Non-binary Quantified CSP: Algorithms and Modelling

Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, Scotland.
email:pn@cs.st-and.ac.uk

Abstract

The Quantified Constraint Satisfaction Problem (QCSP) extends classical CSP in a way which allows reasoning about uncertainty. In this paper I present novel algorithms for solving QCSP. Firstly I present algorithms to perform constraint propagation on reified disjunction constraints of any length. The algorithms make full use of quantifier information to provide a high level of consistency. Secondly I present a scheme to enforce the non-binary pure value rule. This rule is capable of pruning universal variables.

Following this, two problems are modelled in non-binary QCSP: the game of Connect 4, and a variant of job-shop scheduling with uncertainty, in the form of machine faults. The job shop scheduling example incorporates probability bounding of scenarios (such that only fault scenarios above a probability threshold are considered) and optimization of the schedule makespan. These contribute to the art of modelling in QCSP, and are a proof of concept for applying QCSP methods to complex, realistic problems. Both models make use of the reified disjunction constraint, and the non-binary pure value rule. The example problems are used to evaluate the QCSP algorithms presented in this paper, identifying strengths and weaknesses, and to compare them to other QCSP approaches.

1 Introduction

The Quantified Constraint Satisfaction Problem (QCSP) extends classical CSP by allowing the quantification of each variable with the existential ($\exists$) or universal ($\forall$) quantifiers over values. QCSP is PSPACE-complete [18]. QCSP can be used to model various combinatorial problems such as planning or scheduling under uncertainty, design, adversary game playing and model checking. For example, in planning under uncertainty we may want to determine if there exists a plan for all eventualities. In game playing, we may wish to find a strategy which wins the game whatever moves our opponent makes. The uncertainty or contingency in these examples is modelled in QCSP using universal variables. In this way, QCSP extends classical constraint programming in a way which allows reasoning about uncertainty or contingency.

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

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

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. In the baker's puzzle, the universal variable f represents the fact that it must be possible to use the weights to weigh out all 40 quantities of flour.

A *winning strategy* is a structure which specifies the value of existential variables such that the constraints are satisfied, whatever values the universal variables take. The value of each existential is a function of the universals which are quantified to the left of it. In the example above, w variables take only one value in a winning strategy, but c variables take different values depending on the value of f.

Contributions of this work The first part of the paper is concerned with solving QCSP. The general approach taken is the same as many CSP and QCSP solvers: to interleave reasoning and search. The focus of this work is on reasoning, including strong propagation and the pure value rule. These are used with a simple search algorithm. Sophisticated search algorithms which work with non-binary constraints are left for future work. The contributions towards solving QCSP are as follows.

- The reified disjunction propagator, which enforces a high level of consistency on reified disjunction constraints of unlimited length, making full use of quantifier information. This algorithm is instantiated in two different ways, to work with Boolean literals (i.e. x_i or $\neg x_i$ for some Boolean variable x_i) or to work with literals of integer variables (i.e. $x_i = c_i$ or $x_i \neq c_i$ for some integer variable x_i and constant c_i). $x_1 \vee \neg x_2 \vee x_4 \Leftrightarrow \neg x_3$ is an example of a reified disjunction constraint with Boolean literals. This approach subsumes and improves upon the logical primitives of Bordeaux and Monfroy [14, 16].

- The non-binary pure value rule, which allows the solver to identify values of universal variables which are subsumed by another value, and therefore to safely remove these values. Without some way of removing values from universal variables, a top-down search solver would branch exponentially on the universals. The rule also allows existential variables to be assigned to a pure value. A method is given to implement this rule by re-using constraint propagators. The non-binary pure value rule generalizes both the binary pure value rule by Gent et al. [27] and the monotone literal rule for QBF by Cadoli et al. [19].

The algorithms are implemented in a solver called Queso, which is described in section 3.4. It is available from the author on request.

The second part of the paper is concerned with modelling two quite different structured problems in QCSP. This is done for three reasons: to be a proof of concept that complex structured problems can be modelled and solved in QCSP; to contribute towards the art of modelling problems in QCSP; and to identify where improvements might be made to QCSP solvers by analyzing their performance on these problems. Previously, random binary problems have been commonly used [4, 27, 38, 41].

I chose to model the game of Connect 4 since it has received some attention in the area of QBF [25]. It is generalized by parametrizing the width and height of the board. The full size of 6 rows and 7 columns cannot be solved in reasonable time but smaller games can. It is a challenging problem with a non-trivial quantifier structure (with many quantifier alternations) and the model exploits both non-binary constraints and the non-binary pure value rule. Connect 4 is used to compare Queso with QCSP-Solve [27], BlockSolve [41] and Bordeaux's propagators [14, 16]. It would also be interesting to compare Queso with QeCode/QCSP+ [10] but we leave this for future work.

Secondly I modelled a scheduling problem because scheduling is an important application of constraint programming. I use the well-studied job shop scheduling problem because of its simplicity. In addition to the classical problem, machines are tested periodically, and if servicing is required, it is scheduled before the next test. Fault scenarios are bounded according to their probability, to avoid excessive processing time. This problem is modelled in QCSP, and solved with makespan optimization, demonstrating in principle that a realistic contingent scheduling problem can be solved by means of QCSP.

In summary, the contributions of this paper are as follows:

- to advance the art of modelling problems in QCSP;
- to advance algorithms for non-binary QCSP;
- to show that moderately sophisticated problems can be modelled and solved in non-binary QCSP;
- to evaluate the QCSP algorithms presented here on a realistic problem, identifying strengths and weaknesses; and
- to briefly compare the schedule length of a contingent scheduling approach with a non-contingent approach.

The scheduling work in this paper 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. Instead, the contribution is in the QCSP algorithms and modelling.

2 QCSP Background

This section provides the definitions of concepts used in this paper, and a review of previous work on QCSP.

2.1 Theoretical background

It is necessary to define QCSP and some related notions.

Definition 2.1 *Finite Quantified Constraint Satisfaction Problem*
A QCSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{Q} \rangle$ is defined as a sequence of n variables $\mathscr{X} = \langle x_1, \ldots, x_n \rangle$, a sequence of domains $\mathscr{D} = \langle D_1, \ldots, D_n \rangle$ where $D_i \subsetneq \mathbb{Z}$, $|D_i| < \infty$ is the finite set of all potential values of x_i, a conjunction $\mathscr{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints, and a quantifier sequence $\mathscr{Q} = \langle Q_1 x_1, \ldots, Q_n x_n \rangle$ 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 2.2 *Constraint*
Within QCSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{Q} \rangle$, a constraint $C_k \in \mathscr{C}$ consists of a sequence of $r > 0$ variables $\mathscr{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ with respective domains $\mathscr{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle$ s.t. $\mathscr{X}_k$ is a subsequence of $\mathscr{X}$, $\mathscr{D}_k$ is a subsequence of $\mathscr{D}$, and each variable x_{k_i} and domain D_{k_i} matches a variable x_j and domain D_j in $\mathscr{P}$. 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 $\mathscr{X}_k$.

The variables $\mathscr{X}_k$ 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 $\mathscr{P}$ by removing values from the domains to form a smaller problem $\mathscr{P}'$, C_k^S in $\mathscr{P}'$ is reduced accordingly. Once all variables are instantiated, if $\left|C_k^S\right| = 0$ the constraint has *failed* and if $\left|C_k^S\right| = 1$ the constraint is *solved* [3]. (For convenience, in the context of a single constraint C_k the variable x_{k_i} and domain D_{k_i} are referred to as x_i and D_i.)

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

Definition 2.3 *Semantics of the QCSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{Q} \rangle$*

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

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

Definition 2.4 *Solution*
A tuple t is a solution to QCSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{Q} \rangle$ iff $|t| = n$ and $\forall i : t_i \in D_i$ and in the simplified QCSP $\mathscr{P}' = \langle \mathscr{X}, \mathscr{D}', \mathscr{C}', \mathscr{Q} \rangle$ where $\mathscr{D}' = \langle \{t_1\}, \{t_2\}, \dots, \{t_n\} \rangle$, all constraints C'_k are solved.

The solution set $\mathrm{sol}^{\mathscr{P}}$ (defined as the set of all solutions of $\mathscr{P}$) is used later in the definition of the pure value rule.

Bordeaux et al. [15] define *strategies* and *winning strategies*. A strategy S for $\mathscr{P}$ is a family of functions which specify the values of existential variables as a function of the values of outer universal variables, where the value of every existential variable in $\mathscr{P}$ is specified for any valid tuple of values of outer universals. In this way, if all universal variables are assigned, all existential variables have a unique value in the functions. (The function for an existential variable with no outer universals has one mapping from the empty tuple $\langle\rangle$ to some value.)

A winning strategy S' is a strategy such that whenever all universal variables are assigned, and existentials are assigned according to the functions in S', the tuple of assigned values is a solution in $\mathrm{sol}^{\mathscr{P}}$. Such tuples are *scenarios* of S'.

Bordeaux et al. [15] define *inconsistency*, a property of a variable and value pair x_i, a. Informally, a pair is inconsistent iff it is contained in no scenario of any winning strategy.

Finally, Bordeaux et al. [15] show that the inconsistency property can be applied to a single constraint C_k, rather than $\mathscr{P}$, by constructing a QCSP instance $\mathscr{P}_k = \langle \mathscr{X}', \mathscr{D}', \{C_k\}, \mathscr{Q}' \rangle$ containing just the one constraint C_k, where $\mathscr{X}', \mathscr{D}', \mathscr{Q}'$ are restricted to just the variables (or domains, quantifiers) that are in the scope of C_k. This definition of inconsistency over a single constraint will be referred to as QGAC (Quantified Generalized Arc-Consistency). It is QGAC which is enforced by the propagation algorithm for reified disjunction, described below.

2.2 Previous work on QCSP

There have been a number of approaches to solving QCSP within the general scheme of interleaving reasoning with search. Approaches to reasoning include:

- Forward checking on binary (arity two) constraints (employed by the QCSP-Solve solver, primarily developed by Stergiou [27]). Forward checking is a standard technique in CSP [36], but is strengthened by making use of quantifier information in QCSP-Solve.

- Maintain arc-consistency on binary constraints (Mamoulis and Stergiou [31]). Again the definition of arc-consistency is strengthened for QCSP by making use of quantifier information.

- Maintaining consistency of numerical and logical constraints (Bordeaux and Monfroy [14, 16]). Propagation rules are given for short constraints such as $x_1 + x_2 = x_3$ and $x_1 \vee x_2 \Leftrightarrow x_3$ (over Boolean variables). Numerical and logical expressions can be built up from these short primitive constraints.

- Adapting CSP propagation algorithms to QCSP. Benedetti et al. [9] proposed four methods for strengthening a conventional propagation algorithm to take advantage of quantifier information. This is distinct to QCSP+ by the same authors.

- The pure value rule, proposed for binary constraints by Gent et al. [27] and implemented in QCSP-Solve. This identifies values which are not in conflict with any other value via any constraint. For universal variables, a pure value is removed from the domain unless it is the final value in the domain. For existential variables, the variable is assigned to the pure value.

- The dual problem, proposed by Bordeaux and Zhang [17]. The negation (dual) of the QCSP instance is constructed, where each quantifier is flipped from $\exists$ to $\forall$ and vice versa, and the constraint store is negated. Constraint propagation is applied in the dual problem. Values of existentials which are inconsistent in the dual correspond to values of universal variables in the primal problem which satisfy all constraints. Such values can be removed in the primal problem, providing a way of pruning universals.

There have also been innovations in search for QCSP. The simplest search algorithm is a depth-first search where the variables are assigned in quantification order. This is a straightforward generalization of the search procedure for quantified Boolean formulae (QBF) by Cadoli et al. [19]. For an existential variable, values are assigned in turn until the first value is found which allows the remainder of the QCSP to be solved. For a universal variable, all values are assigned in turn. If any one does not allow the remainder of the QCSP to be solved, the search backtracks with a failure. The algorithm is given in section 2.3. It is extended in several ways in the literature:

- Conflict backjumping (CBJ) (by Stergiou [27]) identifies the search assignments which are responsible for a failure, and backtracks directly to the most recent one. In this way the search procedure avoids immediately revisiting the same failure multiple times. The algorithm was specified in terms of binary constraints and implemented in QCSP-Solve.

- Solution directed pruning (SDP) (by Stergiou [27]) avoids immediately re-discovering the same solution multiple times. This is done by matching a solution to other values of universal variables. The algorithm is not specific to binary constraints. It is implemented in QCSP-Solve.

- Repair-based methods (Stergiou [38]) attempt to re-use a solution for different values of universals, by making local repairs to the solution.

- Solution backjumping (SBJ) (Bacchus and Stergiou [4]) makes use of solutions by backjumping in much the same way that CBJ exploits a failure. SBJ subsumes and improves upon SDP.

There are other search algorithms which do not follow the quantifier order. For example, BlockSolve [41] searches the rightmost variables first. A winning strategy is constructed by synthesizing sub-strategies. This is an interesting approach, although the current revision of BlockSolve did not perform well in the experiments reported below.

2.2.1 QCSP+

Benedetti et al. claim that structured problems are difficult to model in QCSP [10]. To illustrate the difficulty, consider an adversarial game where the number of legal moves is not fixed. A move of the adversary would typically be modelled as a universal variable x_i, but the domain of x_i is fixed before search begins. Therefore when branching on x_i it is necessary to ignore some values which correspond to illegal moves. (To disallow illegal moves using a constraint is not correct: this merely falsifies the instance because a winning strategy must satisfy all constraints for all values of universal variables.) The solution proposed by Benedetti et al. is a new language: QCSP+, a generalization of QCSP [10]. Written as first-order logic, a quantifier in QCSP+ has one of the following two forms.

$$\exists X_1 : L_1 \wedge [\ldots] \qquad\qquad \forall X_2 : L_2 \Rightarrow [\ldots]$$

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 allows one 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 values of universal variables can depend on the values of any variables quantified to the left. This solves the modelling difficulty. When solving, the illegal values of universal variables are removed by a form of propagation before the universal variable is branched.

QCSP+ has a practical advantage, which is that CSP propagation algorithms are used without alteration, therefore a large library of constraints is available. 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 [10]. In our experiments with Connect-4, strong quantified consistency was vital (section 4). However, experimental results are promising for QCSP+, and it has proved possible to model a cumulative scheduling problem with an *adversary* which is able to disrupt the schedule in a limited way [11]. (This scheduling problem is substantially different to the one modelled below.) Benedetti et al. also recently proposed an optimization scheme [12] for QCSP+.

2.3 Search algorithm

The search algorithm presented here is very similar conceptually to others in the literature (for example QCSP-Solve [27]), although the presentation is different, therefore I do not claim novelty for this algorithm. The search procedure (algorithm 1) is recursive, and 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.

Algorithm 1 Search for finite QCSP

```
procedure search($i$: variable index): Boolean
if $\neg$propagate():
    return false
if $i > n$: {base case}
    return true
while $D_i = \{a\}$: {One value left in domain}
    $i \leftarrow i+1$
    if $i > n$ then: return true
{recursive cases}
while true:
    $a \leftarrow$pickValue($x_i$) {choose a value heuristically}
    addBacktrackLevel()
    $D_i \leftarrow \{a\}$
    $b \leftarrow$search($i+1$)
    backtrack()
    if $Q_i = \forall$: {variable $x_i$ is universal}
        if $\neg b$ then: return false
        else:
            $D_i \leftarrow D_i \setminus \{a\}$
            if $D_i = \emptyset$ then: return true
            if $\neg$propagate() then: return false
    else: {variable $x_i$ is existential}
        if $b$ then: return true
        else:
            $D_i \leftarrow D_i \setminus \{a\}$
            if $D_i = \emptyset$ then: return false
            if $\neg$propagate() then: return false
```

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

Variables are searched in order $x_1 \dots x_n$ which is the quantification order. (Algorithm 1 would be called with value 1 initially.) 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. Alterations to variable domains are denoted using the domain directly (e.g. $D_i \leftarrow \{a\}$) to distinguish them from pruning performed by constraint propagators (which may cause failure, for example when pruning a universal variable).

Definition 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. The contract with *propagate* is as follows. *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* must 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 the fully instantiated problem. Therefore, assuming that *propagate* performs only sound simplifications, *search* is correct.

3 Solving QCSP

This section is concerned with new algorithms for solving non-binary QCSP. These include an optimization algorithm, but the main contributions are a propagation algorithm for the reified disjunction constraint, and a scheme for implementing the non-binary pure value rule by re-using constraint propagators.

3.1 Optimization

For many problem domains, it is necessary to minimize or maximize a value. For example, when constructing a nurse rota it might be desirable to maximize the fairness of the rota according to some numerical measure. In this section I describe an algorithm to find a winning strategy which minimizes an existential variable. To perform minimization, a winning strategy must be assigned a score. The score is defined as the maximum across all scenarios of the value of the minimization variable.

Minimization for the QCSP instance $\mathscr{P}$ is performed as follows: *search* is called to find a winning strategy S, which has score t. A new QCSP instance $\mathscr{P}'$ is constructed where the upper bound of the minimization variable is $t-1$, and a new search is performed. This is iterated until *search* returns false. I refer to this procedure as *searchOpt*.

SearchOpt performs restarts. It is conventional in constraint programming to backtrack rather than restart the search. However, in QCSP it is not clear how far to backtrack. Since the score is defined as a maximum across all scenarios, it would perhaps be necessary to identify the set of scenarios which have the maximum score, and backtrack to a node that dominates this set before recommencing search. This would require more housekeeping than searchOpt but it may be worthwhile. I leave this for future work.

Another approach is to begin optimizing sub-strategies before completing a winning strategy ([34] section 3.3.2). This algorithm finds a winning strategy which is optimal for all scenarios. However the algorithm spends considerable time performing optimization in branches of the search tree which may not be part of a winning strategy. In informal experiments, it performed much worse than searchOpt on some faulty job shop scheduling instances.

Benedetti et al. recently proposed an optimization algorithm for QCSP+ which could be adapted to QCSP [12]. However it is much more complex than the algorithm here, and the greater flexibility is not necessary for the purposes of this paper.

3.2 The quantified reified disjunction constraint

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. 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 [14, 16]. Solvers for *quantified Boolean formulae* (QBF, a subset of QCSP) [20] also deal with quantified disjunction constraints. Any new approach should compare well (in efficiency, expressiveness or strength of consistency) to both these items of work.

In this section 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 QGAC and executes in linear time (in r), 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 the new constraint is more expressive.

The constraint is defined in terms of literals. A literal l_j is an expression containing a single QCSP variable x_j which evaluates to a Boolean value when x_j is assigned. A constraint C_k with scope $\mathscr{X}_k = \langle x_1 \dots x_r \rangle$ is defined as a set of literals $L = \{l_1 \dots l_{i-1}, l_{i+1} \dots l_r\}$ and a single literal l_i. A tuple $\tau \in D_1 \times \cdots \times D_r$ solves the constraint (i.e. $\tau \in C_k^S$) iff $(x_1 = \tau_1, \dots, x_r = \tau_r) \Rightarrow (\bigvee L \Leftrightarrow l_i)$. In any solution, the disjunction of the values of the literals in L equals the value of l_i.

3.2.1 Motivating examples

In the following examples the variables are Boolean (i.e. with domain $\{0, 1\}$ where 0 represents false and 1 represents true), and literals are either positive ($l_j = x_j$) or negative ($l_j = \neg x_j$). Consider expression (1), which can be broken down into the two constraints shown in (2), by one application of the decomposition rule of Bordeaux and Monfroy [14]. In this example, enforcing QGAC directly on expression (1) 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 QGAC 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 (2) is not allowed in Bordeaux and Monfroy's scheme, because x_4 is not quantified last (further decomposition would be required).

$$\exists x_1, x_2, x_3 \forall x_4 \; : \; x_1 \vee x_2 \vee x_3 \Leftrightarrow x_4 \tag{1}$$

$$\exists x_1, x_2, x_3 \forall x_4 \exists x_5 \; : \; (x_1 \vee x_2 \Leftrightarrow x_5) \wedge (x_3 \vee x_5 \Leftrightarrow x_4) \tag{2}$$

As a second example, consider expression (3). Enforcing QGAC on this expression determines falsity. The expression breaks down into two constraints shown in (4). Enforcing QGAC 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.

$$\exists x_1, x_2 \forall x_3 \; : \; x_1 \vee x_2 \Leftrightarrow \neg x_3 \tag{3}$$

$$\exists x_1, x_2 \forall x_3 \exists x_4 \; : \; (x_1 \vee x_2 \Leftrightarrow x_4) \wedge (\neg x_3 \Leftrightarrow x_4) \tag{4}$$

3.2.2 Proposed variants of reified disjunction

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 QGAC in linear time (time $O(r)$ where r is the number of variables) for either form is given below.

A *Boolean literal* is a positive or negated instance of a variable, represented by l_j for some variable x_j. Boolean reified disjunction is shown in formula 5 below.

$$Q_1 x_1 \dots Q_r x_r : l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_r \Leftrightarrow l_i \tag{5}$$

The other variant is a reified disjunction of comparisons $x_j = v_j$ or $x_j \neq v_j$, where x_j is an integer variable and v_j is a constant. I will call $x_j = v_j$ and $x_j \neq v_j$ *integer literals*. Integer reified disjunction is shown in formula 6, where $(=, \neq)$ represents either $=$ or $\neq$. If $\forall j : v_j = 1$ 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 $=$.

$$Q_1 x_1 \dots Q_r x_r : \tag{6}$$
$$x_1(=,\neq)v_1 \vee \dots \vee x_{i-1}(=,\neq)v_{i-1} \vee x_{i+1}(=,\neq)v_{i+1} \vee \dots \vee x_r(=,\neq)v_r \Leftrightarrow x_i(=,\neq)v_i$$

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 $l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_r$ can be represented as $l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_r \Leftrightarrow 1$. Straightforward conjunctions $l_1 \wedge \cdots \wedge l_{i-1} \wedge l_{i+1} \wedge \cdots \wedge l_r$ can be represented as $l_1 \wedge \cdots \wedge l_{i-1} \wedge l_{i+1} \wedge \cdots \wedge l_r \Leftrightarrow 1$, which is transformed to $\neg l_1 \vee \cdots \vee \neg l_{i-1} \vee \neg l_{i+1} \vee \cdots \vee \neg l_r \Leftrightarrow 0$ by the De Morgan law. (Disjunction and conjunction of integer literals can be handled in exactly the same way.) An implication $l_1 \wedge l_2 \wedge \cdots \Rightarrow l_r$ is rearranged as $[\neg l_1 \vee \neg l_2 \vee \cdots \vee l_r] \Leftrightarrow 1$.

The Boolean reified disjunction primitive can be used in place of the $x_1 \Leftrightarrow \neg x_2$ primitive of Bordeaux and Monfroy, 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.

3.2.3 Propagation algorithm structure

In the following sections I give a coarse-grained, one pass propagation algorithm to achieve QGAC on the constraints given in formulas (5) and (6). The algorithm works on literals $(\neg)x_j$ or $x_j(=,\neq)v_j$, where the literals take values 0 or 1 representing the truth of the literal, depending on the domain of x_j. The useful information about a literal is its index j, quantification Q_j, and whether or not it is fixed to either 0 or 1.

If a literal containing x_j 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_j. Although the algorithm does not dynamically simplify the constraint during search, it does use this fact.

The algorithm is split into four independent parts. These are based on the value and quantification of l_i (where the constraint is $\bigvee L \Leftrightarrow l_i$), since the value of this literal determines the truth or falsity of the disjunction. Each of the four subsections 3.2.6 to 3.2.9 contain the relevant algorithm. 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.

[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.

Normally a reified disjunction constraint would contain no more than one instance of each variable. If an existential variable is repeated, the algorithm will be sound but will not necessarily establish QGAC. For example, establishing QGAC 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. (The implementation checks for repeated universals and prints a warning.)

The algorithm enforces QGAC as defined by Bordeaux et al. [15]. The proof of correctness is given elsewhere [34].

3.2.4 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*. To handle the two different types of literal, two versions of *literalState* are given in section 3.2.10. 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 literal 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 3.2.10.

3.2.5 The central procedure

The algorithm stores no state, and therefore nothing needs to be backtracked. The procedure *propagateOr* (algorithm 2) 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. 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 $\mathscr{P}$ 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.

3.2.6 Case disjunctionFalse

Procedure *disjunctionFalse* (algorithm 3) simply sets all the literals in the disjunction to 0. This is linear time for Boolean literals. The constraint propagator will not be

Algorithm 2 propagateOr

```
procedure propagateOr(): Boolean
{Achieve consistency for the constraint
$Q_1x_1 \dots Q_rx_r : l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_r \Leftrightarrow l_i$ or
$Q_1x_1 \dots Q_rx_r$ :
$x_1(=,\neq)v_1 \vee \cdots \vee x_{i-1}(=,\neq)v_{i-1} \vee x_{i+1}(=,\neq)v_{i+1} \vee \cdots \vee x_r(=,\neq)v_r \Leftrightarrow x_i(=,\neq)v_i$}
$a \leftarrow$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()
```

Algorithm 3 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
```

called again in the current branch of the search, because all its variables have been instantiated.

3.2.7 Case disjunctionTrue

Procedure *disjunctionTrue* (algorithm 4) is called when the disjunction must be satisfied. The disjunction (when applied to Boolean literals) is identical to a QBF clause [19], therefore a *unit propagation* algorithm can be applied (the standard propagation algorithm in QBF). Unit propagation, implemented with backtracking lists, or with watched literals (by Gent et al. [26]) 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 the observation that nearly all of the reified disjunction constraints used to model structured problems (such as Connect 4 [34] and job shop scheduling, section 5) are short, with typically six or fewer literals in the disjunction. The decision was made on the intuition that watched literals would be of no benefit [26] (since watched literals work best on long disjunctions) and the overhead of backtracking lists should be avoided.

3.2.8 Case xiUniversal

Procedure *xiUniversal* (algorithm 5) is called when $Q_i = \forall$ and l_i is not set to 0 or 1. In this case the outer set of literals l_j where $j < i$ must not conflict with $l_i = 0$, therefore l_j is set to 0 for all $j < i$. Therefore the outer part of the disjunction evaluates to 0. The inner set of literals l_j where $j > i$ must be such that the value of the disjunction can

Algorithm 4 disjunctionTrue

```
procedure disjunctionTrue(): Boolean
{Achieve consistency for the reduced constraint $Q_1x_1 \dots Q_rx_r : l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_r$}
$ui \leftarrow nil$ {store indices of a universal and existential literal respectively.}
$ei \leftarrow nil$
for $j \leftarrow 1..i-1,\ i+1..r$ in ascending order:
    $a \leftarrow$literalState($j$)
    if $a = 1$ then: return true {The disjunction is satisfied}
    if $a = \forall$ and $ui = nil$ then: $ui \leftarrow j$
    if $a = \exists$ then:
        if $ei \neq nil$ or $ui \neq nil$ then: return true
        {If there is an existential and an outer existential or universal, then no work can be done, because it cannot be known which literal will satisfy the disjunction}
        $ei \leftarrow j$
{$ui$ contains the outermost universal literal, if one exists}
{$ei$ contains the outermost existential literal, if one exists}
if $ei = nil$ then: return false {No way to satisfy the disjunction}
return removeValue($ei$, 0) {Just one existential literal, with no outer universal, so set it to 1}
```

Algorithm 5 xiUniversal

```
procedure xiUniversal(): Boolean
{The outer set of literals $l_j$ where $j < i$ must evaluate to 0, because 1 is not consistent with $l_i = 0$}
for $j \leftarrow 1 \dots i-1$:
    if not removeValue($j$, 1) then: return false {Set $l_j = 0$}
{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.}
$e \leftarrow$false {Whether an existential has been found}
for $j = i+1 \dots r$:
    $a$=literalState($j$)
    if $a = 1$ or $a = \forall$ then: return false
    if $a = \exists$ then: $e \leftarrow$true
return $e$
```

Algorithm 6 xiExistential

```
procedure xiExistential(): Boolean
{$x_i$ may need to be pruned}
{Check for the two conditions which would lead to pruning $x_i$}
allFalse←true
for $j = 0 \dots i-1,\ i+1 \dots r$:
    $a$ = literalState($j$)
    if $a = 1$ then: return removeValue($i$, 0)
    if ($a = \forall$ and $j > i$) then:
        if not removeValue($i$, 0): return false {Set $l_i = 1$ or return false}
        return disjunctionTrue() {Now $l_i = 1$, call the appropriate procedure to propagate the consequences}
    if $a = \exists$ or ($a = \forall$ and $j < i$) then: allFalse←false
if allFalse then: return removeValue($i$, 1)
return true
```

match l_i in both cases. If any of the inner set of literals is universal, this will conflict with l_i and the constraint fails. Also, $l_{j>i} = 1$ conflicts with $l_i = 0$, so this case also causes failure. Any number of 0 literals are allowed, and at least one $\exists$ literal must be present, so that it can eventually take the same value as l_i to satisfy the constraint. The constraint fails if there is no existential literal.

3.2.9 Case xiExistential

Procedure *xiExistential* (algorithm 6) 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
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 $l_i = 0$.

3.2.10 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 integer literals (i.e. of the form $x_i(=,\neq)v_i$). Algorithm 7 (*literalState*) examines the domain of variable x_i and its quantification, and returns its state. Algorithm 8 (*removeValue*) checks the type of the literal ($=$ or $\neq$) and removes the appropriate values 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.

The Boolean literal x_i is equivalent to $x_i = 1$, and $\neg x_i$ to $x_i \neq 1$. For Boolean literals the procedures are simplified by assuming $v_i = 1$ throughout and replacing the loop in algorithm 8 with a single exclude statement: **return** exclude(C_k, x_i, 0).

Algorithm 7 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 8 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
```

3.3 Pure value rule

The pure value rule for binary constraints is used in QCSP-Solve [27, 28], and the (closely related) pure literal rule is used in many QBF solvers [19,29] (sometimes with the name *monotone*). I re-define the pure value rule for non-binary constraints, and give a scheme for implementing it.

Next I define purity in the context of non-binary QCSP. Purity is defined over an arbitrary QCSP instance $\mathscr{P}$, but the definition is used later on instances $\mathscr{P}_k$ containing only the constraint C_k. The property is quite useless applied to the entire instance $\mathscr{P}$, but when it is applied to $\mathscr{P}_k$ it can be used to prove that values are d-fixable in $\mathscr{P}$.

Definition 3.1 *Purity of $x_i \to a$ in the QCSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{Q} \rangle$*

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

In words, if all possible solutions including value $x_i \mapsto a$ are indeed solutions to $\mathscr{P}$, then $x_i \mapsto a$ is a pure value. The definition of purity is equivalent to the *valid values* property of Bacchus and Walsh [5]. It is simply restated here in a more precise way, and following the terminology of Gent et al. [28] rather than Bacchus and Walsh.

Purity is a sufficient condition for d-fixability (pure$(x_i, a, \mathscr{P}) \Rightarrow$ d-fixable$(x_i, a, \mathscr{P})$), where d-fixability is defined by Bordeaux et al. [15]. This implication is proven elsewhere [34]. Local d-fixability for all constraints (i.e. all restricted QCSP instances $\mathscr{P}_k$) implies d-fixability in $\mathscr{P}$ [15]. If a value $x_i \mapsto a$ is d-fixable in $\mathscr{P}$, then it can be fixed (i.e. all values other than a can be removed from D_i) when x_i is existential. If x_i is universal, and a is not the final value in the domain, then a can be removed. In order to remove a, there must be some other value $b \in D_i, b \neq a$, since a is subsumed by b [34].

The dual problem (proposed by Bordeaux and Zhang [17]) is also capable of removing values from universal variables, on the basis that the value would satisfy all constraints immediately, were it to be assigned. I suspect very few values would be removed by this scheme, because the requirement for removing a value is too strong, since it is global (*all* constraints satisfied by the value) whereas the pure value rule is local (only depending on the constraints containing the variable in question). For this reason, I chose to generalize the pure value rule rather than use a dual problem. In the models of Connect 4 and faulty job shop scheduling, I exploit the fact that the pure value rule is local. Constraints involving universal variables are stated such that values become pure when they are no longer relevant (e.g. they correspond to a cheating move in a game).

In the models below, the pure value rule is used to prune universal variables based on the values of variables quantified to the left. An alternative would be to use QCSP+ (described in section 2.2.1). However, this alternative requires moving to a new language, and the pure value rule is sufficient for the problems modelled below.

3.3.1 Implementing the pure value rule

I give a scheme for implementing the non-binary pure value rule. It re-uses constraint propagation algorithms and avoids constructing the set C_k^S (the satisfying tuples of constraint C_k). Avoiding constructing C_k^S is highly important because C_k^S could be very large.

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

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

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 has failed. $x_{k_i} \mapsto a$ is pure w.r.t. C_k iff $x_{k_i} \mapsto a$ is inconsistent in $\mathscr{PV}_k$. This is because $x_{k_i} \mapsto a$ is consistent in $\neg C_k$ iff there exists a supporting tuple (by the definition of GAC [13]). 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, $x_i \in \mathscr{X}_k$, for $x_i \mapsto a$ to be pure in $\mathscr{P}$.

Figure 1 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 *side* CSPs $\mathscr{PV}_1$, $\mathscr{PV}_2$ and $\mathscr{PV}_3$. The area at the top represents the problem $\mathscr{P}$, 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 $\mathscr{PV}_k$ is constructed for each constraint C_k in $\mathscr{P}$, and it contains the single negated constraint $\neg C_k$. If C_k has scope $\mathscr{X}_k = \langle x_{k_1}, \dots, x_{k_r} \rangle$ then $\neg C_k$ has scope $\langle pm_{k_1}^k, \dots, pm_{k_r}^k \rangle$. (The superscript matches the constraint, and the subscript is the same as the subscript of the x_{k_i} variable in $\mathscr{X}_k$.)

The dashed arrows indicate that values that are pruned in $\mathscr{P}$ 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

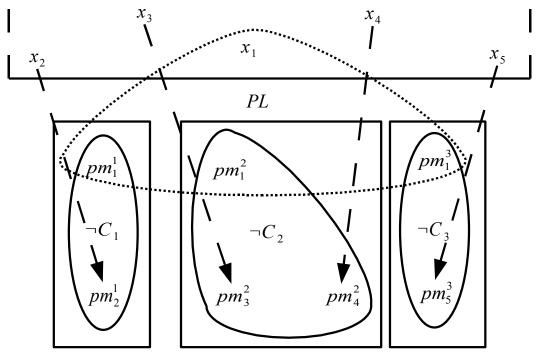


Figure 1: Implementation of pure value rule for one variable x_i

resembles a fine-grained constraint propagation algorithm, and it is called by the constraint queue as needed. It is given in algorithm 9. 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 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 in *pureLink* deals with the quantification of x_i. *Pure* for universal variables does the following: if there exists some value $b \neq a$ where $b \in D_i$, then a is subsumed by b. a is removed from D_i and constraints are queued appropriately. If there does not exist another value b, the *pure* procedure does nothing. For an existential variable, the *pure* procedure 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 $\mathscr{P}$ and the CSP side problems $\mathscr{P}\mathscr{V}$. 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 $\mathscr{P}$. A *pureLink* constraint is queued whenever any of its variables are changed, and is called like any fine-grained constraint.

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 1 as

Algorithm 9 Pure link algorithm

```
procedure pureLink($v$: variable, $a$: value): Boolean
{$x_i$ is the variable in $\mathscr{P}$, $pm_i^1, \ldots pm_i^m$ belong to side problems}
{$\forall a \in D_i$ : $w_a$ is the superscript $1 \ldots m$ of a $pm_i$ variable which contains value $a$}
if $v = x_i$:
    for $j \in \{1 \ldots m\}$:
        exclude($nil$, $pm_i^j$, $a$)
else:
    $v = pm_i^j$ {find the index $j$}
    if $w_a = j$: {$pm_i^j$ no longer contains $a$, watch invalidated}
        for $k$ in $j+1, \ldots, m, 1, \ldots, j-1$:
            if $a \in D_{pm_i^k}$:
                $w_a \leftarrow k$
                return true
        pure($PL_i, x_i, a$)
return true
```

dashed arrows.

The failure of a constraint in a side problem $\mathscr{PV}$ does not imply failure in $\mathscr{P}$. Therefore, if a constraint $\neg C_k$ from a side problem is queued in $\mathscr{P}$, 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 never return false. Also, if the domain of a *pm* variable becomes empty, this does not signify failure in $\mathscr{P}$.

3.4 Queso

The algorithms presented above are all implemented in the prototype solver Queso (*Qu*antified *S*atisfaction and *O*ptimization). In addition, Queso supports non-binary table constraints (enforcing QGAC or another, slightly weaker consistency) and sum constraints (enforcing a quantified bounds consistency) [34], as well as reified comparison ($\leq, <$ or $\neq$). It also has a number of constraints (e.g. max) which only work on existential variables. Queso is able to output a winning strategy in the form of a tree, however the default output is true or false (for the existence of a winning strategy).

Queso supports two types of variable: discrete variables with a small domain, and interval variables whose bounds are represented using integers of arbitrary length[2]. It implements the non-binary pure value rule for individual variables (as described above) and also for all variables as an optimized specialization.

Queso is implemented in Java 1.5 and is heavily tested. The design emphasizes

[2]Discrete variables may be existential or universal. Interval variables may only be existential, and they cannot be branched, they must be instantiated by propagation. There is no theoretical difficulty with interval variables, they are simply not fully implemented at present.

flexibility over performance, making it easy to add new constraints. For details of the design, see [34].

4 The game of Connect 4 in QCSP

In this section I present a model of the game of 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 [25], however the encoding is flawed (it does not forbid placing counters in full columns [2]).

The model is straightforward, with a set of variables representing the state of the board after each move, a set of variables representing the moves taken, and variables to track the state of the game after each move (i.e. if the game is finished, and which player won). The one unusual part of the model is the handling of the universal variables to exploit the pure value rule. This is discussed below the presentation of the constraints. The game is modelled with the following variables, given in quantification order for a single move *i*. This sequence is repeated $row \times col$ times for each *i* from 1 to $row \times col$.

- If *i* is even, move-variable $\forall u^i \in \{1 \dots col\}$ represents the column into which the token is placed.
- Move-variable $\exists m^i \in \{1 \dots col\}$ also represents the column into which the token is placed. If *i* is even, this variable is related by constraints to the one above.
- $row \times col$ variables $\exists b^i_{r,c}$ where $r \in \{1 \dots row\}$ and $c \in \{1 \dots col\}$, each with domain $\{red, black, nil\}$ representing the state of the board after the move. For example $b^i_{1,c}$ represents the piece at the base of column *c*.
- *col* variables $\exists h^i_c$ (height), representing the number of tokens in 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.
- Boolean variable $\exists mh^i_c$ (move-here) representing whether the move *i* was made in column *c*.
- Boolean variable $\exists pos^i_{r,c}$ representing the position of the empty slots in the column. $pos^i_{r,c}$ is 1 if slot *r* is free in column *c* prior to move *i*.

The constraints are given below for a single move i. The entire model is constructed by duplicating the move layers for the required $row \times col$ moves. Constraints are rearranged as reified disjunctions as shown in section 3.2.2.

1. If i is even, for each column c, connect u^i and m^i: $[gamestate^{i-1} = nil \wedge h_c^{i-1} \neq row \wedge u^i = c] \Rightarrow m^i = c$. If it is a legal move to place a token in column c, then $(u^i = c) \Rightarrow (m^i = c)$.

2. If i is even, a black counter is placed on the board: for all columns c, variable mh_c^i (move-here) is set according to whether the move is in this column: $line^{i-1} = 1 \vee h_c^{i-1} = row \vee m^i \neq c \Leftrightarrow mh_c^i \neq 1$

   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: $h_c^{i-1} = r-1 \Rightarrow pos_{r,c}^i = 1$ and
   $pos_{r,c}^i = 1 \Leftrightarrow [b_{r,c}^i \neq red \wedge b_{r+1,c}^i = nil \wedge b_{r+2,c}^i = nil \wedge \cdots \wedge b_{row,c}^i = nil]$. The move-variable m^i is connected to $b_{r,c}^i$: $[mh_c^i = 1 \wedge h_c^{i-1} = r-1] \Rightarrow b_{r,c}^i = black$ and $[mh_c^i \neq 1 \wedge h_c^{i-1} = r-1] \Rightarrow b_{r,c}^i = nil$.

   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, $b_{r,c}^{i-1} = red \Rightarrow b_{r,c}^i = red$ and $b_{r,c}^{i-1} = black \Rightarrow b_{r,c}^i = black$.

4. Link height and board state: for each column c and $r \in 1 \ldots (row+1)$, $b_{r-1,c}^i \neq nil \wedge b_{r,c}^i = nil \Rightarrow h_c^i = r-1$.

5. Detect lines: each set of four board variables that form a line (such as $b_{1,1}^i, b_{2,1}^i, b_{3,1}^i, b_{4,1}^i$) is given a unique number z and I refer to them as $b_{1\ldots4}^z$. If i is odd, for all z, $line^{i-1} = 1 \vee b_1^z \neq red \vee b_2^z \neq red \ \vee b_3^z \neq red \vee b_4^z \neq red \Leftrightarrow l_z^i \neq 1$. In words, $l_z^i = 1$ iff $b_{1\ldots4}^z$ 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_z^i variables, where $y = \max(z)$: $line^{i-1} \vee l_1^i \vee l_2^i \vee \cdots \vee l_y^i \Leftrightarrow line^i$.

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.

Board size		Nodes and time (s)			
col	*row*	Model	Nodes	Setup time	Search time
4	4	Baseline	4196	0.008	1.054
		Baseline+Bordeaux	26359046	0.007	6213.737
		Baseline without PV	92213	0.014	22.066
		QCSP-Solve			8235.647
		BlockSolve			>86400
4	5	Baseline	20856	0.004	9.130
		Baseline without PV	1042992	0.010	355.298
5	4	Baseline	168485	0.004	63.573
		Baseline without PV	28179488	0.017	9425.100
5	5	Baseline	2689288	0.006	1749.937
5	6	Baseline	22197560	0.016	16012.498

Table 1: Comparison of Connect 4 models and solvers for various parameters

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. $gamestate^0 = nil$, $line^0 = 0$, $b^0_{r,c} = nil$, $b^i_{0,c} = red$ (chosen arbitrarily, cannot be nil), $b^i_{row+1,c} = nil$, $h^0_c = 0$. The red player must win, so $gamestate^{row\times col} = red$. For the first move ($m^1$), symmetry is broken by removing the leftmost (lower) $\lfloor \frac{col}{2} \rfloor$ values, because they are equivalent to the higher values.

Constraint type 1 connects u^i to m^i. For each possible move a, if the move is legal and $u^i = a$, then $m^i = a$ is implied. However if move a is not legal, the constraint has no conflicts containing $u^i = a$. Since there are no other constraints containing u^i, a is a pure value, and can be pruned by the pure value rule. This avoids searching a subtree beneath an illegal move. There is always at least one legal move, so the corner case where the final value of a universal variable is pure never arises.

This model is referred to as Baseline. It was solved using Queso for five different board sizes, up to 5 columns and 6 rows. The pure value rule is applied to universal variables, since it is expensive to apply it to all variables, and it does little useful work on existentials. It is not yet feasible to solve the full size game using these methods in a reasonable amount of time. The experiment was run on a Pentium 4 3.06GHz with 1GB of RAM. The solver was set to output only true or false. (All instances used here are false.) The solution times are given in table 1, along with the number of nodes in the search tree. (The node count excludes leaf nodes.)

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 the solver to explore a much larger search tree, as shown in table 1 denoted *Baseline without PV*. Experimental details are the same as for the previous experiment.

Comparing reified disjunction with Bordeaux's primitives

The reified disjunction constraint is intended to improve upon the primitives of Bordeaux and Monfroy [14, 16] by providing significantly stronger consistency.

I will assume that when all variables represented in an expression $\mathscr{E}$ are existential, the consistency achieved on the decomposition is equivalent to applying QGAC 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 constraint rearranged as a reified disjunction (equation (7)) and the decomposition (equation (8)) are shown below.

$$\begin{gathered}\exists gamestate^{i-1}, h_c^{i-1}, \forall u^i, \exists m^i : \\ [(gamestate^{i-1} \neq nil) \vee (h_c^{i-1} = row) \vee (u^i \neq c) \vee (m^i = c) \Leftrightarrow 1\end{gathered} \tag{7}$$

$$\begin{gathered}\exists gamestate^{i-1}, h_c^{i-1}, \forall u^i, \exists m^i, t_1, t_2, t_3, t_4, t_5, t_6 : \\ (gamestate^{i-1} \neq nil \Leftrightarrow t_1) \wedge (h_c^{i-1} = row \Leftrightarrow t_2) \wedge (u^i \neq c \Leftrightarrow t_3) \\ \wedge (m^i = c \Leftrightarrow t_4) \wedge (t_1 \vee t_2 \Leftrightarrow t_5) \wedge (t_3 \vee t_5 \Leftrightarrow t_6) \wedge (t_4 \vee t_6 \Leftrightarrow 1)\end{gathered} \tag{8}$$

The hypothesis is that when solving Connect 4 with Bordeaux's decomposition, the solver will explore more nodes and take more time than with the integer reified disjunction primitive. I used the Baseline model, and simply replaced constraint set 1 with the decomposition in equation (8). QGAC is enforced on the ternary primitive constraints using the reified disjunction algorithm. All experimental details are the same as for the previous experiment. Table 1 shows the search time and number of nodes for Baseline and Baseline+Bordeaux, 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. It is not clear whether there may be another decomposition which performs better.

The Baseline model exploits the pure value rule and QGAC on constraint set 1. Switching off the pure value rule increases the node count 22-fold to 92213. However, using the decomposition gives a node count of 26 million. Consider the situations where $gamestate^{i-1} \neq nil$ or $h_c^{i-1} = row$. In the original formulation all values of u^i become pure, but 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, therefore the effect of the pure value rule is reduced by decomposition. However this is not sufficient to explain the node count; losing QGAC must play a large part as well.

Comparing against binary QCSP solvers

In order to generate a binary QCSP representation of Connect 4, a table (extensional) constraint model was encoded into binary QCSP. The table model is very similar to

Baseline in structure, sharing nearly all variables with Baseline. It is given elsewhere ([34], chapter 4). I do not reproduce it here because of space. Performing QGAC on the table model is slightly stronger than QGAC on Baseline [34].

An encoding into binary QCSP is required. The hidden variable encoding [35] is suitable, since it does not transform the variables (only adding some variables), therefore the quantifier sequence can be preserved (although with the addition of the hidden variables). The hidden variables are all existentially quantified to the right of the other variables. Also, binary constraints are passed through without any modification.

Both QCSP-Solve [27] and BlockSolve [41] have novel search algorithms which are not present in Queso, so both are interesting as a comparison. Both were run on Connect 4 4×4 as shown in table 1. The results were surprisingly bad, with QCSP-Solve taking more than two hours (as compared to 1s for Baseline) and reporting 27068010 search nodes, and BlockSolve running for more than 24 hours before the run was abandoned. The search algorithm of QCSP-Solve is more powerful than that of Queso, therefore the loss must be in reasoning at each node. This is not surprising since QCSP-Solve only implements forward checking. BlockSolve maintains arc-consistency during search, but its approach of bottom-up search does not work well on this instance. Also the pure value rule in QCSP-Solve does not function well on this instance. Switching off the pure value rule increases the node count to 27166112, therefore the rule is much less effective here than in Queso. However, even gaining the 22-fold improvement seen in Queso would not be sufficient for QCSP-Solve to compete with Queso — propagation is also vital, and forward checking is not sufficiently powerful.

It is likely that the problem could be re-modelled for QCSP-Solve or BlockSolve to obtain better results. However, such results would be incomparable with those for Queso, since the model would be significantly different. By using an encoding, I have minimised the difference between Baseline and the experiments with QCSP-Solve and BlockSolve.

5 Faulty Job Shop Scheduling in QCSP

Firstly I describe job shop scheduling, discuss sources of uncertainty in factory scheduling, and introduce the faulty job shop scheduling problem (FJSSP). Secondly some modelling issues are discussed, and FJSSP is modelled in QCSP in two different ways.

5.1 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 $1\dots m$ representing the machine that is required for task i of job j;
- $d(\mathscr{M}, j)$ is an integral constant representing the duration of the task in job j which runs on machine $\mathscr{M}$.

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

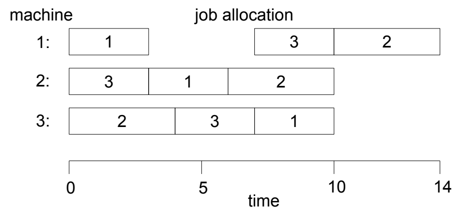


Table 2: Simple job shop scheduling problem and solution with optimal makespan

The symbols *tm* and *d* are indexed differently for convenience in writing out the constraints, but since every job has no more than 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*. The time units are numbered from 1 to *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. The optimization criterion for all experiments in this paper is to minimize the makespan. Table 2 shows a simple example of a job shop problem, and an optimal solution with a makespan of 14.

The JSSP allows at most one task from each job to use a particular machine. This simplifies presentation of the constraints, but the models presented below do not rely on this assumption.

5.2 Uncertainty in factory scheduling

Factory scheduling can have various sources of uncertainty. Each one can have immediate effect or a delayed or schedulable effect. Sources of uncertainty include: staff absences; order changes; machine faults or servicing; early or late delivery of raw materials; and uncertainty in task durations. Davenport and Beck survey approaches to scheduling with uncertainty, with the following broad divisions [24].

- 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.

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.

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: 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 (i.e. with minimal makespan) or close to optimal whether a fault occurs or not. It can be sensibly modelled as a QCSP with universal variables representing faults.

5.3 Faulty job shop scheduling

To introduce fault handling to the JSSP, the time units are grouped into a number of *periods* of equal size, in order to deal with machine testing and servicing. Each machine $\mathscr{M}$ 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 time units 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.

5.4 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 (marginal) probability to each fault, and a probability threshold ϕ to the whole problem. A formula is used to compute the joint probability of a set of events (either faults or non-faults). Only combinations of events which are sufficiently likely (i.e. with probability $\geq \phi$) are considered. For the purposes of this paper, the faults are assumed to be independent, because this makes the joint probability calculation straightforward. However this assumption can be relaxed by replacing the formula which is used to compute the joint probability of the faults. Any formula can be used as long as it can be represented with the constraints supplied by the QCSP solver.

In the models presented in section 5.6, the probability of each fault is represented as a constant. The models could be easily generalized by using a variable for the probability, 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 probabilities are involved, it is reasonable to ask whether Stochastic CSP [6, 32,39,42] would be more suitable than QCSP. 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 considering exactly the set of scenarios whose probability exceeds a threshold. In a QCSP model the modeller can control exactly the set of scenarios which are scheduled, whereas in Stochastic CSP the set of scenarios is chosen by the solver such that the probability of satisfying the chance constraints exceeds a threshold. The penalty for the greater control in QCSP is that the model is likely to be more complex.

One approach to Stochastic CSP is 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 paper 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 5.6.4 below. The other approach to Stochastic CSP, by Balafoutis and Stergiou [6], is similar to top-down search for QCSP, but only table constraints are supported at present. Balafoutis and Stergiou develop a propagation algorithm generalized from GAC2001/3.1. It would be very difficult to compactly represent FJSSP using only stochastic table constraints, therefore I do not compare against this approach. In summary both approaches to Stochastic CSP have significant disadvantages and I did not compare against them empirically.

5.5 CSP model

Before considering FJSSP, I describe a model of JSSP as a constraint optimization problem. This is referred to as the *CSP model*, and provides a basis for modelling FJSSP in QCSP in later sections. The central part of this model (the disjunctive constraint) is given by Baptiste et al. [7] (section 2.1.2). 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 $b^{\mathscr{M}}_{j_1,j_2}$ representing the order of two tasks, belonging to distinct jobs j_1 and j_2, which both contend for machine $\mathscr{M}$.

- mn integer variables $start_j^{\mathscr{M}}$ representing the start time of the task from job j which runs on machine $\mathscr{M}$.
- mn integer variables $end_j^{\mathscr{M}}$ 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 over all tasks.

The integer variables are all initially bounded between 0 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 5.1) that the duration of a task in job j, which runs on machine $\mathscr{M}$, is $d(\mathscr{M}, j)$. The task in position i for job j runs on machine $tm(i, j)$.

1. For each $\mathscr{M}, j$: $start_j^{\mathscr{M}} + d(\mathscr{M}, j) = end_j^{\mathscr{M}}$
2. For each $\mathscr{M}, j_1, j_2 > j_1$: $b_{j_1,j_2}^{\mathscr{M}} \Leftrightarrow [end_{j_1}^{\mathscr{M}} \leq start_{j_2}^{\mathscr{M}}] \wedge \neg b_{j_1,j_2}^{\mathscr{M}} \Leftrightarrow [start_{j_1}^{\mathscr{M}} \geq end_{j_2}^{\mathscr{M}}]$
3. For each $i \in \{1 \ldots m-1\}, j$: $end_j^{tm(i,j)} \leq start_j^{tm(i+1,j)}$
4. $\max(\{end_j^{tm(m,j)} | j \in \{1 \ldots n\}\}) = opt$

The disjunctive constraint of Baptiste et al. [7] is represented as two reified comparisons (constraint type 2). It ensures that two tasks which require the same machine do not run at the same time. Also, the b variable represents the order of the two tasks. 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.

- For each $\mathscr{M}, j_1, j_2 > j_1$: $start_{j_1}^{\mathscr{M}} + d(\mathscr{M}, j_1) \leq start_{j_2}^{\mathscr{M}} \vee start_{j_1}^{\mathscr{M}} \geq start_{j_2}^{\mathscr{M}} + d(\mathscr{M}, j_2)$

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. Despite the availability of edge-finding techniques, this simple model is seen in the literature [8, 37], 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 [21] (first applied in the constraints context by Caseau and Laburthe [22]), and other specialized approaches such as shaving [33]. Both shaving and specialized constraints are applied to a model like the one above, where the start times of tasks are represented directly as variables.

The aim of this paper 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.

5.6 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 and other advanced scheduling constraints. This implies that there are variables representing the starting time of each task. The QCSP model of FJSSP must not be exponentially larger than the CSP model of JSSP, because the additional uncertainty can be represented compactly using universal variables.

5.6.1 Use of CSP constraints

In the models in this section, I use various constraint propagation algorithms without change from CSP. These algorithms are applied to constraints which have no universal variables in their scope. Bordeaux et al. [15] observe that their definition of quantified inconsistency is equivalent to the classical CSP definition of inconsistency when all quantifiers are existential, therefore there is no advantage in defining quantified propagators for these constraints. I use the following CSP constraints: product ($x_1 \times x_2 = x_3$), comparison ($(x_1 < x_2) \Leftrightarrow x_3$), sum ($\sum_{i=0}^{r} c_i x_i = 0$) and max ($\max(x_1 \dots x_{r-1}) = x_r$), all enforcing bounds($\mathbb{R}$)-consistency [23].

5.6.2 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 potential fault with machine $\mathscr{M}$ in period a is modelled with a universal Boolean variable $unifault_a^{\mathscr{M}}$. The constant $p(\mathscr{M}, a)$ represents the estimated probability of the fault. It is assumed that $p(\mathscr{M}, a) \leq 0.5$ so that an additional fault will never increase the probability of a scenario, hence it is possible to know that a scenario falls below the probability threshold without examining all combinations of future faults. This assumption simplifies the model considerably. It is also assumed that $p(\mathscr{M}, a)$ is rational, that all faults are independent, and that $p(\mathscr{M}, a) \neq 0.5$ (because of the form of the constraints containing $p(\mathscr{M}, a)$; these could be reformulated if necessary). A total ordering

$\prec$ is imposed on the faults (specified by the pair $\langle \mathscr{M}, a\rangle$) and this ordering is the same as the ordering of the $\mathit{unifault}_a^{\mathscr{M}}$ variables in the quantifier sequence. Variable $\mathit{precp}_a^{\mathscr{M}}$ is the probability of all events $\langle \mathscr{M}', a'\rangle$ that preceed $\langle \mathscr{M}, a\rangle$: $\langle \mathscr{M}', a'\rangle \prec \langle \mathscr{M}, a\rangle$. This is the product of the probabilities $p(\mathscr{M}', a')$ of those faults which did occur ($\mathit{fault}_{a'}^{\mathscr{M}'} = 1$) with the complement $1 - p(\mathscr{M}', a')$ for those faults which did not occur ($\mathit{fault}_{a'}^{\mathscr{M}'} = 0$).

A constant $\mathit{succp}_a^{\mathscr{M}}$ is calculated for each fault, which is the product of the probabilities of the complement of all succeeding faults $\langle \mathscr{M}', a'\rangle \succ \langle \mathscr{M}, a\rangle$. In words, it is assumed that all later faults do not occur (the most probable outcome) and a probability is calculated for them all based on this.

The variable $\mathit{thisp}_a^{\mathscr{M}}$ is the probability of the scenario where fault $\langle \mathscr{M}, a\rangle$ does occur, all succeeding faults do not occur and the occurrence of preceding faults is decided by their respective *fault* variables. $\mathit{thisp}_a^{\mathscr{M}}$ is computed as follows: $\mathit{thisp}_a^{\mathscr{M}} = \mathit{precp}_a^{\mathscr{M}} \times \mathit{succp}_a^{\mathscr{M}} \times p(\mathscr{M}, a)$. There is a Boolean variable $\mathit{available}_a^{\mathscr{M}}$ which indicates whether the probability of the scenario is above or equal to the threshold. Finally, if $\mathit{available}_a^{\mathscr{M}} = 1$ then the value of $\mathit{unifault}_a^{\mathscr{M}}$ is copied to a second variable $\mathit{fault}_a^{\mathscr{M}}$. $\mathit{fault}_a^{\mathscr{M}}$ determines whether servicing takes place for machine $\mathscr{M}$ during period a.

All variables in Queso are integral. The probabilities (which must be rational) are each represented with an integer variable numerator and a constant denominator. A rational variable q_1 is represented as $\frac{x_1}{c_1}$. The constants c are computed from the constraints and fault probabilities during construction of the model. For example the constraint $q_1 = q_2 \times q_3$ would be translated to $\frac{x_1}{c_1} = \frac{x_2}{c_2} \times \frac{x_3}{c_3}$ then $x_1 = x_2 \times x_3$ and $c_1 = c_2 \times c_3$. The acyclic structure of the model allows all constants c_i to be determined as the model is constructed. This scheme is clearly not sufficient to represent any set of constraints over the rationals, but it is sufficient for this model. All probability variables ($\mathit{thisp}_a^{\mathscr{M}}$, $\mathit{precp}_a^{\mathscr{M}}$ and $\mathit{fault}_a^{\mathscr{M}}$) have initial domain $\{0 \dots c\}$ where c is the constant denominator associated with the variable. (The other variables $\mathit{available}_a^{\mathscr{M}}$, $\mathit{unifault}_a^{\mathscr{M}}$ and $\mathit{fault}_a^{\mathscr{M}}$ are Boolean.)

The constraints linking $\mathit{thisp}_a^{\mathscr{M}}$, $\mathit{precp}_a^{\mathscr{M}}$, $\mathit{available}_a^{\mathscr{M}}$, $\mathit{unifault}_a^{\mathscr{M}}$ and $\mathit{fault}_a^{\mathscr{M}}$ are shown here.

$$\mathit{thisp}_a^{\mathscr{M}} = \mathit{precp}_a^{\mathscr{M}} \times (\mathit{succp}_a^{\mathscr{M}} \times p(\mathscr{M}, a))$$

$$\mathit{available}_a^{\mathscr{M}} \Leftrightarrow \mathit{thisp}_a^{\mathscr{M}} \geq \phi$$

$$\mathit{fault}_a^{\mathscr{M}} \Leftrightarrow \mathit{available}_a^{\mathscr{M}} \wedge \mathit{unifault}_a^{\mathscr{M}}$$

Notice that the only constraint containing a universal variable, the constraint linking $\mathit{unifault}_a^{\mathscr{M}}$ to $\mathit{fault}_a^{\mathscr{M}}$, cannot be represented directly as either a binary constraint or in Bordeaux and Monfroy's framework [14, 16]. (Bordeaux does not give propagation rules for conjunction [14].) It can be reformulated as a disjunction, with three additional negation constraints as shown below.

$$\exists t_1, t_2, t_3 : (t_1 \Leftrightarrow t_2 \vee t_3), (t_1 \Leftrightarrow \neg \mathit{fault}_a^{\mathscr{M}}), (t_2 \Leftrightarrow \neg \mathit{available}_a^{\mathscr{M}}), (t_3 \Leftrightarrow \neg \mathit{unifault}_a^{\mathscr{M}})$$

If $\mathit{fault}_a^{\mathscr{M}} = 0$, the reified disjunction constraint is able to infer $\mathit{available}_a^{\mathscr{M}} = 0$, whereas Bordeaux and Monfroy's primitives are not able to make any inferences. This provides evidence that the reified disjunction constraint is worthwhile.

The $\mathit{precp}_a^{\mathscr{M}}$ variable must be linked to the previous *precp* in the ordering $\prec$. This is done by introducing another variable $\mathit{faultp}_a^{\mathscr{M}}$ which is the probability of the event which occurred (i.e. if the fault occurred then $\mathit{faultp}_a^{\mathscr{M}} = p(\mathscr{M}, a)$ and if not then $\mathit{faultp}_a^{\mathscr{M}} = 1 - p(\mathscr{M}, a)$). The three constraints to achieve this are shown below.

$$(\mathit{faultp}_a^{\mathscr{M}} = p(\mathscr{M}, a)) \Leftrightarrow \mathit{fault}_a^{\mathscr{M}}$$

$$(\mathit{faultp}_a^{\mathscr{M}} = 1 - p(\mathscr{M}, a)) \Leftrightarrow \neg \mathit{fault}_a^{\mathscr{M}}$$

$$\text{if } \mathscr{M} = 1 \text{ and } a \neq 1 : \mathit{precp}_a^{\mathscr{M}} = \mathit{precp}_{a-1}^{\mathscr{M}} \times \mathit{faultp}_{a-1}^{\mathscr{M}}$$

$$\text{if } \mathscr{M} \neq 1 : \mathit{precp}_a^{\mathscr{M}} = \mathit{precp}_a^{\mathscr{M}-1} \times \mathit{faultp}_a^{\mathscr{M}-1}$$

$$\text{if } \mathscr{M} = 1 \text{ and } a = 1 : \mathit{precp}_a^{\mathscr{M}} = 1$$

The quantifier subsequence for these variables is shown below.

$$\exists \mathit{precp}_a^{\mathscr{M}}, \mathit{thisp}_a^{\mathscr{M}}, \mathit{available}_a^{\mathscr{M}}, \forall \mathit{unifault}_a^{\mathscr{M}}, \exists \mathit{fault}_a^{\mathscr{M}}, \mathit{faultp}_a^{\mathscr{M}}$$

The $\mathit{unifault}_a^{\mathscr{M}}$ variable is included in only one constraint. If $\mathit{available}_a^{\mathscr{M}}$ is set to 0, then $\mathit{fault}_a^{\mathscr{M}}$ is set to 0 by propagation and both values of $\mathit{unifault}_a^{\mathscr{M}}$ 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 interval variables, represented in the solver with only their upper and lower bounds.

All the above variables and constraints are shared by model A and model B below.

5.6.3 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. Model A is somewhat similar to a time-indexed formulation of scheduling in integer programming [40].

For each time unit s and each machine $\mathscr{M}$, there is a variable $\exists t_s^{\mathscr{M}} \in \{1, 2, \ldots, n, \mathit{idle}, \mathit{servicing}\}$. $t_s^{\mathscr{M}}$ represents the job that $\mathscr{M}$ is running at time s, or whether it is being serviced or is idle. $\mathit{start}_j^{\mathscr{M}}$, $\mathit{end}_j^{\mathscr{M}}$ and *opt* variables are copied from the CSP model in section 5.5. Constraint types 1, 3 and 4 are copied from the CSP model. One other type of constraint (equation 9) referred to as the *channelling* constraint is required to channel between the $t_s^{\mathscr{M}}$ variables and the *start* and *end* variables. This is assumed to be a single constraint for simplicity.

$$\text{For each } \mathscr{M}, j, s: \left[start_j^{\mathscr{M}} \leq s \wedge end_j^{\mathscr{M}} > s\right] \Leftrightarrow t_s^{\mathscr{M}} = j \tag{9}$$

The $t_s^{\mathscr{M}}$ variables and the $start_j^{\mathscr{M}}$ and $end_j^{\mathscr{M}}$ variables are two representations of the job shop scheduling problem. The reason for having both is that the $t_s^{\mathscr{M}}$ variables can be quantified in chronological order and they enforce mutual exclusion of tasks on machines, and the $start_j^{\mathscr{M}}$ and $end_j^{\mathscr{M}}$ representation enforces that the tasks have the correct duration and occur in the order required for the job.

Each period has length *plen* and $periods = maxmakespan/plen$. The time required for servicing a machine is *servicetime*. For a machine $\mathscr{M}$ and period a, the variable $fault_a^{\mathscr{M}}$ must be connected to the appropriate time unit variables $t^{\mathscr{M}}$. If $fault_a^{\mathscr{M}} = 1$, servicing must be scheduled on machine $\mathscr{M}$ during period a. This is done by introducing variables $servicestart_a^{\mathscr{M}} \in \{((a-1) \times plen) \dots (a \times plen - 1)\}$ and $serviceend_a^{\mathscr{M}}$ with the same domain, and the constraints below. If $fault_a^{\mathscr{M}} = 0$, then $servicestart_a^{\mathscr{M}}$ (and $serviceend_a^{\mathscr{M}}$) are fixed, to avoid the search algorithm branching for each of its values.

$$\text{For each } \mathscr{M}, a: \ servicestart_a^{\mathscr{M}} + servicetime = serviceend_a^{\mathscr{M}}$$

$$\text{For each } \mathscr{M}, a: fault_a^{\mathscr{M}} = 0 \Rightarrow servicestart_a^{\mathscr{M}} = (a \times plen + 1)$$

$$\begin{gathered}\text{For each } \mathscr{M}, a, \text{ for each } s \in \{((a-1) \times plen) \dots (a \times plen - 1)\}: \\ \left[fault_a^{\mathscr{M}} = 1 \wedge servicestart_a^{\mathscr{M}} \leq s \wedge serviceend_a^{\mathscr{M}} > s\right] \Leftrightarrow t_s^{\mathscr{M}} = servicing\end{gathered}$$

To implement the constraints with $\leq$ and $>$, a 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 $fault_a^{\mathscr{M}}$ and $t_s^{\mathscr{M}}$.

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 unit 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) The following sequence is repeated for each machine $\mathscr{M}$:
   $\exists precp_a^{\mathscr{M}}, thisp_a^{\mathscr{M}}, available_a^{\mathscr{M}}, \forall unifault_a^{\mathscr{M}}, \exists fault_a^{\mathscr{M}}, faultp_a^{\mathscr{M}}$
   (b) For each time unit $s \in \{(a-1) \times plen \dots a \times plen - 1\}$ and each machine $\mathscr{M}$: $\exists t_s^{\mathscr{M}} \in \{1 \dots n, idle, servicing\}$
2. For each machine $\mathscr{M}$ and period a: $\exists servicestart_a^{\mathscr{M}}, serviceend_a^{\mathscr{M}} \in \{((a-1) \times plen) \dots (a \times plen)\}$
3. For each machine $\mathscr{M}$ and job j: $\exists start_j^{\mathscr{M}}, end_j^{\mathscr{M}} \in \{0 \dots maxmakespan\}$

4. $\exists opt \in \{0 \ldots maxmakespan\}$

There are two main reasons that model A is problematic. Firstly, the model is not compact enough for propagation to be efficient. There are $O(mn \times maxmakespan)$ 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 $t^1_{2\ldots10}$ are all set to 1 by propagation, which causes $10n$ channelling constraints to be woken up by changes to $t^1_{1\ldots10}$ 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 $t^1_{11\ldots makespan}$ may have value 1 removed, which could potentially wake up a further $(maxmakespan - 10)n$ 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 $t^{\mathscr{M}}_s$ 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.

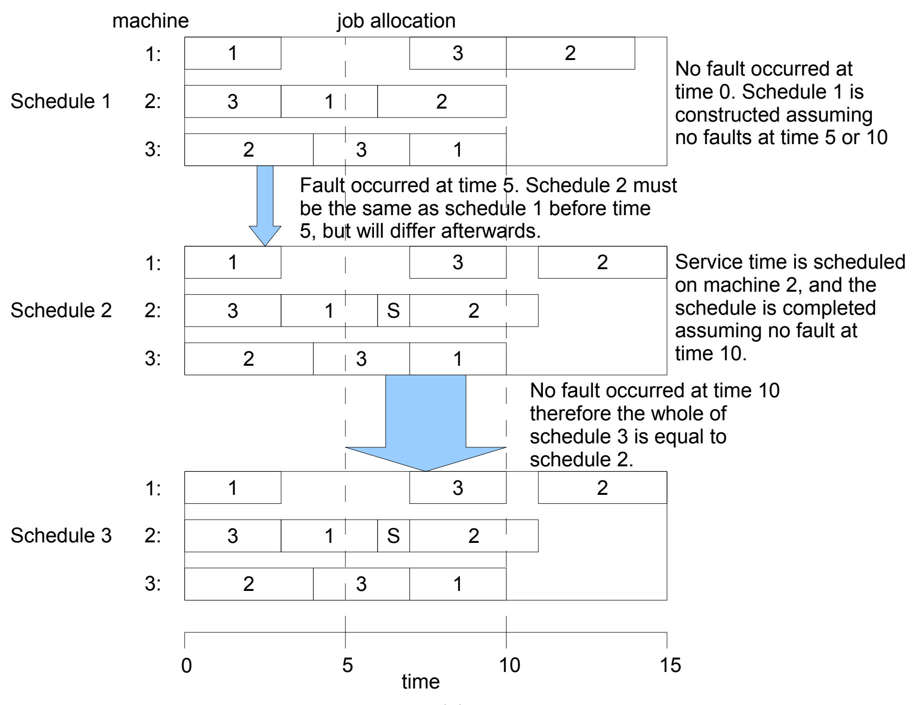
5.6.4 Model B

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. This model contains redundancy, because there are *periods* copies of the entire schedule represented in the QCSP instance. The reason for the copies is to allow rescheduling for faults. The schedules are numbered from $1 \ldots periods$. They are connected by conditional equality constraints: for schedules a and $a+1$, if a fault occurs in period $a+1$, then all tasks in schedule a which started within periods $1 \ldots a$ have the same start time in schedule $a+1$. Other tasks may need to be scheduled differently, with servicing time included. If no fault occurs in period $a+1$, then the entire schedule $a+1$ equals schedule a. When all variables are assigned, the final schedule in the sequence accounts for all occurrences and non-occurrences of faults in the scenario.

The variables to model faults (shown in section 5.6.2) are quantified between instances of the CSP model. The fault variables for period a are quantified between schedule $a-1$ and schedule a. Operationally, this means Queso solves the whole job shop instance for schedule 1, assuming no faults occur for periods $2 \ldots periods$. Then Queso branches on the universal variables $unifault_2$. 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. If it proves impossible to construct schedule 2, then Queso backtracks into schedule 1 and changes that if possible. The b variables are preserved from the CSP model, and are searched on. Model B is illustrated for a small example in figure 2.

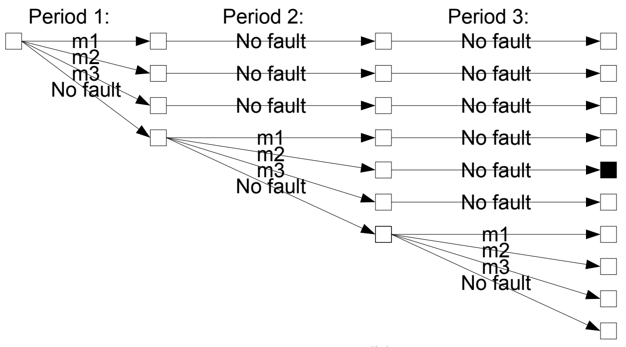
For each period a, an extra existential variable $nofault_a \in \{0,1\}$ is introduced which is used for copying forward the entire schedule when no faults occur. The following constraints are introduced.

$$\text{For each } a : \; nofault_a \Leftrightarrow \neg fault^1_a \wedge \cdots \wedge \neg fault^m_a$$



machine
job allocation
Schedule 1
Schedule 2
Schedule 3
1:
2:
3:
No fault occurred at time 0. Schedule 1 is constructed assuming no faults at time 5 or 10
Fault occurred at time 5. Schedule 2 must be the same as schedule 1 before time 5, but will differ afterwards.
Service time is scheduled on machine 2, and the schedule is completed assuming no fault at time 10.
No fault occurred at time 10 therefore the whole of schedule 3 is equal to schedule 2.
S
0
5
10
15
time

(a)



(b)

Figure 2: (a) Representation of model B on a problem (shown in table 2) with 3 machines, 3 jobs and 3 periods of length 5. Since there are three periods, there are three schedules represented in the model. There are nine possible faults. A solution for the scenario where one fault occurs on machine 2 in period 2 is illustrated. The final schedule (3) is the one which fully accounts for the faults which did and did not occur in this scenario.
(b) Assume that all fault probabilities are equal, and only one fault may occur within the threshold probability. There are ten scenarios in total, nine with one fault and one with no faults. This tree shows an approximation of how Queso branches on the universal variables in model B, assuming no failures occur. After branching for period 1, the solver constructs schedule 1. Then it branches for period 2, constructs schedule 2 etc, in a depth-first search. The scenario illustrated in figure (a) is marked with a solid black box.

Copies of the CSP model are introduced for each period, with each variable superscripted with the period number. For example, $start_j^{\mathscr{M},a}$ is the starting time of the task from job j which requires machine $\mathscr{M}$ 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 $servicestart_a^{\mathscr{M}}$ and $serviceend_a^{\mathscr{M}}$ are introduced with the same meaning and domain as in model A. They are linked to the CSP model with the following constraints. The $\tau_{j,1}^{\mathscr{M},a}, \tau_{j,2}^{\mathscr{M},a}$ variables represent task ordering, and are existentially quantified.

$$\text{For each } \mathscr{M}, a: \; servicestart_a^{\mathscr{M}} + servicetime = serviceend_a^{\mathscr{M}}$$

$$\text{For each } \mathscr{M}, a: \; \neg fault_a^{\mathscr{M}} \Rightarrow servicestart_a^{\mathscr{M}} = (a \times plen + 1)$$

$$\begin{gathered}\text{For each } \mathscr{M}, a, j: \\ fault_a^{\mathscr{M}} \Rightarrow \tau_{j,1}^{\mathscr{M},a} \vee \tau_{j,2}^{\mathscr{M},a} \\ end_j^{M,a} \leq servicestart_a^{\mathscr{M}} \Leftrightarrow \tau_{j,1}^{\mathscr{M},a} \\ serviceend_a^{\mathscr{M}} \leq start_j^{\mathscr{M},a} \Leftrightarrow \tau_{j,2}^{\mathscr{M},a}\end{gathered}$$

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 $start_j^{\mathscr{M},a}$ lies within periods $1 \ldots a$ (i.e. $start_j^{\mathscr{M},a} \leq a \times plen$). σ_2 indicates whether $start_j^{\mathscr{M},a}$ is copied to the next period. $start_j^{\mathscr{M},a}$ must be copied forward if there are no faults in period $a+1$ (therefore $nofault_{a+1} \Rightarrow \sigma_2$) *or* the task started within periods $1 \ldots a$ (therefore $\sigma_1 \Rightarrow \sigma_2$).

$$\begin{gathered}\text{For each } a \in \{1 \ldots periods - 1\}, \mathscr{M}, j: \\ \neg\sigma_1 \vee \sigma_2 \\ \neg nofault_{a+1} \vee \sigma_2 \\ [start_j^{\mathscr{M},a} \leq a \times plen] \Leftrightarrow \sigma_1 \\ [start_j^{\mathscr{M},a} = start_j^{\mathscr{M},a+1}] \Leftrightarrow \sigma_2\end{gathered}$$

Finally some implied constraints are added for efficiency. The ordering variables b and σ entirely specify the order of tasks. When these variables are assigned, all that is needed to complete a schedule is to set the start time of each task. In early experiments we experienced thrashing on the *start* variables. In the context of makespan minimization, a schedule where the tasks are tightly packed is always preferable. The following constraints state that tasks are tightly packed, by stating that the start time of a task $start_j^{\mathscr{M},a}$ is equal to one of the following possibilities: the end time of the previous task in the job; the end time of any other task which runs on machine $\mathscr{M}$ (including servicing tasks); the start of the schedule; or the start of any period $2 \ldots a+1$. This last possibility requires some explanation. For schedule a, it may be necessary to start the task at the beginning of period $a+1$ so that it is not copied forward into schedule $a+1$. Since the start time is not copied forward, the task can be moved in response to servicing in period $a+1$. (It may be necessary for the task to start at the beginning of period $a+1$ if there is no fault, and later if there is a fault in period $a+1$.) The same argument also applies to all schedules $1 \ldots a-1$, therefore the task may begin at the start of any period $2 \ldots a+1$.

The constraints are constructed as follows. For each $a \in \{1 \ldots \mathit{periods}\}, \mathscr{M}, j$, there is a Boolean reified disjunction of the form $\gamma_1 \vee \cdots \vee \gamma_n \Leftrightarrow 1$, and for each γ_d there is a constraint such as $(\mathit{start}_j^{\mathscr{M},a} = \mathit{end}_{j-1}^{\mathscr{M},a}) \Leftrightarrow \gamma_d$. Each γ_d represents one possible value for $\mathit{start}_j^{\mathscr{M},a}$ from the list of possibilities above.

Bounds($\mathbb{D}$)-consistency 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 $\mathscr{M}$: $\exists \mathit{precp}_a^{\mathscr{M}}, \mathit{thisp}_a^{\mathscr{M}}, \mathit{available}_a^{\mathscr{M}}, \forall \mathit{unifault}_a^{\mathscr{M}}, \exists \mathit{fault}_a^{\mathscr{M}}, \mathit{faultp}_a^{\mathscr{M}}$
   (b) $\exists \mathit{nofault}_a \in \{0,1\}$
   (c) For each pair of jobs j_1, j_2 and each machine $\mathscr{M}$, $\exists b_{j_1,j_2}^{\mathscr{M},a} \in \{0,1\}$
   (d) For each machine $\mathscr{M}$ and job j, $\exists \tau_{j,1}^{\mathscr{M},a}, \tau_{j,2}^{\mathscr{M},a} \in \{0,1\}$
   (e) For each machine $\mathscr{M}$ and job j, $\exists \mathit{start}_j^{\mathscr{M},a}, \mathit{end}_j^{\mathscr{M},a} \in \{0 \ldots \mathit{maxmakespan}\}$
   (f) For each machine $\mathscr{M}$ and period a, $\exists \mathit{servicestart}_a^{\mathscr{M}}, \mathit{serviceend}_a^{\mathscr{M}} \in \{((a-1) \times \mathit{plen}) \ldots (a \times \mathit{plen} - 1)\}$
   (g) For all pairs of σ variables for this period, $\exists \sigma_1, \sigma_2 \in \{0,1\}$
   (h) For all γ variables in this period, $\exists \gamma \in \{0,1\}$
2. $\exists \mathit{opt} \in \{0 \ldots \mathit{maxmakespan}\}$

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 branching on the *start* variables. 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.

5.7 Why use QGAC

Why use quantified notions of consistency such as QGAC to solve this problem? Constraint Logic Programming solvers such as Eclipse [1] 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, this is a weak approach because the solver does not distinguish between existential and universal variables, and cannot do any extra reasoning on universal variables.

If the servicing time for machine $\mathscr{M}$ cannot be scheduled in period a, then $\mathit{fault}_a^{\mathscr{M}}$ is set to 0 by propagation on other constraints (which are specific to models A and B). At this point, if $\mathit{available}_a^{\mathscr{M}}$ is set to 1, then the $\mathit{unifault}_a^{\mathscr{M}}$ variable is pruned, and the solver backtracks immediately. In this way the solver exploits the universal variables. Since QGAC on reified disjunction is cheap and search is inherently expensive, it seems very likely that enforcing QGAC is worthwhile, although it remains an empirical question.

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 [32, 39]. 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 or 226 scenarios within the probability bound, therefore this approach is possible if we only consider the 16 or 226 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 may determine failure. This is not the case when applying QGAC to QCSP model B. However, propagation would be more expensive in this scheme, and the set of scenarios must be small.

6 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 pure value rule is applied to universal variables (i.e. $\mathit{unifault}_a^{\mathscr{M}}$ variables) but not existentials. This is because it is expensive to apply the pure value rule to all variables and it rarely does any useful work on existentials. This is the same situation as in Connect 4.

The ten problem instances used here have $n = m = 5$ and are derived from the OR-LIB instances LA01 to LA10[3]. LA01 to LA10 have five machines but ten or fifteen jobs. I used only the first five jobs and deleted the others, creating instances LA01-5×5 to LA10-5×5. 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}.

6.1 Comparing non-contingent with contingent scheduling

A typical approach to machine breakdown is to add extra time to a schedule [24]. Therefore I compare the length of the schedules generated by the two approaches.

[3]Available from http://people.brunel.ac.uk/~mastjjb/jeb/orlib/

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.

6.1.1 Experiments with at most one fault

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 LA01-5×5 to LA10-5×5 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.

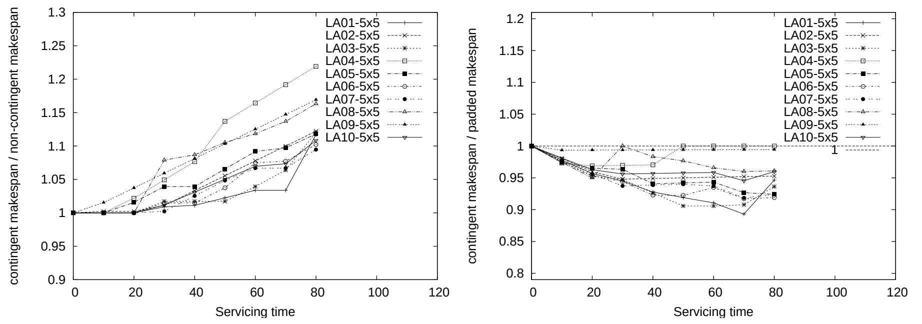
The *searchOpt* algorithm was used to generate contingent schedules. In these schedules, the parameter which is minimized is the maximum of the schedule length for each scenario. The output of the solver is the minimized parameter.

Results

Figure 3(a) 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 [43] between *servicetime* and makespan is 0.8025 (instance LA01-5×5) and the highest is 0.9995 (for LA09-5×5). 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 LA01-5×5 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 $\mathscr{M}$ during a period a when $\mathscr{M}$ is constantly in use, and that the



(a) Effect of servicing on worst-case makespan (b) Contingent worst-case makespan against padded makespan

Figure 3: Comparing contingent and padded non-contingent schedules

fault increases the makespan by *servicetime* time units. Therefore the non-contingent approach is to generate an optimal schedule and add *servicetime* to the makespan to allow for one servicing task. During execution of the schedule, servicing during period a on machine $\mathscr{M}$ would potentially cause tasks on that machine to start later, which would also affect tasks on other machines according to the precedence constraints. This increases the makespan by up to *servicetime* time units. The schedule is padded at the end to allow for overrun, and is referred to as *padded non-contingent schedule*.

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 $\mathscr{M} = 1$ and period $a = 1$ where the machine is constantly in use, and adding servicing to this period does increase the makespan by *servicetime* time units. I did not inspect the schedules for the other instances for this property.

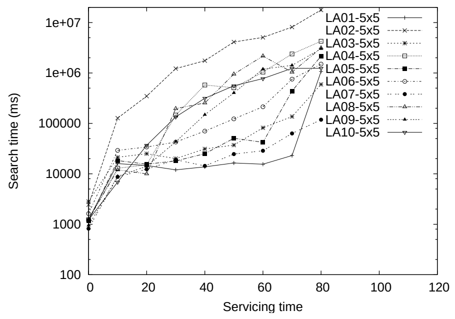
Figure 3(b) plots the ratio between the worst-case makespan with contingency and the makespan of a padded non-contingent schedule. At almost all points, the contingent schedules have a shorter makespan. The improvement is up to 10%. This is evidence in favour of the first hypothesis.

Unfortunately the QCSP algorithms do not scale well as *servicetime* is increased. Search time and search nodes are plotted in figures 4(a) and (b). The experiment was run on a P4 3.06GHz with 1GB of RAM, using Sun Java 1.6 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 0.19 to 0.78.

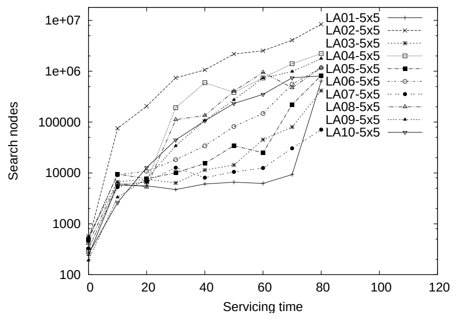
The number of nodes does not scale well for some instances. The time spent at each node decreases for long searches, but the overall effect is that search time does not scale well. This is evidence in favour of the second hypothesis.

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 459 to 8.4 million



(a) Search time



(b) Search nodes

Figure 4: Search time and nodes plotted against *servicetime*

as the servicing time is increased from 0 to 80.

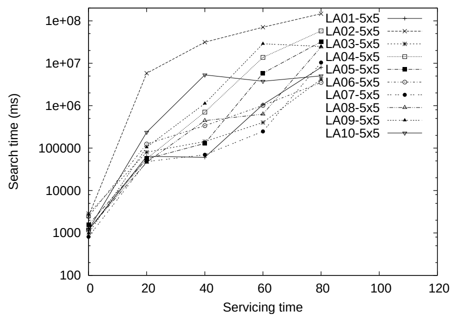
Queso schedules the periods in order. Consider the situation where the first period is scheduled, and a set of decisions made in the first period are incompatible with all valid schedules for the final period. Assume intermediate periods can be scheduled. Queso will reach the final period, detect the 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 effect could be reduced by using an extension to search such as CBJ [27] or conflict learning (used in QBF [29]), both of which reduce thrashing by avoiding re-discovering conflicts, and therefore are likely to be very helpful in this situation.

6.1.2 Experiments with at most two faults

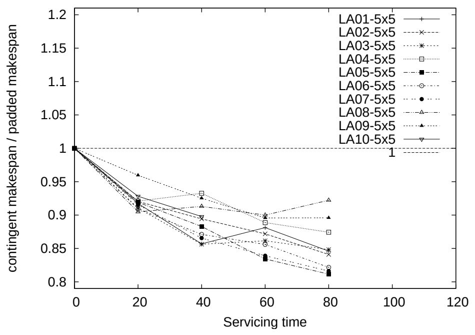
As mentioned in section 5.7, it is possible to solve QCSP by expanding it out into a CSP. This is feasible for the experiments above, where there are 16 scenarios. However, if we allow at most two faults to occur in each scenario, the number of scenarios is 226, approximately 14 times as many. This is likely to make the expansion to CSP infeasible. However it is also likely to degrade the performance of the QCSP algorithms. Also, when comparing contingency to the naive padding approach, there is the potential for greater savings with two faults than one.

Method

The same ten instances are used, with the same value for *maxmakespan* (600) and the same number of periods (3). The fault probability is 0.05 and the threshold is 0.001, therefore each scenario contains at most two faults, and there are 226 scenarios. As before, the *searchOpt* algorithm is used. To save computation time, *servicetime* is



(a) Search time



(b) Contingent worst-case makespan against padded makespan

Figure 5: Results for experiments with at most two faults

increased in increments of 20 rather than 10, but in all other respects the parameters are the same.

Results

Figure 5(a) shows search time. LA02-5×5 is a difficult instance, and when *service-time* is 80 it has the longest search time whether we are considering two faults or one (compare to figure 4(a)). With two faults, the search time is 147030s, and with one fault it is 17744s, a ratio of 8.29. This is less than the 14-fold increase in the number of scenarios. However, the mean ratio of search time is 14.3, averaged for *servicetime* $\in \{20, 40, 60, 80\}$ and for all 10 problems. The ratio of the number of nodes, averaged in the same way, is 13.6. It is more difficult to solve the instances with more scenarios, and the increase in difficulty approximately matches the increase in the number of scenarios.

Figure 5(b) plots the maximum makespan for the contingent schedule against a non-contingent schedule padded with $2 \times$ *servicetime*. As expected, the contingent approach gives a greater saving when we have two faults (compare to figure 3(b)).

7 Conclusion

I have presented two main contributions to solving QCSP. The first is an algorithm to propagate reified disjunction constraints. This dominates the logic primitives of Bordeaux and Monfroy [14, 16], and proves useful with Connect 4 and job shop scheduling. The second contribution is the non-binary pure value rule. This is implemented by reusing constraint propagation algorithms, and again proves useful for both job shop scheduling and Connect 4. Also, I presented a simple optimization algorithm.

A model of Connect 4 was presented, which exploits both the reified disjunction constraint and the pure value rule. This was used to evaluate the pure value rule, by solving Connect 4 at various board sizes with and without the pure value rule. The pure value rule is successful in reducing the number of search nodes by over 150 times on the board with 5 columns and 4 rows. The reified disjunction propagator was evaluated by comparing it with Bordeaux's decomposition. The reified disjunction propagator performed more than 5000 times better in both time and search nodes.

A binary encoding of Connect 4 was developed in order to compare Queso against QCSP-Solve and BlockSolve. On the board with 4 rows and 4 columns, QCSP-Solve was more than 8000 times slower than Queso, and BlockSolve was more than 80000 times slower. In the conversion to a binary QCSP, the structure of the problem is lost and the reasoning algorithms of QCSP-Solve and BlockSolve are not able to simplify it effectively. This provides some evidence that non-binary QCSP is a necessary step forwards, although it is likely that a better model of Connect 4 exists in binary QCSP.

Two models of faulty job shop scheduling were presented, where model A is for pedagogical purposes and model B is intended as a proof of concept for scheduling in QCSP. Both models were designed to make use of the pure value rule on universal variables, and of the reified disjunction constraint. Model B has two main modelling tricks. First, the pure value rule is exploited by using two fault variables, *unifault* and *fault*, with *unifault* being universally quantified and *fault* being existential. There is just one constraint over each universal variable, which makes it very easy to manage when values become pure. The second trick is to copy the CSP model of JSSP for each period. This allows a reasonably compact model (compared to model A), and permits the use of edge finding constraints, variable-value ordering heuristics and a whole range of other techniques from CSP scheduling. To the best of my knowledge, this is the first model of a complex and realistic problem in QCSP.

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. The cost of finding optimal solutions to model B does not scale badly against the number of scenarios. The experiments were performed on instances with 16 and 226 valid scenarios, a 14-fold increase in the number of scenarios, and the average search time increased by only 14.3 times.

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. On the solving side, it would be interesting to investigate making use of conflicts and solutions. In QBF, conflict learning [29] and solution backjumping [30] are very effective. In constraint programming, conflict learning is attracting interest. Also, it would be helpful to incorporate all the constraints of an existing constraint solver into a QCSP solver, to support more effective modelling.

Acknowledgements

Many thanks to Ian Gent who supervised my Ph.D. and made helpful comments on drafts of this paper. Thanks to anonymous reviewers who provided many helpful comments. I would like to thank EPSRC for providing the funding for this work through a doctoral training grant and grant number EP/E030394/1.

References

[1] Eclipse user manual release 5.10, 2006. http://eclipse.crosscoreop.com/doc/.

[2] Carlos Ansótegui. Personal communication.

[3] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[4] Fahiem Bacchus and Kostas Stergiou. Solution directed backjumping for QCSP. In *Proceedings 13th International Conference on the Principles and Practice of Constraint Programming (CP 2007)*, pages 148–163, 2007.

[5] 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.

[6] 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.

[7] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[8] 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.

[9] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Reasoning on quantified constraints. In *Rappresentazione Della Conoscenza e Ragionamento Automatico*, 2006.

[10] 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.

[11] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Modeling adversary scheduling with QCSP+. In *Proceedings 23rd Annual ACM Symposium on Applied Computing*, 2008.

[12] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Quantified constraint optimization. In *Proceedings 14th International Conference on Principles and Practice of Constraint Programming*, pages 463–477, 2008.

[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] Lucas Bordeaux. Boolean and interval propagation for quantified constraints. In *Proceedings 1st International Workshop on Quantification in Constraint Programming (at CP 2005)*, 2005.

[15] 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.

[16] 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.

[17] Lucas Bordeaux and Lintao Zhang. A solver for quantified boolean and linear constraints. In *Proceedings ACM Symposium on Applied Computing (SAC)*, pages 321–325, 2007.

[18] 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.

[19] 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.

[20] 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.

[21] 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 [33].

[22] 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.

[23] 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.

[24] 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.

[25] Ian Gent and Andrew Rowley. Encoding Connect-4 using quantified Boolean formulae. Technical Report APES-68-2003, APES research group, 2003.

[26] 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.

[27] Ian P. Gent, Peter Nightingale, Andrew Rowley, and Kostas Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 2007. To appear.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Peter Nightingale. *Consistency and the Quantified Constraint Satisfaction Problem*. PhD thesis, School of Computer Science, St Andrews University, 2007.

[35] 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.

[36] Francisco Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[37] 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.

[38] 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.

[39] Armagan Tarim, Suresh Manandhar, and Toby Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints*, 11(1):53–80, 2006.

[40] J.M. van den Akker, C.A.J. Hurkens, and M.W.P. Savelsbergh. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*, 12(2):111–124, 2000.

[41] 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.

[42] Toby Walsh. Stochastic constraint programming. In *Proceedings 15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 111–115, 2002.

[43] Eric W. Weisstein. Correlation coefficient. http://mathworld.wolfram.com/CorrelationCoefficient.html.