Solving Quantified Constraint Satisfaction Problems*

Ian P. Gent Peter Nightingale Andrew Rowley
School of Computer Science, University of St Andrews
St Andrews, Fife, KY16 9SS, UK.
{ipg,pn,agdr}@dcs.st-and.ac.uk

Kostas Stergiou
Department of Information and Communication Systems Engineering
University of the Aegean, Greece
konsterg@aegean.gr

November 19, 2008

Abstract

We make a number of contributions to the study of the Quantified Constraint Satisfaction Problem (QCSP). The QCSP is an extension of the constraint satisfaction problem that can be used to model combinatorial problems containing contingency or uncertainty. It allows for universally quantified variables that can model uncertain actions and events, such as the unknown weather for a future party, or an opponent's next move in a game. In this paper we report significant contributions to two very different methods for solving QCSPs. The first approach is to implement special purpose algorithms for QCSPs; and the second is to encode QCSPs as Quantified Boolean Formulas and then use specialized QBF solvers. The discovery of particularly effective encodings influenced the design of more effective algorithms: by analyzing the properties of these encodings, we identify the features in QBF solvers responsible for their efficiency. This enables us to devise analogues of these features in QCSPs, and implement them in special purpose algorithms, yielding an effective special purpose solver, QCSP-Solve. Experiments show that this solver and a highly optimized QBF encoding are several orders of magnitude more efficient than the initially developed algorithms. A final, but significant, contribution is the identification of flaws in simple methods of generating random QCSP instances, and a means of generating instances which are not known to be flawed.

Keywords: Quantified Constraint Satisfaction Problems, Quantified Boolean Formulas, Arc Consistency, Search Algorithms, Random Problems

*Parts of this paper have appeared in the conference papers [28, 29].

1 Introduction

Quantified Constraint Satisfaction Problems (QCSPs) can be used to model various PSPACE-complete combinatorial problems from domains like planning under uncertainty, design, adversary game playing, and model checking. For example, in game playing we may want to determine if a consistent strategy exists for all possible moves of the opponent. In a design problem it may be required that a configuration must be possible for all possible sequences of user choices. As a final example, when planning in a safety critical environment, such as a nuclear station, we may require that an action is possible for every eventuality. QCSPs extend traditional, unquantified, CSPs to deal with the kind of contingency found in the above examples.

The QCSP naturally generalizes the standard CSP formalism by allowing for universally quantified variables. Universal variables are used to model actions or events which are contingent, for which we are uncertain, or just those which are not in our control. Examples would be contingencies such as user choices in a configuration problem, uncertainties such as the weather in a plan to hold a garden party, or opponent moves in an adversary game. In a conventional CSP, all variables are existentially quantified, since all are within our control. The values in the domain of a universal variable capture all the possible outcomes of the event or action modelled by this variable. In this way, QCSPs model *bounded* uncertainty. In a QCSP we try to find a *strategy*, defining the values of the existential variables for all possible sequences of instantiations for the universal variables, so that all the constraints in the problem are satisfied. Such a strategy guarantees that there is a solution whatever values the universal variables take, i.e. whatever the outcome of the uncertain actions and events. The generalization of CSPs to QCSPs increases the expressiveness of the framework, but at the same time the complexity of the decision task rises from NP-complete to PSPACE-complete [8, 26, 38].

There is already considerable interest in quantified constraint reasoning in the case of *Quantified Boolean Formulae* (QBF), which is the generalization of SAT that allows universal quantification (for example, [13, 23, 30, 33, 34]). Also, there is a significant body of work on quantified problems with continuous real domains (e.g. [5, 41]). Ratschan gives numerous references to papers on this subject [40]. As far as QCSPs with discrete finite non-Boolean domains are concerned, there is recent research on theory defining the complexity of various reasoning tasks and also specifying tractable subclasses (e.g. [8, 11, 15–18]). Also, various useful concepts from CSPs, such as global and local consistency, substitutability and interchangeability, have been defined for QCSPs [11, 12]. However, little has been done as far as algorithms for solving QCSPs are concerned. In the few existing works, Bordeaux and Monfroy introduced a framework for implementing arc consistency and described filtering operators for certain classes of constraints [9, 12]. Also, very recently, Verger and Bessiere proposed a bottom-up solver for QCSPs called BlockSolve [44], while Benedetti, Lallouet and Vautard implemented QeCode, a QCSP solver built on top of the CSP solver Gecode [4].

In this paper we report the first comprehensive attempt to build effective QCSP solvers, although we limit ourselves to the case where constraints are binary. We make contributions to two very different approaches to solving QCSPs. These are special pur-

pose solvers for QCSPs; and encoding QCSPs as QBF instances so that existing QBF solvers can be used. In each approach we introduce novel and effective techniques. We also show how experience with the encodings directly influenced the design of better techniques for the specialized solvers.

We first approach QCSPs directly by extending well-known algorithms from the standard to the quantified case. This is analogous to the approach taken at the early stages of research in QBF. We show that some of the most widely used techniques for CSPs can be adapted to deal with quantification. We first describe a generic arc consistency algorithm that can be used to enforce AC in any binary QCSP. We then extend the chronological backtracking (BT), forward checking (FC), and maintaining arc consistency (MAC) algorithms so that they can handle quantification. We also propose modifications of FC and MAC that take advantage of the properties of QCSPs.

Then we follow an orthogonal approach, based on encoding QCSPs as QBFs. A particular advantage of encoding one search problem as another occurs when, as here, search techniques for the target problem are more highly developed than the original. In contrast to QCSP, numerous advanced solvers are available for QBF. We describe a finely-tuned encoding which can be several orders of magnitudes more efficient than the direct methods described so far. The tuning of encodings to be effective for search is considerably more involved than in the case of SAT, where encodings often have an elegant simplicity. A simple way of lifting CSP encodings to QCSP is very ineffective, so we explore and implement new ideas, without analogues in SAT, that make search very effective.

Apart from obtaining efficient tools for QCSP solving, we benefit from the study and development of encodings to learn valuable lessons that can be transferred to direct algorithms. So in the third, and final, stage in the development of algorithms for QCSPs we analyze the advantages offered by our QBF encoding to identify the features responsible for its efficiency. We identify three sophisticated techniques; conflict-based backjumping [39], solution-directed backjumping [33], and most importantly, the pure literal rule [14], as important reasons for the success of QBF solvers in solving encoded QCSP instances. We devise analogues of these features in QCSPs, and implement them on top of direct algorithms, to yield a specialized direct solver, called QCSP-Solve.

A final issue we address in this paper is that of benchmarking, since there is naturally a distinct lack of benchmarks to compare algorithms on. This is a familiar problem that has appeared in the early stages of experimental research in various other areas. As was the case with CSP, SAT, and QBF, we address this problem by proposing and using methods to generate random instances. We show that a simple generalization of random generation models from CSPs or QBF to QCSPs is prone to flaws, which quickly affect all generated instances. We then introduce a random generator that is free from these flaws, although it remains possible that it will suffer from a currently unknown flaw. Experiments run on problems created using this generator reveal a progressive, and dramatic, improvement in the efficiency of our methods; starting with the initial direct algorithms and culminating in QCSP-Solve and a highly optimized QBF encoding.

This paper is structured as follows. In Section 2 we give the necessary definitions and background. We then present progressively more efficient methods of handling QCSPs. In Section 3 we follow the direct approach by extending standard algorithms

from CSPs to QCSPs. In Section 4 we describe some of the existing work on encodings, and develop a finely-tuned encoding which is remarkably more efficient than the direct approaches. In Section 5 we show how lessons learned from the encoding of QCSP into QBF can be utilized to enhance the direct algorithms, resulting in QCSP-Solve; an advanced solver for QCSPs. Section 6 describes a flaw which can arise in random QCSPs, introduces a random generator for QCSPs, and gives indicative experimental results which demonstrate the building of progressively more efficient techniques. Finally, in Section 7 we conclude.

2 Preliminaries

A *Constraint Satisfaction Problem* (CSP) consists of a set of variables, each associated with a domain of possible values, and a set of constraints restricting the combinations of values that the variables can simultaneously take. In CSPs all variables are existentially quantified. QCSPs are more expressive in that they allow universally quantified variables. In this way they enable the formulation of problems where all contingencies must be allowed for. We now give a formal definition of a QCSP instance. As is usual practice in CSPs, we use the name QCSP to denote both particular instances and the decision problem of determining whether an instance is true (i.e. soluble) or not.

Definition 1 A *Quantified Constraint Satisfaction Problem* (QCSP) F is a tuple $\langle V, Q, D, C\rangle$ where:

- V is a linearly ordered set of n variables. In the following we will denote by v_i the i-th element of V with respect to this linear order.
- Q is a mapping from V to the set of quantifiers $\{\exists, \forall\}$. For each variable $v_i \in V$, $Q(v_i)$ is a quantifier ($\exists$ or $\forall$) associated with v_i.
- D is a mapping from V to a set of domains $D = \{D(v_1), \ldots, D(v_n)\}$. For each variable $v_i \in V$, $D(v_i)$ is the finite domain of its possible values.
- $C = \{c_1, \ldots, c_m\}$ is a set of m constraints. Each constraint $c_i \in C$ is defined as a pair $(vars(c_i), rel(c_i))$, where: 1) $vars(c_i) = (v_{j_1}, \ldots, v_{j_k})$ is an ordered subset of V called the constraint *scope*. The size of $vars(c_i)$ is called the *arity* of c_i. 2) $rel(c_i)$ is a subset of the *Cartesian* product $D(v_{j_1}) \times \ldots \times D(v_{j_k})$ and it specifies the allowed combinations of values for the variables in $vars(c_i)$.

The above definition of a QCSP reduces to that of a standard CSP if there are no universally quantified variables in the problem.

A *block* of variables in a QCSP F is a maximal subsequence of variables in V that have the same quantification. The assignment (also called instantiation) of value $a_j \in D(v_j)$ to variable $v_j \in V$ will be denoted by $v_j \mapsto a_j$. Accordingly, the tuple assigning values $a_1, \ldots, a_i$ to variables $v_1, \ldots, v_i$ will be denoted by $\langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i\rangle$. The set of variables over which a tuple τ is defined will be denoted by $vars(\tau)$. For any subset $vars'$ of $vars(\tau)$, $\tau[vars']$ denotes the sub-tuple of τ that includes only

assignments to the variables in $vars'$. A tuple τ is *consistent*, iff for all $c_i \in C$, s.t. $vars(c_i) \subseteq vars(\tau)$, $\tau[vars(c_i)] \in rel(c_i)$. For any constraint $c_i \in C$, variable $v_j \in V$ and value $a \in D(v_j)$, we denote by $c_i[v_j \mapsto a]$ the subset of $rel(c_i)$ that only includes tuples where v_j takes value a. If $v_j \notin vars(c_i)$ then $c_i[v_j \mapsto a] \equiv rel(c_i)$. We write $C[v_j \mapsto a]$ as a shorthand for $c_1[v_j \mapsto a] \wedge \ldots \wedge c_m[v_j \mapsto a]$.

In what follows we will often refer to universally and existentially quantified variables as *universals* and *existentials* respectively.

Definition 2 QCSP semantics.

A QCSP $F = \langle V, Q, D, C \rangle$ represents the logical formula $\phi = Q(v_1)v_1 \in D(v_1) \ldots Q(v_n)v_n \in D(v_n)\ (C)$. The semantics of a QCSP can be defined recursively as follows.

The base case is a QCSP instance with an empty quantifier prefix Q, i.e. all variables assigned. This QCSP instance is true iff, for each constraint $c_i \in C$, the tuple of values of assigned variables in its scope *vars*(c_i) belongs to its relation *rel*(c_i). Note that an empty QCSP is vacuously true. If ϕ is of the form $\exists v_1 \in D(v_1)\ Q(v_2)v_2 \in D(v_2) \ldots Q(v_n)v_n \in D(v_n)\ (C)$ then F is true iff there exists some value $a \in D(v_1)$ such that $Q(v_2)v_2 \in D(v_2) \ldots Q(v_n)v_n \in D(v_n)(C[v_1 \mapsto a])$ is true. Or in words, if under the assignment $v_1 \mapsto a$ the rest of the problem is true. If ϕ is of the form $\forall v_1 \in D(v_1)\ Q(v_2)v_2 \in D(v_2) \ldots Q(v_n)v_n \in D(v_n)\ (C)$ then F is true iff for each value $a \in D(v_1)$, $Q(v_2)v_2 \in D(v_2) \ldots Q(v_n)v_n \in D(v_n)(C[v_1 \mapsto a])$ is true.

To better understand the semantics of a QCSP, we first need to define the notion of a strategy. A *strategy* is a tree with each level of the tree corresponding to a variable. Level 1 corresponds to the first variable v_1 in V, and levels thereafter follow the order of V. A node in the i-th level of the tree corresponds to a tuple of variable assignments $\langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i \rangle$, where $a_1 \in D(v_1), \ldots, a_i \in D(v_i)$. The root of the tree corresponds to the empty tuple, the first level nodes correspond to a 1-tuple assigning a value to the first variable in V, the second level nodes correspond to 2-tuples assigning the first two variables in V, generated by extending the first level assignment, etc. A node in the tree corresponding to tuple $\langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i \rangle$ has as many children as the values in $D(v_{i+1})$, if v_{i+1} is universally quantified, whereas it has a single child if v_{i+1} is existentially quantified. A node corresponding to tuple $\tau = \langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i \rangle$ is *true* iff τ is consistent. Otherwise, the node is *false*. A tuple of assignments to all variables in a QCSP (i.e. a n-tuple) constitutes a *scenario*. Within a scenario, the value of each existential variable depends on the values of the universal variables that precede it in V. A scenario is *consistent* iff all the variable assignments in the scenario satisfy all constraints in the problem.

We can now give an alternative definition of the semantics of a QCSP: A QCSP with n variables is *true* (or *satisfiable*) iff there exists a strategy where all the leaf nodes (i.e. the nodes of level n) are true[1]. Or in other words, iff there exists a strategy such that all the scenarios of the strategy are consistent. Such a strategy is called a *consistent strategy*, or simply a *solution*, to the QCSP. As we will show, backtracking-based algorithms can solve a QCSP by traversing the space of strategies until they discover a consistent one or prove that none exists.

[1]This definition can easily be extended to the case where nodes may be pruned by propagation.

Example 1 Consider the problem $\forall v_1 \exists v_2 \forall v_3 \exists v_4 \ (v_1 \neq v_2 \wedge v_1 \neq v_4 \wedge v_3 \neq v_4)$. This is a QCSP where V consists of four variables, and C is a conjunction of three constraints. The problem reads "for all values of v_1 there exist values of v_2 such that for all values of v_3 there exist values of v_4, such that all constraints are satisfied". Assuming that all variables have domain $\{0, 1, 2\}$ then the problem is true. A solution to this problem is depicted in Figure 1. Each path to a leaf node is a consistent scenario.

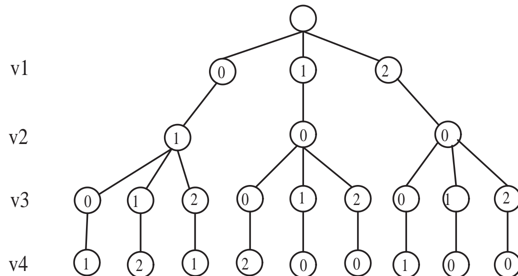

Figure 1: A solution to the problem of Example 1.

Note that, in contrast to standard CSPs, the variables in a QCSP are ordered. This means that changing their order gives rise to a different problem. For example, the problem $\forall v_i \exists v_j \ (v_i = v_j)$, with $D(v_i) = D(v_j) = \{0, 1\}$, is true as for any value of v_i we can find a value for v_j to satisfy the constraint. However, the problem $\exists v_j \forall v_i$ $(v_j = v_i)$, with $D(v_j) = D(v_i) = \{0, 1\}$, is false as no value of v_j satisfies the constraint for both possible assignments to v_i.

In this paper we restrict our attention to binary QCSPs. As is common, we assume that at most one constraint is defined on any pair of variables. In a binary QCSP, each constraint, denoted by c_{ij}, involves two variables (v_i and v_j) which may be universally or existentially quantified. We assume that for any constraint c_{ij}, variable v_i precedes v_j in V, unless explicitly specified otherwise. Some of the techniques described below can be easily extended to constraints of any arity, but for others this extension is much more involved. We briefly elaborate on this in Section 3.1.

3 Extending CSP Algorithms to handle QCSPs

In this section we begin the presentation of techniques for solving QCSPs. We first approach the problem directly by extending standard algorithms from CSPs to deal with quantification. Namely, we describe an arc consistency algorithm for binary QCSPs, and adapt widely used backtracking search algorithms to handle QCSPs. We also show how the notion of value interchangeability can be exploited in QCSPs to break some symmetries.

3.1 Arc Consistency

An important concept in CSPs is the concept of local consistency. Local consistencies are properties that can be achieved in a CSP, using (typically) algorithms with polynomial time complexity, to remove some inconsistent values either prior to or during search. Arc consistency is the most commonly used local consistency property in the existing constraint programming engines.

A constraint c_{ij} in a CSP is *arc consistent* (AC) iff for each value $a \in D(v_i)$ there exists a value $b \in D(v_j)$ so that the assignments $v_i \mapsto a$ and $v_j \mapsto b$ are *compatible* (i.e. satisfy c_{ij}). In this case we say that b is a *support* for a on constraint c_{ij}. Accordingly, a is a support for b on the same constraint. A binary CSP is arc consistent iff all its constraints are arc consistent. The operation performed to determine whether a value $a \in D(v_i)$ is supported by a value $b \in D(v_j)$ with respect to constraint c_{ij} is called a *constraint check*.

Bordeaux and Monfroy extended the definition of AC to QCSPs and described the schema of a generic AC3-based algorithm for QCSPs [12]. This algorithm can be instantiated to achieve AC on specific constraints (not necessarily binary) once filtering operators have been defined for these constraints. A filtering operator for a constraint c is a function that specifies which values in the domains of the variables involved in c are arc inconsistent with respect to c, taking in consideration the quantification of the variables [12]. Consider the following example:

Example 2 Bordeaux and Monfroy defined filtering operators for constraint $\neg v_i = v_j$, where v_i and v_j have Boolean domains [12]. These operators specify the values in $D(v_i)$ and $D(v_j)$ that are arc inconsistent according to the quantification of the two variables. The application of an AC algorithm will prune these values from the domains. For instance, if v_i is existentially quantified and v_j is universally quantified (i.e. the formula is $\exists v_i \forall v_j (\neg v_i = v_j)$) then the following applies: If $D(v_j) = \{0\}$ then value 0 is pruned from $D(v_i)$. If $D(v_j) = \{1\}$ then value 1 is pruned from $D(v_i)$.

The definition of AC and the AC algorithm of [12] are based on decomposing complex constraints (e.g. constraints of high arity), that may be present in a QCSP, into "primitive" constraints for which AC filtering operators have been defined. As noted in [12], this definition is somewhat different from the standard definition of AC in CSPs, and is actually closer to the definition of *relational consistency* [21]. In [12], and later in [9], filtering operators for constraints on Boolean variables and also for linear numerical constraints were defined. The aim of our work on AC for QCSPs is to define filtering operators for arbitrary binary constraints, as opposed to specific constraints with known semantics, and embed these into an efficient algorithm.

We first give an alternative definition of AC for binary QCSPs that closely follows the standard CSP definition. Based on this definition, we then define filtering operators for arbitrary binary constraints in a straightforward way. Finally, we describe QAC-2001, a generic AC algorithm that utilizes these filtering operators to achieve AC in arbitrary binary QCSPs. In contrast to the algorithm of [12] which is based on AC3, QAC-2001 is based on AC2001/3.1, the AC algorithm of [6] for binary CSPs[2].

[2]Note that other AC algorithms, like AC-6, can be used as basis. We chose to use AC2001/3.1 because of its simplicity and optimal time complexity.

Definition 3 A binary QCSP $F = \langle V, Q, D, C \rangle$ is *arc consistent* iff all the constraints $c_{ij} \in C$ are arc consistent. Consider a constraint $c_{ij} \in C$, where v_i is before v_j in V. There are four possible combinations of quantification for v_i and v_j. For each case, constraint c_{ij} is arc consistent iff:

$\exists\, v_i\ \exists\, v_j$: Each value $a \in D(v_i)$ is supported by at least one value in $D(v_j)$, and each value $b \in D(v_j)$ is supported by at least one value in $D(v_i)$.

$\forall\, v_i\ \forall\, v_j$: Each value $a \in D(v_i)$ is supported by all values in $D(v_j)$, and each value $b \in D(v_j)$ is supported by all values in $D(v_i)$.

$\forall\, v_i\ \exists\, v_j$: Each value $a \in D(v_i)$ is supported by at least one value in $D(v_j)$, and each value $b \in D(v_j)$ is supported by at least one value in $D(v_i)$.

$\exists\, v_i\ \forall\, v_j$: Each value $a \in D(v_i)$ is supported by all values in $D(v_j)$, and each value $b \in D(v_j)$ is supported by at least one value in $D(v_i)$.

Matching the four cases of the definition above, we can define filtering operators for an arbitrary binary constraint c_{ij} as follows. These filtering operators specify the values that are arc inconsistent, and thus must be removed from the domains of the variables involved in the constraint.

$\exists\, v_i\ \exists\, v_j\ (c_{ij})$: If a value $a \in D(v_i)$ has no support in $D(v_j)$ then a is removed from $D(v_i)$. Similarly, if a value $b \in D(v_j)$ has no support in $D(v_i)$ then b is removed from $D(v_j)$. If any of the two domains becomes empty then the problem is false.

$\forall\, v_i\ \forall\, v_j\ (c_{ij})$: If any value $a \in D(v_i)$ is not supported by all values in $D(v_j)$ (in which case at least one value $b \in D(v_j)$ will also not be supported by all values in $D(v_i)$) then the problem is false. Note that it suffices to check the constraint only in one direction. If all values of v_i are supported by all values of v_j then, obviously, the opposite also holds.

$\forall\, v_i\ \exists\, v_j\ (c_{ij})$: If a value $a \in D(v_i)$ has no support in $D(v_j)$ then the problem is false. If a value $b \in D(v_j)$ has no support in $D(v_i)$ then b is removed from $D(v_j)$. If $D(v_j)$ becomes empty then the problem is false.

$\exists\, v_i\ \forall\, v_j\ (c_{ij})$: If a value $a \in D(v_i)$ is not supported by all values in $D(v_j)$ then a is removed from $D(v_i)$. If $D(v_i)$ becomes empty then the problem is false. If a value $b \in D(v_j)$ has no support in $D(v_i)$ then the problem is false. Note that it suffices to check the constraint only in one direction. If there is at least one value in $D(v_i)$ that is supported by all values in $D(v_j)$ then, obviously, all values in $D(v_j)$ have at least one support in $D(v_i)$.

In Figure 2 we sketch algorithm QAC-2001. The algorithm takes as input a QCSP $F = \langle V, Q, D, C \rangle$ and removes unsupported values from the domains of the variables using the filtering operators described above. If the domain of an existential becomes empty or a value is removed from the domain of a universal then the algorithm returns FALSE.

```
Boolean QAC-2001(F = ⟨V, Q, D, C⟩)
input: A QCSP F
output: TRUE if AC is successfully applied on F and FALSE if there a domain wipeout
        of an existential or a value is removed from the domain of a universal
put all constraints of C in S
S' ← ∅
for each c_ij ∈ C where Q(v_i) = ∀ and Q(v_j) = ∀
    for each a ∈ D(v_i)
      if a is not supported by all values in D(v_j) return FALSE
    add c_ij to S'
for each c_ij ∈ C where Q(v_i) = ∃ and Q(v_j) = ∀
    for each a ∈ D(v_i)
      if a is not supported by all values in D(v_j)
        remove a from D(v_i)
    if D(v_i) = ∅ return FALSE
    else add c_ij to S'
S ← S \ S'
return Propagation(F, S)

function Propagation(F, S : stack)
input: A QCSP F and a stack of variables S
output: TRUE if AC is successfully applied on F and FALSE if there a domain wipeout
        of an existential or a value is removed from the domain of a universal
for each constrained pair of variables v_i, v_j ∈ V
    for each a ∈ D(v_i)
      currentSupport_{v_i,a,v_j} ← NIL
while S ≠ ∅
    pop a constraint c_ij from S
    if Revise(v_i, v_j, currentSupport)
      if Q(v_i) = ∀ or D(v_i) = ∅ return FALSE
      put each constraint c_ki in S
return TRUE

function Revise(v_i, v_j, currentSupport)
input: A pair of variables v_i,v_j and the data structure currentSupport
output: TRUE if a value is removed from a domain and FALSE otherwise
DELETION ← FALSE
for each a ∈ D(v_i)
    if currentSupport_{v_i,a,v_j} is NIL or is no longer in D(v_j)
      if exists b(∈ D(v_j)) > currentSupport_{v_i,a,v_j} and b supports a
        currentSupport_{v_i,a,v_j} ← b
      else remove a from D(v_i)
        if Q(v_i) = ∀ return TRUE
        DELETION ← TRUE
return DELETION
```

Figure 2: QAC-2001: An arc consistency algorithm for binary QCSPs.

The algorithm performs constraint-based propagation. That is, it uses a stack of constraints that are propagated instead of a stack of variables. Apart from this, it is similar to AC2001/3.1 augmented with the handling of universal variables. As in AC2001/3.1, we use a structure, called $currentSupport$ (corresponding to structure $Last$ in [6]), to keep track of the most recently discovered supports for the values of the variables. To be precise, if $c_{ij} \in C$ then $currentSupport_{v_i,a,v_j}$ is the value in $D(v_j)$ that currently supports value $a \in D(v_i)$. For each $v_i \in V$, $a \in D(v_i)$, and $v_j \in V$ (s.t. v_i is constrained with v_j) $currentSupport_{v_i,a,v_j}$ is initialized to NIL. As in [6], value NIL is defined as a dummy value which precedes any value in any domain. We assume that there exists an order of the values in the domains.

Initially, all constraints are added to the stack S. Constraints of the form $\exists\, v_i \forall\, v_j$ (c_{ij}) and $\forall\, v_i \forall\, v_j$ (c_{ij}) are dealt with by a preprocessing step in function *QAC-2001*. For the former, each value of $D(v_i)$ that is not supported by all values of $D(v_j)$ is removed from $D(v_i)$. For the latter, if there is a value of $D(v_i)$ that is not supported by all values of $D(v_j)$ then we can determine that the problem is false. Such constraints are thereafter removed from S and are not considered during the propagation phase.

In the propagation phase, function ***Revise*** is called for each constraint c_{ij} in the stack. This function looks for a support in $D(v_j)$ for each value a of $D(v_i)$. This is done by first looking at value $currentSupport_{v_i,a,v_j}$. If this is NIL or it has been removed from $D(v_j)$ because of propagation then the values of v_j are examined starting with the one immediately after $currentSupport_{v_i,a,v_j}$. If no support is found for a, it is removed from $D(v_i)$. In this case, if v_i is universally quantified or $D(v_i)$ becomes empty then we can determine that the problem is false. Otherwise, each constraint c_{ki} involving v_i and some other variable v_k is added to the stack so that it can be revised. The algorithm terminates successfully if the stack becomes empty.

We now show that, despite the presence of universal quantifiers, the worst-case time complexity of *QAC-2001* is the same as that of AC-2001/3.1. We assume that m is the number of binary constraints in a problem, and d is the maximum domain size.

Proposition 1 The worst-case time complexity of algorithm *QAC-2001* is $O(md^2)$.

Proof: The worst-case time complexity of the algorithm can be determined by examining the constraint checks executed in the two **for** loops in function *QAC-2001* and also in function *Propagation*.

In the first **for** loop we iterate through all constraints c_{ij} that involve two universal variables v_i and v_j. For each value $a \in D(v_i)$, we check if a is supported by all values in $D(v_j)$ or not. Therefore, each iteration costs $O(d^2)$ constraint checks. Hence, the first **for** loop costs $O(md^2)$ checks. In the second **for** loop we iterate through all constraints c_{ij} that involve an existential variable v_i and a universal v_j. As with the first loop, it is easy to see that the cost of the second loop is again $O(md^2)$.

Function *Revise* is called at most d times for each constraint $c_{ij} \in C$; once for every deletion of a value from $D(v_j)$. In each call to *Revise* the algorithm performs at most d checks (one for each value $a \in D(x_i)$) to see if $currentSupport_{v_i,a,v_j}$ is still in the domain of $D(v_j)$. If it is not (or it is NIL), the algorithms tries to find a new support for a in $D(v_j)$ starting from the value immediately after $currentSupport_{v_i,a,v_j}$. Since we use structure $currentSupport$, each time $Revise$ is called for c_{ij}, and for each value

$a \in D(x_i)$, we only check values that have not been checked before. In other words, we can check each of the d values in $D(v_j)$ at most once for each value of v_i. So overall, in the worst case, we have d checks plus the d checks to test the validity of the current support. For the d values of v_i the upper bound in checks performed to make one variable AC is therefore $O(d^2)$. For m constraints the worst-case complexity bound of *Propagation* is O(md^2). Hence, the worst-case time complexity of *QAC-2001* is O($md^2 + md^2 + md^2$)=O(md^2). *QED*

The generalization of AC to non-binary constraints is usually referred to as *Generalized Arc Consistency* (GAC). Processing a non-binary constraint to achieve GAC according to the definition by Bordeaux et. al. [9, 11, 12] is much more challenging than the binary case for several reasons. Firstly, since this level of consistency is a generalization of GAC for CSP, enforcing it takes exponential time in general, whereas binary quantified arc consistency can be enforced in O(md^2). Secondly, an algorithm that achieves GAC on non-binary QCSPs would be considerably more complex than a similar algorithm for GAC in CSP, because (for a constraint with arity k) it should be able to handle up to 2^k different quantifier sequences. Thirdly, a support for a value in a non-binary constraint c is no longer simply a tuple that includes that value and is allowed by c, as in the CSP case. Here, we need a more complex definition of a support that takes into account the quantified variables in the constraint's scope. Hence we restrict our attention to binary QCSPs in this paper and leave consistency algorithms for non-binary QCSPs as future work.

Finally, compared to the work of [12] on AC, we can note the following differences: We only deal with binary constraints whereas the definition of [12] is generic (i.e. it covers GAC). We have defined filtering operators for arbitrary binary constraints, whereas [12] defined filtering operators for specific binary (and ternary) Boolean and numerical constraints. The AC algorithm of [12] is based on AC3 while ours is based on AC2001/3.1.

3.2 Search Algorithms

Numerous search algorithms have been developed for CSPs. Most of them are based on backtracking search. In this section we adapt chronological backtracking (BT), forward checking (FC) [36] and maintaining arc consistency (MAC) [42] to deal with binary QCSPs. Also, we show that by slightly modifying FC and MAC we get algorithms that can discover inconsistencies earlier, and therefore can be more efficient.

For any algorithm, we assume that before commencing search the input QCSP has been made AC using algorithm QAC-2001. Under this assumption, we do not have to consider constraints of the form $\exists\ v_i \forall\ v_j\ (c_{ij})$ or $\forall\ v_i \forall\ v_j\ (c_{ij})$ in the algorithms. All values of variable v_i, in such constraints, are definitely consistent with all values of variable v_j. If some value was not consistent then it would have been removed by the application of QAC-2001. This implies that, during search, we can safely ignore the last variables in V if they are universally quantified. For instance, if a problem $\exists v_i \forall v_j \exists v_k \forall v_l\ (c_{ij} \wedge c_{jk} \wedge c_{kl})$ is AC, then we can remove constraint c_{kl} and ignore variable v_l. Hence it suffices to apply search on the simplified problem $\exists v_i \forall v_j \exists v_k\ (c_{ij} \wedge c_{jk})$.

3.2.1 Chronological Backtracking

BT is a straightforward extension of the corresponding algorithm for standard CSPs. It takes as input a QCSP F and traverses the space of strategies until the truth of the problem is proved or disproved. To simplify the description of the algorithm (and the ones that follow), we assume that variables are assigned values following their order in V. However, consecutive variables with the same quantification can be instantiated in any order. The variable that is currently instantiated is called the *current variable* and is denoted by v_{cur}. The variables in V after v_{cur} are called *future* variables, while the ones before v_{cur} are called *past* variables. A situation where all values of the current variable are deemed inconsistent is called a *dead-end*. We make use of the following functions:

next: For any variable v_i, function *next*(v_i) returns the variable immediately after v_i in V.

last_u: For any variable v_i, function *last_u*(v_i) returns the variable $v_j \in V$ such that v_j is universally quantified, it precedes v_i in V, and there is no universal variable after v_j and before v_i in V. If v_{fu} is the first universal in V then *last_u*(v_{fu}) is NIL.

last_e: For any variable v_i, function *last_e*(v_i) returns the variable $v_j \in V$ such that v_j is existentially quantified, it precedes v_i in V, and there is no existential variable after v_j and before v_i in V. If v_{fe} is the first existential in V then *last_e*(v_{fe}) is NIL.

BT terminates successfully if all the values of the first universal variable are found to be part of a consistent scenario (line 15). In case there are no universals in the problem, the algorithm terminates successfully once the last existential has been consistently instantiated (line 10), as the problem is a standard CSP.

If the current variable is existential and a dead-end occurs then the algorithm backtracks to the previously instantiated existential variable, possibly jumping over some universal variables (lines 4–5). Detecting a dead-end means that the algorithm determines that the currently explored strategy cannot be extended to a solution. Therefore, it backtracks to the previous existential to assign it a new value and explore an alternative strategy. If there is no dead-end, the next available value of the current variable is checked against the previous assignments (line 8). If the value is compatible with all assignments to past variables and BT has reached a true leaf node then it backtracks to the previous universal variable (line 11). If BT is not at a leaf node, it proceeds by moving to the next variable (line 12). In case some constraint check fails, BT tries the next value of the current variable in the next iteration of the **while** loop.

If the current variable is universal then there are two cases. If all of its values have been proved to be part of a consistent scenario, BT backtracks to the previous universal variable (lines 14,16) to assign it its next value. If not all of the current variable's values have been tried, BT assigns it with its next value and proceeds with the next variable (lines 17,18). Note that when BT assigns a value to a universal variable it does not check this value against the previously made assignments. The reason is that, due to

```
Boolean BT(F = ⟨V, Q, D, C⟩)
input: A QCSP F
output: TRUE if a solution to F exists and FALSE otherwise
1: v_cur ← v_1
2: while v_cur ≠ NIL
3:   if Q(v_cur) = ∃
4:     if all values in D(v_cur) have been tried
5:       v_cur ← last_e(v_cur)
6:     else
7:       assign v_cur with the next value a ∈ D(v_cur)
8:       if v_cur ↦ a is compatible with the assignments to all past variables
9:         if v_cur = v_n
10:          if there are no universals in V return TRUE
11:          else v_cur ← last_u(v_cur)
12:        else v_cur ← next(v_cur)
13:  else //Q(v_cur) = ∀//
14:    if all values in D(v_cur) have been tried
15:      if v_cur is the first universal in V return TRUE
16:      else v_cur ← last_u(v_cur)
17:    else assign v_cur with the next value a ∈ D(v_cur)
18:        v_cur ← next(v_cur)
19: if v_cur = NIL return FALSE
```

Figure 3: Chronological Backtracking for binary QCSPs.

AC preprocessing, all values of a universal variable v_i are definitely consistent with all values of the variables before v_i in V.

Correctness of BT We now demonstrate, informally, the correctness of BT. To show soundness, we need to demonstrate that whenever BT returns true after traversing a strategy, this strategy is indeed consistent. Or, in other words, that all the scenarios in the strategy are consistent. Take any scenario in the strategy and consider any tuple of assignments $\tau = \langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i \rangle$ along this scenario. BT extends this tuple to variable v_{i+1} by assigning it a value a_{i+1} only if a_{i+1} is consistent with all assignments in τ. Therefore, when a tuple $\langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_{n-1} \rangle$ is extended to an $n-$tuple, all assignments in the tuple will be consistent with each other. This means that the tuple is a consistent scenario.

To show completeness, we need to demonstrate that if a consistent strategy exists, BT will correctly verify it by returning true once it has traversed it. It suffices to show that BT traverses the entire search space apart from some sub-spaces that are not part of any consistent strategy. BT systematically explores the search space trying to verify that for any sequence of assignments to the universals we can find a consistent scenario that includes these assignments. Search sub-spaces are skipped 1) when a value of the current variable fails a constraint check with an assignment of a past variable, and 2) when there is a backtrack to an existential. In the first case, let $\tau = \langle v_1 \mapsto a_1, \ldots, v_{i-1} \mapsto a_{i-1} \rangle$ be the current tuple of assignments and assume that

value a_i of the current variable v_i fails a constraint check with an assignment in τ. Tuple τ cannot be extended to a consistent scenario and hence a_i (and the sub-tree below the corresponding node) is correctly pruned. A backtrack to an existential v_i assigned value a_i means that the currently explored strategy cannot be extended to a consistent strategy and therefore a_i (and the sub-tree below the corresponding node) is correctly pruned.

3.2.2 Forward Checking and MAC

Many ways to improve the performance of BT have been proposed in the CSP literature. Most of them are classified as either *look-ahead* or *look-back* methods. The former try to detect inconsistencies early by performing some amount of local reasoning after each variable instantiation. The latter try to deal with a dead-end in an intelligent way by identifying the variables that are responsible for the dead-and and directly backtracking to one of these variables. We now show how the most commonly used look-ahead algorithms, FC and MAC, can be adapted to QCSPs. Look-back methods for QCSPs are discussed in Section 5.

The algorithm FC0, shown in Figure 4, is an extension of standard FC to QCSPs. It operates in a way similar to BT with the difference that, as in standard CSPs, constraint checks are made against future instead of past variables. To be precise, once a variable assignment to an existential or universal is made, it is checked against values of future existentials using function *Forward_Check0* (lines 9 and 25). In this function any value that is not compatible with the current assignment is temporarily removed from the domain of the corresponding variable. As mentioned, constraints of the form $\exists\ v_i \forall\ v_j\ (c_{ij})$ or $\forall\ v_i \forall\ v_j\ (c_{ij})$ have already been handled by preprocessing. Therefore, no checks against universals are performed. If all the values are removed from the domain of a variable (domain wipe-out) then the current assignment is rejected. In this case, if v_{cur} is an existential, procedure *Restore* is called to undo any changes made in the domains of the variables (line 16). Then the algorithm will try the next value of v_{cur} in the next iteration of the **while** loop. If v_{cur} is a universal, the algorithm will backtrack to the previous existential *last_e*(v_{cur}) in V (line 29). Before backtracking, all values that were temporarily removed because of the assignments to variables between *last_e*(v_{cur}) and v_{cur} are restored in their domains using procedure *Restore* (line 28).

Note that *Restore* must be called whenever a backtrack occurs. That is, apart from the case described above, restoration of values to domains is required when a dead-end is encountered (line 5), when a true leaf node is reached (line 13), and when all the assignments to a universal have been proved to be part of a consistent scenario (line 21).

By slightly modifying the forward checking function of FC0 we get an algorithm, which we call FC1, that can discover inconsistencies earlier than FC0. Algorithm FC1 has exactly the same behavior as FC0 when the current variable is existentially quantified. If the current variable v_{cur} is universally quantified then we first check every value of v_{cur} against all future variables before assigning a specific value to it. This is done using Function *Forward_Check1*, depicted in Figure 5. If one of v_{cur}'s values causes a domain wipe-out then we backtrack to the last existential variable. Other-

Boolean **FC0**$(F = \langle V, Q, D, C\rangle)$
input: A QCSP F
output: TRUE if a solution to F exists and FALSE otherwise
1: $v_{cur} \leftarrow v_1$
2: **while** $v_{cur} \neq NIL$
3: **if** $Q(v_{cur}) = \exists$
4: **if** all values in $D(v_{cur})$ have been tried
5: *Restore*$(F, v_{cur}$,*last_e*$(v_{cur}))$
6: $v_{cur} \leftarrow$ *last_e*(v_{cur})
7: **else**
8: assign v_{cur} with the next value $a \in D(v_{cur})$
9: **if** *Forward_Check0*(F, v_{cur}, a)
10: **if** $v_{cur} = v_n$
11: **if** there are no universals in V **return** TRUE
12: **else**
13: *Restore*$(F, v_{cur}$,*last_u*$(v_{cur}))$
14: $v_{cur} \leftarrow$ *last_u*(v_{cur})
15: **else** $v_{cur} \leftarrow$ *next*(v_{cur})
16: **else** *Restore*(F, v_{cur}, v_{cur})
17: **else** //$Q(v_{cur}) = \forall$//
18: **if** all values in $D(v_{cur})$ have been tried
19: **if** v_{cur} is the first universal in V **return** TRUE
20: **else**
21: *Restore*$(F, v_{cur}$,*last_u*$(v_{cur}))$
22: $v_{cur} \leftarrow$ *last_u*(v_{cur})
23: **else**
24: assign v_{cur} with the next value $a \in D(v_{cur})$
25: **if** *Forward_Check0*(F, v_{cur}, a)
26: $v_{cur} \leftarrow$ *next*(v_{cur})
27: **else**
28: *Restore*$(F, v_{cur}$,*last_e*$(v_{cur}))$
29: $v_{cur} \leftarrow$ *last_e*(v_{cur})
30: **if** $v_{cur} = NIL$ **return** FALSE

function *Forward_Check0*(F, v_{cur}, a)
input: A QCSP F, the current variable v_{cur} and its assigned value a
output: TRUE if no domain is wiped out and FALSE otherwise
1: **for** each existential v_i after v_{cur} in V
2: **for** each $b \in D(v_i)$
3: **if** $v_i \mapsto b$ is incompatible with $v_{cur} \mapsto a$
4: temporarily remove b from $D(v_i)$
5: **if** $D(v_i)$ is wiped out **return** FALSE
6: **return** TRUE

procedure *Restore*(F, v_{cur}, v_{back})
input: A QCSP F, the current variable v_{cur} and the variable where the algorithm will backtrack v_{back}
output: -
1: **for** $v_i = v_{back}$ to v_{cur}
2: **for** each existential v_j after v_i in V
3: restore to $D(v_j)$ any value that was removed because of v_i's instantiation

Figure 4: FC0: Forward Checking for binary QCSPs.

wise, we proceed in the usual way by instantiating v_{cur} with its next available value a and removing all values of future variables that are incompatible with the assignment $v_{cur} \mapsto a$. In this way we can discover dead-ends earlier and avoid fruitless exploration of search tree branches.

Note that the look-ahead of FC1 need only be applied once when the algorithm reaches a universal at some branch of the search tree. That is, immediately before trying the first available assignment of the universal. Assuming that none of the universal's possible assignments causes a domain wipe-out, then applying the FC1 type of look-ahead again after a backtrack to the universal occurs is redundant. This is obvious since the restoration of values guarantees that the result will be the same as before, i.e. none of the remaining possible assignments for the universal will cause a domain wipe-out.

It is easy to see that FC1 will always visit at most the same number of search tree nodes as FC0 as it may discover an inconsistency earlier than FC0 but never later. The two algorithms are incomparable in the number of constraint checks they perform. That is, depending on the problem, FC0 may perform less checks than FC1 and vice versa.

function *Forward_Check1*(F, v_{cur})
input: A QCSP F and the current variable v_{cur}
output: FALSE if some value of v_{cur} is incompatible with all values of an existential, and TRUE otherwise
1: **for** each $a \in D(v_{cur})$
2: **for** each existential v_i after v_{cur} in V
3: **if** $v_{cur} \mapsto a$ is incompatible with all values in $D(v_i)$ **return** FALSE
4: **return** TRUE

Figure 5: Forward checking function of algorithm FC1.

Correctness of FC The correctness of FC can be informally demonstrated following similar arguments as in the case of BT. In addition, we need to show that the forward checking functions of FC0 and FC1 are correct. That is, they prune parts of the search space that do not belong to a consistent strategy. Function *Forward_Check0* is called after assigning an existential or universal variable v_i with a value a_i. Assume that the current tuple of assignments is $\tau = \langle v_1 \mapsto a_1, \ldots, v_i \mapsto a_i \rangle$. *Forward_Check0* will prune any value from the domain of a future existential that fails a constraint check with assignment $v_i \mapsto a_i$. This means that any such value is not consistent with τ and therefore τ cannot be extended to a consistent scenario that includes this value. Hence, it is correctly pruned. *Forward_Check1* is called before assigning a universal variable v_i and for each value $a_i \in D(v_i)$ it temporarily prunes any value from the domain of a future existential that fails a constraint check with a_i. If the domain of a future existential v_j is wiped out then the algorithm backtracks. Assume that the assignment $v_i \mapsto a_i$ causes the wipeout of $D(v_j)$. This means that no value of v_j can participate in a consistent scenario of the currently explored strategy that includes the assignment $v_i \mapsto a_i$. Therefore, no consistent scenario that includes $v_i \mapsto a_i$ exists in the current strategy and, hence, the algorithm correctly backtracks to try an alternative strategy.

Maintaining Arc Consistency Based on the above description of FC, we can easily adapt the MAC algorithm to QCSPs. MAC is the most widely used complete search algorithm for CSPs. It reduces the domains of future variables during search by applying an AC algorithm on the problem after each variable instantiation. In this way inconsistencies are discovered early and search effort is saved.

To implement MAC for QCSPs we need a simple modification in the pseudo-code of FC0. We need to replace the calls to *Forward_Check0* in lines 9 and 25 of Figure 4 with calls to function *Propagation* of QAC-2001. In this case the stack of constraints would have to be initialized by adding to it only constraints that involve the current variable. MAC can also be modified in the same way as FC to yield MAC1, an algorithm analogous to FC1. That is, when the current variable v_{cur} is universally quantified we can (temporarily) enforce AC for each instantiation $v_{cur} \mapsto a$, where $a \in D(v_{cur})$, before committing to a particular instantiation. If one of the instantiations causes a domain wipe-out then we backtrack. Otherwise, we commit to one of the values and proceed with the next variable.

3.3 Symmetry Breaking

Many CSPs contain symmetries which means that for a given solution there are equivalent solutions. This can have a profound effect on the search cost when looking for one or (even more) all solutions to a CSP. Various methods for symmetry breaking have been proposed. Most of these methods add symmetry breaking constraints to the problem either statically, before search, or dynamically during search. A survey of work on symmetry in standard CSPs has recently been published, giving extensive references to the large body of work in that area [31].

QCSPs, in particular, can greatly benefit from symmetry breaking techniques, since we have to check if there exists a consistent scenario for all values of all universally quantified variables. We propose the exploitation of value interchangeability as a static and dynamic symmetry breaking technique in QCSPs. However, we consider only a simple type of symmetry in this paper, leaving until future work extension of more powerful techniques devised for constraint satisfaction. Some advanced concepts, like value substitutability, that can be used for symmetry breaking in QCSPs have been defined (though not implemented) in [11].

The notion of interchangeable values in CSPs was defined by Freuder in [24]. A value a of a variable v_i is *fully interchangeable* with a value b of v_i, iff every solution which contains the assignment $v_i \mapsto a$ remains a solution if we substitute b for a, and vice versa. Since determining full interchangeability is **coNP**-complete [10], Freuder also defined various local interchangeabilities that are polynomially computable.

Definition 4 Given a variable $v_i \in V$, a value $a \in D(v_i)$ is *neighborhood interchangeable* (NI) with a value $b \in D(v_i)$, iff for each $v_j \in V$, such that v_j is constrained with v_i, a and b are supported by exactly the same values in $D(v_j)$.

Neighborhood interchangeability is a sufficient (but not necessary) condition for full interchangeability [24]. A set of NI values can be replaced by a single representative of the set without losing any solutions. Experiments showed that this can reduce the

search effort in standard CSPs when applied as a preprocessing step or during search, and especially when looking for all solutions to a problem [2, 37]. In what follows we will often refer to NI values simply as interchangeable.

In the context of QCSPs we can exploit interchangeability to break symmetries by pruning the domains of universal variables. That is, for each set (sometimes called bundle) of NI values we can keep one representative and remove the others, either permanently before search, or temporarily during search. If the algorithm finds a consistent scenario for the representative value then surely there exists one for the rest of the NI values as well. Therefore, branching on these values is redundant. Consider the following example.

Example 3 We have the QCSP $\forall v_1 \exists v_2 \exists v_3 \ (v_1 \neq v_2 \wedge v_1 \neq v_3)$, where the domains of the variables are $D(v_1) = \{0, 1, 2, 3, 4\}$, $D(v_2) = \{0, 1\}$, $D(v_3) = \{0, 2\}$. Values 3 and 4 of v_1 are NI since they are supported by the same values in both v_2 and v_3. Therefore, they can be replaced by a single value or, to put it differently, one of them can be pruned out of the domain.

The cost of computing all neighborhood interchangeable values in a CSP, using the algorithm of [24], is O(d^2n^2). In QCSPs we can detect NI values as a preprocessing step and thus remove values from the domains of universal variables, and we can also detect them dynamically during search to avoid repeated exploration of similar subtrees.

Example 4 Assume that variables $\forall v_i \exists v_j \exists v_k \exists v_l$ are part of a QCSP and their domains are $D(v_i) = \{a_1, a_2\}$, $D(v_j) = D(v_k) = D(v_l) = \{a_3, a_4, a_5, a_6\}$. Also, the QCSP includes constraints c_{ij}, c_{ik}, c_{il}. Assume that value a_1 is supported by values a_3, a_4, a_5 in each of v_j, v_k and v_l, and a_2 is supported by a_3, a_4, a_6. If the current variable at some stage of search is v_i and values a_5 and a_6 have been previously removed from the domains of v_j, v_k and v_l then at that stage a_1 and a_2 are NI. We can proceed to search for a consistent scenario that includes assignment $v_i \mapsto a_1$. If one is found then when we backtrack to universal variable v_i we do not need to perform a similar search for assignment $v_i \mapsto a_2$. If we backtrack further back and undo the deletions of values a_5, a_6 from the domains of v_j, v_k and v_l then the next time we reach variable v_i the values a_1 and a_2 may not be NI.

Naturally, we can also use NI to reduce the domains of existential variables as proposed by Freuder. However, our experiments showed that this is an overhead that slows down the algorithms. The (small) reduction in the number of search tree node visits is outweighed by the cost of computing the NI values of existentials.

NI-based symmetry breaking can be embedded in the search algorithms described previously using two simple procedures. The first one detects bundles of NI values as a preprocessing step, keeps one representative of each bundle and removes the rest from the domains of universals. To check if two values a, b of a universal variable v_i are NI, this procedure iterates through the domain of any existential variable $v_j \in V$ that is constrained with v_i and is after v_i in V. If a value is found that is a support for a but not for b, or vice versa, then a and b are not NI. Otherwise, they are NI, so one of them is removed. This is repeated for all pairs of values of v_i. In a similar way, the second procedure dynamically detects bundles of NI values each time the algorithm reaches a

universal variable v_i. In this case, one representative of each bundle is kept, and the rest of the values are temporarily removed from $D(v_i)$. The values are restored when the algorithm backtracks to a variable before v_i in V. The worst-case time complexity of these symmetry detecting procedures, as they are currently implemented, is O(d^3n^2).

4 Encoding QCSP as QBF

In this section we first give some general background on QBFs. Then we briefly elaborate on the difficulties in encoding QCSP into QBF and describe the features of the QBF solver we used in our experiments. In the main part of the section we present the previous best encoding of QCSP into QBF (the adapted log encoding), and we introduce a new encoding which improves on it, both in simplicity and performance.

4.1 Quantified Boolean Formulae

A special case of a QCSP is a Quantified Boolean Formula (QBF). A QBF is of the form $\langle V, Q, D, C\rangle$ where V and Q are defined as in Definition 1, but each domain in D has only two elements $\{F, T\}$ (or $\{0, 1\}$). C is a Boolean formula in conjunctive normal form (CNF), a conjunction of clauses where each clause is a disjunction of literals. Each literal is a variable and a sign. A literal is said to be negative if negated and positive otherwise. A *universal literal* is a literal whose variable is universally quantified and an *existential literal* is a literal whose variable is existentially quantified. The semantic definition is the same as for QCSPs. Note that 2-QBF (i.e. QBF problems with at most two literals per clause) is solvable in polynomial time. However, binary QCSPs are PSPACE-complete [8].

A QBF is vacuously true if it consists of an empty set of clauses. It is vacuously false if the set of clauses contains either an empty clause (i.e. a clause with no literals) or an all universal clause (i.e. a clause with only universal literals).

4.2 The difficulty in encoding QCSP to QBF

Gent, Nightingale and Rowley introduced a number of different ways to encode a QCSP instance into QBF [28]. To encode an existential QCSP variable to a set of QBF variables, some assignments to the QBF variables represent values of the original variable, and other assignments are ruled out by adding clauses to the formula. For example, if an assignment to the QBF variables indicates that the original QCSP variable has no values in its domain, the assignment is invalid and is ruled out with a clause. However this approach is not possible for a universal QCSP variable.

To see why, consider the following example. In QBF instance ϕ, we have two universal variables x_i and x_j. ϕ represents an adversarial game, and the assignment $x_i = T$, $x_j = T$ represents a cheating move in the game. Naively we might use a clause $(\neg x_i \vee \neg x_j)$ to rule out this assignment. Unfortunately, such a clause is trivially false, and therefore would render ϕ false.

The encodings introduced in [28] were able to overcome this difficulty. However, the *global acceptability encoding* and the *local acceptability encoding* were very in-

efficient compared to direct QCSP algorithms. In contrast, the *adapted log encoding*, which we describe below, turned out to be very efficient.

4.3 QBF Solver

Many advanced solvers for QBF have been proposed in the literature [3,7,22,27,33,35]. We concentrate on search-based solvers, which interleave search (by instantiating variables individually, in quantifier order) with reasoning on the formula (local reasoning). They are based on the Davis-Putnam-Logemann-Loveland algorithm [19, 20], adapted to QBF [14]. Local reasoning cuts down the search space. Also, in certain situations, backjumping is applied which allows the solver to jump several levels up the search tree (undoing several search operations at once), by identifying the cause of a success or failure.

We give a brief overview of some of the literature to set the encodings in context. We use the solver CSBJ [27], which implements the following two local reasoning techniques. We only sketch each technique in the broadest way and refer the reader elsewhere for full details.

Unit propagation A literal l of variable x is *unit* if it appears alone in a clause, or if the other literals in the clause are universal, and their corresponding variables are quantified after x [14]. When l is unit, it is instantiated ($l = T$) and the formula is simplified. If x is universal, the formula simplifies to false. Otherwise, clauses containing l are removed, and all literals $\neg l$ are removed. This may cause other literals to become unit.

The pure literal rule A literal l is called *pure* (or *monotone*) if its complementary literal does not appear in any clause [14]. Such literals are important because they can immediately be assigned a value without any need for branching. This is what the *pure literal rule* does. If an existential pure literal l is found, it is set to true. If a false leaf node is then reached, assigning l to false will be unnecessary since it is certain that this will again lead to a false leaf node. If a universal pure literal l is found, it is set to false. If a true leaf node is then reached, assigning l to true will be unnecessary since it is certain that this will again lead to a true leaf node.

As we mentioned, local reasoning and search is commonly augmented with backjumping. CSBJ implements conflict and solution backjumping. We informally describe these two techniques. This is only intended to give a flavor of the techniques.

Conflict-based backjumping Conflict-based backjumping (CBJ) is a look-back technique, originally proposed for CSPs, that tries to reduce the number of backtracks performed by a search algorithm [39]. CBJ tries to deal with dead-ends in an intelligent way by recording and exploiting *conflict sets*. A conflict set is a set of existential literals whose assignments are responsible for a contradiction in the formula, i.e. an empty or all universal clause. When a contradiction is encountered, CBJ backjumps to one of the existential literals in the conflict set of the current variable, instead of blindly backtracking to the last assigned existential. In this way, search effort can be saved.

Solution-directed backjumping Solution-directed backjumping (SBJ) is a specialized technique for QBF that tries to avoid redundant search after a true leaf node is reached in the search tree [33]. This is accomplished by recording and exploiting *solution sets*. A solution set is a set of universals such that all clauses not satisfied by the current assignment of the existentials are satisfied by at least one of the universals. After a true leaf node is reached, a solution set is calculated and SBJ backjumps to one of the variables in the set, possibly jumping over some universals.

Efficient implementation is crucial in the SAT domain, and various techniques have been carried across into the solver we used. For example, watched literals give us efficient lazy unit propagation, and watched clauses do the same for the pure literal rule [27].

4.4 Adapted Log Encoding

This section describes a previous contribution, the adapted log encoding (by Gent, Nightingale and Rowley [28]). It is described here in order to set the enhanced log encoding (in the following section) in context. We briefly explain the main groups of clauses in the encoding. For full technical details we refer the reader to Gent et al. [28].

In order to deal with the difficulty described above, the adapted log encoding uses indicator variables (first described by Rowley [32]) to indicate when a universal assignment is not valid. An indicator variable takes value T iff a particular (invalid) assignment is made to universal variables. There is one indicator variable z^v for each of the original universal QCSP variables v. z^v is existentially quantified in a final block at the end of the variable sequence. All clauses representing constraints contain a literal z^v. Hence, they are true under any assignment setting z^v to true. In this way, if an invalid assignment is made to universal variables, the formula simplifies to true by unit propagation, as required to deal with the difficulty.

In SAT, it has often been noted that just three variables are needed to encode 8 values of a CSP variable, instead of the 8 in the direct encoding [25, 45]. This is known as the log encoding. Walsh proves that unit propagation on the log encoding does less work than on the direct encoding [45], and hence it is rarely used. However we adapt the log encoding for QCSP with good results.

Each variable in a QCSP is encoded to a set of variables in QBF, with these sets quantified in the same way and in the same order as in the QCSP. Additional existential variables are added to the end of the variable sequence. For an existential variable, each QBF variable represents one value. For a universal, each value is represented by a unique assignment to the QBF variables, and also each value is represented by an existential QBF variable quantified at the end. There are clauses which maintain correspondence between these two representations, named *channelling* clauses here.

We now describe the encoding in more detail. We first show how the QCSP variables are encoded and then present the clauses of the encoding. The notation w^u_* is used for the set of w variables with superscript u and any subscript. The $*$ is used in the same way with x^u_* and i^u_*.

- **Quantification**
  To encode some existential variable $v \in \{1 \dots d\}$:
  - We use existential variables $\exists x^v_1, \dots, \exists x^v_d$

  To encode some universal variable $u \in \{1 \dots d\}$:
  - We use universal variables $\forall w^u_{\lceil \log_2 d \rceil - 1}, \dots, \forall w^u_0$
  - We also use existential variables representing each value: $\exists x^u_1, \dots, \exists x^u_d$
  - Finally, We use indicator variables for each invalid assignment to w^u_*, and one overall indicator variable: $\exists i^u_*, \exists z_u$

For some universal variable v, the following clauses map assignments of w^v_* to x^v_* and i^v_*. These *channelling* clauses ensure that at least one of the x^v_* or i^v_* variables is set to T, and that an indicator variable in i^v_* is only set to T when the w^v_* variables take the corresponding invalid assignment. Variable u is the universal that directly precedes v in the QCSP variable order. The indicator variable z_u is T only when u or a previous universal has an invalid assignment: it is used to make the channelling clauses true in this situation.

The clauses are given as an example for $d = 5$, but the general form is easy to infer from the example.

- **Channelling clauses**

$$
\begin{array}{l}
z_u \lor ((\neg w^v_2 \land \neg w^v_1 \land \neg w^v_0) \Rightarrow x^v_1) \\
z_u \lor ((\neg w^v_2 \land \neg w^v_1 \land w^v_0) \Rightarrow x^v_2) \\
z_u \lor ((\neg w^v_2 \land w^v_1 \land \neg w^v_0) \Rightarrow x^v_3) \\
z_u \lor ((\neg w^v_2 \land w^v_1 \land w^v_0) \Rightarrow x^v_4) \\
z_u \lor ((w^v_2 \land \neg w^v_1 \land \neg w^v_0) \Rightarrow x^v_5) \\
z_u \lor ((w^v_2 \land \neg w^v_1 \land w^v_0) \iff i^v_6) \\
z_u \lor ((w^v_2 \land w^v_1 \land \neg w^v_0) \iff i^v_7) \\
z_u \lor ((w^v_2 \land w^v_1 \land w^v_0) \iff i^v_8)
\end{array}
$$

These eight expressions correspond to all possible assignments to $\{w^v_2, w^v_1, w^v_0\}$, from $\langle F, F, F\rangle$ for the first expression, to $\langle T, T, T\rangle$ for the last. Each assignment is linked either to an x^v variable if it is valid, or an i^v variable otherwise. The expressions are expanded into clauses in the encoding.

The variables i^v_* indicate when the assignment is invalid in a particular way. These variables are accumulated into a single indicator variable z_v which is T iff at least one of i^v_* is T, or the previous accumulated indicator variable z_u is T.

- **Indicator collector clauses**

$$z_v \iff i^v_6 \lor i^v_7 \lor i^v_8 \lor z_u$$

Note that if z_v is set to T, this assignment will be propagated to the next indicator collector clause and thus will make the next accumulated indicator variable T. This assignment will be propagated further, and so on.

For each existential variable v, at least one of the QBF variables x^v_* must be set to true to ensure that v is assigned a value. This is accomplished with an at-least-one (ALO) clause.

- **ALO clause**

$$\bigvee_{i\in 1\ldots d} x_i^v$$

Constraints are represented as follows. Consider a constraint c_{uv} between variables u and v, where u precedes v in the variable order. A pair of values $\langle i, j\rangle$, where $i \in D(u)$ and $j \in D(v)$, that do not satisfy the constraint (i.e. they do not belong to $rel(c_{uv})$) is represented with a single clause in the QBF. Assume variable t is universally quantified and directly precedes v in the variable order. Note that t may be the same as u. The indicator variable z_t for t is used, so that if a preceding universal variable is set in an invalid way, the conflict clause is satisfied. (When a universal is set invalidly, the remaining part of the QBF must be true). Given this, conflict clauses only contain indicator variables and negative literals.

- **Conflict clauses**

$$\begin{array}{l}\forall\langle i, j\rangle \notin rel(c_{uv}) \ : \\ z_t \vee \neg x_i^u \vee \neg x_i^v\end{array}$$

For channelling and conflict clauses, if there is no preceding universal variable in the QCSP, the indicator variable is omitted. To illustrate the encoding, we give an example of a simple QCSP.

Example 5 Consider the QCSP $\forall v \exists u : v \neq u$ where $D(v) = D(u) = \{1, \ldots, 5\}$. This is encoded as follows:

- QBF variables:

$$\forall w_2^v, w_1^v, w_0^v, \exists x_1^v, x_2^v, x_3^v, x_4^v, x_5^v, \exists x_1^u, x_2^u, x_3^u, x_4^u, x_5^u, \exists z^v, i_6^v, i_7^v, i_8^v$$

- Channelling clauses for v:

$$\begin{array}{l}(\neg w_2^v \wedge \neg w_1^v \wedge \neg w_0^v) \Rightarrow x_1^v \\ (\neg w_2^v \wedge \neg w_1^v \wedge w_0^v) \Rightarrow x_2^v \\ (\neg w_2^v \wedge w_1^v \wedge \neg w_0^v) \Rightarrow x_3^v \\ (\neg w_2^v \wedge w_1^v \wedge w_0^v) \Rightarrow x_4^v \\ (w_2^v \wedge \neg w_1^v \wedge \neg w_0^v) \Rightarrow x_5^v \\ (w_2^v \wedge \neg w_1^v \wedge w_0^v) \iff i_6^v \\ (w_2^v \wedge w_1^v \wedge \neg w_0^v) \iff i_7^v \\ (w_2^v \wedge w_1^v \wedge w_0^v) \iff i_8^v\end{array}$$

- Indicator collector clauses for v:

$$z^v \iff i_6^v \vee i_7^v \vee i_8^v$$

- At-least-one clause for u:

$$x_1^u \vee x_2^u \vee x_3^u \vee x_4^u \vee x_5^u$$

- Conflict clauses representing $v \neq u$:

$$z^v \vee \neg x_1^v \vee \neg x_1^u$$
$$z^v \vee \neg x_2^v \vee \neg x_2^u$$
$$z^v \vee \neg x_3^v \vee \neg x_3^u$$
$$z^v \vee \neg x_4^v \vee \neg x_4^u$$
$$z^v \vee \neg x_5^v \vee \neg x_5^u$$

The subtlety of this encoding is that we omit the clauses which force equivalence between x_a^v and the corresponding values of w_b^v: the implication is one way in the channelling clauses involving x_a^v. So we omit clauses such as $z_u \vee \neg x_1^v \vee \neg w_2^v$. It might seem that this is erroneous, as it allows a universal to take two values, if x_1^v and x_2^v are both true. But there is no way that setting of the universal variables w_*^v can *force* more than one x_a^v to be true. The advantage of this arrangement is that x_*^v variables occur only positively in the channelling clauses. Therefore if a particular variable x_v^1 did not occur in any conflict clause, it would be pure. Furthermore, this could lead to w_*^v variables becoming pure, reducing the need for search. So, with sufficient care, we can use the pure literal rule included in our QBF solver, and have it work in the QCSP case. This property carries over to the enhanced log encoding, described below.

4.5 Enhanced Log Encoding

The *enhanced log encoding* is a refinement of adapted log which has not been previously published. Each universal variable v is encoded by $\lceil \log_2 d \rceil$ variables w_*^v which are consecutively universally quantified. The order of variables is preserved. We also introduce $x_1^v \dots x_d^v$ variables for each universal QCSP variable v, which are existentially quantified at the end of the variable sequence. These x_*^v variables are used in the conflict clauses. The w_*^v variables are channelled to the x_*^v variables with a set of d clauses. For the following example $d = 5$.

- **Channelling clauses**

$$(\neg w_2^v \wedge \neg w_1^v \wedge \neg w_0^v) \Rightarrow x_1^v$$
$$(\neg w_2^v \wedge \neg w_1^v \wedge w_0^v) \Rightarrow x_2^v$$
$$(\neg w_2^v \wedge w_1^v) \Rightarrow x_3^v$$
$$(w_2^v \wedge \neg w_1^v) \Rightarrow x_4^v$$
$$(w_2^v \wedge w_1^v) \Rightarrow x_5^v$$

There are 8 possible assignments to the w_*^v variables, and 5 values, so for the values 3, 4 and 5 there are two w_*^v assignments mapped onto each, hence all 8 assignments are valid. In contrast to adapted log, no local acceptability variable (z_v in the previous subsection) is present, because no assignments to previous universal variables can be invalid.

To state this formally, we represent a QBF with the tuple $F' = \langle Q', V', C' \rangle$ where Q' is the quantifier mapping, V' is the ordered set of Boolean variables and C' is the set of disjunctive clauses, to mirror the QCSP $F = \langle Q, V, D, C \rangle$. Domains are excluded as they are always $\{0, 1\}$.

The ordered set of variables V for the QCSP is encoded by an ordered set of Boolean variables as shown by the following recursive rules where translate$(V) = V'$.

translate($\exists v \in \{1 \dots d\}, V_1) = \exists x_1^v \dots \exists x_d^v$, translate($V_1$)
translate($\forall v \in \{1 \dots d\}, V_2) = \forall w^v_{\lceil \log_2 d \rceil - 1} \dots \forall w_0^v$, translate($V_2$), $\exists x_1^v \dots \exists x_d^v$

An existential variable v in the QCSP instance is mapped to d existential variables ($x_1^v \dots x_d^v$) in the encoding. These represent each value in the domain. The enhanced log encoding also has the at-least-one clause ($\bigvee_{i \in 1 \dots d} x_i^v$). In this respect, the enhanced log encoding is identical to the adapted log.

A universal variable v in the QCSP is mapped to $l = \lceil \log_2 d \rceil$ variables w_*^v. Every complete assignment A to variables w_*^v (of which there are 2^l) is mapped to a value $b \in D(v)$. All values b map to one assignment, or two assignments with only one literal different. It is never necessary to have a value b mapping to more than two assignments, whatever the value of d. $2^l - d$ values must map to two assignments. An assignment A is represented as a conjunction of literals (e.g. $w_0^v \wedge \neg w_1^v$). For some value b which maps to just one assignment A, the channelling clause is as follows.

$$\neg A \vee x_b^v$$

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

$$\neg(A_1 \vee A_2) \vee x_c^v$$

The simplification of $\neg(A_1 \vee A_2)$ ends with a disjunction of $l - 1$ literals.

For a constraint c_{ij}, with satisfying tuples $rel(c_{ij})$ the conflict clauses are:

- **Conflict clauses** For all tuples $\langle a, b \rangle \notin rel(c_{ij})$,

$$(\neg x_a^{v_i} \vee \neg x_b^{v_j})$$

To illustrate the encoding, we encode the QCSP of Example 5.

Example 6 We have the QCSP $\forall v \exists u : v \neq u$ where $D(v) = D(u) = \{1, \dots, 5\}$. This is encoded as follows:

- QBF variables:

$$\forall w_2^v, w_1^v, w_0^v, \exists x_1^v, x_2^v, x_3^v, x_4^v, x_5^v, \exists x_1^u, x_2^u, x_3^u, x_4^u, x_5^u$$

- Channelling clauses for v:

$$\begin{array}{l} (\neg w_2^v \wedge \neg w_1^v \wedge \neg w_0^v) \Rightarrow x_1^v \\ (\neg w_2^v \wedge \neg w_1^v \wedge w_0^v) \Rightarrow x_2^v \\ (\neg w_2^v \wedge w_1^v) \Rightarrow x_3^v \\ (w_2^v \wedge \neg w_1^v) \Rightarrow x_4^v \\ (w_2^v \wedge w_1^v) \Rightarrow x_5^v \end{array}$$

- At-least-one clause for u:

$$x_1^u \vee x_2^u \vee x_3^u \vee x_4^u \vee x_5^u$$

- Conflict clauses representing $v \neq u$:

$$\neg x_1^v \vee \neg x_1^u$$
$$\neg x_2^v \vee \neg x_2^u$$
$$\neg x_3^v \vee \neg x_3^u$$
$$\neg x_4^v \vee \neg x_4^u$$
$$\neg x_5^v \vee \neg x_5^u$$

Theorem 1 A QCSP is true if and only if the encoded QBF is true, for the enhanced log encoding.

Proof: The proof is recursive and closely follows the definition of QCSP semantics (definition 2). A QCSP $F = \langle V, Q, D, C \rangle$ represents the logical formula $\phi = Q(v_1)v_1{\in}D(v_1)\dots Q(v_n)v_n{\in}D(v_n)$ (C) which is encoded as a QBF $F' = \langle V', Q', C' \rangle$ representing $\phi' = Q'(x_1)x_1 \dots Q'(x_n)x_n$ (C'). The encoding of the empty QCSP (containing no variables or constraints) is the empty QBF which is vacuously true.

Existential case:

- Assume ϕ is of the form $\exists v_1 Q(v_2) \dots (C)$ (domains are omitted for simplicity).
- Now ϕ' must be of the form $\exists x_1^{v_1} \dots \exists x_d^{v_1}$translate$(Q(v_2)\dots)(C')$.
- By definition 2, F is true iff there exists some value $a \in D(v_1)$ such that $Q(v_2)\dots Q(v_n)(C[v_1 \mapsto a])$ is true.
- Equivalently, in the encoding F' is true iff there exists an assignment $A = x_1^{v_1} \mapsto b_1 \dots x_d^{v_1} \mapsto b_d$ such that the ALO clause is true and translate$(Q(v_2)\dots)$ $(C'[A])$ is true.

The QCSP value a can be any value such that $x_a^{v_1} \mapsto 1$. If there is more than one a s.t. $x_a^{v_1} \mapsto 1$, then all these values can be extended to a solution.

Universal case:

- Assume ϕ is of the form $\forall v_1 Q(v_2) \dots (C)$.
- Now ϕ' must be of the form $\forall w_l^{v_1} \dots \forall w_0^{v_1}$translate$(Q(v_2)\dots)(C')$ where $l = \lceil \log_2 d \rceil - 1$.
- By definition 2, F is true iff for all values $a \in D(v_1) : Q(v_2) \dots (C[v_1 \mapsto a])$ is true.
- Equivalently, in the encoding F' is true iff for all assignments $A = w_l^{v_1} \mapsto b_l \dots w_0^{v_1} \mapsto b_0$: translate$(Q(v_2)\dots)(C'[A])$ is true.

Note that each value a is covered by some assignment A. If the assignment A is made, the additional $x_a^{v_1}$ variable introduced by the encoding must be 1 because of the channelling clauses. Other variables $x_{b \neq a}^{v_1}$ are not constrained by the channelling clauses, and therefore can be set to 0 if they are contained in any conflict clause.

Therefore, by examination of definition 2, the encoding is true iff the original QCSP is true, because each step of the recursion of definition 2 can be performed equivalently in the QCSP and in the encoding. *QED*

The problem with this encoding is that the QBF solver can search two equivalent subtrees in some cases, for example when $w_2^v \mapsto T$ and $w_1^v \mapsto T$, the solver can branch on w_0^v which is not contained in any clause.

After setting w_2^v and w_1^v, if either is set to T then the first two clauses above are satisfied and in the reduced set of clauses w_0^v does not exist. Both w_0^v and $\neg w_0^v$ are pure, so if the solver implements the pure literal rule then it will not branch on this variable. This solves the repeated subtree problem mentioned above, on the condition that w_0^v is set last. Also, in common with the adapted log encoding, the channelling works only from w^v to x^v variables, so only positive x^v literals are included in the clause set above, therefore the pure literal rule can detect cases where the x_a^v is involved in no conflicts. In some circumstances, this can also lead to the elimination of w^v variables. For example, if x_4^v and x_5^v become pure, then w_2^v becomes pure as well and the search is reduced accordingly.

5 QCSP-Solve: A Direct Solver for QCSPs

The efficiency of the adapted and enhanced log encodings is largely due to their ability to exploit sophisticated techniques offered by the underlying QBF solver; namely, the pure literal rule, conflict-based backjumping, and solution-directed backjumping. Two questions that immediately arise are: what do these techniques correspond to in QCSPs, and how can we implement them within direct algorithms? In this section we try to answer these questions and describe the resulting efficient direct solver, which we call QCSP-Solve.

QCSP-Solve performs a backtracking search, as described in Section 3, augmented with various capabilities. First of all, QCSP-Solve always applies algorithm QAC-2001 as a preprocessing step. As explained in Section 3.1, apart from reducing the problem size by deleting values from the domains of existentials, QAC-2001 removes from the problem all constraints of the form $\exists\, v_i \forall\, v_j\ (c_{ij})$ and $\forall\, v_i \forall\, v_j\ (c_{ij})$. During search, QCSP-Solve can apply any of the basic forms of look-ahead described in Section 3, i.e. FC0, MAC0, and their enhancements FC1 and MAC1. In what follows we will describe how new look-ahead and look-back techniques are combined with an FC-based look-ahead. Most of these techniques can be combined with a MAC-based look-ahead in a very similar way.

5.1 The Pure Value Rule

Our experiments showed that the most important QBF technique, in terms of its practical effectiveness in the encoded QCSPs, is the pure literal rule. We now explain what this corresponds to in a binary QCSP, and how we can exploit it to prune the search space. We first define the notion of a *pure value*.

Definition 5 A value $a \in D(v_i)$ in a QCSP $F = \langle V, Q, D, C\rangle$ is *pure* iff for each $v_j \in V$, where $v_j \neq v_i$ and for each $b \in D(v_j)$, the assignments $v_i \mapsto a$ and $v_j \mapsto b$ are compatible.

Bordeaux et al. introduced the notion of a *fixable* value in a CSP [10]. In few words, a value a of a variable v is fixable if for any solution which includes the assignment of a value b to v, we still have a solution if $v \mapsto b$ is substituted with $v \mapsto a$. As noted in [10], a sufficient (but not necessary) condition for determining the fixability of a value can be computed through local reasoning in polynomial time. This is similar to the pure literal rule in SAT. The same authors defined the notion of fixability for QCSPs [11]. Following the terminology of [11], if a value is pure this is a sufficient but not necessary condition for the value to be $d-$fixable.

In a way analogous to the pure literal rule in QBF, we have devised and implemented a look-ahead technique, which we call the *pure value (PV) rule*, that detects and exploits pure values. The actions taken are dual for existential and universal pure values. An existential variable with a pure value can be set to that value as it will not violate any constraint in any scenario. On the other hand, a pure value is removed from the domain of a universal variable as it will certainly be part of any solution and thus we do not need to search for a consistent scenario that includes it. This duality reflects the dual semantics of existential and universal variables. For a universal variable, showing that a value a is pure does not prove that it leads to a consistent scenario, only that if some other value of the same variable leads to a consistent scenario then a does. Hence it is a subsumption rule and the last value in the domain cannot be removed.

Note that values can become pure dynamically during search as variable assignments and constraint propagation remove values from the domains of the variables (see Example 9 in Section 5.3). Therefore, the PV rule is applied both as a preprocessing technique and as a dynamic look-ahead technique during search. The PV rule works as follows.

- If a pure value a of an existential v_i is discovered during preprocessing (*search*), then the assignment $v_i \mapsto a$ is made and all other values of v_i are permanently (*temporarily*) removed from $D(v_i)$. To check, during search, if a value a of an existential v_i is pure, we only need to check if the assignment $v_i \mapsto a$ is compatible with all values of future variables. FC (or MAC) guarantee that $v_i \mapsto a$ is compatible with the instantiations of the past variables.

- If a pure value a of a universal v_i is discovered during preprocessing (*search*), then a is permanently (*temporarily*) removed from $D(v_i)$ unless it is the final value in $D(v_i)$. To check if a value of a universal is pure, we only need to check against future variables since preprocessing with AC guarantees that there are no constraints between a universal and a past variable. If we discover during preprocessing that all the values of a universal are pure then we can ignore it thereafter as it is certain that all its values can be part of any consistent scenario.

In both cases, any value that was temporarily removed because of the pure value rule is restored once a backtrack to a variable before v_i in V occurs.

Currently, the PV rule is implemented within two simple functions; one for preprocessing and another for the dynamic application of the rule during search. In both cases, to detect the pure values of a variable v_i we iterate through the domains of the other variables that are constrained with v_i (only variables after v_i in case of dynamic application). During preprocessing we have to repeat this for all the variables in the

problem, which gives a worst-case time complexity of O(n^2d^2). During search we can restrict PV detection to the values of the current variable. This gives a worst-case time complexity of O(nd^2).

The function that applies the PV rule during search need only be called before assigning the current variable v_{cur} with its first available value. That is, immediately after line 2 in Figure 4, assuming that the underlying algorithm is FC0. Calling the function again when a backtrack to v_{cur} later occurs is redundant as the restoration of the domains after a backtrack guarantees that the pure values that were previously detected will remain pure and no new values will become pure.

Relation between the pure value and the pure literal rule The PV rule applied to a QCSP F has a similar effect to the application of the pure literal rule (PL) to the enhanced log encoding (E). In some cases, PV and PL are equivalent, and in other cases details are different and an exact equivalence is elusive. To discuss this, we consider four cases, two each for existential and universal variables.

If an existential variable v_e has one pure value a in F, v_e will be assigned to a in F. In the encoding PL will assign $x_a^{v_e} \mapsto T$, since if $v_e \mapsto a$ is not contained in any conflict in F, then $\neg x_a^{v_e}$ is not contained in any conflict clause in E. The instantiation $x_a^{v_e} \mapsto T$ makes the ALO clause for v_e true.

If an existential variable v_e in F has more than one pure value $a, b, \ldots$, then one of the pure values is instantiated by the PV rule. The value instantiated would typically be the first discovered by the algorithm. In the encoding, *all* corresponding variables $x_a^{v_e}, x_b^{v_e}, \ldots$ are positively pure, and are instantiated to T by the PL rule. Again, this makes the ALO clause true.

If a universal variable v_u in F has a set of pure values P, and $P \subsetneq D(v_u)$, then all values in P are removed. In the encoding, all variables $x_a^{v_u}$, where $a \in P$, are positively pure, and are instantiated to T by the PL rule. Thus the channelling clauses containing each of these variables are true, and the variables are entirely removed from E. This may cause variables in the set $w_*^{v_u}$ to become pure, reducing the number of branches explored by the QBF solver. However, there is not an exact equivalence here between PV and PL.

If all values of a universal variable v_u in F are pure, then all but one are removed in F. In the encoding, all variables $x_*^{v_u}$ are positively pure and all are set to true by the PL rule. Consequently, all channelling clauses are true, and all variables $w_*^{v_u}$ are pure. Therefore they are also instantiated, and the QBF solver does not branch on any variable in the set $w_*^{v_u}$. Similarly, the QCSP solver does not branch on v_u after instantiating it.

5.2 CBJ and Solution-Directed Pruning

5.2.1 Conflict-Based Backjumping

CBJ has been successfully combined with FC in CSPs [39], and a DLL-based procedure in QBF [33] to deal with dead-ends in an intelligent way and, thus, avoid redundant search. We now explain how CBJ is implemented in QCSP-Solve.

As in CSPs, for each variable $v_i \in V$ we keep a set of variables called *conflict set* and denoted by $conf_set(v_i)$. This holds the past existentials that are responsible for the deletion of values from $D(v_i)$. Initially all conflict sets are empty. When encountering a dead-end at an existential or when a value of a universal is rejected, the algorithm exploits information kept in the conflict set of the current variable v_{cur} to backjump to one of the past existentials that are responsible for the dead-end instead of blindly backtracking chronologically to the previous existential in V. To be precise, the algorithm backjumps to the most recently instantiated existential, say v_k, among the existentials in $conf_set(v_{cur})$ and reassigns it with its next available value. As v_k's previous assignment caused the deletion of (at least one) value from $D(v_{cur})$, if we reassign it, one or more values in $D(v_{cur})$ may now become available. In contrast, an algorithm that always backtracks chronologically, as BT and the two FC variants do, may repeatedly encounter a dead-end since the existential immediately before v_{cur} in V may not belong to $conf_set(v_{cur})$. In this case, reassigning this existential will not "free" any of v_{cur}'s values and therefore the dead-end at v_{cur} will be encountered again.

Conflict sets are updated as follows.

- If the current variable v_{cur} is existentially quantified and, during forward checking, a value of a future variable v_j is found to be incompatible with the assignment of v_{cur} then v_{cur} is added to $conf_set(v_j)$. This is straightforward as now the assignment of v_{cur} is responsible for the removal of a value from $D(v_j)$.

- If, after assigning a value a to v_{cur} (which may be existential or universal) and forward checking, the domain of a future existential v_j is wiped out then the existentials in $conf_set(v_j)$ are added to the conflict set of the current variable. This is done because the domain wipe-out of v_j will result in value a being rejected. The past existentials that can be considered responsible for this rejection are the ones whose instantiations removed values from $D(v_j)$. To understand this consider that were it not for the assignments to these existentials, a might not be rejected since $D(v_j)$ would not have been wiped out. Note that this is the only way in which the conflict set of a universal can be updated.

Backjumping can occur in either of the following two cases:

1. If the current variable v_{cur} is existential and there are no more values to be tried for it then the algorithm backjumps to the latest (i.e. the most recently instantiated) existential v_k in V that belongs to $conf_set(v_{cur})$. At the same time all variables in $conf_set(v_{cur})$ (except v_k) are copied to $conf_set(v_k)$ so that no information about conflicts is lost. This requires some explanation.

   Assume that v_k was added to $conf_set(v_{cur})$ because the assignment $v_k \mapsto a$ resulted in the removal of value b from $D(v_{cur})$. If after the backjump to v_k all remaining values of v_k are rejected then we have a dead-end and must jump further back. Now assume that the most recent existential v_l in $conf_set(v_k)$, where the algorithm will backjump, was copied to $conf_set(v_k)$ from $conf_set(v_{cur})$. Since v_l was in $conf_set(v_{cur})$, its current assignment resulted in the removal of (at least one) value from v_{cur}. When we change v_l's assignment it is possible

that one or more of these values will not be removed from $D(v_{cur})$. Therefore, it may be possible to reassign v_k with a in the future without causing a dead-end at v_{cur} further down the search tree. This gain of search effort would not be possible if $conf_set(v_{cur})$ was not copied to $conf_set(v_k)$ since we would have to backjump to an existential before v_l in V once encountering the dead-end at v_k. Example 7 further demonstrates this reasoning.

2. If the current variable v_{cur} is universal and one of its values fails (because it results in the domain wipe-out of an existential v_j) then the algorithm backjumps to the latest existential v_k in V that belongs to $conf_set(v_{cur})$. That is, to the most recent existential whose instantiation removed a value from $D(v_j)$. Again all variables in $conf_set(v_{cur})$ (except v_k) are copied to $conf_set(v_k)$ so that no information about conflicts is lost. The reasoning behind this is similar to above.

Example 7 Consider the following QCSP where V consists of 6 quantified variables, and C is a conjunction of 5 constraints: $\exists v_1 \exists v_2 \exists v_3 \exists v_4 \forall v_5 \exists v_6 (v_1 = v_3 \wedge v_2 \neq v_6 \wedge v_3 \neq v_6 \wedge v_4 \leq v_6 \wedge v_5 \neq v_6)$. Assume that the domains of the variables are as follows: $D(v_1) = D(v_3) = D(v_4) = D(v_5) = D(v_6) = \{0, 1, 2\}$, $D(v_2) = \{2, 3\}$. Algorithm FC1 equipped with CBJ will proceed to solve the problem as follows.

Variable v_1 is assigned its first value 0. Forward checking removes values 1 and 2 from $D(v_3)$ and we set $conf_set(v_3) = \{v_1\}$. Variable v_2 is assigned its first value 2. Forward checking removes value 2 from $D(v_6)$ and we set $conf_set(v_6) = \{v_2\}$. Variable v_3 is assigned its first value 0. Forward checking removes value 0 from $D(v_6)$ and v_3 is added to $conf_set(v_6)$. So now we have $conf_set(v_6) = \{v_2, v_3\}$. Variable v_4 is assigned its first value 0. Forward checking does nothing. We now reach variable v_5 which is universally quantified. FC1 will forward check each of v_5's values against v_6. Value 1 of v_5 results in the domain wipe-out of v_6. Therefore, $conf_set(v_6)$ will be copied to $conf_set(v_5)$ and we have $conf_set(v_5) = \{v_2, v_3\}$. Since one of v_5's values failed we must backjump to the most recent variable in $conf_set(v_5)$ which is v_3. All variables in $conf_set(v_5)$ (except v_3) will be copied to $conf_set(v_3)$ and we now get $conf_set(v_3) = \{v_1, v_2\}$. There are no more available values in $D(v_3)$ and therefore the algorithm will jump further back to the most recent variable in $conf_set(v_3)$, which is v_2.

Variable v_2 is assigned its next value 3. Forward checking does nothing. Variable v_3 is assigned its first value 0. Forward checking removes value 0 from $D(v_6)$ and we set $conf_set(v_6) = \{v_3\}$. Variable v_4 is assigned its first available value 0. Forward checking does nothing. We now reach variable v_5 again so FC1 will forward check each of v_5's values against v_6. None of its values results in the domain wipe-out of v_6. Therefore, there is no dead-end and the consistent strategy shown in Figure 6 will be found.

Note that if we had not added the variables in $conf_set(v_5)$ to $conf_set(v_3)$, we would backjump to v_1 when encountering the dead-end at v_3. This would result in a different solution being found, with more search effort.

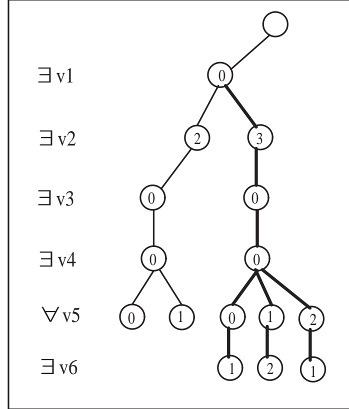

Figure 6: The search tree explored by FC1 with CBJ on the problem of Example 7. The consistent strategy found is depicted with bold lines.

5.2.2 Solution-Directed Pruning

As discussed in Section 4.3, Giunchiglia et al. introduced solution-directed backjumping for QBF [33]. This allows backjumps over universally quantified literals once reaching a true leaf node. Inspired by this idea, we have implemented a technique that can prune values from universal variables when reaching a true leaf node and may also perform solution-directed backjumps. We call this *solution-directed pruning* (SDP). SDP is based on the following idea.

Assume that v_i is the last universal in V and $q = (v_{i+1} \dots v_n)$ is the sequence of existentials after v_i in V. Also, assume that a consistent scenario including assignment $v_i \mapsto a_i$ has been found and $\langle v_{i+1} \mapsto a_{i+1}, \dots, v_n \mapsto a_n \rangle$ are the assignments of the existential variables $(v_{i+1} \dots v_n)$ in this scenario. Then any value of v_i that is compatible with all these assignments will, obviously, also be part of a consistent scenario. Therefore, for each such value we can avoid running a search in the remaining existentials (i.e. any such value can be pruned). Based on this, after reaching a true leaf node, SDP first computes the values of the last universal v_i in V that have the above property. All such values are temporarily pruned from $D(v_i)$. If there are no available values in $D(v_i)$, SDP proceeds with the universal immediately before v_i in V, say v_j.

SDP then checks if v_j's remaining values are compatible with the assignments of all existentials after v_j. Each such value is pruned from $D(v_j)$, under the condition that **all** values of $D(v_i)$, after the first one, were previously pruned by SDP. Or in other words, if all values of v_i were found to be compatible with the same set of assignments $\langle v_{i+1} \mapsto a_{i+1}, \dots, v_n \mapsto a_n \rangle$ for the existentials after v_i. Essentially this means that to prune a value from $D(v_j)$ it must be compatible with all the assignments in the previously discovered strategy for setting the variables after v_j. This is repeated recursively until a universal is found which has available values left in its domain after SDP has been applied. The algorithm then backjumps to this universal. Example 8 illustrates how SDP operates.

Example 8 Consider the QCSP $\forall v_1 \exists v_2 \forall v_3 \exists v_4 \exists v_5\ (C)$. Assume that all variables have the domain $\{0, 1, 2\}$ except v_1 whose domain is $\{0, 1, 2, 3\}$. C includes some constraints which we don't mention for simplicity reasons. Imagine that BT, coupled with SDP, is used to solve the problem. Figure 7 depicts a solution to the problem, and the nodes pruned by SDP together with the subtrees that are not searched.

Assume that the consistent scenario $\langle v_1 \mapsto 0, v_2 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1, v_5 \mapsto 2\rangle$ has been discovered. The algorithm will now backtrack to the last universal (i.e. v_3) and apply SDP. Assuming that values 1 and 2 of v_3 are compatible with assignments $v_4 \mapsto 1$, $v_5 \mapsto 2$, SDP will prune values 1 and 2 from $D(v_3)$ and, thus, avoid searching the subtrees below the corresponding nodes. Since there are no values left in $D(v_3)$, the algorithm will apply SDP to the previous universal (i.e. v_1). Assuming that value 1 of v_1 is compatible with assignments $v_2 \mapsto 1, v_4 \mapsto 1, v_5 \mapsto 2$, SDP will prune value 1 from $D(v_1)$. According to the definition of SDP the pruning is possible because **all** values of $D(v_3)$, after the first one, were pruned by SDP previously. That is, assignment $v_1 \mapsto 1$ is compatible with all the assignments in the previously discovered strategy for setting the variables after v_1.

Now assuming value 2 of v_1 is not compatible with assignments $v_2 \mapsto 1$, $v_4 \mapsto 1$, $v_5 \mapsto 2$, the algorithm will backjump to v_1 and proceed by making the assignment $v_1 \mapsto 2$. As shown in Figure 7, the algorithm will then find consistent scenarios for values 0 and 1 of v_3, while SDP will prune value 2 of v_3 because it is compatible with assignments $v_4 \mapsto 1, v_5 \mapsto 1$. Since there are no more values in $D(v_3)$, the algorithm will apply SDP to v_1. However, value 3 of v_1 cannot be pruned because not all of v_3's values, after the first one, had been previously pruned by SDP. Therefore, the algorithm will proceed as usual to explore the subtree below the node corresponding to assignment $v_1 \mapsto 3$. Note that SDP is not able to detect that the subtrees below the nodes corresponding to $v_1 \mapsto 2$ and $v_1 \mapsto 3$ are similar because it only uses information about the most recently discovered consistent scenario.

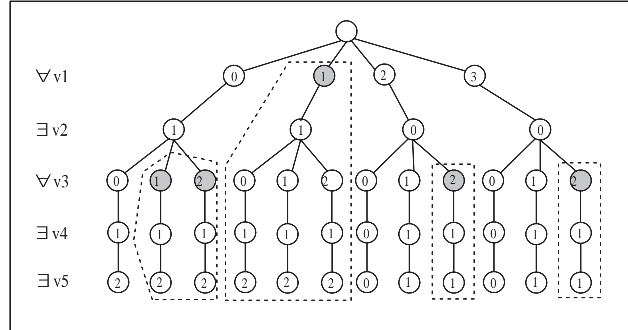

Figure 7: A solution to the problem of Example 8. Dark nodes are pruned by SDP and the subtrees below them (enclosed in dotted areas) are not searched.

The way SDP operates, illustrated in Example 8, immediately suggests possible enhancements. For example, an algorithm that stores a history of consistent scenarios

discovered earlier, as opposed to only the last one, may be able to perform more pruning than SDP, albeit with greater spatial requirements. We plan to investigate such learning techniques in the future.

5.3 The Algorithm of QCSP-Solve

A high level description of QCSP-Solve's algorithm is shown in Figure 8. It takes a QCSP $F = \langle V, Q, D, C \rangle$ and determines whether the problem is true or false. The version of QCSP-Solve shown in Figure 8 is based on FC. In Figure 8,

- $preprocess()$ is a function that preprocesses the problem by applying algorithm QAC-2001, and computing pure and NI values.
- *compute_PV* computes the pure values of v_{cur} during search. If v_{cur} is existential and one of its values (say a) is pure then *compute_PV* assigns v_{cur} with a and temporarily removes the rest of $D(v_{cur})$'s values. If v_{cur} is universal then *compute_PV* temporarily removes all the pure values, except the last one, from $D(v_{cur})$. The **if** statement of line 4 ensures that *compute_PV* is only called before v_{cur} is assigned with its first available value.
- *Forward_Check0* is the function of Figure 4 and implements the FC0-type look-ahead. It is called after the current variable (existential or universal) is assigned and checks this assignment against all future existentials constrained with v_{cur}. If a value of a variable v_i is deleted then v_{cur} is added to $conf_set(v_i)$. If $D(v_i)$ is wiped out then each $v_j \in conf_set(v_i)$ is added to $conf_set(v_{cur})$.
- *Forward_Check1* is the function of Figure 5 and implements the FC1-type look-ahead. It is called before v_{cur} is assigned (if it is a universal) and checks all of $D(v_{cur})$'s available values against the future variables constrained with v_{cur}. If the domain of a variable v_i is wiped out then each $v_j \in conf_set(v_i)$ is added to $conf_set(v_{cur})$. The **if** statement of line 30 ensures that $fc1()$ is called only before v_{cur} is assigned with its first available value.
- *SDP* implements solution-directed pruning. *SDP* prunes values from universals according to the reasoning described in Section 5.2 and returns the first universal found that has values left in its domain after SDP has been applied.
- *Restore* is the procedure depicted in Figure 9 used to restore values to the domains of variables upon backtracks. This procedure is slightly different from the one used by FC as it has to restore any values pruned by the PV rule in addition to the ones pruned by forward checking.

QCSP-Solve works as follows. It takes as input a QCSP $F = \langle V, Q, D, C \rangle$ and, after preprocessing the problem (line 1), it proceeds by making assignments of values to variables until the truth of the problem is proved or disproved. Before assigning a value to v_{cur}, QCSP-Solve calls *compute_PV* to compute the pure values of v_{cur} (lines 4–5). If v_{cur} is existential and there are no available values in $D(v_{cur})$ then the algorithm backtracks to the latest variable in V belonging to $conf_set(v_{cur})$ (lines

Boolean **QCSP-Solve**$(F = \langle V, Q, D, C \rangle)$
input: A QCSP F
output: TRUE if a solution to F exists and FALSE otherwise

1: *preprocess*(F)
2: $v_{cur} \leftarrow v_1$
3: **while** $v_{cur} \neq NIL$
4:   **if** the previously assigned variable was v_{cur-1}
5:     *compute_PV*(F, v_{cur})
6:   **if** $Q(v_{cur}) = \exists$
7:     **if** all values in $D(v_{cur})$ have been tried
8:       $v_{back} \leftarrow$ latest variable in V belonging to $conf_set(v_{cur})$
9:       *Restore*(F, v_{cur}, v_{back})
10:       $v_{cur} \leftarrow v_{back}$
11:     **else**
12:       assign v_{cur} with the next available value $a \in D(v_{cur})$
13:       **if** *Forward_Check0*(F, v_{cur}, a)
14:         **if** $v_{cur} = v_n$
15:           **if** there are no universals in V **return** TRUE
16:           **else**
17:             $v_{back} \leftarrow$ *SDP*(F)
18:             *Restore*(F, v_{cur}, v_{back})
19:             $v_{cur} \leftarrow v_{back}$
20:         **else** $v_{cur} \leftarrow$ *next*(v_{cur})
21:       **else** *Restore*(F, v_{cur}, v_{cur})
22:   **else** //$Q(v_{cur}) = \forall$//
23:     **if** all values in $D(v_{cur})$ have been tried
24:       **if** v_{cur} is the first universal in V **return** TRUE
25:       **else**
26:         *Restore*$(F, v_{cur},$*last_u*$(v_{cur}))$
27:         $v_{cur} \leftarrow$ *last_u*(v_{cur})
28:     **else**
29:       FC_result ←TRUE
30:       **if** the previously assigned variable was v_{cur-1}
31:         FC_result ←*Forward_Check1*(F, v_{cur})
32:       **if** FC_result
33:         assign v_{cur} with the next available value $a \in D(v_{cur})$
34:         *Forward_Check0*(F, v_{cur}, a)
35:         $v_{cur} \leftarrow$ *next*(v_{cur})
36:       **else**
37:         $v_{back} \leftarrow$ latest variable in V belonging to $conf_set(v_{cur})$
38:         *Restore*(F, v_{cur}, v_{back})
39:         $v_{cur} \leftarrow v_{back}$
40: **if** $v_{cur} = NIL$ **return** FALSE

Figure 8: The algorithm of QCSP-Solve.

procedure *Restore*(F, v_{cur}, v_{back})
input: A QCSP F, the current variable v_{cur} and the variable where the algorithm will backtrack v_{back}
output: -
1: **for** $v_i = v_{back}$ to v_{cur}
2: **for** each variable v_j after v_i in V
3: restore to $D(v_j)$ any value that was removed because of v_i's instantiation
4: restore to $D(v_j)$ any value that was removed because of the PV rule

Figure 9: Restoration procedure of QCSP-Solve.

7–10). Otherwise, v_{cur} is assigned with its next available value and the assignment is checked against future variables (lines 12–13). If there is no domain wipe-out and the algorithm has reached a true leaf node (i.e. v_{cur} is the last variable in V) then *SDP* is called to perform solution-directed pruning (lines 16–19). If QCSP-Solve is not at a leaf node, it proceeds by moving to the next variable (line 20). If there is a domain wipe-out, the next value of v_{cur} will be tried in the next iteration of the **while** loop. Note that if there are no universals in the problem (i.e. it is a standard CSP), QCSP-Solve terminates when a true leaf node is reached (line 15).

If v_{cur} is a universal and consistent scenarios have been found for all of its values, then there are two cases. If v_{cur} is the first universal, QCSP-Solve terminates successfully (line 24). Otherwise, it backtracks to the last universal (line 27). Before assigning any value to a universal variable, QCSP-Solve calls *Forward_Check1* to perform the FC1-type look-ahead (lines 30–31). If there is a domain wipe-out, the algorithm backtracks to the latest variable in V belonging to $conf_set(v_{cur})$ (lines 36–39). If there is no domain wipe-out, or *Forward_Check1* has already been called at this level, v_{cur} is assigned with its next available value (line 33), the assignment is checked against future variables (line 34), and QCSP-Solve proceeds with the next variable (line 35).

Although it is not shown in Figure 8, QCSP-Solve can also employ the dynamic symmetry-breaking technique based on computing NI values, described in Section 3.3. However, the experiments we have run so far have showed that the time overheads of this technique outweigh the benefits it offers, when the PV rule is also used. That is why it is not included in the pseudo-code of Figure 8. However, in other problems than the ones we tried, and with better implementation, it is quite possible that dynamic NI-based symmetry breaking may be useful.

The following example demonstrates how QCSP-Solve operates.

Example 9 Consider the following QCSP where V consists of 7 quantified variables, and C is a conjunction of 9 constraints. $\exists v_1 \exists v_2 \forall v_3 \forall v_4 \forall v_5 \exists v_6 \exists v_7 (v_1 \neq v_6 \wedge v_1 \neq v_7 \wedge v_2 \neq v_6 \wedge v_3 \neq v_6 \wedge v_3 < v_7 \wedge v_4 \neq v_6 \wedge v_4 \neq v_7 \wedge v_5 \neq v_6 \wedge v_5 < v_7)$. Assume that the domains of the variables are as follows: $D(v_1) = \{2, 3\}$, $D(v_2) = \{0, 1, 2\}$, $D(v_3) = \{0, 3\}$, $D(v_4) = \{0, 1, 6\}$, $D(v_5) = \{4, 5\}$, $D(v_6) = \{0, 1, 2, 3\}$, $D(v_7) = \{0, 2, 3, 6\}$.

Let us trace the execution of QCSP-Solve for a few steps to understand how its various features prune the search space. Figures 10a to 10k demonstrate how the search tree explored by QCSP-Solve is built, how certain nodes are pruned, and the way the domains of the variables change during search.

Figure 10a Preprocessing is applied (line 1 of the algorithm). There are no arc inconsistent or pure values, so no pruning is performed[3].

Figure 10b The assignment $v_1 \mapsto 2$ is made (line 12). *Forward_Check0* reduces $D(v_6)$ and $D(v_7)$ to $\{0, 1, 3\}$ and $\{0, 3, 6\}$ respectively (line 13). We now have the following: $conf_set(v_6) = conf_set(v_7) = \{v_1\}$.

Figure 10c Now, value 2 of v_2 becomes pure because it is supported by all values in future variables (lines 4–5). The PV rule will immediately make the assignment $v_2 \mapsto 2$.

Figure 10d The next variable is a universal. *Forward_Check1* (lines 30–31) does not wipe out any future domain, so the assignment $v_3 \mapsto 0$ is made (line 33). *Forward_Check0* reduces $D(v_6)$ and $D(v_7)$ to $\{1, 3\}$ and $\{3, 6\}$ respectively (line 34).

Figure 10e Value 0 of v_4 is pure (lines 4–5). Therefore, it is removed. *Forward_Check1* (lines 30–31) does not wipe out any future domain, so the assignment $v_4 \mapsto 1$ is made (line 33) and *Forward_Check0* reduces $D(v_6)$ to $\{3\}$ (line 34).

Figure 10f The next variable is v_5. *Forward_Check1* does not wipe out any future domain (lines 30–31), so the assignment $v_5 \mapsto 4$ will be made (line 33). *Forward_Check0* reduces $D(v_7)$ to $\{6\}$ (line 34).

Figure 10g v_6 and v_7 are assigned their only available values (in line 12) and a true leaf node is found (line 14).

Figure 10h Now function *SDP* is called (line 17). *SDP* discovers that value 5 of the last universal (v_5) is compatible with the assignments of all the existentials after v_5. Therefore, this value is removed from $D(v_5)$. *SDP* is then applied to the previous universal v_4. Value 6 of v_4 is not compatible with the assignments to v_6 and v_7. Therefore, a solution-directed backjump to v_4 is performed (line 19).

Figure 10i The assignment $v_4 \mapsto 6$ is made (line 33). *Forward_Check0* reduces $D(v_6)$ and $D(v_7)$ to $\{1, 3\}$ and $\{3\}$ respectively (line 34).

Figure 10j *Forward_Check1* (lines 30–31) applied at v_5 wipes out $D(v_7)$ because value 4 of v_5 is incompatible with the only value in $D(v_7)$. Therefore, we have a dead-end and $conf_set(v_7)$ will be added to $conf_set(v_5)$.

Figure 10k The algorithm will backjump to the latest variable in V belonging to $conf_set(v_5)$, which is v_1 (line 37–39).

Figure 10l shows the part of the search tree traced in the example and illustrates how subtrees are pruned by applying look-ahead and look-back techniques.

[3] Values 4 and 5 of v_5 are NI, but let us ignore this for the sake of the example.

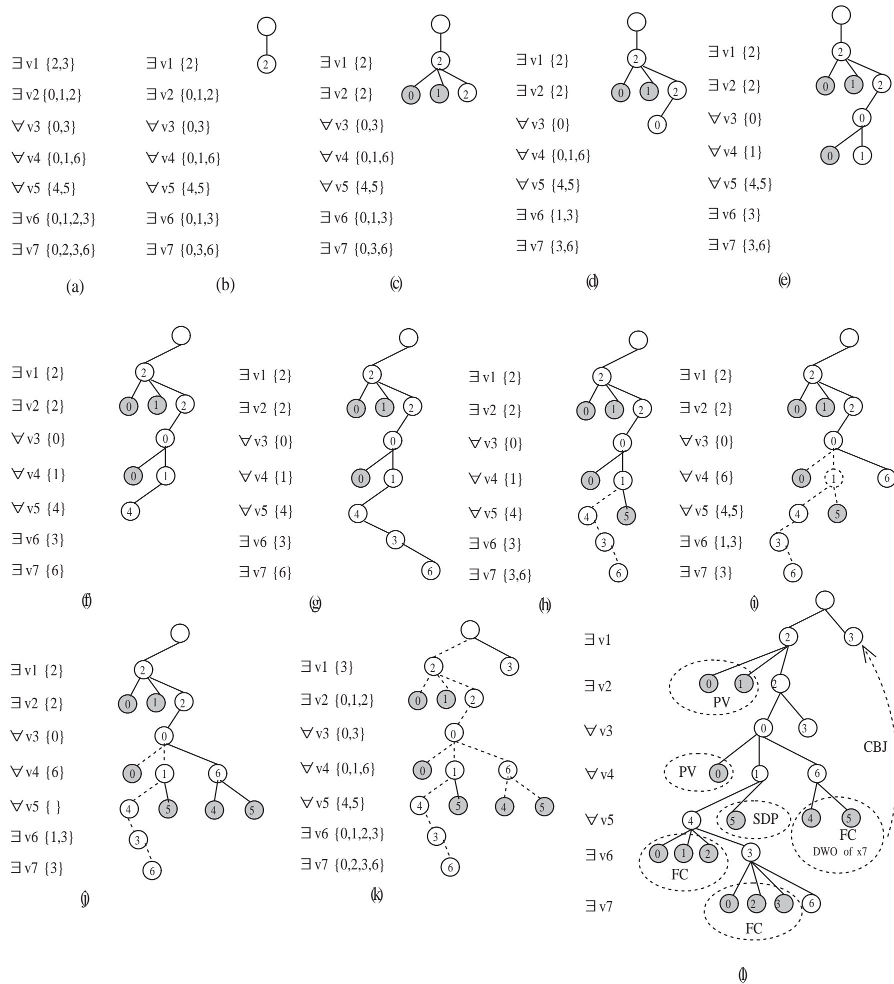

Figure 10: Search tree of Example 9. Dark nodes are pruned by QCSP-Solve. Such nodes together with the feature responsible for their pruning are included in dotted ovals in the last figure. Dotted edges denote parts of the tree that were visited in the past. DWO stands for domain wipe-out.

6 Experimental Evaluation

To compare the performance of the methods presented in the previous sections, we ran experiments on randomly generated QCSPs. Before presenting the results, we discuss the issue of flaws in random instances, which is familiar from other search problems such as CSP and QBF and can have a significant impact on experimental studies. We show that random generators derived by extending standard generators for QBF and CSP give rise to flaws, which quickly infect all generated problems. Since this is an important problem for experiments in QCSP, we propose a random generator that is free from these flaws.

6.1 Flaws in Random QCSP Generation

Local flaws have been discovered in random generation models for search problems, such as CSPs [1] and QBF [30]. We show that random generation models for QCSPs that are based on standard generators for QBF and CSPs can suffer from a local flaw (specific to QCSPs) that makes almost all of the generated instances false, even for small problem sizes.

Consider, for example, the k-QBF random generation model [30] which has been widely used in experiments with QBF. In this model there are $k+1$ blocks of variables with alternating quantification, with the variables in the first block being existentially quantified. For example, in a 2-QBF problem we have a block of existential variables followed by a block of universal variables followed by another block of existential variables. This model can be easily adapted to generate QCSP instances. The blocks of variables are generated as in k-QBF and the binary constraints can be generated using a standard model for binary CSPs (e.g. model B [43]).

However, the k-QCSP generator is subject to a local flaw. Suppose we can find assignments $\langle v_1 \mapsto 7, v_2 \mapsto 2, \ldots, v_k \mapsto 3\rangle$ for universals $v_1, \ldots, v_k$, and there exists an existential v_e appearing later in V than all variables v_i where $i \in 1 \ldots k$. If every value of v_e conflicts with one of the chosen values of one of the universals, this tuple of assignments is inconsistent. But it remains inconsistent irrespective of assignments to other universals or existentials and so the problem is trivially false as a whole. Even taking the extreme case of only one conflict per constraint, this can happen as long as there are as many universals before v_e as values in its domain.

Assume there are d universals, and there is a constraint between a pair of variables with probability p. Each constraint has one nogood. Assume that $D(v_e) = \{1, \ldots d\}$. The probability of a conflict between some universal and value 1 in $D(v_e)$ is pd/d (i.e. we pick a universal u from d universals, with probability p there is a constraint between u and v_e, and with probability $1/d$ the single conflict involves value $1 \in D(v_e)$). For value 2, the set of available universals has size $d-1$, so the probability is $p(d-1)/d$. Overall the probability of a flaw between existential v_e and d particular universals is $P(v_e) = \prod_{i=1}^{d} pi/d$.

With k existential variables quantified after d universals, the probability of no flaw occurring is $(1 - P(v_e))^k$. Since $P(v_e)$ does not depend on k, with fixed d and p this probability tends to 0 as $k \to \infty$. Not only are flaws certain to occur, but there is no phase transition: i.e. for any $p > 0$ almost all problems are false asymptotically.

This flaw is not only very common, but also discovering its presence is an **NP**-complete problem. Let us repeat the description of the flaw slightly more formally.

Definition 6 Flawed Problem Suppose that we have a set of variable assignments $S = \{v_1 \mapsto a_1, \ldots, v_i \mapsto a_i\}$, where for each a_j, $j = 1 \ldots i$, $a_j \in D(v_j)$, and each v_j, $j = 1 \ldots i$, is universally quantified and appears in V before an existential variable v_e. If every value $b \in D(v_e)$ is incompatible with *at least one* value assignment $v_j \mapsto a_j \in S$ ($\langle a_j, b\rangle \notin c_{je}$), then the entire problem is false, and is said to be flawed.

Notice that the only case of this which is detected by either a QCSP technique or encoding is the case where the set of variable-value pairs is a singleton. I.e. some $v_j \mapsto a_j$ is inconsistent with every value of v_e. In this case algorithm QAC-2001 reports failure. For the case where each constraint contains only one conflict, it is easy to check each existential variable for a flaw. That is how we computed the proportion of flawed problems above. However, in general it is hard to confirm the existence of a flaw:

Theorem 2 Checking for the presence of a flaw in a QCSP is NP-complete.

Proof: Consider any SAT instance. We convert this into a QCSP such that the SAT instance has a solution iff the QCSP is flawed. For each SAT variable v we have a corresponding universal QCSP variable v with two values, 0 and 1. We have a single existential variable v_e quantified last, with domain size equal to the number of clauses in the SAT problem. We have a constraint between every universal variable and v_e. This constraint has a conflict for each SAT clause that the variable occurs in. If the literal in clause i is $\neg v$, the conflict rules out the pair $\langle v \mapsto 0, v_e \mapsto i\rangle$, while if the literal is v, the conflict rules out $\langle v \mapsto 1, v_e \mapsto i\rangle$.

Now consider any satisfying assignment to the SAT instance. This is a set of literals such that at least one occurs in each clause. Say $\neg v$ is in the set and occurs in clause i. Then the translation ensures that $v \mapsto 0$ rules out $v_e \mapsto i$. Similarly, if u is in the assignment and occurs in clause j, then $u \mapsto 1$ is in conflict with $v_e \mapsto j$. So each value of v_e is ruled out. We never need to set any variable to 0 and 1 simultaneously, as the satisfying assignment does not contain both a variable and its negation.

The reverse direction is similar. Say that the translated QCSP is flawed. Then there is a set of assignments $\{v_i \mapsto a_i\}$ ruling out each value of v_e. If $a_i = 0$ then, by construction, the literal $\neg v$ occurs in clause i and satisfies it. And if $a_i = 1$ then v occurs in clause i. As all values of v_e are ruled out, the SAT instance is satisfied.

The flaw is easily witnessed, by a choice of values for universal variables, so the problem of instances being flawed is also in NP so is NP-complete. *QED*

Note that the flaw is simply a situation in which search can be terminated. As such it might give rise to interesting new propagation techniques in QCSP, or valuable new clauses in QBF encodings.

6.1.1 Random problem generator

The random generator we used controls the probability of flaws. Variables appear in blocks with alternating quantification. For simplicity, we describe the model in the

case of three blocks. That is, a block of existentials followed by a block of universals then another block of existentials. The generator takes 7 parameters: $\langle n, n_\forall, n_{pos}, d, p, q_{\forall\exists}, q_{\exists\exists}\rangle$ where n is the total number of variables, $n_\forall$ is the number of universally quantified variables, n_{pos} is the position of the first universally quantified variable in V, d is the uniform domain size, p is the number of binary constraints as a fraction of all possible constraints.

$q_{\exists\exists}$ is the number of *goods* (i.e. satisfying tuples) in $\exists\, v_i \exists\, v_j\ (c_{ij})$ constraints as a fraction of all possible tuples, and $q_{\forall\exists}$ is a similar quantity for $\forall\, v_i \exists\, v_j\ (c_{ij})$ constraints explained below. The other two types of binary constraint can be removed entirely by preprocessing and so we do not generate them.

Since the flaw is a characteristic of $\forall\, v_i \exists\, v_j\ (c_{ij})$ constraints, we restrict these in the following way: we generate a random total bijection (i.e. a one-to-one correspondence) from one domain $D(v_i)$ to the other $D(v_j)$. Conflicts are chosen only from those pairs in the bijection. All 2-tuples not in the bijection are goods. Now $q_{\forall\exists}$ is the fraction of goods from the d tuples in the bijection.

Notice that the p, $q_{\exists\exists}$, $q_{\forall\exists}$, and $q_{\forall\exists}$ parameters are proportions rather than probabilities, hence this model is similar in style to model B for random CSPs.

To control the probability p_f of the flaw, we write down an expression for p_f, approximating proportions $p, q_{\forall\exists}, q_{\exists\exists}$ as probabilities. $n_\forall$ is the number of universal variables, and $n_\exists$ is the number of existential variables in the second existential block.

For each existential assignment $v_e \mapsto 1$, the probability that it is covered by a universal v_u is $p(1 - q_{\forall\exists})$. If the variable v_e is flawed, then all its values are in conflict with some value of some universal variable. However, each universal variable can only cover one value (since we use a bijection).

For an individual existential variable v_e (in the second existential block), and representing domain values using positive integers, we start by writing down the following equation. It places an ordering on the values and represents the probability of all values in $D(v_e)$ being flawed as a product of the probabilities of each value a, given that all values less than a are flawed. So for example, if $a = 5$, the probability that value 5 is flawed (given values $1, 2, 3, 4$ are flawed) is written as $p(5|1, 2, 3, 4)$.

$$p(v_e\, flaw) = p(1)p(2|1)p(3|1, 2)\dots \quad (1)$$

The probability that value a is flawed, given that the previous $a - 1$ values are flawed, is given by equation (2). $1 - q_{\forall\exists}$ is the probability of the particular value $a \in D(v_e)$ being in a nogood of any particular constraint. This is multiplied by p_1 to obtain an approximate probability of a particular universal v_g and constraint c_{ge} having a nogood containing $a \in D(v_e)$.

The exponent $n_\forall - (a-1)$ is the number of universal variables, minus those $(a-1)$ variables which are already instantiated to conflict with the $(a - 1)$ lower values in $D(v_e)$. The probability $p_1(1 - q_{\forall\exists})$ of a particular universal having a conflict with $a \in D(v_e)$ is complemented, raised to the exponent and complemented again to obtain the probability of any remaining universal variable having a conflict with $a \in D(v_e)$.

$$p(a|1 \dots a - 1) = 1 - (1 - p_1(1 - q_{\forall\exists}))^{n_\forall - (a-1)} \quad (2)$$

Substituting equation (2) into equation (1) gives the probability of one particular existential variable being flawed.

$$p(v_e\, flaw) = \prod_{i=0}^{d-1} \left(1 - (1 - p_1(1 - q_{\forall\exists}))^{n_\forall - i}\right) \tag{3}$$

The probability that no existential variables are flawed is given below. This formula gives incorrect results when $d > n_\forall$. In this case, $p_f = 1$ since there are not enough universal variables to cover all elements of a domain.

$$p_f = (1 - p(v_e\, flaw))^{n_\exists} \tag{4}$$

6.2 Experimental Results

In this section we present experimental results from problems generated using the model described above. Our aim is to demonstrate the huge progress in the efficiency of QCSP solving that was made, starting from our first methods and culminating in the most advanced ones. Therefore, we only give indicative results for the various techniques.

6.2.1 Direct Algorithms

Figure 11 presents a comparison of algorithms FC1, FC1+PV, MAC1+PV, and QCSP-Solve on problems generated according to the model described above. All algorithms apply AC, and NI preprocessing. For each value of $q_{\exists\exists}$ shown in the figures, 100 problem instances were generated and we use the mean average. The generation parameters are $n = 21$, $d = 8$, $p = 0.2$, and $q_{\forall\exists} = 0.5$. Variables $v_1 \ldots v_7$ are existentials, $v_8 \ldots v_{14}$ are universals, and $v_{15} \ldots v_{21}$ are existentials. These parameters ensure that the instances are unflawed. Finally, $q_{\exists\exists}$ is varied across the satisfiability phase transition. We include FC1+PV and MAC1+PV in the comparison to illustrate the power of the PV rule. Note that, for the problems we tried, the FC-based algorithms are more efficient than the corresponding MAC-based ones. However, for larger problems this may easily be reversed.

In the problems of Figure 11 the execution of FC1 was stopped at the cut-off limit of 2 hours in more than 50% of the instances. As we can see, QCSP-Solve is many orders of magnitude faster than FC1. The speed-up obtained is largely due to the application of the PV rule. Similar results were obtained with various parameter settings.

At this point we should note that both the recently proposed QCSP solvers BlockSolve and QeCode, of [44] and [4] respectively, achieved very good results on randomly generated QCSPs. Both these solvers are considerably different than QCSP-Solve. BlockSolve is a bottom-up solver that displays better performance than QCSP-Solve on satisfiable instances, but as a downside requires exponential space. QeCode is built on top of Gecode and hence is equipped with many advanced CSP techniques such as GAC algorithms for certain global constraints. On the other hand, it lacks specialized features for QCSPs, such as pure value handling.

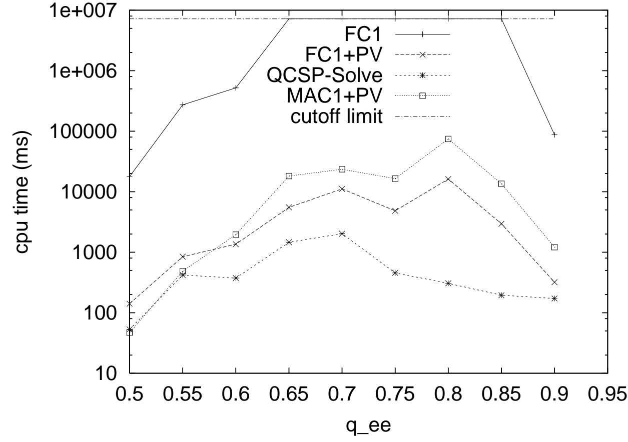

Figure 11: Comparison of direct algorithms for QCSPs. $n = 21$, $n_\forall = 7$, $d = 8$, $p = 0.20$, $q_{\forall\exists} = 1/2$.

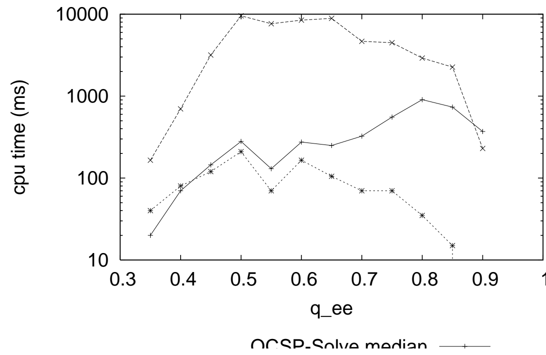

Figure 12: Comparison of the enhanced log encoding with adapted log and QCSP-Solve

6.2.2 Encodings of QCSP as QBF

As explained in Section 4, the global and local acceptability encodings perform poorly compared to the other encodings and the direct methods. Therefore, we do not include results for these two encodings. The enhanced log encoding gives a remarkable improvement over adapted log. It is also competitive with QCSP-Solve and can be two orders of magnitude better. Figure 12 shows results using the three methods. The generation parameters are $n = 24$, $d = 9$, $p = 0.2$, and $q_{\forall\exists} = 0.5$. Variables $v_1 \ldots v_8$ are existentials, $v_9 \ldots v_{16}$ are universals, and $v_{17} \ldots v_{24}$ are existentials. $q_{\exists\exists}$ is varied across the satisfiability phase transition. For each point, 100 instances were generated. The median average is used because of high outliers. The time taken to encode the instances is not included, but since it is a linear encoding this is negligible for difficult instances.

The closest setting of $q_{\exists\exists}$ to the phase transition is 0.55, with 43 instances out of

100 being true. This also approximately coincides with the difficulty peaks for the encodings, but not for QCSP-Solve. For lower values of $q_{\exists\exists}$ fewer of the instances are true and the enhanced log encoding is less competitive with QCSP-Solve. For example at $q_{\exists\exists} = 0.35$, 99 of the instances are false, and QCSP-Solve outperforms the enhanced log encoding. Where $q_{\exists\exists} = 0.8$ all the instances are true and the enhanced log encoding outperforms QCSP-Solve. At $q_{\exists\exists} = 0.9$, the median for the enhanced log encoding fell below the resolution of the timer, so it is not shown on the graph.

This suggests that the QBF solver CSBJ is more effective in pruning or backjumping over universal variables, because in a loosely-constrained instance the main cost is branching on universals. Testing this, and identifying which rules in CSBJ are responsible, remains for future work.

We very briefly experimented with two other solvers, the resolution solver Quantor, and the hybrid (search and resolution) solver sKizzo. The aim was to gather some initial evidence as to whether the enhanced log encoding is as efficient with non search-based solvers as it is with search-based ones like CSBJ. The random instances run had the following parameters: $n = 24$, 3 blocks of 8 variables with alternating quantification, and $d = 9$, ensuring that the instances are unflawed. $p = 0.2$, $q_{\forall\exists} = 0.5$, and $q_{\exists\exists} = 0.5$ (at the phase transition and difficulty peak for CSBJ). 10 instances were generated. CSBJ solved 9 instances in under half a second each, and the tenth in 8.98 seconds. Quantor quickly ran out of memory (>1GB) on eight of the instances, solved one in 2.98s and ran out of time for the other (>60s). sKizzo was unable to solve any instance within 60s. From this we conjecture that search-based solvers are preferable for the encoding. However, further experiments are necessary to validate this conjecture.

7 Conclusions

In this paper we studied various methods for solving QCSPs with finite discrete non-Boolean domains. Our first approach was based on adapting techniques from CSPs to deal with QCSPs. We described an AC algorithm for QCSPs that can deal with arbitrary binary constraints. We then extended the BT, FC, and MAC algorithms so that they can handle quantification. We also proposed modifications of FC and MAC that are better suited to QCSPs.

Our second approach was based on encoding QCSPs as QBFs. Our motivation was that at an early stage of research into a new problem like QCSP, encoding into a more studied problem like QBF would very likely provide competitive performance. We introduced progressively more efficient encodings, culminating in the enhanced log encoding, which can be several orders of magnitude faster than the direct QCSP algorithms. Through this study it was also demonstrated that the effective encoding of QCSP into QBF can be a complex process, since simple generalizations of CSP-to-SAT encodings are very inefficient.

Apart from giving us efficient tools for QCSP solving, the performance and properties of encodings and techniques used in QBF solving indicated significant enhancements to the direct QCSP algorithms. We identified two features of the log encodings and the underlying QBF solver as largely responsible for their success; first, their ability to take advantage of the pure literal rule in QBF, and second, their backjumping

capabilities, manifested by CBJ and SBJ. We devised and implemented analogues of these features within direct QCSP algorithms, resulting in QCSP-Solve, an efficient direct solver.

Finally, we proposed a model for the random generation of QCSPs that is free from known flaws. Experiments with problems generated using this model demonstrated the dramatic improvement in performance when comparing our initial QCSP solving attempts to the sophisticated techniques developed later.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and suggestions that helped greatly improve the presentation of this paper.

References

[1] D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: A more accurate picture. In *Proc. CP97*, pages 107–120. Springer, 1997.

[2] A. Beckwith and B. Choueiry. On the dynamic detection of interchangeability in finite constraint satisfaction problems. In *Proceedings of CP-2001*, page 760, 2001.

[3] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Proc. of 20th International Conference on Automated Deduction (CADE05)*, 2005.

[4] M. Benedetti, A. Lallouet, and J. Vautard. Reusing CSP propagators for QCSPs. In *Proceedings of CSCLP-2006*, 2006.

[5] F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proceedings of CP-2000*, pages 67–82, 2000.

[6] C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[7] A. Biere. Resolve and expand. In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542, pages 59–70, 2005.

[8] F. Boerner, A. Bulatov, P. Jeavons, and A. Krokhin. Quantified Constraints: Algorithms and Complexity. In *Proceedings of CSL-2003*, pages 244–258, 2003.

[9] L. Bordeaux. Boolean and Interval Propagation for Quantified Constraints. In *Proceedings of the CP-2005 Workshop on Quantification in Constraint Programming*, 2005.

[10] L. Bordeaux, M. Cadoli, and T. Mancini. Exploiting Fixable, Removable, and Implied Values in Constraint Satisfaction Problems. In *Proceedings of LPAR-2004*, pages 270–284, 2004.

[11] L. Bordeaux, M. Cadoli, and T. Mancini. CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of AAAI-2005*, pages 360–365, 2005.

[12] L. Bordeaux and E. Monfroy. Beyond NP: Arc-consistency for Quantified Constraints. In *Proceedings of CP-2002*, pages 371–386, 2002.

[13] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of AAAI-98*, pages 262–267, 1998.

[14] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[15] H. Chen. The Computational Complexity of Quantified Constraint Satisfaction, Phd Thesis, 1994.

[16] H. Chen. Collapsibility and Consistency in Quantified Constraint Satisfaction. In *Proceedings of AAAI-04*, pages 155–160, 2004.

[17] H. Chen. Quantified Constraint Satisfaction and Bounded Treewidth. In *Proceedings of ECAI-04*, pages 161–165, 2004.

[18] N. Creignou, S. Khanna, and M. Sudan, editors. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. SIAM Monographs in Discrete Mathematis and Applications, 2001.

[19] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[20] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):201–215, 1960.

[21] R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997.

[22] A. Tacchella E. Giunchiglia, M. Narizzano. Learning for quantified boolean logic satisfiability. In *Proc. 18th National Conference on Artificial Intelligence (AAAI - 02)*.

[23] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *Proceedings of AAAI-2000*, pages 417–422, 2000.

[24] E. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of AAAI-91*, pages 227–233, 1991.

[25] A. Frisch and T.J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *Proc. IJCAI-01*, pages 282–288, 2001.

[26] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[27] I. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tacchella. Watched data structures for qbf solvers. In *Proceedings SAT 2003*, pages 25–36. Springer, 2003.

[28] I. Gent, P. Nightingale, and A. Rowley. Encoding Quantified CSPs as Quantified Boolean Formulae. In *Proceedings of ECAI-2004*, pages 176–180, 2004.

[29] I. Gent, P. Nightingale, and K. Stergiou. QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems. In *Proceedings of IJCAI-2005*, 2005.

[30] I. Gent and T. Walsh. Beyond NP: The QSAT phase transition. In *Proceedings of AAAI-99*, pages 648–653, 1999.

[31] I.P. Gent, J.-F. Puget, and K.E. Petrie. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.

[32] I.P. Gent and A.G.D. Rowley. Encoding connect-4 using quantified boolean formulae. In *Proceedings of Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 78–93, 2003.

[33] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proceedings of IJCAI-2001*, pages 275–281, 2001.

[34] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Proceedings of AAAI-2001*, pages 649–654, 2002.

[35] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research*, 26:371–415, 2006.

[36] R.M. Haralick and G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[37] A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI-93*, pages 282–287, 1993.

[38] C. Papadimitriou, editor. *Computational Complexity*. Addison Wesley, 1994.

[39] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[40] S. Ratschan. Applications of quantified constraint solving over the reals bibliography, http://www.cs.cas.cz/ ˜ratschan/appqcs.html, 2003.

[41] S. Ratschan. Efficient Solving of Quantified Inequality Constraints over the Reals. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.

[42] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of ECAI-94*, pages 125–129, 1994.

[43] B. Smith. Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings of ECAI-94*, pages 100–104, 1994.

[44] G. Verger and C. Bessière. Blocksolve: a Bottom-Up Approach for Solving Quantified CSPs. In *Proceedings of CP-2006*, pages 635–649. Springer, 2006.

[45] T. Walsh. SAT v CSP. In *Proceedings of CP-2000*, number 1894 in LNCS, pages 441–456. Springer, 2000.