

Algorithm 28 (*literalState*) examines the domain of variable x_i and its quantification, and returns its state. Algorithm 29 (*removeValue*) simply checks whether the variable occurs negatively

Algorithm 28 literalState for $(\neg)x_i$

```
procedure literalState($i$: variable index): $\{0, 1, \forall, \exists\}$
if $0 \in D_i$:
    if $1 \in D_i$ then:
        return $Q_i$
    else:
        if $l_i = \neg x_i$ then: return 1 else: return 0
else:
    if $l_i = \neg x_i$ then: return 0 else: return 1
```

Algorithm 29 removeValue for $(\neg)x_i$

```
procedure removeValue($i$: variable index, $b$: Boolean): Boolean
if $(b = 0 \ \wedge \ l_i = \neg x_i) \vee (b = 1 \ \wedge \ l_i = x_i)$ then:
    return exclude($C_k, \ x_i, \ 1$)
else:
    return exclude($C_k, \ x_i, \ 0$)
```

Algorithm 30 literalState for $x_i(=, \neq)v_i$

```
procedure literalState($i$: variable index): $\{0, 1, \forall, \exists\}$
if $D_i = \{v_i\}$ then:
    if $l_i = (x_i \neq v_i)$ then: return 0 else: return 1
else:
    if $v_i \in D_i$ then:
        return $Q_i$
    else:
        if $l_i = (x_i \neq v_i)$ then: return 1 else: return 0
```

Algorithm 31 removeValue for $x_i(=, \neq)v_i$

```
procedure removeValue($i$: variable index, $b$: Boolean): Boolean
if $(b = 0 \ \wedge \ l_i = (x_i \neq v_i)) \vee (b = 1 \ \wedge \ l_i = (x_i = v_i))$ then:
    return exclude($C_k, \ x_i, \ v_i$)
else:
    for all $a \in D_{k_i}$ where $a \neq v_i$ then:
        if not exclude($C_k, \ x_i, \ a$) then: return false
    return true
```

$(l_i = \neg x_i)$ or positively in the constraint, and removes the appropriate value using the *exclude* procedure which is assumed to be part of the constraint solver infrastructure. The *exclude* procedure returns false if the domain is emptied or a universal is pruned.

5.2.9.2. *Integer literals.* For literals of the form $x_i(=, \neq)v_i$, the two procedures are given in algorithms 30 and 31. These are very similar to the above.

Other instantiations. The Boolean and integer literals were chosen because of the requirements for modelling Connect 4 (section 5.3 below). However, other types of literal such as $x_i \leq v_i$ or $x_i \geq v_i$ may be useful in other circumstances. The most general form of literal would be set membership, $x_i \in A_i$ where A_i is a set of elements from the original domain: $A_i \subseteq D_i^0$. The NB-SAT formalism (Frisch and Peugniez [**43**], and developed further by Frisch, Peugniez, Doggett and Nightingale [**44**]) uses set membership literals. Each constraint in an NB-SAT instance is a disjunction of set membership literals.

5.3. Empirical evaluation

The proposed algorithm could be compared with the propagation algorithms in the previous chapter, the decomposition approach, QBF, and binary QCSP.

First I compare the proposed algorithm against SQGAC using Connect 4, finding that the new algorithm can be significantly more efficient. Second, I use Connect 4 again to compare against the decomposition approach. In this case the new algorithm can be hugely more efficient.

I did not compare the proposed algorithm against QBF solvers, although this would be interesting. The QBF solvers are very efficient at handling this type of relation, so I would expect them to perform very well. Also they have features such as clause learning which should allow them to outperform the plain search algorithm of Queso. However, the aim of the reified disjunction algorithm is to bring strong reasoning on such constraints to QCSP, where the algorithm can be used with other constraint propagators to solve complex problems, such as the job shop scheduling problem in chapter 7.

I also did not compare against binary QCSP solvers, but in chapter 4 it was shown that binary QCSP solvers are very poor with long constraints, when using the hidden variable encoding. There may be a better encoding specifically for reified disjunction, but it is not clear that any encoding could achieve a high level of consistency (approaching SQGAC).

5.3.1. Connect 4 experiments. To compare the proposed algorithm with the general SQGAC algorithm in chapter 4, I have constructed a second model of Connect 4 using the disjunction primitive. The previous model will be referred to as the *arbitrary constraint model*, and the one presented here as the *disjunction model*. Recall that SQGAC was most effective of the algorithms in chapter 4.

The set of variables is a superset of those of the arbitrary constraint model. There are some additional variables for each move, and they are existentially quantified directly after m^i. They are listed below. Also there are more l^i_z variables in this model than previously, one for each possible line whereas previously there was one for each row, column and diagonal on the board.

- Boolean variables $\exists mh^i_c$ (move-here) representing whether the move i was made in column c.
- Boolean variables $\exists pos^i_{r,c}$ representing the position of the empty slots in the column. $pos^i_{r,c}$ is 1 if slot r is free in column c.

Again the constraints are given for a single move $i \in \{1 \dots row \times col\}$. They are grouped together into seven sets, each of which is interchangeable with the appropriate set from the other model. An implication $A \wedge B \wedge \dots \Rightarrow C$ is rearranged as $[\neg A \vee \neg B \vee \dots \vee C] \Leftrightarrow 1$ to match the primitives. Also, conjunction $A \wedge B \wedge \dots \Leftrightarrow C$ is rearranged as $\neg A \vee \neg B \vee \dots \Leftrightarrow \neg C$ to match the primitives.

(1) If i is even, for each column c, $[gamestate^{i-1} = nil \wedge h^{i-1}_c \neq row \wedge u^i = col] \Rightarrow m^i = col$.

(2) If i is even, a black counter is placed on the board: for all columns c, variable mh^i_c (move-here) is set according to whether the move is in this column: $line^{i-1} = 1 \vee h^{i-1}_c = row \vee m^i \neq c \Leftrightarrow mh^i_c \neq 1$

Also, for all columns c and rows r, the following four constraints are posted. If the column height at move $i-1$ is r, then the rest of the column is filled appropriately: $h^{i-1}_c = r-1 \Rightarrow pos^i_{r,c} = 1$ and $pos^i_{r,c} = 1 \Leftrightarrow [b^i_{r,c} \neq red \wedge b^i_{r+1,c} = nil \wedge b^i_{r+2,c} = nil \wedge \dots \wedge b^i_{row,c} = nil]$. The move variable m^i is connected to $b^i_{r,c}$: $[mh^i_c = 1 \wedge h^{i-1}_c = r-1] \Rightarrow b^i_{r,c} = black$ and $[mh^i_c \neq 1 \wedge h^{i-1}_c = r-1] \Rightarrow b^i_{r,c} = nil$.

If i is odd, a similar set of constraints is posted with red and $black$ substituted for each other.

(3) Map pieces from the board at $i-1$ to the board at move i: for all rows and columns r, c, $b^{i-1}_{r,c} = red \Rightarrow b^{i}_{r,c} = red$ and $b^{i-1}_{r,c} = black \Rightarrow b^{i}_{r,c} = black$.

(4) Link height and board state: for each column c and $r \in 1 \dots (row+1)$, $b^{i}_{r-1,c} \neq nil \wedge b^{i}_{r,c} = nil \Rightarrow h^{i}_{c} = r-1$.

(5) Detect lines: each set of four board variables that form a line (such as $b^{i}_{1,1}, b^{i}_{2,1}, b^{i}_{3,1}, b^{i}_{4,1}$) is given a unique number z and I refer to them as $b^{z}_{1\dots4}$. If i is odd, for all z, $line^{i-1} = 1 \vee b^{z}_{1} \neq red \vee b^{z}_{2} \neq red \ \vee b^{z}_{3} \neq red \vee b^{z}_{4} \neq red \Leftrightarrow l^{i}_{z} \neq 1$. In words, $l^{i}_{z} = 1$ iff $b^{z}_{1\dots4}$ are all red, and there is no line at the previous move. $line^{i-1}$ is included so that when a line is found, all future l_z variables are set to 0 and cannot be branched.

If i is even, similar constraints are posted with $black$ substituted for red.

(6) Connect the main line variable $line^{i}$ with the l^{i}_{z} variables, where $y = \max(z)$: $line^{i-1} \vee l^{i}_{1} \vee l^{i}_{2} \vee \cdots \vee l^{i}_{y} \Leftrightarrow line^{i}$.

(7) Set the $gamestate$ variables: if i is odd, the following four constraints are posted.
$gamestate^{i-1} = red \Rightarrow gamestate^{i} = red$,
$gamestate^{i-1} = black \Rightarrow gamestate^{i} = black$,
$gamestate^{i-1} = nil \wedge line^{i} = 1 \Rightarrow gamestate^{i} = red$,
$gamestate^{i-1} = nil \wedge line^{i} = 0 \Rightarrow gamestate^{i} = nil$.

If i is even, similar constraints are posted with $black$ and red substituted for each other.

In some of these constraints, variables are referred to with indices which are out of range, for example $gamestate^{0}$. These are set as follows. $gamestate^{0} = nil$, $l^{0} = nil$, $b^{0}_{r,c} = nil$, $b^{i}_{0,c} = red$ (chosen arbitrarily, cannot be nil), $b^{i}_{row+1,c} = nil$, $h^{0}_{c} = 0$. The red player must win, so $gamestate^{row \times col} = red$. Symmetry is broken as in the arbitrary constraint model.

5.3.1.1. *Comparing efficiency using Connect 4.* For each of the 7 sets of constraints, I investigate how the level of consistency achieved, and the overall performance, differs from the arbitrary constraint model. Stronger consistency implies a smaller (or equal) number of nodes in the search tree. Efficiency is measured by time to solve the instance.

Hypothesis. For each set of constraints, the new model is more efficient, despite any differences in the level of consistency.

Method. I take the arbitrary constraint model and construct seven hybrid models by substituting each set of constraints by the corresponding set from the disjunction model. These are compared with the arbitrary constraint model and disjunction model, using the *search* algorithm in chapter 3 and the pure value rule applied to universal variables only. For the arbitrary constraints, the SQGAC algorithm was used. The same value selection heuristic was applied as in chapter 4.

I chose the parameters $col = 5$ and $row = 4$. The arbitrary constraint model takes approximately 4 minutes to solve with SQGAC with these parameters. The machine and other experimental details are the same as in chapter 4.

Results. Table 5 contains node counts and run times for the arbitrary constraint model, the seven hybrid models and the disjunction model. For constraint set 5 (detect lines), the disjunction model is unambiguously better since it has a lower node count and lower time per node. For sets 1, 3, 4 and 6, the disjunction model is better in terms of the time per search node, although for set 4 the search time difference is unlikely to be significant. This leaves sets 2 and 7 (place pieces on board, and constrain $gamestate$ variables). For both of these, they are weaker in terms of consistency but more efficient per node. In line with these results, the disjunction model explores more nodes than the arbitrary constraint model, but is more than an order of magnitude faster per node, and takes around a quarter of the time to solve overall. In the trade-off between propagation strength and efficiency, the disjunction model wins because its greater efficiency outweighs the effect of its weaker propagation.

The results do not support the hypothesis because of constraint sets 2 and 7.

5.3.1.2. *Scaling behaviour with Connect 4.* I informally investigate whether the disjunction model scales better than the arbitrary constraint model. Table 6 shows (in order of increasing search time) four sets of parameters, the search time and number of nodes for both models, and the ratios between the models. The disjunction model is faster for all five parameter sets, but the search time ratio is increasing, suggesting that the arbitrary constraint model may be faster at some

Set of constraints substituted	Nodes and time			
	Nodes	Setup time (s)	Search time (s)	Search time per node (ms)
None	47712	0.324	243.709	5.107
1	47712	0.529	238.306	4.994
2	121739	0.447	457.507	3.758
3	47712	0.478	209.293	4.387
4	47712	0.467	243.135	5.095
5	46557	0.393	168.260	3.614
6	47712	0.569	223.023	4.674
7	62258	0.561	284.760	4.574
All	168485	0.004	63.573	0.377

TABLE 5. Comparison of Connect 4 models where $row = 4$, $col = 5$

Board size		Nodes and time (s)					
col	*row*	Model	Nodes	Setup time	Search time	Search time ratio	Nodes ratio
4	4	Arbitrary	1692	0.395	5.812		
		Disjunction	4196	0.008	1.054	0.181	2.480
4	5	Arbitrary	7744	0.794	41.104		
		Disjunction	20856	0.004	9.130	0.222	2.693
5	4	Arbitrary	47712	0.324	243.709		
		Disjunction	168485	0.004	63.573	0.261	3.531
5	5	Arbitrary	1027266	1.221	6204.549		
		Disjunction	2689288	0.006	1749.937	0.282	2.618
5	6	Arbitrary	4874626	12.805	54606.482		
		Disjunction	22197560	0.016	16012.498	0.293	4.554

TABLE 6. Comparison of Connect 4 models for various parameters

point. The nodes ratio does not show a clear pattern. Connect 4 is unsatisfiable for all five sets of parameters.

5.3.1.3. *Comparing reified disjunction with Bordeaux's primitives.* The hypothesis is that the reified disjunction provides significantly stronger consistency than Bordeaux's primitives. Various examples of this were given in section 5.1.1. The aim here is to see if this also applies in the context of the Connect 4 model.

I will assume that when all variables represented in an expression $\mathcal{E}$ are existential, the consistency achieved on the decomposition is equivalent to applying SQGAC to the expression directly.

(This only applies when the expression contains no repeated variables.) There is only one expression in the Connect 4 disjunction model which contains a universal variable. This is the expression connecting the universal move variable u^i to the existential move variable for the same move, m^i. In the disjunction model, this is expressed with a single reified disjunction. The original expression followed by its expression as a reified disjunction (equation (18)) and the decomposition (equation (19)) is shown below.

$$\exists gamestate^{i-1}, h_c^{i-1}, \forall u^i, \exists m^i :$$
$$[gamestate^{i-1} = nil \wedge h_c^{i-1} \neq row \wedge u^i = col] \Rightarrow m^i = col$$

$$\exists gamestate^{i-1}, h_c^{i-1}, \forall u^i, \exists m^i :$$
$$[gamestate^{i-1} \neq nil \vee h_c^{i-1} = row \vee u^i \neq col \vee m^i = col \Leftrightarrow 1 \tag{18}$$

$$\exists gamestate^{i-1}, h_c^{i-1}, \forall u^i, \exists m^i, t_1, t_2, t_3, t_4, t_5, t_6 :$$
$$(gamestate^{i-1} \neq nil \Leftrightarrow t_1) \wedge (h_c^{i-1} = row \Leftrightarrow t_2) \wedge (u^i \neq col \Leftrightarrow t_3) \tag{19}$$
$$\wedge (m^i = col \Leftrightarrow t_4) \wedge (t_1 \vee t_2 \Leftrightarrow t_5) \wedge (t_3 \vee t_5 \Leftrightarrow t_6) \wedge (t_4 \vee t_6 \Leftrightarrow 1)$$

Hypothesis. When solving Connect 4 with the decomposition, the solver will explore more nodes and take more time than with the integer reified disjunction primitive.

Method. I used the disjunction model, and simply replaced constraint set 1 with the decomposition in equation (19). SQGAC is enforced on the ternary primitive constraints using the reified disjunction algorithm. All experimental details are the same as for the previous experiments. As before, the pure value rule is applied to universal variables only.

Board size		Nodes and time (s)			
col	row	Model	Nodes	Setup time	Search time
4	4	Disjunction	4196	0.008	1.054
		Disjunction+Bordeaux	26359046	0.007	6213.737

TABLE 7. Comparison of reified disjunction with Bordeaux's decomposition on Connect 4

Results. Table 7 shows the search time and number of nodes for the two models, for a 4×4 board. The experiment took so long for this board size that I did not run it for larger sizes. The decomposition gives surprisingly bad results at this size.

The disjunction model is set up so that the pure value rule can work, but this does not carry through to the decomposition. Whenever the column is full or the game is over, the constraint (equation (18)) in the disjunction model becomes trivially true and all values of u^i become pure with respect to the constraint. In the decomposition, t_1 or t_2 is set to 1 and hence t_5 and t_6 are set to 1. t_3 and t_4 remain uninstantiated, and none of the values of u^i become pure. I can see no way of avoiding this problem within Bordeaux's method.

Without the effect of the pure value rule, the *search* algorithm will assign u^i a cheating move, then explore a subtree where m^i is assigned a legal move and then the search proceeds as usual, from move $i+1$. The size of the subtree can be very large, depending on the number of remaining moves and the size of the board. The size of the subtree scales up exponentially as the board size is increased, therefore the problem worsens.

The other possible problem is that consistency is weaker in the decomposition. For example this could arise if $gamestate^{i-1} = nil$ and $m^i \neq col$, in which case $h_c^{i-1} = row$ is inferred by the reified disjunction, and nothing is inferred by the decomposition. However I do not know if a situation like this occurs during search.

5.3.1.4. *The effect of the pure value rule.* The disjunction model is designed such that the pure value rule can prune certain values (corresponding to invalid moves) of universal variables. Disabling the pure value rule causes Queso to explore a much larger search tree, as shown in table 8. Experimental details are the same as for the previous experiments.

Board size		Nodes and time (s)			
col	*row*	Model	Nodes	Setup time	Search time
4	4	Disjunction with PV	4196	0.008	1.054
		Disjunction without PV	92213	0.014	22.066
4	5	Disjunction with PV	20856	0.004	9.130
		Disjunction without PV	1042992	0.010	355.298
5	4	Disjunction with PV	168485	0.004	63.573
		Disjunction without PV	28179488	0.017	9425.100

TABLE 8. The effect of the pure value rule with Connect 4

5.4. Summary

The aim of this chapter is to bring strong reasoning on logic constraints (such as QBF clauses) to QCSP. I have presented a new algorithm for logic constraints in QCSP, and evaluated it on a game, showing that there are QCSP instances where the new algorithm performs significantly better than the SQGAC algorithm in chapter 4. I have also shown that the new algorithm performs significantly better than the decomposition approach of Bordeaux et al. [**19**], when used on the game of Connect 4.

CHAPTER 6

Bounds consistency and the sum constraint

6.1. Introduction

In classical CSP, numerical constraints and bounds consistency are commonplace. In section 2.1.2.3 of the literature review I cover three notions of bounds consistency for CSP [**30**], and show the derivation of a simple propagation algorithm which enforces one of the notions of bounds consistency [**76**].

The presence of quantifiers gives an opportunity to strengthen the definition of bounds consistency. In this chapter I define a new bounds consistency notion for QCSP, and propose a propagation algorithm for the sum constraint, proving that the algorithm enforces the new consistency notion. This is compared with the existing work in QCSP by Bordeaux [**19**].

6.1.1. Motivating examples. The only other work on numerical constraints in QCSP is Bordeaux et al. [**19**, **22**] (reviewed in chapter 2 section 2.3.3). In chapter 4, the following example is given.

$$\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\} \; : \; x_1 - x_2 + x_3 = 3$$

It is false because $x_1 = 4$, $x_2 = 3$ cannot be extended to a satisfying tuple. In chapter 4 section 4.2, it is shown that this expression can be decomposed in three ways in Bordeaux's framework, but the propagation rules are unable to detect the inconsistency [**19**].

When the only available primitives are $a+b=c$ and $a=-b$, the variables x_1 and x_2 cannot be contained in the same primitive constraint. This difficulty could be solved by introducing negation to the addition primitive.

The following example (equation (20)) is trivially false since all three variables are universal. Since the three variables are interchangeable, all three decompositions are equivalent. When the expression is decomposed (as in equation (21)), the propagation rules are insufficient to detect the inconsistency.

(20)
$$\forall x_1, x_2, x_3 \in \{0,1\} \ : \ x_1 + x_2 + x_3 = 2$$

(21)
$$\forall x_1, x_2, x_3 \in \{0,1\} \exists t_1 \in \{0,1,2\} \ : \\ x_1 + x_2 = t_1, t_1 + x_3 = 2$$

A sum primitive with unlimited length would resolve this problem.

6.1.2. Proposed sum primitive. The proposed primitive, shown below, introduces constants $c_i \in \mathbb{Z} \setminus \{0\}$ and can be of any length. This allows it to be applied directly to both the examples above.

$$\sum_{r}^{i=1} c_i x_{k_i} = 0$$

I refer to this as a sum of terms $c_i x_{k_i}$, and each term as being universal or existential depending on the quantification of the variable in the term. As in definition 3.2.2 in chapter 3, for constraint C_k the sequence of variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ is a subsequence of the variables $\mathcal{X}$ in the QCSP instance, and therefore the numbering $1 \ldots r$ is consistent with the quantifier sequence.

6.2. Definition of bounds consistency

Three definitions of bounds consistency in CSP are given by Choi et al. [**30**] and reviewed in chapter 2 section 2.1.2.3. I generalize bounds($\mathbb{R}$) consistency for QCSP. Briefly, a constraint C_k is bounds($\mathbb{R}$) consistent iff a real-valued solution to the constraint can be constructed for each bound of each variable, where the solution contains only values within the existing bounds of the variables.

The definition of Qbounds($\mathbb{R}$) consistency depends on relaxing some properties of the constraint. The local instance $\mathcal{P}_k$ containing just the constraint C_k and its variables $\mathcal{X}_k$ (defined in chapter 3 section 3.2.4) is relaxed in three ways, to form r relaxed instances, one for each variable, referred to as $\mathcal{P}_{k_1} \dots \mathcal{P}_{k_r}$. The idea is that new bounds for x_{k_i} will be found by a computation on the instance $\mathcal{P}_{k_i}$. The reason to have a relaxed instance for each variable x_{k_i} is that the relationship between x_{k_i} and outer universals is different to that with inner universals.

DEFINITION 6.2.1. Relaxed QCSP instance $\mathcal{P}_{k_i}$ derived from $\mathcal{P}_k$ w.r.t. variable x_{k_i}.

For $\mathcal{P}_{k_i} = \langle \mathcal{X}_k, \mathcal{D}_{k_i}, \{C_k\}, \mathcal{Q}_{k_i} \rangle$, the set of variables $\mathcal{X}_k$ is shared with $\mathcal{P}_k$. Some of the quantifiers are changed: $\mathcal{Q}_{k_i} = \langle \exists x_{k_1} \dots \exists x_{k_{i-1}} Q_{k_i} x_{k_i} \dots Q_{k_r} x_{k_r} \rangle$ and the individual quantifier symbols are referred to as $Q_1^{k^i} \dots Q_r^{k^i}$. The domains of $\mathcal{P}_{k_i}$ are derived from the bounds of the variables: $\mathcal{D}_{k_i} = \langle D_1^{k^i} \dots D_r^{k^i} \rangle$. Existential variables can take any real value between their bounds: $\forall j \ : \ (Q_j^{k^i} = \exists) \Rightarrow (D_j^{k^i} = \{z | z \in \mathbb{R} \wedge \underline{x_{k_i}} \leq z \leq \overline{x_{k_i}}\})$, and universal variables can only take their two bounds: $\forall j \ : \ (Q_j^{k^i} = \forall) \Rightarrow (D_j^{k^i} = \{\underline{x_{k_i}}, \overline{x_{k_i}}\})$.

In words, the following three relaxations are applied.

- For each variable x_{k_i}, all outer universals $Q_{k_j} = \forall$ where $j < i$ are changed to existentials to form the instance $\mathcal{P}_{k_i}$.
- Existential variables in $\mathcal{P}_{k_i}$ may take any real value between their upper and lower bounds.
- Universal variables in $\mathcal{P}_{k_i}$ are relaxed by discarding all values except the upper and lower bounds.

In the context of bounds consistency, set C_k^S includes all real-valued solutions to the constraint which are within the bounds of the variables. This is in contrast to the use of C_k^S in other chapters where it only contains solutions which are within the domains of the variables (i.e. $C_k^S \subseteq D_{k_1} \times \dots \times D_{k_r}$).

The definition of winning strategy (definition 3.2.8 in chapter 3) is central to the definition of Qbounds($\mathbb{R}$) consistency. I will call a winning strategy for $\mathcal{P}_{k_i}$ a *real-valued partial winning strategy* to avoid confusion with a winning strategy for $\mathcal{P}_k$. The definition of Qbounds($\mathbb{R}$) consistency depends on the existence of real-valued partial winning strategies containing each bound of each variable.

DEFINITION 6.2.2. Qbounds($\mathbb{R}$) consistency of constraint C_k

A constraint C_k with variables $\mathcal{X}_k = \langle x_{k_1}, \dots, x_{k_r} \rangle$ is Qbounds($\mathbb{R}$) consistent iff for each variable x_{k_i} and each bound $b_i \in \{\underline{x_{k_i}}, \overline{x_{k_i}}\}$ there exists a real-valued partial winning strategy for $\mathcal{P}_{k_i}$, $S = \{s_j | Q_j^{k^i} = \exists\}$ such that S contains the bound: $\exists \tau : \tau \in \text{sce}(S) \wedge \tau_i = b_i$.

For example, the constraint below is Qbounds($\mathbb{R}$) consistent but not SQGAC.

$$\forall x_{k_1} \in \{-10, 2, 10\} \exists x_{k_2} \in \{-10, -4, 10\} : x_{k_1} + x_{k_2} = 0$$

Consider the real-valued partial winning strategy for $\mathcal{P}_{k_1}$, $S = \{s_2\}$ where $s_2(\langle -10 \rangle) = 10$, $s_2(\langle 10 \rangle) = -10$. S supports both bounds of x_{k_1} because $\text{sce}(S) = \{\langle -10, 10 \rangle, \langle 10, -10 \rangle\}$. For x_{k_2}, in the relaxed problem $\mathcal{P}_{k_2}$ both variables are existential. To support the lower bound of x_{k_2}, a winning strategy can be constructed with one scenario: $\langle 10, -10 \rangle$, and similarly for the upper bound.

The following constraint is not Qbounds($\mathbb{R}$) consistent but it is bounds($\mathbb{R}$) consistent.

$$\exists x_{k_1} \in \{1, 2\} \forall x_{k_2} \in \{1, 2\} \exists x_{k_3} \in \{1, 2\} : x_{k_1} + x_{k_2} + x_{k_3} = 5$$

The lower bound of x_{k_1} is not supported by a real-valued partial winning strategy for $\mathcal{P}_{k_1}$, because $x_{k_1} \mapsto 1$ and $x_{k_2} \mapsto 1$ are not compatible, and in $\mathcal{P}_{k_1}$, x_{k_2} is universal. To establish Qbounds($\mathbb{R}$) consistency, value 1 is removed from x_{k_1}.

In the case where all variables in the constraint are existential, Qbounds($\mathbb{R}$) consistency is equivalent to bounds($\mathbb{R}$) consistency. All variables of all relaxed instances $\mathcal{P}_{k_i}$ are existential, therefore a winning strategy for $\mathcal{P}_{k_i}$ can be constructed from a single solution $\tau \in C_k^S$ where $\forall j : \tau_j \in \mathbb{R}, \underline{x_{k_j}} \leq \tau_j \leq \overline{x_{k_j}}$. Therefore the definition of Qbounds($\mathbb{R}$) consistency requires a real-valued solution for each bound of each variable. This is identical to the definition of bounds($\mathbb{R}$) consistency in chapter 2 section 2.1.2.3.

Finally, it is straightforward to define Qbounds($\mathbb{Z}$) and Qbounds($\mathbb{D}$) consistency in a similar way to Qbounds($\mathbb{R}$) consistency. This is done by adapting the definition of $\mathcal{P}_{k_i}$ such that for Qbounds($\mathbb{Z}$) consistency each domain is a subset of the integers, $\forall j : D_j^{k^i} \subseteq \mathbb{Z}$, or for Qbounds($\mathbb{D}$) consistency each domain is a subset of the equivalent domain in $\mathcal{P}$, $\forall j : D_j^{k^i} \subseteq D_{k_j}$.

6.3. Propagation rules

Rearranging the sum expression for a variable x_{k_i} gives the following.

$$c_i x_{k_i} = -\sum\{c_j x_{k_j} | j \neq i\} \tag{22}$$

In CSP, the maximum and minimum values of the right-hand side of this expression could be used to compute new bounds for x_{k_i}. A derivation of bounds consistency rules in CSP is shown in chapter 2 section 2.1.2.3. However in QCSP it is slightly more complex.

For each variable $x_{k_i} \in \mathcal{X}_k$, there are two rules to update the upper and lower bounds. In these, the function max$(c_j x_{k_j})$ is $c_j \overline{x_{k_j}}$ if c_j is positive, and $c_j \underline{x_{k_j}}$ if c_j is negative. min is defined similarly. The two rules below compute new bounds on $c_i x_{k_i}$. The first provides a lower bound for $c_i x_{k_i}$, which is a lower bound on x_{k_i} if c_i is positive, and an upper bound if c_i is negative. This is referred to as rule 1, shown in equation (23).

$$\begin{aligned} c_i x_{k_i} \quad \geq \quad & -\sum\{\text{max}(c_j x_{k_j}) | (j < i) \vee (Q_j = \exists)\} \\ & -\sum\{\text{min}(c_j x_{k_j}) | (j > i) \wedge (Q_j = \forall)\} \end{aligned} \tag{23}$$

The only difference between this and the equivalent CSP rule is when dealing with variables x_{k_j} which are universal, and quantified inside x_{k_i} ($j > i$). In this case, the new bound of x_{k_i} must be consistent with *all* values of x_{k_j}. Because of the nature of the sum constraint, it suffices to use a single value for x_{k_j}. The value is the one which most restricts $c_i x_{k_i}$, which in this case is the value which minimizes $c_j x_{k_j}$.

The rule to compute the new upper bound of $c_i x_{k_i}$ is very similar to equation (23), with min and max transposed (rule 2).

$$\begin{aligned} c_i x_{k_i} \quad \leq \quad & -\sum\{\min(c_j x_{k_j}) | (j < i) \vee (Q_j = \exists)\} \\ & -\sum\{\max(c_j x_{k_j}) | (j > i) \wedge (Q_j = \forall)\} \end{aligned} \tag{24}$$

From these two rules, new bounds of x_{k_i} are computed by dividing by c_i. The pairs of new bounds are referred to as $\underline{b_i}$ and $\overline{b_i}$ where $\underline{b_i} \leq \overline{b_i}$. If $c_i < 0$ then $\underline{b_i}$ is derived from $\overline{c_i x_{k_i}}$, and $\overline{b_i}$ is derived from $\underline{c_i x_{k_i}}$. These two rules are applied to all variables regardless of quantification.

For each universal variable there is a single rule. To illustrate this rule, consider the expression below.

$$\exists x_{k_1}, x_{k_2}, x_{k_3} \in \{-2, -1, 0, 1, 2\} \forall x_{k_4} \in \{0, 1, 2\} \exists x_{k_5} \in \{-1, 0\} \; : \\ x_{k_1} + x_{k_2} + x_{k_3} + x_{k_4} + x_{k_5} = 0$$

This expression is false because (whatever the instantiation of x_{k_1}, x_{k_2} and x_{k_3}) x_{k_4} has a wider interval than x_{k_5}, so there can not exist a value in D_{k_5} for each value in D_{k_4}. Rules 1 and 2 applied to all variables do nothing, so a third rule is required. In more general terms, the range of a universal term must be smaller or equal to the combined range of all terms of variables quantified inside it. A sum constraint is false if the range of any universal variable is too great given the ranges of variables quantified inside. The range is defined on terms, rather than variables: range$(c_j x_{k_j})$ $=$ max$(c_j x_{k_j})$ $-$ min$(c_j x_{k_j})$. If the universal variable in question is x_{k_i}, the combined range of all variables $x_{k_{j>i}}$ must be calculated. If x_{k_j} is existential, their range is summed, but if it is universal, range$(c_j x_{k_j})$ is *subtracted* because the values of x_{k_i} must be compatible with both bounds of x_{k_j}. This is illustrated with the following example.

$$\exists x_{k_1}, x_{k_2}, x_{k_3} \in \{-2,-1,0,1,2\} \forall x_{k_4} \in \{0,1,2\} \exists x_{k_5} \in \{-1,0\}$$
$$\forall x_{k_6} \in \{0,1\} \exists x_{k_7} \in \{-1,0\} \; : \; x_{k_1} + x_{k_2} + x_{k_3} + x_{k_4} + x_{k_5} + x_{k_6} + x_{k_7} = 0$$

Working from the innermost variable outwards, x_{k_7} and x_{k_6} both have a range of 1. In any winning strategy, when x_{k_6} is set to 1 then x_{k_7} must be set to -1, and when x_{k_6} is set to 0 then x_{k_7} must be set to 0 since other variables are fixed by the winning strategy. Therefore the sub-expression $x_{k_6} + x_{k_7}$ always takes value 0 and has range 0 since the universal variable counteracts the existential. Therefore the range of $x_{k_5} + x_{k_6} + x_{k_7}$ is range$(c_5 x_{k_5})$ − range$(c_6 x_{k_6})$ + range$(c_7 x_{k_7}) = 1$.

The complete rule for universal variable x_{k_i} is given below (rule 3).

$$\begin{aligned} & \sum \{\text{range}(c_j x_{k_j}) | (j > i) \wedge (Q_{k_j} = \exists)\} \\ (25) \qquad & - \sum \{\text{range}(c_j x_{k_j}) | (j \geq i) \wedge (Q_{k_j} = \forall)\} \quad \geq \quad 0 \end{aligned}$$

Rule 3 only ever fails or succeeds, never leads to pruning. If the inequality is not true, then the constraint is false.

6.3.1. Proof of correctness. I claim that applying the three rules in equations (23), (24) and (25) either detects falsity or enforces exactly Qbounds($\mathbb{R}$) consistency. The rules are applied to exhaustion, with the computed bounds rounded to an integer (the upper bound rounded down, and the lower bound rounded up), before being applied to variable domains.

The proof is by constructing sets of scenarios with certain properties, such that they correspond to real-valued partial winning strategies for $\mathcal{P}_{k_i}$. First, for each variable x_{k_i}, two sets are constructed. In the first set each tuple contains the most recently computed lower bound $\underline{b_i}$, and similarly for the second set and $\overline{b_i}$. Second, $\underline{x_{k_i}} \geq \underline{b_i}$ and $\overline{x_{k_i}} \leq \overline{b_i}$ (the new bounds are not necessarily tight), so a further two sets are constructed where the tuples contain $\underline{x_{k_i}}$ and $\overline{x_{k_i}}$ respectively. It is shown that these sets have the necessary properties to be sets of scenarios of real-valued partial winning strategies of $\mathcal{P}_{k_i}$, and hence the definition of Qbounds($\mathbb{R}$) consistency is met. (Henceforth real-valued partial winning strategies for $\mathcal{P}_{k_i}$ will be referred to as winning strategies for $\mathcal{P}_{k_i}$ for brevity.)

The proof below assumes that the constants c_i are positive, for simplicity. For terms with negative constants, all references to upper and lower bounds, and increasing or decreasing, are simply reversed.

For the following proof, I observe that the set of scenarios $\text{sce}(S)$ of a winning strategy S for $\mathcal{P}_{k_i}$ has the following properties, following directly from the definitions in chapter 3. These properties are complete in the sense that a set of scenarios meeting these properties can be combined into a winning strategy. This follows by examination of the properties and the definitions of winning strategy and scenario (definitions 3.2.8 and 3.2.7 in chapter 3).

(1) $\forall \tau \,:\, \tau \in \text{sce}(S) \Rightarrow \tau \in C_k^S$ (definition 3.2.8 in chapter 3).

(2) For any two distinct tuples τ and τ' in $\text{sce}(S)$, the leftmost value to differ corresponds to a universal variable: $\forall j \,:\, \tau_j \neq \tau'_j \Rightarrow [\exists l : (l \leq j) \wedge (Q_l^{k^i} = \forall) \wedge (\tau_l \neq \tau'_l)]$. This follows from the definition of strategy as a family of functions determining the values for existential variables from those for outer universals (definition 3.2.6 in chapter 3).

(3) There is a tuple in $\text{sce}(S)$ for each combination of values of universal variables (definition 3.2.7 in chapter 3).

The following proof constructs sets of scenarios with the three properties, and as part of this construction it is required to construct a tuple by adapting another tuple. The proof requires adapting a tuple τ to σ by increasing one value at position i, and decreasing others to restore the sum. The lex-least adapted tuple σ is required, and for indices $j \in \{1 \ldots \alpha\}$, $\sigma_j = \tau_j$. In the following, set realsol_k contains all real solutions to the constraint, regardless of the domains, therefore $C_k^S \subset \text{realsol}_k$. Lex-least refers to the least tuple in the lexicographic ordering. The lex-least adapted tuple is defined below (definition 6.3.1).

DEFINITION 6.3.1. Lex-least adapted tuple σ from τ with parameters α, i and v_i.

The tuple σ is the lex-least tuple subject to the following:

Algorithm 32 Algorithm to construct the lexicographically least adapted tuple

```
procedure lexLeastAdapted(tuple $\tau$, value $v_i$, index $i$, index $\alpha$): tuple $\sigma$
Construct $\sigma$ by adapting $\tau$, using value $v_i$ at index $i$ and changing no other values in the range
$\tau_1 \dots \tau_\alpha$. The values of $\sigma$ are bounded as follows:
$\sigma_i = v_i$ where $v_i \geq \tau_i$,
$\forall j \, : \, (Q_j^{k^i} = \forall \vee j \leq \alpha) \Rightarrow (\sigma_j = \tau_j)$,
$\forall j \, : \, (Q_j^{k^i} = \exists) \Rightarrow (\underline{x_{k_j}} \leq \sigma_j \leq \tau_j)$.
Within these bounds, values for existential variables are set as follows. For each $j \in \{\alpha+1 \dots r\}$
and where $Q_j^{k^i} = \exists$ in ascending order: there is a sequence of 0 or more indices where $\sigma_j = \underline{x_{k_j}}$,
followed by one where $\underline{x_{k_j}} \leq \sigma_j \leq \tau_j$, followed by 0 or more $\sigma_j = \tau_j$ such that $\sigma \in \text{realsol}_k$.
```

- $\forall j \, : \, (Q_j^{k^i} = \forall \vee j \leq \alpha) \Rightarrow (\sigma_j = \tau_j)$,
- $\forall j \, : \, (Q_j^{k^i} = \exists) \Rightarrow (\underline{x_{k_j}} \leq \sigma_j \leq \tau_j)$,
- $\sigma_i = v_i$,
- $\sigma \in \text{realsol}_k$.

A procedure for constructing σ in the context of $\mathcal{P}_{k_i}$ is given in algorithm 32. An important property of algorithm 32 is that the changes between σ and τ are concentrated to the left of the tuple (except for those positions which must be equal). For any two indices j, l where $i \neq j$, $i \neq l$, $j < l$, $\sigma_j \neq \tau_j$ and $\sigma_l \neq \tau_l$, then $\sigma_j = \underline{x_{k_j}}$. In words, the value at index j is reduced as far as possible before changing the value at index l.

THEOREM 6.3.2. *A sum primitive constraint C_k is Qbounds($\mathbb{R}$) consistent after applying the propagation rules to exhaustion.*

PROOF. For each variable x_{k_i}, two non-integral bounds $\underline{b_i}$ and $\overline{b_i}$ are computed by the final applications of rules 1 and 2. The lower bound rule (rule 1, equation (23)) implicitly constructs a tuple $\tau \in \text{realsol}_k$. In the context of $\mathcal{P}_{k_i}$, $\tau_i = \underline{b_i}$, $\forall j \, : \, (Q_j^{k^i} = \forall) \Rightarrow (\tau_j = \underline{x_{k_j}})$ and $\forall j \, : \, (Q_j^{k^i} = \exists) \Rightarrow (\tau_j = \overline{x_{k_j}})$. τ is adapted to form a set of scenarios $\mathcal{S}_1$ as follows.

For the outermost universal x_{k_j} where $j > i$ in $\mathcal{P}_{k_i}$, by rule 3 (equation (25)) $\underline{x_{k_j}}$ can be replaced with $\overline{x_{k_j}}$ in τ. Algorithm 32 is used to construct a new tuple τ' where $\tau'_j = \overline{x_{k_j}}$, with parameter $\alpha = j$, with the result that $\tau' \in \text{realsol}_k$. Rule 3 guarantees that it is possible to restore the sum to 0 when $\alpha = j$, while allowing the same to be done for all other universal variables quantified after x_{k_j} (since the range of all such variables is subtracted). Using the lex-least tuple τ' ensures that the leftmost values after j are changed to restore the sum. This ensures that the procedure can be repeated for other universal variables quantified after x_{k_j}.

This procedure is repeated on both τ and τ' for the next universal after x_{k_j} in the quantifier sequence, producing four tuples, and so on to the final universal in the quantifier sequence. This yields a set of scenarios $\mathcal{S}_1$ where $\tau_i = \underline{b_i}$. By symmetry there exists a second set $\mathcal{S}_2$ of scenarios where $\tau_i = \overline{b_i}$, derived from the upper bound rule (equation 24). Now it can be seen that both $\mathcal{S}_1$ and $\mathcal{S}_2$ each satisfy property 2 but not necessarily 1 and 3.

It is possible to construct a third set of scenarios $\mathcal{S}_3$ each containing some intermediate value v_i ($\underline{b_i} \leq v_i \leq \overline{b_i}$) at position i. This is done by adapting all the tuples in set $\mathcal{S}_1$. For each tuple $\tau \in \mathcal{S}_1$, there exists $\sigma \in \mathcal{S}_3$ with following properties. All values for universal variables are the same: $\forall j\ :\ (Q_j^{k^i} = \forall \wedge j \neq i) \Rightarrow (\tau_j = \sigma_j)$, $\sigma_i = v_i$, and since $\sigma_i \geq \tau_i$, other values in σ must be adjusted downwards such that $\sigma \in \text{realsol}_k$. This must be possible since there is a tuple $\gamma \in \mathcal{S}_2$ such that $\gamma_i \geq \sigma_i$ and values for universal variables are the same in γ and τ. The lex-least σ is constructed by algorithm 32 using $\alpha = 0$ such that $\sigma_i = v_i$. For any two tuples τ and τ' in $\mathcal{S}_1$, which first differ at position j, two tuples σ and σ' are constructed with $\sigma_i = \sigma'_i = v_i$. From the algorithm, σ and σ' also first differ at position j, therefore set $\mathcal{S}_3$ must have property 2. $\mathcal{S}_3$ also has property 1 iff $\underline{x_{k_i}} \leq v_i \leq \overline{x_{k_i}}$.

If x_{k_i} is existential ($Q_i^{k^i} = \exists$), $\underline{\mathcal{S}_3}$ is constructed where $v_i = \underline{x_{k_i}}$, and $\overline{\mathcal{S}_3}$ where $v_i = \overline{x_{k_i}}$, and both these sets have the required three properties, and therefore winning strategies exist containing each bound.

If x_{k_i} is universal ($Q_i^{k^i} = \forall$), then $\underline{\mathcal{S}_3}$ is constructed where $v_i = \underline{x_{k_i}}$and $\overline{\mathcal{S}_3}$ where $v_i = \overline{x_{k_i}}$. $\underline{\mathcal{S}_3} \cup \overline{\mathcal{S}_3}$ has the three required properties, proving that a winning strategy exists containing both bounds of x_{k_i}. □

6.3.2. Comparison with previous work. In section 6.1.1 two examples were given where the propagation rules and decomposition of Bordeaux [**19**] are inadequate. They are reproduced below.

$$\forall x_1, x_2 \in \{3,4\}, \exists x_3 \in \{3,4\} \; : \; x_1 - x_2 + x_3 = 3$$

$$\forall x_1, x_2, x_3 \in \{0,1\} \; : \; x_1 + x_2 + x_3 = 2$$

Applying the new rules to these examples detects falsity for both. Applying rule 3 to x_1 in both examples is sufficient. These examples illustrate that decomposition damages propagation.

Bordeaux gives specialized propagation rules for the ternary sum primitive $x_1 + x_2 = x_3$ for eight different quantifier sequences. By inspection, none of Bordeaux's rules are individually stronger than the equivalent new rules. Therefore, even without the problems introduced by decomposition, Bordeaux's scheme must be equivalent or weaker than the new scheme.

The only advantage of Bordeaux's scheme is that it is potentially faster, since the rules are specialized and the propagator does no multiplication or division.

Benedetti et al. [**11**] described four ways of adapting an existing CSP propagation algorithm for QCSP (reviewed in chapter 2 section 2.3.2). If we had a bounds($\mathbb{R}$) propagator for long sum, this could be adapted as follows.

Existential analysis: This just applies bounds($\mathbb{R}$) consistency, and if a universal variable is pruned, it fails, hence existential analysis is much weaker than Qbounds($\mathbb{R}$) consistency.

Functional analysis: The long sum is not necessarily functional on any of its variables, so this cannot be applied.

Look-ahead analysis: Performing look-ahead analysis for all universal variables yields a propagator which takes exponential time.

Dual analysis: This form of analysis negates the constraint and inverts each quantifier without changing the order. The long sum becomes $\sum c_i x_{k_i} \neq 0$. Some form of propagation is performed on the negated constraint. Existential analysis with bounds($\mathbb{R}$) consistency could be applied, but the negated constraint is such that no values could be pruned.

The only form of analysis that can be usefully applied, look-ahead analysis, is comparable to the SQGAC and WQGAC-Schema algorithms in chapter 4. It is much more expensive than the Qbounds($\mathbb{R}$) algorithm presented here.

6.4. Implementation

The three rules can be implemented in a coarse-grained multi-pass propagator such that the propagator runs in $O(r)$ time, assuming arithmetic operations take constant time, and also discounting the effect of waking up other constraints when bounds are changed.

Algorithm 33 applies rules 1 and 2 once to each variable, in reverse quantification order. If the *upper total*, ut (i.e. the negative of the right hand side of equation 23) and *lower total*, lt (equation 24) were calculated individually for each variable, the result would be a $O(r^2)$ algorithm. Instead the ut and lt for variable x_{k_j} are computed from the values for $x_{k_{j+1}}$. ut and lt for x_{k_r} are computed on the second and third line in $O(r)$ time. By factoring out a common subformula between two formulas, a time factor of r is saved.

For each variable, *reviseBounds* is called to perform the division by c_i, rounding and applying to the variable. If this causes a domain to become empty or a universal to be pruned, *reviseBounds* returns false, therefore so does *propagateSum*.

In the same way, rule 3 is applied once to each universal variable, in reverse quantification order. The innerRange quantity (i.e. the right hand side of equation 25) for variable x_{k_j} is computed from the same quantity for $x_{k_{j+1}}$. If the inequality does not hold, the constraint is false so *propagateSum* returns false.

The procedure *reviseBounds* (algorithm 34) performs the division and rounding necessary to apply the new bounds to a variable. The upper total, lower total and variable index are passed in. For the upper bound ub, if it is less than $\overline{x_{k_i}}$ then the *excludeUpper* procedure is called to reduce $\overline{x_{k_i}}$. If this empties the domain, or if x_{k_i} is universal, then *excludeUpper* returns false, therefore *reviseBounds* returns false. The same is done for the lower bound lb.

Algorithm 33 Propagation algorithm for sum constraint

```
procedure propagateSum(): Boolean
$ut \leftarrow \sum_{j=1}^{r-1}$ max$(c_j x_{k_j})$ {Upper total, excluding max$(c_r x_{k_r})$}
$lt \leftarrow \sum_{j=1}^{r-1}$ min$(c_j x_{k_j})$ {Lower total, excluding min$(c_r x_{k_r})$}
if ¬reviseBounds$(lt, ut, r)$: return false {Apply rules 1 and 2 to $x_{k_r}$}
for $j$ in $(r-1)\dots 1$:
    if $Q_{k_{j+1}} = \forall$:
        $ut \leftarrow ut +$ min$(c_{j+1} x_{k_{j+1}})$ $-$ max$(c_j x_{k_j})$
        $lt \leftarrow lt +$ max$(c_{j+1} x_{k_{j+1}})$ $-$ min$(c_j x_{k_j})$
    else:
        $ut \leftarrow ut +$ max$(c_{j+1} x_{k_{j+1}})$ $-$ max$(c_j x_{k_j})$
        $lt \leftarrow lt +$ min$(c_{j+1} x_{k_{j+1}})$ $-$ min$(c_j x_{k_j})$
    if ¬reviseBounds$(lt, ut, j)$: return false {Apply rules 1 and 2 to $x_{k_j}$}
{Apply rule 3 to all universal variables}
innerRange $\leftarrow 0$
for $j$ in $r \dots 1$:
    if $Q_{k_j} = \forall$:
        innerRange $\leftarrow$ innerRange $-$ range$(c_j x_{k_j})$
        if innerRange $< 0$: return false
    else:
        innerRange $\leftarrow$ innerRange $+$ range$(c_j x_{k_j})$
return true
```

Algorithm 34 reviseBounds procedure

```
procedure reviseBounds$(lt, ut, i)$: Boolean
{Revise the bounds for variable $x_{k_i}$ using $lt$ and $ut$}
if $c_i > 0$:
    $ub \leftarrow \left\lfloor \frac{-lt}{c_i} \right\rfloor$
    $lb \leftarrow \left\lceil \frac{-ut}{c_i} \right\rceil$
else:
    $ub \leftarrow \left\lfloor \frac{-ut}{c_i} \right\rfloor$
    $lb \leftarrow \left\lceil \frac{-lt}{c_i} \right\rceil$
if $ub < \overline{x_{k_i}}$:
    if ¬excludeUpper$(nil, x_{k_i}, ub)$: return false
if $lb > \underline{x_{k_i}}$:
    if ¬excludeLower$(nil, x_{k_i}, ub)$: return false
return true
```

Assuming that the division and rounding take constant time, the time complexity of *revise-Bounds* is the same as that for *excludeUpper* (algorithm 7), which is proportional to the size of the wakeUp(x_{k_i}) set.

6.4.1. Implementation decisions. The algorithm above is coarse-grained and non-incremental (i.e. it stores no state between invocations). The alternative here is to build a fine-grained algorithm which is notified of each change to the bound of a variable, and which stores the ut and lt values between invocations. Considering each variable separately, each variable has a ut and lt value, as well as innerRange for universal variables. These 3 values would be maintained incrementally, and backtracked when the search backtracks. Unfortunately, a change in either bound of any other variable would require updates to ut or lt at least. It seems likely that maintaining all these values incrementally would be very inefficient.

The proposed algorithm is multi-pass rather than one-pass. When a bound is changed by *reviseBounds*, the long sum is re-queued so that it will be called again later, possibly after other constraints have been called. The alternative here is to repeat the entire contents of *propagateSum* until a fixed point is reached, avoiding the need to re-queue the constraint. It is difficult to say which alternative would be more efficient without experimentation.

6.4.2. Comparison with SQGAC. As a proof of concept, the algorithm described has been implemented and compared with the SQGAC propagation algorithm in chapter 4. The QCSP instances used contain one sum constraint, with six variables. For each variable, its quantifier is chosen randomly, with probability 0.8 for the existential quantifier. The domain of each variable is chosen with uniform probability from the following five: $\{3 \dots 12\}$, $\{0 \dots 9\}$, $\{-3 \dots 6\}$, $\{-9 \dots 0\}$ and $\{-12 \dots -3\}$. The constants c_1 to c_6 for the constraint are non-zero integers chosen with uniform probability in the range $-10 \dots 10$. 10,000 such instances were generated.

SQGAC propagation is able to determine the satisfiability of each instance at the root node, so no search is required. 4412 instances are satisfiable, and SQGAC propagation took 1045 seconds to decide all 10,000 instances. The bounds consistency algorithm was able to solve 4800 at the root node, and the remainder required search (algorithm 3). For the instances that required search, 249,763 nodes were explored in total. 7.86 seconds were required to decide all 10,000 instances. This is 0.8% of the time required by SQGAC propagation.

6.5. Summary

In this chapter I have defined a form of bounds consistency for quantified constraints. This is similar to bounds($\mathbb{R}$) consistency in that both definitions require an object to exist for each bound of each variable. In bounds($\mathbb{R}$) consistency, the object is a satisfying tuple, and in Qbounds($\mathbb{R}$) consistency it is a winning strategy. Both objects pertain to a relaxed form of the constraint, where some variables may take any real value between their bounds. This similarity suggests that Qbounds($\mathbb{R}$) consistency, or a close variant, may be as widely applicable as bounds($\mathbb{R}$) consistency is.

The propagation algorithm for long sum is significantly stronger than the existing ternary sum predicate, as well as being designed with efficiency in mind. It extends the input language of Queso significantly. The brief experiment shows that the bounds propagation algorithm can be much more efficient than the general SQGAC propagation algorithm. The work in this chapter could serve as a proof of concept for further arithmetic constraints.

CHAPTER 7

Application of QCSP to factory scheduling

7.1. Introduction

Scheduling is an important application of constraint programming. I use the well-studied job shop scheduling problem because of its simplicity. The aims are as follows:

- to advance the art of modelling problems in QCSP;
- to show that a moderately sophisticated problem can be modelled and solved with Queso;
- to evaluate some of the QCSP algorithms in this thesis on a realistic problem;
- and to briefly compare the schedule length of a contingent scheduling approach with a non-contingent approach.

The work in this chapter is intended as a proof of concept, rather than as a contribution to scheduling. Job shop scheduling is very well studied, so I will not be able to compete in terms of optimizing large instances.

In general, scheduling is applying resources to tasks over a period of time, respecting constraints such as task order and resource capacity. It is often framed as an optimization problem, with the aim of minimizing the total length of the schedule (the *makespan*). In this chapter I consider problems with uncertainty (particularly machine servicing at unpredictable times), where the uncertainty is modelled using universally quantified variables.

Constraint programming is a leading technique for scheduling, with a great deal of research on the subject. One important technique is *edge finding* constraint propagation algorithms (introduced by Carlier and Pinson [**26**]). Edge finding propagation algorithms perform powerful propagation on the resources by exploiting rules about start and end times for tasks. While I do not use these algorithms here, the models presented in section 7.2 could be trivially adapted by

Job number	machine/duration		
1	1/3	2/3	3/3
2	3/4	2/4	1/4
3	2/3	3/3	1/3

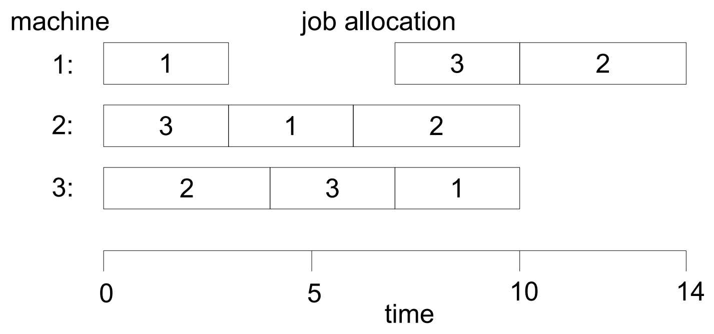


TABLE 9. Simple job shop scheduling problem and solution with optimal makespan

adding edge finding constraints over the start time variables of the tasks. The start time variables are all existential, so the existing edge finding algorithm can be used without alteration.

7.1.1. Job shop scheduling. The job shop scheduling problem (JSSP) is a simple and well-studied form of factory scheduling, with n jobs and m machines. A job consists of a chain of m tasks, each assigned to a distinct machine. Each task has two constants associated with it:

- the constant $tm(i, j)$ is an integer from $1 \ldots m$ representing the machine that is required for task i of job j;
- $d(\mathcal{M}, j)$ is an integral constant representing the duration of the task in job j which runs on machine $\mathcal{M}$.

The symbols tm and d are indexed differently for convenience in writing out the constraints, but since every job has exactly one task on each machine, the duration of task i of job j is simply $d(tm(i, j), j)$. The tasks cannot be interrupted, and must be executed in order. All tasks must be completed within a time bound *maxmakespan*.

A schedule is found which maps each task of each job to a starting time, such that no two tasks are running on the same machine at the same time. The duration (*makespan*) of the schedule is often optimized or approximately optimized.

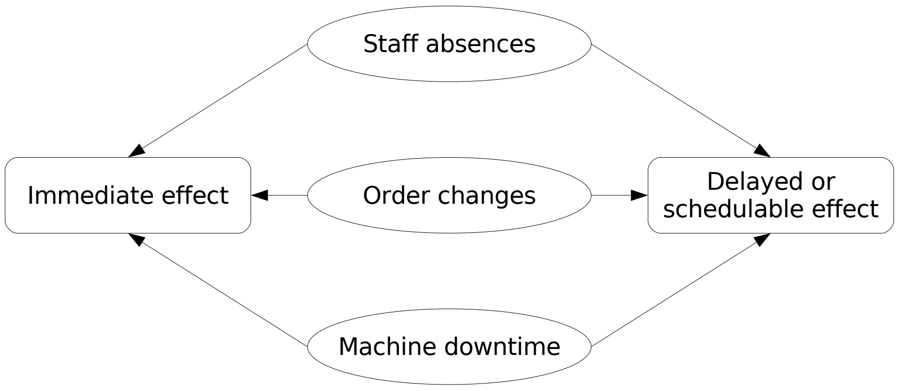


FIGURE 38. Sources of uncertainty

I will refer to the basic time units as *slots*, numbered from 1 to *maxmakespan*. Table 9 shows a simple example of a job shop problem, and an optimal solution with a makespan of 14.

7.1.2. Uncertainty in factory scheduling. Factory scheduling can have various sources of uncertainty, such as the ones listed below. Each one can have immediate effect or a delayed or schedulable effect (figure 38).

- Staff absences
- Order changes
- Machine faults or servicing
- Early or late delivery of raw materials
- Uncertainty in task durations

Davenport and Beck survey approaches to scheduling with uncertainty, with the following broad divisions [**33**].

- Redundancy-based techniques (which typically reserve time to re-execute tasks that fail).
- Probabilistic techniques, where the aim is to maximize the probability that a schedule will be able to execute.
- Contingent scheduling, where multiple schedules or schedule fragments are generated which optimally respond to anticipated events.

In this chapter I model a contingent scheduling problem with QCSP. One of the aims is to show that QCSP has potential in the area of contingent scheduling.

Faults. For simplicity I will focus on machine faults and ignore other sources of uncertainty. It is possible to broadly divide machine faults into two sets as shown in figure 38: faults which have an immediate effect; and faults which have a delayed or schedulable effect. I focus on faults with a delayed or schedulable effect.

Various types of machine fault may allow the machine to continue running for a period of time. For example, if a machine is running low on oil and needs to be refilled, or it is becoming less accurate and needs to be calibrated but the accuracy is still within acceptable bounds. In these situations it is desirable to have an optimal schedule whether the fault occurs or not. If the fault occurs, the schedule includes some servicing time but not otherwise. Contingent scheduling is ideal for this situation because a contingent schedule can be optimal or close to optimal whether a fault occurs or not. It can be sensibly modelled as a QCSP with universal variables representing faults.

7.1.3. Robust vs. contingent schedules. QCSP is well suited to generating contingent schedules. The advantage is that the schedule can be optimal in different circumstances. However such schedules can be large and take a long time to generate. Another approach is to generate robust schedules which are intelligently padded to guarantee certain conditions. Both these approaches have been investigated with various optimization techniques [**33**] but I will focus on constraint-based techniques here.

One type of robust solution is a *supersolution*. Hebrard et al. [**61**] develop the concept of an *(a,b)-supersolution*, which is a solution to a CSP such that if the values assigned to a variables are no longer available, the solution can be repaired by assigning these variables with a new values and also changing the value for b other variables. They develop a solver based on constraint propagation and search, which produces a supersolution in the form of a solution and a table of $O(n^a)$ repairs containing $a + b$ new values each. Deciding the existence of a supersolution is NP-complete, assuming a is fixed.

Hebrard et al. [**60**] go on to extend the framework, to allow restrictions on the set of variables which can be repaired, and on the new values that they will take. These restrictions make the framework much more attractive for factory scheduling. If a fault occurs on a machine part way through executing the schedule, the schedule modifications must be restricted to the part of the schedule that has not yet been executed. This means certain variables may not be changed. Others can be changed in a restricted way (the new value is restricted).

Finding supersolutions is a general technique, suitable for many types of problems, but it is less general than QCSP since it focusses on finding a single robust solution rather than a winning strategy. Since the two approaches address different problems, I do not compare them further.

Developing a comprehensive solution to scheduling with faults might require a hybrid approach where the schedule is padded intelligently, and also the schedule is contingent for certain situations. However, constructing a hybrid solver is beyond the scope of this thesis.

7.1.4. Faulty job shop scheduling. Recall that the basic unit of time in the JSSP is the slot. The slots are grouped into a number of *periods* of equal size, in order to deal with machine testing and servicing. For example, slots could be hours, and periods days.

To introduce faults to the JSSP, each machine $\mathcal{M}$ is tested at the beginning of each period a, and if servicing is required it is scheduled entirely within period a, and cannot be interrupted. The amount of time required to service a machine is *servicetime*. There are $2^{m \times periods}$ possible subsets of faults, but I will not define how many subsets must be covered by a schedule. I refer to this problem as the faulty job shop scheduling problem (FJSSP). I am not aware of this problem in the literature. It would be excessively difficult to schedule for every scenario, so to reduce the number of scenarios I use a simple probability-based approach.

More formally, an instance of FJSSP is an instance of JSSP with the following additional information:

- an integral constant *periods* which divides *maxmakespan*;
- a positive integer *servicetime* which defines the number of slots required for servicing when a fault occurs. *servicetime* must be less than *maxmakespan*/*periods*.

The problem of FJSSP is that of finding a set of schedules, each with a different subset of all possible faults. Each individual schedule meets the requirements of JSSP, and schedules contiguous servicing time for each occurring fault, within the period of the fault. For any pair of schedules, if they have the same fault configuration until period a, then the schedules must be equal until the beginning of period a. This set of schedules is hereafter referred to as a contingent schedule.

7.1.5. Probability bounding. When dealing with large numbers of possible faults in any scheduling problem, it would be very difficult (and unnecessary) to schedule for every possible combination of faults. To avoid this, I assign a probability to each fault, and a probability threshold ϕ to the whole problem. Only combinations of faults which are sufficiently likely (i.e. with probability $\geq \phi$) are considered. The faults are assumed to be independent, which makes the probability calculation straightforward. This assumption could be relaxed if necessary.

In the models presented in section 7.2, the probability of each fault is represented as a constant but it could trivially be a variable, whose value is a function of previous faults, machine workload, or other factors. In these models, the probability threshold can be used to control the amount of time it takes to generate the contingent schedule, since it controls how many scenarios are covered by the contingent schedule.

Since we are dealing with probabilities, it is reasonable to ask whether Stochastic CSP [**8**, **75**, **96**, **99**] (reviewed in chapter 2 section 2.4.1) would be more suitable. The treatment of probability is somewhat different in Stochastic CSP: when following a *policy* (analogous to a winning strategy), the probability of satisfying certain constraints (chance constraints, as opposed to hard constraints) exceeds some threshold. This is distinct from only considering scenarios whose probability exceeds a threshold.

One approach to Stochastic CSP has been to encode it into CSP. This approach can generate an exponentially large CSP instance, so it may not always be feasible. Since the QCSP algorithms used in this chapter scale polynomially in space, they can potentially be applied to larger problems. However, applying the CSP encoding approach can potentially yield stronger constraint propagation. I discuss this further in section 7.2.2.3 below.

The other approach of Balafoutis and Stergiou [**8**] is similar to top-down search for QCSP, but only arbitrary constraints are supported at present, with a propagation algorithm generalized from GAC2001/3.1.

7.2. Modelling Faulty Job Shop Scheduling in QCSP

First I will describe a model of JSSP in CSP, then develop a QCSP model of FJSSP with probability bounding which is closely related to the CSP model. When I refer to the *CSP model* this is the CSP model of JSSP, and the *QCSP models* A and B are of FJSSP with probability bounding.

7.2.1. CSP model. The model of job shop scheduling used by Lecoutre and Prosser [**69**] is presented here. n refers to the number of jobs, and m is the number of machines. The model has three sets of variables, and one optimization variable:

- $mn(n-1)/2$ Boolean variables $b^{\mathcal{M}}_{j_1,j_2}$ representing the order of two tasks, belonging to jobs j_1 and j_2 where $j_1 < j_2$, which both contend for machine $\mathcal{M}$.
- mn integer variables $\mathit{start}^{\mathcal{M}}_j$ representing the start time of the task from job j which runs on machine $\mathcal{M}$.
- mn integer variables $\mathit{end}^{\mathcal{M}}_j$ representing the end time of each task. Since the duration is a constant, this is simply the start time plus the duration.
- One integer variable *opt*, to be minimized, representing the maximum end time of all tasks.

The integer variables are all initially bounded between 1 and *maxmakespan*. The start time of a task may be equal to the end time of the previous task on the same machine.

The constraints are given below. Recall (from section 7.1.1) that the duration of a task in job j, which runs on machine $\mathcal{M}$, is $d(\mathcal{M}, j)$. The task in position i for job j runs on machine $tm(i, j)$.

(1) $\forall \mathcal{M}, j : \mathit{start}^{\mathcal{M}}_j + d(\mathcal{M}, j) = \mathit{end}^{\mathcal{M}}_j$

(2) $\forall \mathcal{M}, j_1, j_2 > j_1 : b^{\mathcal{M}}_{j_1,j_2} \Leftrightarrow [\mathit{end}^{\mathcal{M}}_{j_1} \leq \mathit{start}^{\mathcal{M}}_{j_2}] \wedge \neg b^{\mathcal{M}}_{j_1,j_2} \Leftrightarrow \mathit{start}^{\mathcal{M}}_{j_1} \geq \mathit{end}^{\mathcal{M}}_{j_2}$

(3) $\forall i \in \{1 \ldots m-1\}, j : \mathit{end}^{tm(i,j)}_j \leq \mathit{start}^{tm(i+1,j)}_j$

(4) $\max(\{\mathit{end}^{tm(m,j)}_j | j \in \{1 \ldots n\}\}) = \mathit{opt}$

A reasonable static variable ordering for this model would be to branch on b variables first, using 0 as the first value, then branch on the *start* variables using the smallest remaining value first. Once all the *start* variables have been instantiated, the *end* and *opt* variables are set by constraint propagation.

Another possibility would be to omit the b variables and replace constraint type 2 with a binary constraint between the *start* variables of any two tasks which run on the same machine. The new type 2 constraints are shown below.

- $\forall \mathcal{M}, j_1, j_2 > j_1 : \mathit{start}^{\mathcal{M}}_{j_1} + d(\mathcal{M}, j_1) \leq \mathit{start}^{\mathcal{M}}_{j_2} \vee \mathit{start}^{\mathcal{M}}_{j_1} \geq \mathit{start}^{\mathcal{M}}_{j_2} + d(\mathcal{M}, j_2)$

This would halve the number of constraints of type 2, and potentially allow more propagation. The *end* variables, and constraint type 1 can also be removed, and constraint types 3 and 4 re-stated in terms of *start* variables. This model is also seen in the literature [**9**, **87**], combined with sophisticated branching schemes.

Branching on the *start* variables using a simple numerically ascending value ordering results in far more search than branching on b variables in the reified model. Queso does support dynamic value ordering heuristics, so it would be possible to use the binary constraint model, but I have chosen to use the reified model and branch on the b variables, using value 0 first.

More advanced models. Job shop scheduling is very well studied and there are various more advanced models, with specialized non-binary constraints such as edge finding [**26**] (first applied in the constraints context by Caseau and Laburthe [**27**]), and other specialized approaches such as shaving [**77**].

The aim of this chapter is not to re-implement all this work in the context of QCSP, but to demonstrate that a contingent QCSP variant of a CSP model can be constructed, while preserving the important features of the CSP model, such as constraint propagators and variable and value ordering heuristics.

7.2.2. QCSP model. Some properties of a good contingent model for job shop scheduling are the following. It should:

- be not much larger than the CSP model;

- allow similar propagation to the CSP model among variables that are common to the two models;
- allow similar variable and value orderings as successful CSP models;
- and allow edge finding constraints. This implies that there are variables representing the starting time of each task.

The QCSP model must not be exponentially larger than the CSP model.

7.2.2.1. *Modelling faults and probability bounding.* First I will describe how the faults and probability bounding are modelled. The aim is to find a contingent schedule for all combinations of faults with probability greater than or equal to the threshold ϕ.

A fault with machine $\mathcal{M}$ in period a is modelled with a universal Boolean variable $\mathit{fault}_a^{\mathcal{M}}$. The constant $p(\mathcal{M}, a)$ represents the estimated probability of the fault. It is assumed that $p(\mathcal{M}, a) < 0.5$ (because of the form of constraints linking $p(\mathcal{M}, a)$ with $\mathit{faultp}_a^{\mathcal{M}}$), and that all faults are independent. A total ordering $\prec$ is imposed on the faults (specified by the pair $\langle \mathcal{M}, a \rangle$) and this ordering is the same as the ordering of the $\mathit{fault}_a^{\mathcal{M}}$ variables in the quantifier sequence. Variable $\mathit{precp}_a^{\mathcal{M}}$ is the probability of all events $\langle \mathcal{M}', a' \rangle$ that preceed $\langle \mathcal{M}, a \rangle$: $\langle \mathcal{M}', a' \rangle \prec \langle \mathcal{M}, a \rangle$. This is the product of the probabilities $p(\mathcal{M}', a')$ of those faults which did occur ($\mathit{fault}_{a'}^{\mathcal{M}'} = 1$) with the complement $1 - p(\mathcal{M}', a')$ for those faults which did not occur ($\mathit{fault}_{a'}^{\mathcal{M}'} = 0$).

A constant $\mathit{succp}_a^{\mathcal{M}}$ is calculated for each fault, which is the product of the probabilities of the complement of all succeeding faults $\langle \mathcal{M}', a' \rangle \succ \langle \mathcal{M}, a \rangle$. In words, we assume that all later faults do not occur and compute a probability for them all based on this.

$\mathit{thisp}_a^{\mathcal{M}}$ is the probability of the scenario where fault $\langle \mathcal{M}, a \rangle$ does occur, all succeeding faults do not occur and the occurrence of preceding faults is decided by their respective *fault* variables. $\mathit{thisp}_a^{\mathcal{M}}$ is computed as follows: $\mathit{thisp}_a^{\mathcal{M}} = \mathit{precp}_a^{\mathcal{M}} \times \mathit{succp}_a^{\mathcal{M}} \times p(\mathcal{M}, a)$. There is a Boolean variable $\mathit{available}_a^{\mathcal{M}}$ which indicates whether the probability of the scenario is above or equal to the threshold. Finally, if $\mathit{available}_a^{\mathcal{M}} = 1$ then the value of $\mathit{fault}_a^{\mathcal{M}}$ is copied to a second variable $\mathit{shadow}_a^{\mathcal{M}}$. $\mathit{shadow}_a^{\mathcal{M}}$ determines whether servicing takes place for machine $\mathcal{M}$ during period a.

The constraints linking $\mathit{thisp}_a^{\mathcal{M}}$, $\mathit{precp}_a^{\mathcal{M}}$, $\mathit{available}_a^{\mathcal{M}}$, $\mathit{fault}_a^{\mathcal{M}}$ and $\mathit{shadow}_a^{\mathcal{M}}$ are shown here.

$$\mathit{thisp}_a^{\mathcal{M}} = \mathit{precp}_a^{\mathcal{M}} \times (\mathit{succp}_a^{\mathcal{M}} \times p(\mathcal{M}, a))$$

$$\mathit{available}_a^{\mathcal{M}} \Leftrightarrow \mathit{thisp}_a^{\mathcal{M}} \geq \phi$$

$$\neg\mathit{shadow}_a^{\mathcal{M}} \Leftrightarrow \neg\mathit{available}_a^{\mathcal{M}} \vee \neg\mathit{fault}_a^{\mathcal{M}}$$

$\mathit{precp}_a^{\mathcal{M}}$ must be linked to the previous *precp* in the ordering $\prec$. This is done by introducing another variable $\mathit{faultp}_a^{\mathcal{M}}$ which is the probability of the event which occurred (i.e. if the fault occurred then $\mathit{faultp}_a^{\mathcal{M}} = p(\mathcal{M}, a)$ and if not then $\mathit{faultp}_a^{\mathcal{M}} = 1 - p(\mathcal{M}, a)$). The three constraints to achieve this are shown below.

$$\mathit{shadow}_a^{\mathcal{M}} \Leftrightarrow (\mathit{faultp}_a^{\mathcal{M}} = p(\mathcal{M}, a))$$

$$\neg\mathit{shadow}_a^{\mathcal{M}} \Leftrightarrow (\mathit{faultp}_a^{\mathcal{M}} = 1 - p(\mathcal{M}, a))$$

$$\text{if } \mathcal{M} = 1 : \ \mathit{precp}_a^{\mathcal{M}} = \mathit{precp}_{a-1}^{m} \times \mathit{faultp}_{a-1}^{m}$$

$$\text{if } \mathcal{M} \neq 1 : \ \mathit{precp}_a^{\mathcal{M}} = \mathit{precp}_a^{\mathcal{M}-1} \times \mathit{faultp}_a^{\mathcal{M}-1}$$

The quantifier subsequence for these variables is shown below.

$$\exists \mathit{precp}_a^{\mathcal{M}}, \mathit{thisp}_a^{\mathcal{M}}, \mathit{available}_a^{\mathcal{M}}, \forall \mathit{fault}_a^{\mathcal{M}}, \exists \mathit{shadow}_a^{\mathcal{M}}, \mathit{faultp}_a^{\mathcal{M}}$$

The $\mathit{fault}_a^{\mathcal{M}}$ variable is included in only one constraint. If $\mathit{available}_a^{\mathcal{M}}$ is set to 0, then $\mathit{shadow}_a^{\mathcal{M}}$ is set to 0 by propagation and both values of $\mathit{fault}_a^{\mathcal{M}}$ become pure. One value will be removed by the pure value rule. This avoids unnecessary search. The pure value rule is very significant because without it $2^{\mathit{periods} \times m}$ scenarios would be explored.

The variables *thisp*, *precp* and *faultp* are represented using only their upper and lower bounds, as described in chapter 3 section 3.7.1.2. The multiplication constraint enforces bounds($\mathbb{R}$) consistency, which is sufficient since it is only used on existential variables.

All the above variables and constraints are shared by model A and model B below.

7.2.2.2. *Model A.* Model A is naive and ineffective, but since it is more obvious than model B I will describe it and explain why it is ineffective. This motivates the more complex model B.

For each time slot s and each machine $\mathcal{M}$, there is a variable $\exists t_s^{\mathcal{M}} \in \{1, 2, \ldots, n, \mathit{idle}, \mathit{servicing}\}$. $t_s^{\mathcal{M}}$ represents the job that $\mathcal{M}$ is running at time s, or whether it is being serviced or is idle. $\mathit{start}_j^{\mathcal{M}}$, $\mathit{end}_j^{\mathcal{M}}$ and *opt* variables are copied from the CSP model in section 7.2.1. Constraint types 1, 3 and 4 are copied from the CSP model. One other type of constraint (equation 26) referred to as the *channelling* constraint is required to channel between the $t_s^{\mathcal{M}}$ variables and the *start* and *end* variables. This is assumed to be a single constraint for simplicity.

$$\forall \mathcal{M}, j, s : \left[\mathit{start}_j^{\mathcal{M}} \leq s \wedge \mathit{end}_j^{\mathcal{M}} > s\right] \Leftrightarrow t_s^{\mathcal{M}} = j \tag{26}$$

The $t_s^{\mathcal{M}}$ variables and the $\mathit{start}_j^{\mathcal{M}}$ and $\mathit{end}_j^{\mathcal{M}}$ variables are two representations of the job shop scheduling problem. The reason for having both is that the $t_s^{\mathcal{M}}$ variables can be quantified in chronological order and they enforce mutual exclusion of tasks on machines, and the other representation enforces that the tasks occur in order on the appropriate machines.

Each period has length *plen* and $\mathit{periods} = \mathit{maxmakespan}/\mathit{plen}$. For a machine $\mathcal{M}$ and period a, the variable $\mathit{shadow}_a^{\mathcal{M}}$ must be connected to the appropriate time slots. This is done with variables $\mathit{servicestart}_a^{\mathcal{M}} \in \{((a-1) \times \mathit{plen} + 1) \ldots a \times \mathit{plen}\}$ and $\mathit{serviceend}_a^{\mathcal{M}}$ with the same domain, and the constraints below. The time required for servicing a machine is *servicetime*. If $\mathit{shadow}_a^{\mathcal{M}}$ is 0, then $\mathit{servicestart}_a^{\mathcal{M}}$ (and $\mathit{serviceend}_a^{\mathcal{M}}$) are fixed, to avoid the search algorithm branching for each of its values.

$$\forall \mathcal{M}, a : \mathit{servicestart}_a^{\mathcal{M}} + \mathit{servicetime} = \mathit{serviceend}_a^{\mathcal{M}}$$

$$\forall \mathcal{M}, a:\ \mathit{shadow}_a^{\mathcal{M}} = 0 \Rightarrow \mathit{servicestart}_a^{\mathcal{M}} = (a \times \mathit{plen} + 1)$$

$$\forall \mathcal{M}, a, \forall s \in \{((a-1) \times \mathit{plen} + 1) \ldots a \times \mathit{plen}\} :$$
$$\big[\mathit{shadow}_a^{\mathcal{M}} = 1 \ \wedge\ \mathit{servicestart}_a^{\mathcal{M}} \leq s \ \wedge\ \mathit{serviceend}_a^{\mathcal{M}} > s\big] \Leftrightarrow t_s^{\mathcal{M}} = \mathit{servicing}$$

To implement the constraints with $\leq$ and $>$, a GAC reified comparison constraint is used. For each $\leq$ or $>$ symbol, an additional existential variable is introduced. A single reified disjunction constraint is used to link the additional variables with $\mathit{shadow}_a^{\mathcal{M}}$ and $t_s^{\mathcal{M}}$.

The quantifier sequence is given below. The variables associated with each period are quantified, in chronological order. Within each period, the variables associated with faults are quantified first, then the time slot variables for the period. The other variables are existentially quantified at the end of the quantifier sequence.

(1) For each period a in ascending order, the following two groups of variables are quantified:
  (a) First, the following sequence is repeated for each machine $\mathcal{M}$:
  $\exists \mathit{precp}_a^{\mathcal{M}}, \mathit{thisp}_a^{\mathcal{M}}, \mathit{available}_a^{\mathcal{M}}, \forall \mathit{fault}_a^{\mathcal{M}}, \exists \mathit{shadow}_a^{\mathcal{M}}, \mathit{faultp}_a^{\mathcal{M}}$
  (b) This is followed by: $\exists t^{1\ldots m}_{(a-1)\times \mathit{plen}+1 \ldots a \times \mathit{plen}}$
(2) $\exists \mathit{servicestart}^{1\ldots m}_{1\ldots \mathit{periods}}, \mathit{serviceend}^{1\ldots m}_{1\ldots \mathit{periods}}$
(3) $\exists \mathit{start}^{1\ldots m}_{1\ldots n}, \mathit{end}^{1\ldots m}_{1\ldots n}, \mathit{opt}$

There are two main reasons that model A is problematic. Firstly, the model is not compact enough for propagation to be efficient. There are $O(mn \times \mathit{maxmakespan})$ channelling constraints, and the *maxmakespan* can be large. As an example, if variable t_1^1 is set to 1 by the search procedure, and $d(1,1) = 10$, then $t^1_{2\ldots 10}$ are all set to 1 by propagation, which causes $10n$ channelling constraints to be woken up by changes to $t^1_{1\ldots 10}$ variables, and nm channelling constraints (and various others) to be woken up by bound changes on the start_1^1 and end_1^1 variables. In addition, variables $t^1_{11\ldots \mathit{makespan}}$ may have value 1 removed, which could potentially wake up a further

$(\textit{maxmakespan} - 10)n$ channelling constraints. The *maxmakespan* and the durations can be large, so model A is highly inefficient for propagation.

The other reason is search. To make intelligent branching decisions for the $t_s^{\mathcal{M}}$ variables would require a variable and value ordering heuristic which is aware of the start and end times of tasks. This is not currently implemented in Queso, although it would not be difficult.

7.2.2.3. *Model B.* Model B is much more compact, and preserves the CSP model structure much better. The trick here is to duplicate the whole CSP model once for each period, and post constraints to copy forward the relevant values from period a to $a + 1$. If a fault occurs in period $a + 1$, then only tasks which started before the beginning of period $a + 1$ are copied forward. The rest may have to be scheduled differently, with servicing time included. If no fault occurs in period $a + 1$, the whole schedule is copied forward.

Operationally, this means Queso solves the whole job shop instance for period 1, assuming no faults occur for periods $2 \ldots \textit{periods}$. If no faults occur, then the whole schedule is copied forward and there is no more work to do. If a fault occurs in period 2, then the section of the schedule before the start of period 2 is copied forward, and the rest is rescheduled. The b variables are preserved, and are searched on.

For each period a, an extra existential variable $\textit{nofault}_a \in \{0, 1\}$ is introduced which is used for copying forward the entire schedule when no faults occur. The following constraints are introduced.

$$\forall a: \; \textit{nofault}_a \Leftrightarrow \neg \textit{shadow}_a^1 \wedge \cdots \wedge \neg \textit{shadow}_a^m$$

Copies of the CSP model are introduced for each period, with each variable superscripted with the period number. For example, $\textit{start}_j^{\mathcal{M},a}$ is the starting time of the task from job j which requires machine $\mathcal{M}$ from period a. Constraint types 1,2 and 3 are used for all periods. Constraint type 4 and the *opt* variable are only present for the last period.

For each period, additional variables $\textit{servicestart}_a^{\mathcal{M}}$ and $\textit{serviceend}_a^{\mathcal{M}}$ are introduced with the same meaning and domain as in model A. They are linked to the CSP model with the following constraints. The $\tau_{j,1}^{\mathcal{M},a}, \tau_{j,2}^{\mathcal{M},a}$ variables represent task ordering, and are existentially quantified.

$$\forall \mathcal{M}, a : \textit{servicestart}_a^{\mathcal{M}} + \textit{servicetime} = \textit{serviceend}_a^{\mathcal{M}}$$

$$\forall \mathcal{M}, a : \textit{shadow}_a^{\mathcal{M}} = 0 \Rightarrow \textit{servicestart}_a^{\mathcal{M}} = (a \times \textit{plen} + 1)$$

$$\forall \mathcal{M}, a, j :$$

$$\neg \textit{shadow}_a^{\mathcal{M}} \vee \tau_{j,1}^{\mathcal{M},a} \vee \tau_{j,2}^{\mathcal{M},a}$$

$$\textit{end}_j^{M,a} \leq \textit{servicestart}_a^{\mathcal{M}} \Leftrightarrow \tau_{j,1}^{\mathcal{M},a}$$

$$\textit{serviceend}_a^{\mathcal{M}} \leq \textit{start}_j^{\mathcal{M},a} \Leftrightarrow \tau_{j,2}^{\mathcal{M},a}$$

Adjacent periods are connected with the following constraints. The σ variables are local and are existentially quantified at the end of the period set. σ_1 and σ_2 are both Boolean variables. σ_1 indicates whether the value of $\textit{start}_j^{\mathcal{M},a}$ lies within periods $1 \ldots a$ (i.e. $\textit{start}_j^{\mathcal{M},a} \leq a \times \textit{plen}$). σ_2 indicates whether $\textit{start}_j^{\mathcal{M},a}$ is copied to the next period. $\textit{start}_j^{\mathcal{M},a}$ must be copied forward if there are no faults in period $a+1$ (therefore $\textit{nofault}_{a+1} \Rightarrow \sigma_2$) *or* the task started within periods $1 \ldots a$ (therefore $\sigma_1 \Rightarrow \sigma_2$).

$$\forall a \in \{1 \ldots \textit{periods} - 1\}, \mathcal{M}, j :$$

$$\neg \sigma_1 \vee \sigma_2$$

$$\neg \textit{nofault}_{a+1} \vee \sigma_2$$

$$[\textit{start}_j^{\mathcal{M},a} \leq a \times \textit{plen}] \Leftrightarrow \sigma_1$$

$$[\textit{start}_j^{\mathcal{M},a} = \textit{start}_j^{\mathcal{M},a+1}] \Leftrightarrow \sigma_2$$

GAC is applied to the reified $\leq$ constraints. All others are implemented using reified disjunction. The quantification sequence is given below.

(1) For each period a in ascending order:

  (a) For each machine $\mathcal{M}$: $\exists \mathit{precp}_a^{\mathcal{M}}, \mathit{thisp}_a^{\mathcal{M}}, \mathit{available}_a^{\mathcal{M}}, \forall \mathit{fault}_a^{\mathcal{M}}, \exists \mathit{shadow}_a^{\mathcal{M}}, \mathit{faultp}_a^{\mathcal{M}}$

  (b) $\exists \mathit{nofault}_a$

  (c) For each pair of jobs j_1, j_2 and each machine $\mathcal{M}$, $\exists b_{j_1,j_2}^{\mathcal{M},a}$

  (d) For each machine $\mathcal{M}$ and job j, $\exists \tau_{j,1}^{\mathcal{M},a}, \tau_{j,2}^{\mathcal{M},a}$

  (e) For each machine $\mathcal{M}$ and job j, $\exists \mathit{start}_j^{\mathcal{M},a}, \mathit{end}_j^{\mathcal{M},a}$

  (f) For each machine $\mathcal{M}$, $\exists \mathit{servicestart}_a^{\mathcal{M}}, \mathit{serviceend}_a^{\mathcal{M}}$

  (g) For all pairs of σ variables for this period, $\exists \sigma_1, \sigma_2$

(2) $\exists \mathit{opt}$

This model is much more compact than model A, and allows branching on b and τ variables first, thus deciding the ordering of tasks before setting the start variables to their lowest values. Edge-finding constraints could be trivially added to this model, over the *start* variables. If an edge-finding constraint supported variable durations, it could also be used on *servicestart* where the duration would be 0 if no servicing is required, and *servicetime* if it is required.

Search on model B. Unfortunately, when searching on model B (with either *search* or *branch-Bound*, algorithm 3 or 4), if schedule infeasibility is discovered when branching on variables representing period $a > 1$, then the solver backtracks to period $a - 1$ and typically tries a different value for a *servicestart* or *start* variable. This leads to fruitless search, because the ordering of tasks on each machine is not changed. After setting b and τ variables, and after propagation, instantiating *servicestart* and *start* variables to their lowest value will certainly find a minimal length solution if a solution exists, therefore there is no reason to try other values for these variables.

To solve this difficulty, the search procedure was altered to branch for only the lowest value of *servicestart* and *start* variables. Thus the values of *servicestart* and *start* variables are a function of the values of b and τ variables. This new procedure is referred to as *search2*.

A second difficulty arises when applying the *branchBound* algorithm to model B. Consider the situation where the algorithm makes a decision for period 1 which does not extend to a winning strategy. After making this decision, and before detecting the conflict, optimization is performed, making *branchBound* considerably slower than *search* to detect a conflict.

To solve this difficulty, the *search2* algorithm is used instead of *branchBound*, and optimization is performed as follows. Search2 is called to find a winning strategy S within *maxmakespan*. S has length *len*, and the *end* variables (for all periods) are all given a new upper bound of *len* -1, and *search2* is called again. When *search2* fails to find a winning strategy, the winning strategy from the previous iteration is an optimal one. This new procedure is called *searchOpt*. The experiment in section 7.3 uses *searchOpt*.

Notice that both of these difficulties are caused by late detection of conflicts. In the first case, the algorithm performs branches which lead to a conflict that has already been discovered in another branch, and in the second case work is done on optimization in areas of the search tree which cannot be part of a winning strategy. *searchOpt* also suffers from this issue to a lesser extent. This is discussed in section 7.3.2.

7.2.3. Why use SQGAC. Why use quantified notions of consistency such as SQGAC to solve this problem, instead of simply branching both ways on universal variables, and using CSP constraint propagation? Constraint Logic Programming solvers such as Eclipse [**3**] would allow the programmer to construct an ad-hoc solution using GAC or weaker consistency, where the solver branches for those combinations of faults which are within the probability threshold. However, maintaining SQGAC can be useful.

If the servicing time for machine $\mathcal{M}$ cannot be scheduled in period a, then $\mathit{shadow}_a^{\mathcal{M}}$ is set to 0 by propagation on other constraints (which are specific to models A and B). At this point, if $\mathit{available}_a^{\mathcal{M}}$ is set to 1, then the $\mathit{fault}_a^{\mathcal{M}}$ variable is pruned, the simplified QCSP is false, and the solver backtracks. In this way the solver checks forward for the universal variables, and should outperform a solver which simply branches for all values on a universal variable. Stronger local reasoning over the *start* variables would potentially increase this effect as well.

A second alternative would be to expand out the QCSP model for all scenarios within the probability bound, creating a CSP. This is very similar to the approach used by Tarim, Manandhar and Walsh to solve Stochastic CSP [**75**, **96**]. The size of the resultant CSP is the size of the QCSP model multiplied by the number of scenarios. There can be exponentially many scenarios. However, in the experiments below, there are 2^{15} scenarios but only 16 scenarios within the probability bound, therefore this approach is possible if we only consider the 16 scenarios.

This approach would probably yield more powerful propagation. Search decisions would be propagated for every scenario. If the decision is incompatible with any sufficiently probable combination of future faults, then the propagation would determine failure. This is not the case when applying SQGAC to QCSP model B. However, propagation would be more expensive in this scheme, and the set of scenarios must be small.

7.3. Empirical evaluation

The aim of this section is to show that applying QCSP to job shop scheduling can be useful, compared to simpler approaches, when dealing with machine faults and servicing.

I use model B for all experiments. To optimize schedules, *searchOpt* is used. The variable *opt* is minimized. When applying a new upper bound, all *end* variables for all periods are pruned with the new upper bound.

The ten problem instances used here have $n = m = 5$ and are derived from the ORLIB instances LA01 to LA10[1]. LA01 to LA10 have five machines but ten or fifteen jobs. I used only the first five jobs and deleted the others. The instances are small for non-contingent scheduling, but become challenging for Queso when contingency is introduced. The number of Boolean variables b in the CSP model would be $mn(n-1)/2 = 50$, so the space of assignments to them is 2^{50}.

7.3.1. Comparing non-contingent with contingent scheduling. A typical approach to machine breakdown is to add extra time to a schedule [**33**]. Therefore I compare the length of the schedules generated by the two approaches.

The performance of the QCSP algorithms on model B is likely to degrade as the disruption caused by servicing increases. This is because the schedule for each period is constructed assuming that no further faults will occur, so if a very disruptive fault does occur in a later period, the solver is likely to search extensively.

[1]Available from http://people.brunel.ac.uk/~mastjjb/jeb/orlib/

Hypothesis 1. Contingent scheduling will generate schedules with significantly shorter makespans and similar robustness to a non-contingent approach.

Hypothesis 2. The performance of the QCSP algorithms will degrade as *servicetime* is increased, all else remaining the same.

Method. The ten instances derived from LA01 to LA10 are used. The *maxmakespan* is 600 in all cases. The schedule is divided into three periods of 200 time units, and for each period, each machine has a fault probability of 0.05. The threshold probability is 0.01. The effect of this is that every scenario of a winning strategy contains at most one fault. With five machines and three periods, there are fifteen scenarios where some machine has a fault, and one scenario where no machine has a fault.

The *searchOpt* algorithm was used to generate contingent schedules. In these schedules, the parameter which is minimized is the maximum of the schedule length for each scenario.

Results. Table 10 shows the makespan for each instance and each servicing time. Figure 39 plots the ratio between the worst-case makespan with contingency and the makespan with no contingency. The makespan often increases as *servicetime* is increased, but the makespan is clearly not exactly proportional to *servicetime*. The lowest value (across the 10 instances) of Pearson's correlation coefficient between *servicetime* and makespan is 0.8025 (instance LA01-5 $\times$ 5) and the highest is 0.9995 (for LA09-5 $\times$ 5). Therefore the two are highly correlated.

In some cases, increasing the servicing time by 10 results in an increase of more than 10 in the makespan. For example, for instance LA01-5 $\times$ 5 between *servicetime* $=$ 70 and 80 the makespan increases by 37. This is counterintuitive, and could not happen in non-contingent scheduling. In this example, a particular partial schedule (which is associated with a low worst-case makespan) becomes infeasible for some scenario as *servicetime* is increased, so the schedule where *servicetime* $= 80$ is significantly different to the one where *servicetime* $= 70$.

In this experiment, every scenario of a winning strategy contains at most one fault. To generate a non-contingent schedule with similar robustness, I assume that a single fault occurs on machine $\mathcal{M}$ during a period a when $\mathcal{M}$ is constantly in use, and that the fault increases the makespan by

Servicing time per fault	Optimal worst-case makespan per instance									
	1	2	3	4	5	6	7	8	9	10
0	444	450	407	365	381	401	433	380	455	517
10	444	451	407	365	381	401	433	380	462	517
20	444	451	407	373	387	401	433	380	472	517
30	448	455	414	383	396	407	434	410	482	523
40	449	465	414	393	396	407	444	413	492	533
50	454	475	414	415	406	416	454	420	502	543
60	459	485	423	425	416	431	462	425	512	553
70	459	495	433	435	418	432	462	432	522	555
80	496	505	456	445	426	442	474	442	532	573

TABLE 10. The optimal makespan for each instance and each servicing time

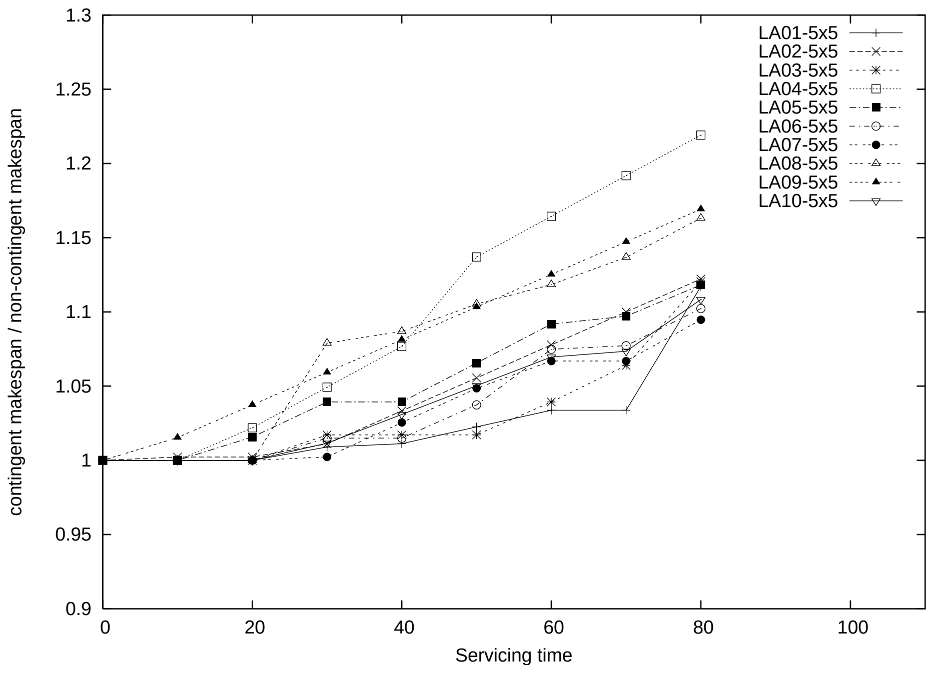


FIGURE 39. Worst-case makespan as servicing time is increased

servicetime time units. Therefore the non-contingent approach is to generate an optimal schedule and add *servicetime* to the makespan.

This approach may seem to be too pessimistic. However, for the instance LA01-5×5, for the optimal non-contingent schedule generated by Queso (with makespan 444), there is a machine $\mathcal{M} = 1$ and period $a = 1$ where the machine is constantly in use, and adding servicing to this

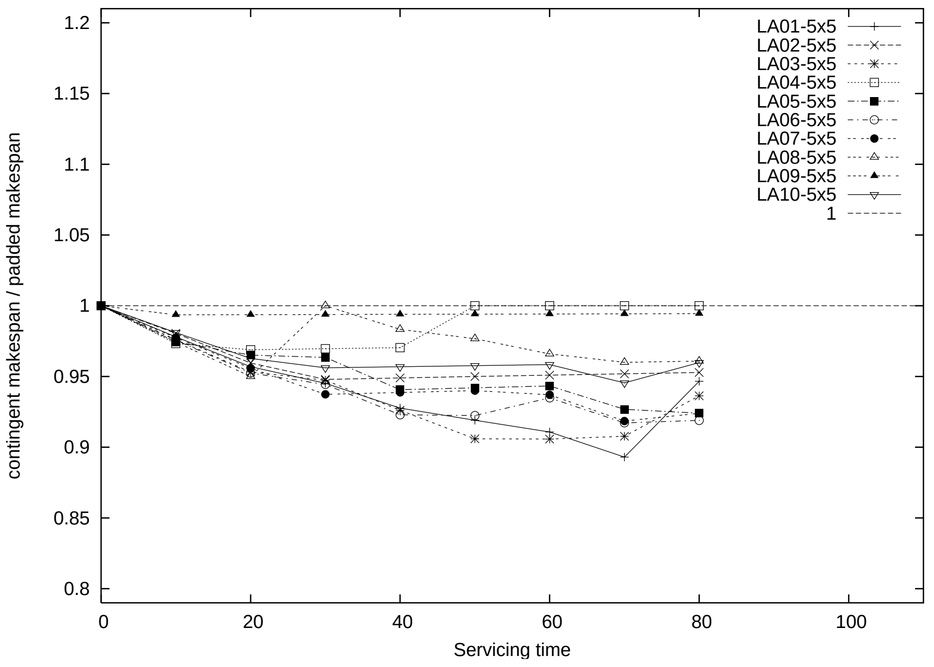


FIGURE 40. Comparing contingent and non-contingent schedules

period does increase the makespan by *servicetime* time units. I did not inspect the schedules for the other instances for this property.

Figure 40 plots the ratio between the worst-case makespan with contingency and the makespan of a padded non-contingent schedule. At almost all points, the contingent schedules have a shorter makespan. The improvement is up to 10%. This is evidence in favour of the first hypothesis.

Unfortunately the QCSP algorithms do not scale well as *servicetime* is increased. Tables 11 and 12 show search nodes explored and search time. These are plotted in figures 41 and 42(a), with figure 42(b) showing the number of nodes explored per millisecond. The experiment was run on a P4 3.06GHz with 1GB of RAM, using Sun Java 1.5 in server mode. Note that the search time includes all setup processes. Therefore for short searches, the number of nodes explored per millisecond can be low. This quantity varies in the range 0.17 to 2.55.

The number of nodes does not scale well for some instances. The time spent at each node decreases for long searches, but the overall effect is that search time does not scale well. This is evidence in favour of the second hypothesis.

Servicing time per fault	Total search nodes per instance				
	1	2	3	4	5
0	493	720	909	619	1027
10	13563	85103	11590	11261	19324
20	14083	178815	12890	13018	15377
30	11365	1494159	11114	310903	19422
40	13130	1183084	42673	1437076	31957
50	16177	2101186	43539	1043067	120872
60	12896	3778879	297610	1815212	98949
70	19547	1953620	270144	4267905	1336178
80	1122100	17333478	860395	8507842	2611152

Servicing time per fault	Total search nodes per instance				
	6	7	8	9	10
0	931	485	735	384	355
10	17348	8342	10946	9296	4689
20	22571	10778	9357	33090	21482
30	27344	25443	275270	116424	40153
40	62555	13309	412352	221565	65323
50	127142	16432	1458357	402092	124348
60	493937	18832	9462189	722570	206244
70	2409509	32666	3712182	1738985	566648
80	3291325	126681	7967568	2277102	691551

TABLE 11. Number of search nodes for each instance and servicing time

7.3.2. Observations. In some cases, Queso scales very poorly when servicing time is increased. This is most dramatic on instance 2, where the number of nodes increases from 720 to 17 million as the servicing time is increased from 0 to 80.

Queso schedules the periods in order. Consider the situation where the first period is scheduled, and a set of decisions made in period 1 are incompatible with all valid schedules for the final period. Assume intermediate periods can be scheduled. Queso will reach the final period, detect the conflict, and backtrack. However, because of chronological backtracking, it will explore every possibility for periods between the first and the last, before backtracking to the first period. It will thrash, detecting the same conflict many times. Situations like this are impossible when the servicing time is 0, but as servicing time is increased, the disruption caused by faults is increased so I believe it is more likely that this type of thrashing will occur.

This issue is discussed further in the future work section of chapter 8.

Servicing time per fault	Total search time per instance				
	1	2	3	4	5
0	1899	2708	5383	1231	1027
10	23642	74941	23997	16201	19594
20	17941	162070	24570	16172	15983
30	14712	1013624	19569	151221	17970
40	15746	833299	40663	710142	24658
50	19369	1644480	43241	535671	59414
60	17099	2687633	152972	922112	54205
70	25887	1556651	161298	2054317	524124
80	526963	13346121	462707	4123366	1151898

Servicing time per fault	Total search time per instance				
	6	7	8	9	10
0	2479	851	1256	1006	1209
10	28453	8844	12322	9718	7065
20	33993	12373	10034	20639	34869
30	36649	20133	159587	57353	69440
40	57729	14189	214746	116706	89916
50	92003	23329	845631	234751	132137
60	257263	25828	4975136	472589	191211
70	1095927	42241	1923378	974972	464795
80	1570807	98908	3497858	1416355	480744

TABLE 12. Total search time for each instance and servicing time

7.4. Summary

I have demonstrated the applicability of QCSP by modelling a real and well studied scheduling problem in it, using the SQGAC reified disjunction constraint, and applying the pure value rule to universal variables. The *searchOpt* algorithm was used to solve this problem. The algorithm rapidly finds the first winning strategy, and can be stopped at any time after that, giving an anytime behaviour. If the algorithm is allowed more time, the makespan can be reduced.

Applying contingent scheduling via QCSP can yield schedules with a shorter makespan than a naive padding approach, since it can optimize the schedule separately for each scenario. Unsurprisingly, the computational cost is higher. The search time grows with the servicing time required for each fault, all else being equal.

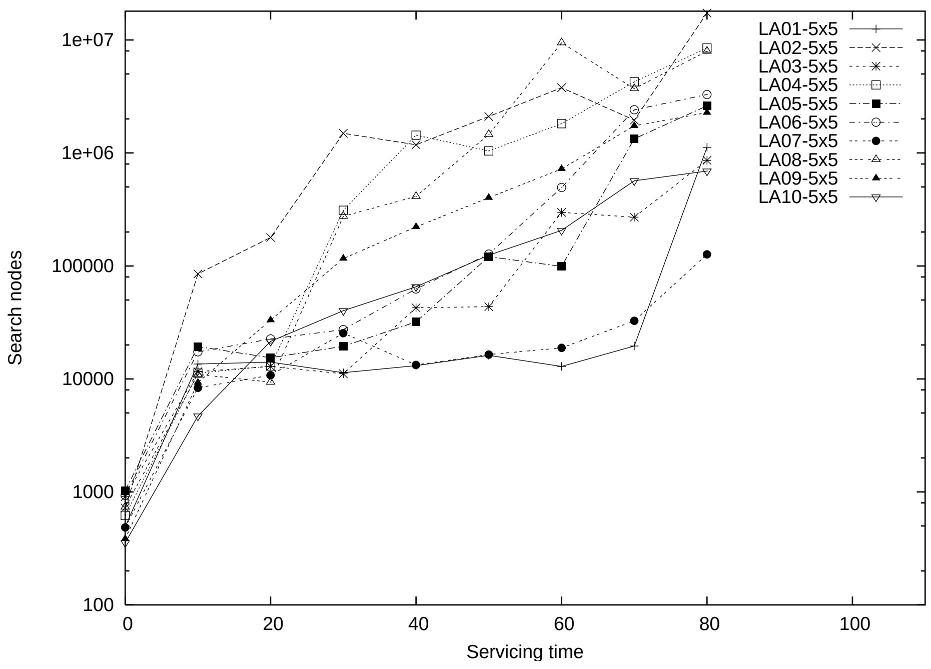
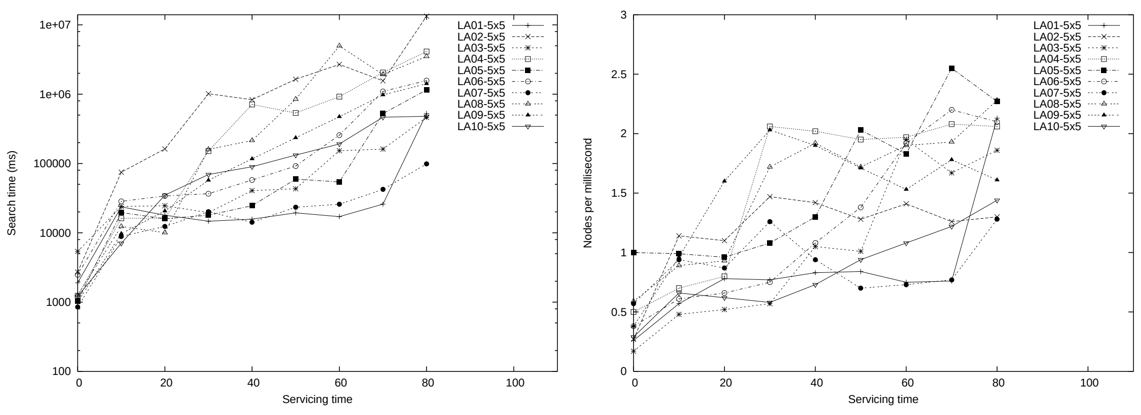


FIGURE 41. Search nodes against *servicetime*



(a) Search time (b) Nodes per millisecond

FIGURE 42. Search time and nodes per millisecond plotted against *servicetime*

As discussed in the introduction, real-world factory scheduling problems have many sources of uncertainty. Contingent scheduling with QCSP could be an interesting approach to these problems, combining the efficiency of CSP with contingency.

CHAPTER 8

Conclusion

As stated in the introduction, the aim of this thesis is to investigate the usefulness of QCSP as a formalism for reasoning with uncertainty. This breaks down into two main questions: can QCSP be solved efficiently, and can problems containing uncertainty be modelled effectively in QCSP? All the work in this thesis addresses one or other of these questions.

This chapter begins by summarizing the work reported within this thesis. This is followed by suggestions of future work and how these may be executed. This is divided into improving the techniques developed in this thesis, and developing new modelling ideas and conventions, to improve the applicability of QCSP.

8.1. Summary

The major items of work in this thesis are summarized here.

8.1.1. Definitions of consistency. The definitions of WQGAC and Qbounds($\mathbb{R}$) consistency are novel, and have allowed the development of new constraint propagation algorithms. It is hoped that these definitions are a useful contribution to the field.

8.1.2. Constraint propagators. The bulk of the work in this thesis is about constraint propagation. Table 13 summarizes the algorithms. Initially I considered propagation algorithms for arbitrary constraints.

Constraint form	Consistency	Time per node
Arbitrary (set of tuples)	SQGAC	$O(d^n)$
Arbitrary (set of tuples or predicate)	WQGAC	$O(nd^n)$
Reified disjunction	SQGAC	$O(r)$
Long sum	Qbounds($\mathbb{R}$)	$O(r)$

TABLE 13. Constraint propagators

In chapter 4, the algorithm SQGAC-propagate was developed, which is fine-grained and incrementally maintains a multiple winning strategy tree. Following this, WQGAC-Schema was developed from GAC-Schema [**13**]. WQGAC-Schema is also fine-grained and incremental, but enforces WQGAC, and is able to share its large data structure among many constraints, thereby reducing memory usage. WQGAC-Schema is instantiated for predicates, and three types of lists of satisfying tuples.

These five algorithms compared favourably against QBF and binary QCSP solvers, on random instances. They were also compared together on structured instances (Noughts and Crosses and Connect 4). The results of this comparison favour SQGAC-propagate in most cases, but on one model of Noughts and Crosses with an arity 12 constraint, WQGAC-Schema proves to be more efficient.

Unfortunately, both SQGAC-propagate and WQGAC-Schema take $O(d^r)$ time, and their evaluation on Connect 4 shows them to be less useful than reified disjunction. Also, neither algorithm found a use in solving the faulty job shop scheduling problem. However, these are the only algorithms for arbitrary constraints in QCSP and they may prove very useful in some other context. Indeed both algorithms compared very favourably against QBF and binary QCSP solvers.

One of the instantiations of WQGAC-Schema (the Next-Difference list) is also applicable to CSP, and has been shown experimentally to be very competitive in that context [**51**].

In chapter 5, two variants of the reified disjunction constraint were presented. This constraint is coarse-grained and runs in $O(r)$ time. It proves to be much more efficient for Connect 4 than either SQGAC-propagate or WQGAC-Schema. It is also shown (analytically and experimentally on Connect 4) to dominate the logic constraints of Bordeaux et al. [**19**, **22**].

Finally, in chapter 6 the long sum constraint is proposed. This is significantly stronger than the existing ternary sum predicate [**19**], and a brief experiment shows it to be much more efficient than SQGAC-propagate on some arity 6 constraints.

8.1.3. Pure value rule. In chapter 3, the pure value rule was defined for non-binary constraints, and two schemes were given to efficiently implement it by re-using constraint propagation algorithms. It finds an important use pruning universal variables in all the experiments involving

Connect 4, Noughts and Crosses, and faulty job shop scheduling. The pure value rule is shown to make a huge difference to the size of the search tree. With Connect 4, the difference can be several orders of magnitude. With job shop scheduling, the pure value rule is shown analytically to be of huge benefit in terms of the size of the search tree.

8.1.4. Search and optimization. The pure value rule and the propagation algorithms are embedded into search and optimization algorithms. While similar search algorithms have been published in various papers, optimization has never been addressed to my knowledge, even though it is likely to be very important for real-world problems expressed in QCSP.

8.1.5. Application of QCSP. Connect 4 was proposed by Walsh as a challenge for QBF, at the SAT 2003 conference [**100**]. Walsh claims that it is natural to express Connect 4 in QBF, but an attempt at this yielded a very complicated encoding [**46**]. The encoding into QCSP is simpler, and was solved with some success with board sizes up to 5 columns and 6 rows. Solving the full board size (7 columns and 6 rows) instance of Connect 4 remains a significant challenge in QCSP.

In modelling Connect 4, it was necessary to ensure that certain values of universal variables became pure during search. Those values represent cheating moves. A second, existential variable was used, which takes the same value as the universal if that value is not a cheating move. This simple pattern allows the pure value rule to prune universal variables effectively.

Job shop scheduling with faults serves as evidence that a complex, realistic scheduling problem can be modelled in QCSP and solved. This is a proof of concept, and the QCSP model shows potential to be developed much further by incorporating CSP scheduling techniques. The aims of this work were to advance the art of modelling problems in QCSP, to evaluate the algorithms proposed in this thesis, and to compare lengths of contingent schedules with non-contingent robust schedules.

In modelling job shop scheduling with faults, it is possible to adapt a simple CSP model. The CSP model is duplicated for each period, and some constraints and variables pertaining to faults are added. The pure value rule is exploited in the same way as for Connect 4. It may be possible to use this approach with other problems where a CSP model already exists.

8.2. Future work

There are many opportunities for development of the QCSP formalism, both in terms of algorithms and in modelling and application. In constraint programming, there is a large body of research on solving CSP instances, but also a great deal of research on modelling problems effectively and selecting appropriate propagation algorithms. Both these strands of research are in their infancy in QCSP.

The following items of future work concentrate on solving QCSP more effectively, rather than modelling, because modelling is dependent on the constraints available in the solver, and reasoning techniques employed by the solver. The suggestions are sorted into order of decreasing importance, in my opinion.

8.2.1. Conflict and solution learning. In QBF, conflict learning [**56**] and solution backjumping [**57**] are very effective. Several variants of solution learning [**55**, **56**] have been proposed, but none have been shown to be always beneficial. (The overheads of solution learning can often outweigh the gains.) Also, in constraint programming, conflict learning is beginning to attract interest.

It is reasonable to expect that conflict learning would be effective in QCSP, since it has been in QBF. Combined with strong propagation, conflict learning should be able to derive short reasons for each conflict. Similarly, some kind of solution learning or backjumping could be very beneficial, and I believe it would be worth investigating.

It may be easier to develop learning or backjumping algorithms if all constraints are table constraints. This would be a high price to pay for learning or backjumping. Ideally, the algorithms would work with any kind of constraint, thus enabling efficient propagation as well as the learning or backjumping. I do not know if this will be possible, but I believe it is worth investigation.

Since conflict learning is very successful in SAT and QBF, I have placed this item first among the items of future work.

8.2.2. Incorporating CSP techniques. With faulty job shop scheduling, it was observed that CSP edge finding constraints could be incorporated without change, because they would contain only existential variables. Also, variable and value ordering heuristics could be used without change within each period. It would be useful to have the full range of CSP propagation algorithms available. The most straightforward way to do this may be to incorporate some quantified constraints and the pure value rule into a CSP solver such as Minion or Gecode. Indeed Benedetti et al. take this approach by adapting Gecode [**11**]. (However, they have no way of pruning universal variables.)

8.2.3. Stronger reasoning. It was observed with faulty job shop scheduling that the consistency reasoning was not adequate in some cases, despite using SQGAC. Edge finding constraints would probably be useful, improving propagation within each period. However, propagation to future periods is limited since the solver does not know the value of the fault variables.

Debruyne and Bessière first proposed singleton arc-consistency (SAC) [**37**], where an assignment $x_i \mapsto a$ is consistent iff the constraint network can be made arc-consistent with x_i instantiated to a. Assuming SAC could be adapted to QCSP, it would be possible to apply SAC to the fault variables during search, and hence have a much stronger look-ahead from one period to the next. In CSP Lecoutre and Prosser apply SAC during search with some success [**69**].

8.2.4. Automated modelling. Modelling in QCSP is difficult. It would be interesting to explore the possibility of automating some aspects of modelling. For example, ensuring that values of universal variables become pure at the appropriate time is not straightforward. Ideally, one would develop a framework which takes an abstract specification and creates a QCSP model which can be efficiently solved. In constraint programming, the equivalent effort is underway, for example the Essence/Conjure system [**41**] and G12 [**36**].

8.3. Final words

Is QCSP a practical, useful formalism? My answer to this question is a tentative yes. In this section I summarize the lessons learned while doing this work, and evidence in favour of the thesis, as well as the points against.

8.3.1. Two key lessons. I have assumed a top-down search framework, and have devoted most of this thesis to local reasoning. Within this framework, the first key lesson I have learned is that strong reasoning is essential for constraints which contain universal variables. This is shown most strikingly with Connect 4 (section 5.3.1.3).

I found no way of decomposing such non-binary constraints into binary constraints without losing propagation — the straightforward adaptation of the hidden variable encoding proved ineffective. Also, decomposing constraints into the logical and arithmetic primitives of Bordeaux et al. [**19**] was shown to be ineffective with many examples.

Searching over each value of each universal variable would yield an exponential explosion of the search space. The second key lesson is that this combinatorial explosion must be avoided in some way. In this thesis, the non-binary pure value rule is used to prune subsumed values from the universal domains. It performs the function adequately and with little overhead.

Benedetti et al. [**12**] identified the same difficulty and solved it in an almost identical way with QCSP+. Both approaches allow the modeller to specify (using constraints) which values of a universal variable are valid and which are not, as a function of the values of outer variables. This allows the user to arbitrarily specify which scenarios are important and should be included in a winning strategy.

Other solutions to this difficulty alter the search framework (Blocksolve [**97**] for binary QCSP, and the method of Ansótegui et al. [**5**] for QBF).

8.3.2. Modelling in QCSP. The model B of faulty job shop scheduling provides a proof of concept for modelling optimization problems in QCSP. There are two main modelling tricks: the shadow variable, which is used so that the pure value rule can be effective; and the device of duplicating the entire schedule once for each period. It is possible to adapt the model to any scheduling problem, with any kind of resource, or any timetabling problem. It is also possible to add edge-finding and other state of the art constraint propagation algorithms to the model without modifying the propagation algorithms.

On the other hand, the optimization algorithm I proposed in chapter 3 is not suitable for this model. This fragility is a sign of the immaturity of QCSP algorithms. To avoid this problem I proposed another optimization algorithm. There was a second difficulty with branching on start and end variables, where equivalent solutions were discovered many times. This was avoided by defining start and end variables as functions of the Boolean variables.

Finally, and perhaps most seriously, at each period the scheduling instance is solved assuming no future faults. This assumption may lead to inappropriate schedules which cannot cope with future faults, and therefore excessive backtracking to repair the schedule. This suggests that the propagation is not strong enough, despite an effort throughout the thesis to develop strong algorithms. This difficulty led me to suggest singleton consistency as an item of future work.

These three difficulties in solving faulty job shop scheduling illustrate the relative immaturity of QCSP algorithms at present, however they are not fatal to the thesis because each can be addressed.

8.3.3. The state of QCSP. The main point in favour of the thesis is that it has been possible to model and solve a realistic contingent scheduling problem in QCSP. This is only a prototype, but it shows in principle that QCSP can be applied to contingent scheduling.

The model is fragile, and was constructed carefully with the reasoning algorithms of the solver in mind. This is far from the *declarative* aspiration of traditional constraint programming. Similarly, algorithms were changed specifically to fit the model, because the unmodified algorithms were not suitable. Therefore, Queso is not robust to any QCSP instance you give to it, but careful modelling is required to take advantage of the algorithms. I believe that the problem is fragility of the solver rather than a basic unsuitability of QCSP.

The original DPLL algorithm [**34**, **35**] for SAT has proven very useful for solving various interesting problems for many years. However, the development of conflict learning [**79**] and its related heuristics were very important in the evolution of robust and powerful solvers, with a much greater reach than plain DPLL. QCSP still requires a breakthrough of this kind.

As stated above, conflict learning has been very effective in SAT and QBF, and solution backjumping has been very useful in QBF. I would expect these techniques to be successful in QCSP as well, and to contribute to the robustness of a solver. I believe that solution learning is worth investigation as well, although the picture is somewhat less promising at the moment. It could be that solution backjumping would be preferable to solution learning. All techniques of this type have the potential to improve robustness because they can substantially reduce the size of the search tree based on similarities between branches. I think that learning and backjumping is the most promising avenue of further research in QCSP.

Overall, I expect the following elements to make up a robust QCSP solver: strong propagation algorithms for non-table constraints; some way of pruning universals; conflict learning; solution learning or backjumping; and a heuristic to order variables within blocks.

In summary, I believe the case for QCSP is reasonably strong, although there is still much work to be done. I hope that the material presented in this thesis will be a useful contribution to the field.

Bibliography

[1] ILOG solver 6.0 user manual, 2003.

[2] Dimitris Achlioptas, Michael S. O. Molloy, Lefteris M. Kirousis, Yannis C. Stamatiou, Evangelos Kranakis, and Danny Krizanc. Random constraint satisfaction: A more accurate picture. *Constraints*, 6(4):329–344, 2001.

[3] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. Eclipse user manual release 5.10, 2006. http://eclipse-clp.org/.

[4] Carlos Ansótegui. Personal communication.

[5] Carlos Ansótegui, Carla P. Gomes, and Bart Selman. The achilles' heel of QBF. In *Proceedings 20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 275–281, 2005.

[6] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[7] Fahiem Bacchus and Toby Walsh. A constraint algebra. Technical Report APES-77-2004, APES Research Group, 2004. Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.

[8] Thanasis Balafoutis and Kostas Stergiou. Algorithms for stochastic CSPs. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, pages 44–58, 2006.

[9] J. Christopher Beck and Mark S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117:31–81, 2000.

[10] Marco Benedetti. sKizzo: a suite to evaluate and certify QBFs. In *Proceedings 20th International Conference on Automated Deduction (CADE 2005)*, pages 369–376, 2005.

[11] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Reasoning on quantified constraints. In *Rappresentazione Della Conoscenza e Ragionamento Automatico*, 2006.

[12] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Qcsp made practical by virtue of restricted quantification. In *Proceedings 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 38–43, 2007.

[13] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 398–404, 1997.

[14] Christian Bessière and Jean-Charles Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *Proceedings 5th International Conference on the Principles and Practice of Constraint Programming (CP 99)*, pages 103–117, 1999.

[15] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 309–315, 2001.

[16] Christian Bessière, Jean-Charles Régin, Roland Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[17] Christian Bessière and Guillaume Verger. Strategic constraint satisfaction problems. In *Proceedings 5th International Workshop on Constraint Modelling and Reformulation (at CP 2006)*, 2006.

[18] Armin Biere. Resolve and expand. In *Proceedings 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 238–246, 2005.

[19] Lucas Bordeaux. Boolean and interval propagation for quantified constraints. In *Proceedings 1st International Workshop on Quantification in Constraint Programming (at CP 2005)*, 2005.

[20] Lucas Bordeaux, Marco Cadoli, and Toni Mancini. Exploiting fixable, substitutable and determined values in constraint satisfaction problems. In *Proceedings 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 270–284, 2004.

[21] Lucas Bordeaux, Marco Cadoli, and Toni Mancini. CSP properties for quantified constraints: Definitions and complexity. In *Proceedings 20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 360–365, 2005.

[22] Lucas Bordeaux and Eric Monfroy. Beyond NP: Arc-consistency for quantified constraints. In *Proceedings 8th International Conference on the Principles and Practice of Constraint Programming (CP 2002)*, pages 371–386, 2002.

[23] F Börner, A Bulatov, Peter Jeavons, and Andrei Krokhin. Quantified constraints: Algorithms and complexity. In *Proceedings 17th International Workshop on Computer Science Logic (CSL 2003)*, pages 58–70, 2003.

[24] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proceedings 15th National Conference on Artificial Intelligence (AAAI 98)*, pages 262–267, 1998.

[25] Marco Cadoli, Marco Schaerf, Andrea Giovanardi, and Massimo Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[26] Jacques Carlier and Eric Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, 26:269–287, 1990. Cited by [**77**].

[27] Yves Caseau and Francois Laburthe. Improved CLP Scheduling with Task Intervals. In *Proceedings 11th International Conference on Logic Programming (ICLP 94)*. The MIT press, 1994.

[28] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings 12th International Joint Conference on Artificial Intelligence (IJCAI 91)*, pages 331–337, 1991.

[29] Hubie Chen and Víctor Dalmau. From pebble games to tractability: An ambidextrous consistency algorithm for quantified constraint satisfaction. In *Proceedings 19th International Workshop on Computer Science Logic (CSL 2005)*, pages 232–247, 2005.

[30] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *Proceedings 19th Australian Joint Conference on Artificial Intelligence (AI 2006)*, pages 49–58, 2006.

[31] Lecoutre Christophe and Szymanek Radoslaw. Generalized arc consistency for positive table constraints. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, pages 284–298, 2006.

[32] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint programming. In *Proceedings 11th International Conference on the Principles and Practice of Constraint Programming (CP 2005)*, pages 17–31, 2005. Best Paper award.

[33] Andrew J. Davenport and J. Christopher Beck. A survey of techniques for scheduling with uncertainty. Unpublished manuscript. Available from http://tidel.mie.utoronto.ca/publications.php.

[34] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[35] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(1):201–215, 1960.

[36] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, pages 700–705, 2006.

[37] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 412–417, 1997.

[38] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[39] Hélène Fargier, Jérôme Lang, and Thomas Schiex. Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In *Proceedings 13th National Conference on Artificial Intelligence (AAAI 96)*, pages 175–180, 1996.

[40] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[41] Alan M. Frisch, Matthew Grum, Chris Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proceedings 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 80–87, 2007.

[42] Alan M. Frisch, Christopher Jefferson, and Ian Miguel. Symmetry-breaking as a prelude to implied constraints: A constraint modelling pattern. In *Proceedings 16th European Conference on Artificial Intelligence (ECAI 2004)*, 2004.

[43] Alan M. Frisch and Timothy J. Peugniez. Solving non-Boolean satisfiability problems with stochastic local search. In *Proceedings 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 282–290, 2001.

[44] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter Nightingale. Solving non-Boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, 2005.

[45] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings 12th Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974. Cited by [**85**], chapter 3.

[46] Ian Gent and Andrew Rowley. Encoding Connect-4 using quantified Boolean formulae. Technical Report APES-68-2003, APES research group, 2003.

[47] Ian P. Gent, Enrico Giunchiglia, Massimo Narizzano, Andrew G. D. Rowley, and Armando Tacchella. Watched data structures for QBF solvers. In *Proceedings 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 25–36, 2003.

[48] Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD using computational group theory. In *Proceedings 9th International Conference on the Principles and Practice of Constraint Programming (CP 2003)*, pages 333–362, 2003.

[49] Ian P. Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M. Smith, and S. Armagan Tarim. Search in the patience game 'black hole'. Technical Report CPPOD-21-2006, CPPOD research group, 2006.

[50] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 98–102, 2006.

[51] Ian P. Gent, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. Technical Report CPPOD-19-2006-A, CPPOD research group, 2006.

[52] Ian P. Gent, Peter Nightingale, and Andrew Rowley. Encoding quantified CSPs as quantified Boolean formulae. In *Proceedings 16th European Conference on Artificial Intelligence (ECAI 2004)*, pages 176–180, 2004.

[53] Ian P. Gent, Peter Nightingale, Andrew Rowley, and Kostas Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 2007. To appear.

[54] Ian P. Gent, Peter Nightingale, and Kostas Stergiou. QCSP-Solve: A solver for quantified constraint satisfaction problems. In *Proceedings 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 138–143, 2005.

[55] Ian P. Gent and Andrew G. D. Rowley. Local and global complete solution learning methods for QBF. In *Proceedings 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, pages 91–106, 2005.

[56] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *Journal of Artificial Intelligence Research (JAIR)*, 26:371–417, 2006.

[57] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified Boolean logic satisfiability. In *Proceedings 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 275–281, 2001.

[58] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for quantified Boolean logic satisfiability. In *Proceedings 18th National Conference on Artificial Intelligence (AAAI 2002)*, pages 649–654, 2002.

[59] Warwick Harvey. Symmetry breaking and the social golfer problem. In *Proceedings SymCon-01: Symmetry in Constraints, co-located with CP 2001*, pages 9–16, 2001.

[60] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Robust solutions for constraint satisfaction and optimization. In *Proceedings 16th European Conference on Artificial Intelligence (ECAI 2004)*, 2004.

[61] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in constraint programming. In *Proceedings 1st International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, pages 157–172, 2004.

[62] Hirosi Hitotumatu and Kohei Noshita. A technique for implementing backtrack algorithms and its application. *Information Processing Letters*, 8(4):174–175, 1979. Cited by [**66**].

[63] Brahim Hnich, Ian Miguel, Ian P. Gent, and Toby Walsh. CSPLib: a problem library for constraints. http://csplib.org/.

[64] Joey Hwang and David G. Mitchell. 2-way vs. d-way branching for CSP. In *Proceedings 11th International Conference on the Principles and Practice of Constraint Programming (CP 2005)*, pages 343–357, 2005.

[65] Claire Kenyon and Meinolf Sellmann. Plan B: Uncertainty/time trade-offs for linear and integer programming. In *Proceedings 3rd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2006)*, pages 126–138, 2006.

[66] Donald Knuth. Dancing links. In *Millennial Perspectives in Computer Science*, pages 187–214. Palgrave, 2000.

[67] Ludwig Krippahl and Pedro Barahona. Chemera: Constraints in protein structural problems. In *Proceedings of WCB06 Workshop on Constraint Based Methods for Bioinformatics*, pages 30–45, 2006.

[68] Francoise Laburthe. Choco: a constraint programming kernel for solving combinatorial optimization problems. http://choco.sourceforge.net/.

[69] Christophe Lecoutre and Patrick Prosser. Maintaining singleton arc-consistency. Technical Report CPPOD-14-2006, CPPOD research group, 2006.

[70] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings 20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 405–410, 2005.

[71] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the AllDifferent constraint. In *Proceedings 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 306–319, 2003.

[72] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. Cited by [**85**], chapter 3.

[73] A. K. Mackworth. On reading sketch maps. In *Proceedings 5th International Joint Conference on Artificial Intelligence (IJCAI 77)*, pages 598–606, 1977. Cited by [**85**], chapter 3.

[74] Nikos Mamoulis and Kostas Stergiou. Algorithms for quantified constraint satisfaction problems. In *Proceedings 10th International Conference on the Principles and Practice of Constraint Programming (CP 2004)*, pages 752–756, 2004.

[75] Suresh Manandhar, Armagan Tarim, and Toby Walsh. Scenario-based stochastic constraint programming. In *Proceedings 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 257–262, 2003.

[76] Kim Marriott and Peter J. Stuckey. *Programming with constraints: an introduction*. MIT Press, 1998.

[77] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings 5th International Conference on Integer Programming and Combinatorial Optimization (IPCO 96)*, pages 389–403, 1996.

[78] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings 8th European Conference on Artificial Intelligence (ECAI 88)*, pages 651–656, 1988.

[79] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings 39th Design Automation Conference (DAC 2001)*, 2001.

[80] Les Proll and Barbara Smith. ILP and constraint programming approaches to a template design problem. *INFORMS Journal of Computing*, 10:265–275, 1998.

[81] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[82] Stefan Ratschan. Continuous first-order constraint satisfaction. In *Proceedings of Artificial Intelligence and Symbolic Computation 2002*, 2002.

[83] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings 12th National Conference on Artificial Intelligence (AAAI 94)*, pages 362–367, 1994.

[84] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings 9th European Conference on Artificial Intelligence (ECAI 90)*, pages 550–556, 1990.

[85] Francisco Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[86] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, 1995.

[87] Norman M. Sadeh, Katia P. Sycara, and Yalin Xiong. Backtracking techniques for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 76(1-2):455–480, 1995.

[88] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.

[89] Barbara Smith, Kostas Stergiou, and Toby Walsh. Modelling the Golomb Ruler problem. In *Proceedings of Workshop on Non Binary Constraints (at IJCAI 99)*, 1999.

[90] Barbara M. Smith. A dual graph translation of a problem in 'Life'. In *Proceedings 8th International Conference on the Principles and Practice of Constraint Programming (CP 2002)*, pages 402–414, 2002.

[91] Barbara M. Smith. Constraint programming models for graceful graphs. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, pages 545–559, 2006.

[92] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977. Cited by [**85**] chapter 4.

[93] Kostas Stergiou. Repair-based methods for quantified CSPs. In *Proceedings 11th International Conference on the Principles and Practice of Constraint Programming (CP 2005)*, pages 652–666, 2005.

[94] Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings 16th National Conference on Artificial Intelligence (AAAI 99)*, pages 163–168, 1999.

[95] Stanley Smith Stevens. On the theory of scales of measurement. *Science*, 103:677–680, 1946.

[96] Armagan Tarim, Suresh Manandhar, and Toby Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints*, 11(1):53–80, 2006.

[97] Guillaume Verger and Christian Bessière. Blocksolve: a bottom-up approach for solving quantified CSPs. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, pages 635–649, Nantes, France, 2006.

[98] T. Walsh. SAT v CSP. In *Proceedings 6th International Conference on the Principles and Practice of Constraint Programming*, number 1894 in LNCS, pages 441–456. Springer, 2000.

[99] Toby Walsh. Stochastic constraint programming. In *Proceedings 15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 111–115, 2002.

[100] Toby Walsh. Challenges for SAT and QBF, 2003. Talk at SAT 2003 conference, slides available from http://www.cse.unsw.edu.au/∼tw/sat2003.ppt.

[101] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report MAC AI-271, MIT, 1972. Cited by [**85**], chapter 3.

[102] Neil Yorke-Smith and Carmen Gervet. Closures of uncertain constraint satisfaction problems. In *Proceedings 1st International Workshop on Quantification in Constraint Programming (at CP 2005)*, 2005.