Consistency for Quantified Constraint Satisfaction Problems

Peter Nightingale

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

Abstract. The generalization of the constraint satisfaction problem with universal quantifiers is a challenging PSPACE-complete problem, which is interesting theoretically and also relevant to solving other PSPACE problems arising in AI, such as reasoning with uncertainty, and multi-player games. In this paper I define two new levels of consistency for QCSP, and give an algorithm to enforce consistency for one of these definitions. The algorithm is embedded in backtracking search, and tested empirically. The aims of this work are to increase the facilities available for modelling and to increase the power of constraint propagation for QCSPs. The work is motivated by examples from adversarial games, contrasting the different levels of consistency by their ability to prune unfruitful moves.

1 Introduction

The finite quantified constraint satisfaction problem (QCSP) is a generalization of the finite constraint satisfaction problem (CSP), in which variables may be universally quantified. The CSP is a very successful paradigm for solving many real world problems. The QCSP can be used to model problems containing uncertainty, in the form of variables which have a finite domain but whose value is unknown at solution time. Therefore a QCSP solver finds solutions suitable for each value of these variables.

A QCSP has a quantifier sequence which quantifies (existentially, $\exists$, or universally, $\forall$) each variable in the instance. For each possible value of a universal variable, we find a solution for the later variables in the sequence. Therefore the solution is no longer a sequence of assignments to the variables, but a tree of assignments where the variables are set in quantification order, branching for each value of the universal variables. A route from the root node to a leaf node assigns all variables such that all constraints are satisfied. This is known as a winning strategy, and can be exponential in size. This generalization increases the computational complexity[1]: QCSP is PSPACE-complete rather than NP-complete.

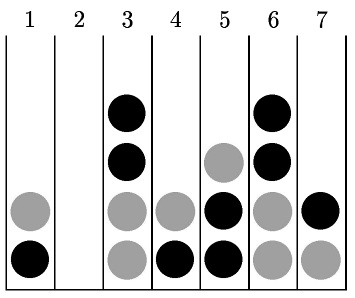
There has been a great deal of work on quantified Boolean formulae (QBF), which is the PSPACE-complete generalization of Boolean satisfiability (SAT).

[1] Under the usual assumption that P$\subsetneq$NP$\subsetneq$PSPACE.

SAT is complementary to CSP, and has better optimized solvers at the cost of a much less expressive language. As a result, many NP-complete problems cannot be sensibly modelled in SAT but can in CSP. QCSP has the potential to have the same advantages over QBF as CSP has over SAT.

The QCSP can be used to model PSPACE-complete problems from areas such as multiplayer games, planning with uncertainty and model checking. For the first two of these, the models can have a similar character, where we need to find successful actions whatever the actions of the opponent or changes in the environment, and our actions are interleaved with opponent actions or changes in the environment. In QCSP, our actions are represented with existential variables and the others are represented with universal variables, and the quantifier sequence interleaves the two. The constraints express that the outcome is successful, i.e. the player wins or the goal is met. Intuitively, this corresponds to the question: Does there exist an action, such that for any eventuality, does there exist a second action, such that for any eventuality, etc, I am successful?

The aim of this paper is to contribute to the tools available to model problems such as these. Bordeaux and Monfroy [3] define arc-consistency for QCSP, and give algorithms for two classes of constraints: QBF (Quantified Boolean Formulae) with quantified (generalized)arc-consistency rules for constraints such as $a = \neg b$ and $a = b \vee c$; and BAF (Bounded Arithmetic Formulae) with interval consistency rules for constraints such as $a = b + c$. For both these schemes, constraints of arity greater than three are broken down. Mamoulis and Stergiou [1] extended arc-consistency for binary constraints from CSP to QCSP. Non-binary constraints can be encoded using an adapted hidden variable encoding (defined below). Unfortunately, when breaking down a longer constraint into binary or ternary constraints, propagation may be lost. In this paper I introduce a general consistency algorithm for quantified constraints of any arity, based on Bessière and Régin's GAC-Schema [5,8].



$\exists grey1 \forall black1 \exists grey2 \forall black2 \exists grey3 : \text{greywins}(grey1, black1, grey2, black2, grey3)$

Fig. 1. Connect-4 endgame

Connect-4 For example consider the Connect-4 endgame in figure 1. The aim of Connect-4 is to make a line of four counters, vertically, horizontally or diagonally. The two players take turns, and can only place a counter at the bottom of a column on the board. It is played on a board with six rows and seven columns. It is grey to move, and it can be seen that columns 2 and 4 are the only moves allowing grey to win in 3 moves if black defends perfectly[2]. The seven such winning sequences are 2-2-4-4-5, 2-2-5-4-4, 4-4-5-2-2, 4-4-5-2-6, 4-4-5-6-2, 4-4-2-2-5 and 4-4-2-5-2. As shown below the figure, this problem can be modelled as a QCSP, with 5 variables representing the column numbers of the 5 moves, with just one 5-ary constraint representing that grey wins. This is similar to a 5-move lookahead constraint, but with the additional restriction that grey must win within the 5 moves. Ideally, the propagation algorithm would be able to restrict all three of the grey move variables. GAC infers nothing. I define two stronger levels of consistency, WQGAC, which infers that $grey1 \in \{2, 4\}$, and the stronger SQGAC, which also infers $grey2 \in \{4, 5\}$, and $grey3 \in \{2, 4, 5, 6\}$. I give an algorithm for WQGAC in section 4. The example is unusual in that global reasoning and the equivalent local reasoning on the greywins constraint are equivalent, since there is only one constraint.

2 Definitions

Definition 1. *Quantified Constraint Satisfaction Problem*

A QCSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q} \rangle$ is defined as a set of n variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, a set of current domains $\mathcal{D} = \{D_1, \ldots, D_n\}$ where D_i is the finite set of all values which x_i can take, a conjunction $\mathcal{C}$ of constraints between variables in $\mathcal{X}$, and a quantifier sequence $\mathcal{Q} = \phi_1 x_1, \ldots, \phi_n x_n$ where ϕ_i is a quantifier, $\exists$ (existential, 'there exists') or $\forall$ (universal, 'for all').

Before defining the semantics of a QCSP, it is necessary to define constraints.

Definition 2. *Constraint*

A constraint $C \in \mathcal{C}$ on the ordered set of variables $\mathcal{X}_C = (x_i, \ldots, x_j)$ has an associated set $C_S \subseteq D_i^0 \times \ldots \times D_j^0$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_C$, where D_i^0 is the initial domain of i.

C_S may be represented implicitly, for example by an algebraic expression.

Definition 3. *QCSP semantics*

If $\mathcal{C}$ is empty, the problem is true. If $\mathcal{Q} = \exists x_1, \phi_2 x_2 \ldots$ then $\mathcal{P}$ is true iff there exists a value $a \in D_1$ such that the problem $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}', \mathcal{Q}' \rangle$ with $\mathcal{Q}' = \phi x_2, \ldots$ and $\mathcal{C}' = \mathcal{C}[x_1 = a]$[3] is true. If $\mathcal{Q} = \forall x_1, \phi_2 x_2 \ldots$ then $\mathcal{P}$ is true iff for all values $a \in D_1$ the problem $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}', \mathcal{Q}' \rangle$ with $\mathcal{Q}' = \phi x_2, \ldots$ and $\mathcal{C}' = \mathcal{C}[x_1 = a]$ is true.

[2] Black chooses, whenever possible, a move which prevents grey from winning immediately afterwards.

[3] Where $\mathcal{C}[x_1 = a]$ denotes the instantiation of a to x_1.

CSPs are most commonly solved by interleaving constraint propagation and search. Work on QCSPs has followed the same path so far [1,2,3], and I take the same approach here. I define some local consistency conditions based on a single constraint C. The following definitions of consistency are taken from López-Ortiz et. al. [9] and generalized to the quantified case. When a variable is instantiated, it is considered to have a unit domain.

Definition 4. *Support*

Given some constraint C, a value $a \in D_i$ for a variable $x_i \in \mathcal{X}_C$ has domain support *in C iff there exists a tuple $t \in C_S$ such that $t_i = a$* [4] *and $\forall x_j \in \mathcal{X}_C : t_j \in D_j$. Similarly, a partial assignment p (which is a set of pairs $\langle x_i, a\rangle$) over C has* domain support *in C iff there exists a tuple $t \in C_S$ such that for all pairs (x_i, a) in p, $t_i = a$ and $\forall x_j \in \mathcal{X}_C : t_j \in D_j$.*

Definition 5. *Generalized arc-consistency (GAC)*

A constraint C is generalized arc-consistent iff for each $x \in \mathcal{X}_C$, each value $a \in D_x$ has a domain support in C.

Consistency is achieved for the entire problem by removing inconsistent domain values. For the running example in figure 1, GAC is not able to prune any values. It is possible to define stronger local consistencies in the presence of universal variables. In the definition of a constraint, the variables $\mathcal{X}_C = (x_i, \ldots, x_j)$ are ordered in the same way as the quantifier sequence. Intuitively, if some variable x_i has an inner variable x_j which is universal, then each value of x_i must have domain support for each value of x_j, by the semantics of the QCSP. If x_i has more than one inner universal variable, each value must have domain support for all possible choices of values for the inner universals, hence the definition of domain support over partial assignments is used. This is a stronger consistency than GAC because each literal (a variable and value pair, $\langle x, a\rangle$) may require many tuples to support it.

Definition 6. *Weak Quantified GAC*

A constraint C is weak quantified GAC (WQGAC) iff for each variable $x \in \mathcal{X}_C$ and value $a \in D_x$, with inner universal variables $x_i, x_j, \ldots$, each partial assignment $p = \{\langle x, a\rangle, \langle x_i, b\rangle, \langle x_j, c\rangle | b \in D_i, c \in D_j \ldots\}$ has domain support.

For the example in figure 1, WQGAC is able to prune the following values from $grey1 : 1, 3, 5, 6, 7$, but is unable to prune the other existential variables.

The definition of WQGAC is not the strongest possible for quantified GAC, as this example illustrates: consider a constraint with variables $\forall x_1, \exists x_2, \forall x_3, \exists x_4$, and all variables have initial domain $D_i^0 = \{0, 1\}$. The set C_S is given below:

$\forall x_1$	$\exists x_2$	$\forall x_3$	$\exists x_4$
0	0	0	0
0	1	1	0
1	1	0	1
1	0	1	1

[4] t_k is used to refer to the element of t corresponding to variable x_k.

For the literal $x_1 = 0$, a tuple is required for each value of x_3. The first two tuples in the table meet the requirement. However, the value for x_2 is different, so these two tuples could not form part of the same winning strategy for the QCSP. There does not exist a value of x_2 such that all values of x_3 can be extended to a satisfying assignment. WQGAC can make no inferences, however the definition can be strengthened to SQGAC, which finds the constraint false.

Definition 7. *Strong Quantified GAC*
A constraint C is strong quantified GAC (SQGAC) iff for each variable $x \in \mathcal{X}_C$ and value $a \in D_x$, with inner universal variables $x_i, x_j, \ldots$, each partial assignment $p = \{\langle x, a\rangle, \langle x_i, b\rangle | b \in D_i, \langle x_j, c\rangle | c \in D_j, \ldots\}$ has domain support and *all the supporting tuples can form part of the same winning strategy. For any two supporting tuples τ and τ' this is the case iff $\exists\lambda\forall i < \lambda : \tau_i = \tau'_i \wedge \tau_\lambda \neq \tau'_\lambda \wedge \forall(x_\lambda)$ (i.e. the leftmost difference between the tuples must correspond to a universal variable).*

A QCSP $\mathcal{P}$ is (W/S)QGAC iff each constraint $C \in \mathcal{C}$ is (W/S)QGAC. This allows us to locally reason about non-binary arbitrary constraints in the problem and therefore avoid expensive reasoning on the whole problem. This local reasoning can be interleaved with search to provide a sound and complete solver for QCSP.

For the example in figure 1, SQGAC is able to prune from all three existential variables, in contrast to WQGAC. SQGAC can infer *grey*$2 \in \{4, 5\}$, and *grey*$3 \in \{2, 4, 5, 6\}$. Any value which cannot form part of a winning strategy covering C are pruned, therefore I believe that SQGAC is the strongest form of consistency which considers each constraint individually and posts only unary constraints.

The definition of WQGAC is designed to be enforced by an algorithm that processes tuples, specifically one generalized from a general GAC algorithm for CSP. The motivation for this weaker definition is therefore an engineering concern. It may also be the case that WQGAC has a lower complexity than SQGAC. The definition of SQGAC in similar terms to WQGAC allows comparison between the two. I believe that SQGAC is equivalent to the definition of local inconsistency given by Bordeaux et. al. 2005 [4], and the definition of quantified arc-consistency given by Bordeaux and Monfroy [3].

In CSP, the hidden variable encoding [6] encodes a non-binary constraint into n binary constraints and one extra (hidden) variable representing the supporting tuples.

Definition 8. *The hidden variable encoding*
Encoding non-binary constraint C over variables $x_1, \ldots, x_n$ into n binary constraints, with additional variable v. The quantification of the variables does not affect the encoding. The additional variable v represents the supporting tuple, so its initial domain is the set of all supporting tuples for C. A binary constraint r_i links variable x_i to v. A literal $x_i = a$ conflicts with the tuple $v = \tau$ in r_i iff $\tau_i \neq a$. Variables $x_1, ..., x_n$ retain their quantification. v is existentially quantified after $x_1, ..., x_n$ since a supporting tuple must exist for the assignment to $x_1, \ldots, x_n$.

If a domain removal is made from one of the variables x_i, the constraint r_i then propagates the changes to v, removing tuples which are no longer valid. This propagates back to the other x variables, establishing GAC when all constraints are satisfied. The encoding scales with the size of the tuple set, potentially d^n where d is the domain size [6].

3 Comparing effectiveness of local consistencies

In this section, the running example of figure 1 (expressed with one constraint as shown below the figure) and the arithmetic expression in equation (1) (reformulated in equation (2)) are used to compare different levels of consistency.

$$\exists x \in \{1..5\}, \forall y \in \{1..5\}, \exists z \in \{1..5\} : 2x + 5y + 3z = 30 \tag{1}$$

$$2 \times x = t_1,\ 5 \times y = t_2,\ 3 \times z = t_3,\ t_1 + t_2 = t_4,\ t_3 + t_4 = 30 \tag{2}$$

Consistency	Form of constraint	Inference	Form of constraint	Inference
	Figure 1		Equation (1)	
BAF			Equation (2)	none
QAC	Hidden variable encoding	none	Hidden variable encoding	none
GAC	unchanged	none	unchanged	none
WQGAC	unchanged	$grey1 \neq 1,3,5,6,7$	unchanged	False
SQGAC	unchanged	$grey1 \neq 1,3,5,6,7$ $grey2 \neq 1,2,3,6,7$ $grey3 \neq 1,3,7$	unchanged	False

Table 1. Comparison of consistency levels

Table 1 shows the inferences of each algorithm for these two examples. The binary quantified arc-consistency defined by Mamoulis and Stergiou is referred to as QAC. GAC refers to ignoring the quantifiers and applying a GAC algorithm.

Lemma 1. *GAC on a constraint C and QAC on the hidden variable encoding of C are equivalent.*

Proof. From the definition of GAC, each consistent pair $\langle x_i, a\rangle$ where $x_i \in \mathcal{X}_C$ and $a \in D_i$ must have domain support in C, i.e. there exists some tuple $t \in C_S$, where $t_i = a$ and $\forall j : t_j \in D_j$. In the hidden variable encoding, each constraint r_i is a one-to-many mapping from values $\langle x_i, a\rangle$ to tuples $t \in C_S$ where $t_i = a$. Therefore if $\langle x_i, a\rangle$ remains after performing AC, then there exists a tuple t where $t_i = a$. If AC is performed to completion on all r constraints, a tuple t is removed from the domain of the hidden variable iff $t_j \notin D_j$. Therefore we have that $\langle x_i, a\rangle$is supported by a tuple $t \in C_S$ where $t_i = a$ and $\forall j : t_j \in D_j$, which is equivalent to GAC.

Of the consistency rules given by Bordeaux and Monfroy [3], the BAF rules can be sensibly applied to equation 1, by reforming it as equation 2. Unfortunately they do very little inference, as shown in table 1.

4 A general schema for enforcing WQGAC

This section describes the proposed WQGAC-Schema algorithm, derived from GAC-Schema[5], a successful framework for GAC. In this section most attention will be given to the differences between WQGAC-Schema and GAC-Schema. On constraints with no universal variables, the behaviour of WQGAC-Schema is identical to GAC-Schema. The key feature of (WQ)GAC-Schema is multidirectionality, defined below.

Definition 9. *Multidirectionality*

1. *WQGAC-Schema never looks for a support for a partial assignment* p *on a constraint* C *when a tuple supporting* p *has already been found, and*
2. *it never checks whether a tuple is a support for a value when it has already been checked for another value [8].*

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

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

Point (1) of multidirectionality is taken into account with the S_C data structure. The seekInferableSupport procedure (algorithm 4) ensures that a new support is not sought if one is already stored in S_C. Point (2) is dealt with by the individual constraint representations described in section 5.

Algorithm 1 procedure establishWQGAC

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

Initialization To initialize the above data structures, the required supports must be found for all pairs $\langle x_i, a\rangle$, and recorded appropriately (or the value a must be removed from D_i). This is achieved with establishWQGAC (algorithm 1), which calls findSupport for each pair $\langle x_i, a\rangle$ (algorithm 2). If findSupport cannot find all supports for $\langle x_i, a\rangle$, establishWQGAC calls exclude$\langle x_i, a\rangle$, which returns false if the removal falsifies the QCSP (possibly by domain wipeout of an existential or pruning a universal). The Boolean returned by establishWQGAC represents whether the QCSP has been falsified.

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

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

Propagation After initialization, removing an element from a domain may result in one or more supports becoming invalid. The procedure propagate(x_i, a) (algorithm 3) replaces the invalid supports if possible, otherwise prunes the unsupported values. The procedure generatePA generates the set of partial assignments containing (x_i, a) for all possible sequences of inner universal assignments. This is the set whose supports are invalidated by the removal. For each of these partial assignments p, $S_C(p)$ contains all the tuples τ containing p which were previously supporting something and are now invalid. τ is removed from S_C, then all the partial assignments r which are currently supported by τ are processed: if r is still valid, a new support is required. $S_C(r)$ may contain another valid support: this would be discovered by seekInferableSupport. If not, seekNextSupport is called to find the next support in lexicographic order. If one is found, it is added to the relevant sets in S_C and S, and $last_C$ is updated. If no support for r is found, the appropriate value is pruned.

The procedure seekInferableSupport (algorithm 4) searches the set $S_C(p)$ to find a valid support for p which was found earlier to support some other partial

Algorithm 2 procedure findSupport

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

Algorithm 3 procedure propagate

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

Algorithm 4 procedure seekInferableSupport

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

Algorithm 5 procedure seekNextSupport for the predicate instantiation

```
procedure seekNextSupport($p$: partial assignment, $\tau$: tuple): tuple
if $\tau \neq nil$:
    $(\tau, dummy) = \text{nextTuple}(p, \tau, |\tau|)$
else:
    $\tau =$ smallest valid tuple containing $p$
$\tau = \text{seekCandidateTuple}(p, \tau, 1)$
while $\tau \neq nil$:
    if $f_C(\tau)$:
        return $\tau$
    else:
        $(\tau, k) = \text{nextTuple}(p, \tau, |\mathcal{X}_C|)$
        $\tau = \text{seekCandidateTuple}(p, \tau, k)$
return $nil$
```

assignment. It clears invalid tuples from the set as they are found. This satisfies part 1 of the the definition of multidirectionality (definition 9).

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

5 How to deal with specific constraint representations

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

Predicates The constraint is defined by an arbitrary expression for which no specific propagation algorithm is known. The user provides a black box function $f_C(\tau)$, which returns *true* iff the tuple τ satisfies the constraint, *false* otherwise. This is used in the seekNextSupport procedure shown in algorithm 5. seekNextSupport(p: partial assignment, τ: tuple) returns the smallest (in lexicographic order) tuple greater than τ which is checked to be allowed by C. The only change from the GAC-Schema version in [5] is that the variable y and value b have been replaced everywhere with p.

The procedure seekCandidateTuple (algorithm 6) uses the $last_C$ data structure to jump forward, skipping tuples which have already been checked and found not to satisfy C. This satisfies part 2 of multidirectionality (definition

Algorithm 6 procedure seekCandidateTuple

```
procedure seekCandidateTuple($p$: partial assignment, $\tau$:tuple, $k$: index): (tuple, index)
while $\tau \neq nil$ and $k \leq |\tau|$:
    $q = (x_k, \tau_k)$ {construct a partial assignment starting with index $k$}
    for all $i$ s.t. $k < i \leq |\tau|$:
        if $\forall(x_i)$: $q = q + (x_i, \tau_i)$ {add all inner universal assignments}
    $\lambda = last_C(q)$
    if $\lambda \neq nil$:
        $split = 1$
        while $\tau_{split} = \lambda_{split}$: $split = split + 1$
        if $split > |\tau|$ or $\tau_{split} < \lambda_{split}$:
            if $split < k$:
                $(\tau, k') =$ nextTuple($p$, $\tau$, $k$)
                $k = k' - 1$
            else:
                $(\tau, k') =$ nextTuple($p$, $\tau$, $|\tau|$)
                $k = min(k, k' - 1)$
    $k = k + 1$
return $\tau$
```

9). Together with the other part, a tuple will not be checked more than once, therefore limiting the total number of checks to d^n.

A *candidate* is a valid tuple which has not been checked. The procedure seekCandidateTuple(p, τ, k) returns the smallest candidate greater than or equal to τ, assuming τ is valid and includes p and the prefix $\tau_{1..k-1}$ is a possible prefix for a candidate.

The change to the algorithm is that when retrieving the relevant entry of $last_C$, the algorithm constructs a partial assignment q from $\langle x_k, \tau_k \rangle$ and the inner universal assignments. The intuition is that τ contains q so for $\lambda = last_C(q)$, if $\lambda >_{lo} \tau$ then τ has already been checked or jumped over when searching for support for q. If the difference between λ and τ occurs before k, nextTuple(p, τ, k) is called which ensures that the prefix $\tau_{1..k}$ is increased while respecting p. Otherwise, nextTuple(p, τ, $|\tau|$) is called to get the valid tuple following λ. k decreases to the smallest index where the value of τ has changed. Bessière and Régin give a sketch proof which shows that seekCandidateTuple cannot miss any candidates when jumping forwards with the calls to nextTuple [5]. This applies here unchanged, apart from the substitution of p for the variable and value y, b.

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

Positive constraints Here the set of allowed tuples (C_S) is given explicitly. Again, this is generalized from the algorithm given by Bessière and Régin [5], with the data structure from Mohr and Masini [10]. In practice this will only be practical for very loose constraints, but it has better time complexity than the

Algorithm 7 procedure seekNextSupport for the positive instantiation

```
procedure seekNextSupport($p$: partial assignment, $dummy$): tuple
while $elt(p) \neq nil$:
    $\sigma = C_S(elt(p))$
    if $\sigma$ is valid against current domains: return $\sigma$
    $elt(p) = next(elt(p))$
return $nil$
```

predicate instantiation. The set C_S is sorted by partial assignment, to match the requirements of supporting a value. For each pair $\langle x_i, a \rangle$, the tuples matching $\langle x_i, a \rangle$ are divided into each possible sequence of inner universal assignments. (This does not increase the asymptotic space consumption because each tuple of length n has n references to it.) The seekNextSupport procedure is given in algorithm 7.

Negative constraints The set of disallowed tuples is given explicitly. Bessière and Régin give an efficient method based on hashing which uses the predicate instantiation and can be used without any modification [5].

Other instantiations Bessière and Régin's more recent work instantiates GAC-Schema to process a conjunction of constraints (a subproblem) [8], which gives the same capabilities as the predicate instantiation but with much greater efficiency — the search performed by the predicate instantiation is like a generate-and-test, whereas the subproblem instantiation can use CSP propagation algorithms to prune the inner search space. Adapting the subproblem instantiation remains for future work.

6 Space and time complexities

The space and time properties of WQGAC-Schema+predicate are compared with GAC-Schema+predicate. Let C have arity n and contain variables with domain size d. GAC-Schema+predicate requires $O(n^2d)$ space, with the greatest cost being the S_C data structure. WQGAC-Schema maintains more supports: for a constraint with $u < n$ universal variables, WQGAC-Schema maintains support for $O(nd^{u+1})$ partial assignments compared to nd values for GAC-Schema. Therefore there are potentially $O(nd^{u+1})$ tuples stored. Since each tuple supports n partial assignments, S_C contains n references to it, giving a space requirement of $O(n^2d^{u+1})$. However, there can be no more than d^n tuples, so if $d^n < nd^{u+1}$ then the space requirement is $O(nd^n)$.

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

The other cost of enforcing multidirectionality is in seekNextSupport and seekCandidateTuple. For each variable x_i and each value a, the space of assignments to other variables (size d^{n-1}) is divided up among the partial assignments supporting (x_i, a), but the whole space is covered. I assume no jumping forward is possible. Finding the lexicographically next tuple takes constant time[5], so the total time taken is $O(nd \times d^{n-1}) = O(nd^n)$.

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

7 Examples

In this section the predicate instantiation of WQGAC-Schema is tested on some example games. Firstly the algorithm is tested on Connect-4 endgames, for constraints with 3, 5 and 7 variables. This shows its effectiveness in minimizing the number of tuples tested, and also gives some indication of time requirements. Secondly it is embedded in backtracking search, and tested on an encoding of noughts and crosses.

CPU times are given for an implementation in Java, running with the Java 5.0 HotSpot compiler on a Pentium 4 3.06GHz with 1Gb of memory. Although some attention was paid to efficiency in the implementation, this was not the main concern and the CPU times could be improved.

Connect-4 endgames To illustrate the strength of WQGAC and the efficiency of WQGAC-Schema, I give some endgames for the game Connect-4. The predicate instantiation of WQGAC-Schema is used.

The first and simplest endgame is shown in figure 2(a). It is grey to move, and the question is: can grey win whatever move black makes? Grey can win in two moves if black defends perfectly. Representing a move by the column number, grey wins with the sequence 5-2-2. The problem is modelled as shown below, with three variables (representing the column number of the move) and one constraint, which forbids black winning or drawing. If a grey (black) cheats (by placing a counter in a full column) then the constraint is unsatisfied (satisfied), that is black (grey) wins the game.

$$\exists grey1 \forall black1 \exists grey2 : \text{greywins}(grey1,\ black1,\ grey2)$$

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

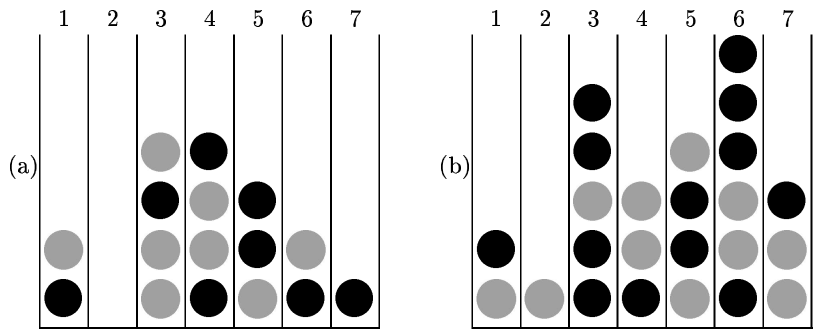


Fig. 2. Connect-4 Endgames

This approach of encoding all game rules into one constraint is not feasible for the full game of Connect-4, since the constraint in that case would be far too large for WQGAC-Schema to handle. The full game could be encoded as a conjunction of constraints, with local reasoning performed on each constraint with WQGAC-Schema, as with the noughts and crosses example below.

WQGAC-Schema determines the two moves grey must make in 0.008s (table 2(a)). From $7^3 = 343$ possible tuples, 99 are tested against the predicate.

The second endgame is the running example in figure 1. Grey can win in three moves if black defends perfectly, and in two moves if black makes a mistake. There are five move sequences where black defends perfectly: 2-2-4-4-5, 2-2-5-4-4, 4-4-5-2-2, 4-4-5-2-6 and 4-4-5-6-2. Table 2(running example) shows three consecutive actions on the greywins constraint. (Asserting a value includes calling propagate to exhaustion.) Apart from two additional move variables, the model remains the same as above.

The total number of tuples is $7^5 = 16807$, of which 2554 were tested in total, which is 15.2%. The algorithm is effective in minimizing the number of tests. This example also shows the weakness of WQGAC: when first establishing WQGAC, only the outermost variable is pruned. SQGAC would be able to prune $grey2 : 1, 2, 3, 6, 7$ and $grey3 : 1, 3, 7$ as well.

The third endgame, shown in figure 2(b), is used to test WQGAC-Schema with a longer constraint. There are seven variables in total, and again grey has the first move. Apart from two additional move variables, the model remains the same. In this case it is easy for grey to win so the greywins constraint is loose, and establishing WQGAC only prunes the cheating move (table 2(b)). SQGAC is also unable to prune. While moves such as $grey1 : 6, 7$ are wasted, grey is still able to win. The grey player can waste a move at any turn so no pruning is possible. During the three actions, the algorithm tests a very small percentage of tuples, although time is becoming a problem. To execute the three actions requires (at peak) 4097 kilobytes of memory.

Noughts and crosses WQGAC-Schema is embedded in a backtracking search procedure, in which variables are instantiated in quantification order, and uni-

Action	Tuples tested	Total tested %	Values pruned	CPU time
Board (a)				
establishWQGAC()	99	28.9%	$grey1 : 1, 2, 3, 4, 6, 7$ and $grey3 : 1, 3, 4, 5, 6, 7$	0.008s
Running example (figure 1)				
establishWQGAC()	2196	15.2%	$grey1 : 1, 3, 5, 6, 7$	0.046s
assert $grey1 \neq 2$	207		none	0.008s
assert $black1 = 4$	151		$grey2 : 1, 2, 3, 4, 6, 7$ and $grey3 : 1, 3, 4, 5, 7$	0.016s
Board (b)				
establishWQGAC()	20611	4.1%	$grey1 : 6$	0.167s
assert $grey1 = 4$	12796		none	0.133s
assert $black1 = 4$	629		$grey2 : 1, 3, 4, 6, 7$	0.018s

Table 2. Connect-4 results

versal variables branch for every value, closely following definition 3. The propagation algorithm is constraint-oriented, maintaining a queue of (C, x_i, a) records to be processed[6]. Noughts and crosses (tic tac toe) is played on a 3×3 board. The aim is to make a line of three counters, including diagonal lines. The two players take turns to place a counter on any free slot. The first player is crosses ($\times$), followed by noughts ($\circ$). The aim is to find if crosses can win however noughts defends, therefore the constraints are all satisfied if crosses wins the game, and some constraint is unsatisfied if crosses cheats or noughts wins or draws. The case where noughts cheats is discussed below. All the constraints are implemented with WQGAC-Schema+predicate.

To test the potential benefit of enforcing WQGAC on high-arity constraints, I compare two models. First the game is modelled with 9 move variables (with alternating quantification) each with domain size 9. The state of the board is represented with 9 variables per move, $b^i_{pos} \in \{\times, \circ, nil\}$ where $pos \in 1..9$. $w^i \in \{\times, \circ, nil\}$ indicates the winner at move i, and Boolean variables xl^i and ol^i indicate if a player has a line at move i. The quantifier sequence is $\diamond m^i \exists b^i_{1..9}, w^i, (x \text{ or } o)l^i$ where $\diamond = \exists$ for $\times$ moves and $\forall$ for $\circ$ moves.

The moves are modelled with 8 variables m^i, the board states with 9 variables per move, $b^i_{pos} \in \{\times, \circ, nil\}$ where $pos \in 1..9$, $w^i \in \{\times, \circ, nil\}$ indicates the winner at move i, and Boolean variables xl^i and ol^i indicate if a player has a line at move i. The quantifier sequence for the first 6 moves is $\diamond m^i \exists b^i_{1..9}, w^i, (x \text{ or } o)l^i$ where $\diamond = \exists$ for $\times$ moves and $\forall$ for $\circ$ moves. For $\times$ moves, there are 11 constraints: $\forall pos : (m^i = pos) \Rightarrow (b^{i-1}_{pos} = nil \wedge b^i_{pos} = \times)$, findline$(b^i_{1..9}, xl^i)$, wins$(w^{i-1}, xl^i, w^i)$. For $\circ$ moves, we have: $\forall pos : (b^{i-1}_{pos} = nil \wedge m^i = pos) \Rightarrow (b^i_{pos} = \circ)$, findline$(b^i_{1..9}, ol^i)$, wins$(w^{i-1}, ol^i, w^i)$. For the first 4 moves, the findline constraints are omitted, and $w^{1..4} = nil$ and $w^5 \neq \circ$. For values a of m^i where the corresponding board position is already filled (i.e. cheating moves), a

[6] The queue is a stack. No attempt was made to optimize queue behaviour.

is not contained in any nogoods[7]. Such values are dynamically pruned by the pure value rule [2]. When searching with this model, only the move variables are instantiated by the search algorithm, all others are set by propagation. The value ordering is line by line, left to right on the board. At the root node, establishing WQGAC for all constraints took 19ms. The search explored 4107 internal nodes in 26.205s.

In the second model, the final three moves are represented with a single constraint, over variables $w^6, b^6_{1..9}, m^7, m^8$. All other variables and constraints for these moves are eliminated. In particular, m^9 is removed because it is unit. To avoid cheating moves, m^7 has only 3 values and m^8 only 2, and these are mapped to the free slots. The constraint is satisfied iff noughts wins the game. To establish WQGAC now takes 134ms, and the search explores 3403 internal nodes explored in 13.782s. To an extent, this shows the potential of consolidating a set of constraints into a single high-arity constraint, because better propagation is achieved and the time to reach local consistency at each node is reduced.

The results above should not be oversold because WQGAC-Schema was used for all constraints. A simpler algorithm may be more efficient for shorter constraints, aiding the first model more than the second.

8 Conclusion

Generalized arc-consistency has been well studied and is very important in CSP. To my knowledge, this is the first time a GAC-like consistency has been brought to QCSP. I have defined two new levels of consistency based on GAC, and have developed an algorithm for one. This is empirically tested on game problems and embedded in backtracking search.

9 Acknowledgements

This work is funded by EPSRC, and I would like to thank my supervisor Ian Gent, Ian Miguel and the anonymous reviewers who all made helpful comments on this paper. I would like to thank Björn Assmann for help with footnote 4.

References

1. Nikos Mamoulis and Kostas Stergiou, Algorithms for Quantified Constraint Satisfaction Problems, in Proc. 10th CP, pages 752-756, 2004.

[7] If the remainder of the game is satisfiable for all non-cheating moves, then it must also be satisfiable for move a since no counter is placed, therefore the remainder of the game is easier to win for noughts. Therefore move a can only falsify the remainder of the game if some non-cheating move does as well. Because of the universal quantification, this is equivalent to the desired semantics: that for a cheating move the remainder of the game is satisfied.

2. Ian P. Gent, Peter Nightingale and Kostas Stergiou, QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems, to appear in Proc. 19th IJCAI, 2005.
3. Lucas Bordeaux and Eric Monfroy, Beyond NP: Arc-Consistency for Quantified Constraints, in Proc. 8th CP, pages 371-386, 2002.
4. Lucas Bordeaux, Marco Cadoli and Toni Mancini, CSP Properties for Quantified Constraints: Definitions and Complexity, in Proc. 20th AAAI, pages 360-365, 2005.
5. Christian Bessière and Jean-Charles Régin, Arc consistency for general constraint networks: preliminary results, in Proc. 15th IJCAI, pages 398-404, 1997.
6. Nikos Mamoulis and Kostas Stergiou, Solving Non-binary CSPs Using the Hidden Variable Encoding, in Proc. 7th CP, pages 168-182, 2001.
7. Ian Gent and Andrew Rowley, Encoding Connect-4 using Quantified Boolean Formulae, APES Technical Report APES-68-2003, 2003.
8. Christian Bessière and Jean-Charles Régin, Enforcing Arc Consistency on Global Constraints by Solving Subproblems on the Fly, in Proc. 5th CP, pages 103-117, 1999.
9. Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp and Peter van Beek, A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint, in Proc. 18th IJCAI, pages 306-319, 2003.
10. Roger Mohr and Gérald Masini, Good Old Discrete Relaxation, in Proc. 8th ECAI, pages 651-656, 1988.