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. 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.

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. QCSP can be used to model problems containing uncertainty, in the form of variables which have a finite domain but whose value is unknown. Therefore a QCSP solver finds solutions suitable for all values 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. This is known as a strategy, and can be exponential in size. This generalization increases the computational complexity (under the usual assumption that P$\subsetneq$NP$\subsetneq$PSPACE): QCSP is PSPACE-complete. QCSP can be used to model problems from areas such as planning with uncertainty and multiplayer games. Intuitively, these problems correspond 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? Actions are represented with existential variables, and eventualities with universals. Bordeaux and Monfroy [3] and Mamoulis and Stergiou [1] define levels of consistency for various quantified constraints of bounded arity. I introduce a general consistency algorithm for quantified constraints of any arity, based on Bessière and Régin's GAC-Schema [4].

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 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. The five such winning sequences are 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. 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 (i.e. the constraint is satisfied iff grey wins, thus all the rules of the game are exactly encoded in one constraint). 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 restrict all three of the grey move variables. Ignoring the quantification and applying 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 3.

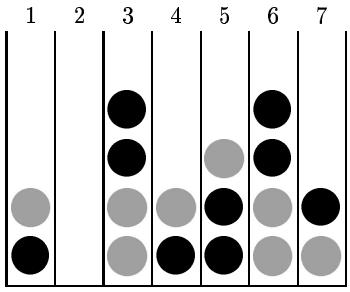
2 Defining Quantified Generalized Arc Consistency

A flavour of the definitions of weak and strong quantified generalized arc consistency is given below, based on the full definitions in [8]. $\mathcal{X}_C$ is the variables within the scope of the constraint C. C_S is the set of supporting tuples of the constraint C, with each tuple sorted in quantification order. The domain of variable x_i is D_i.

Definition 1. *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 $t_i = a$* [1] *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 $\langle x_i, a \rangle$ in p, $t_i = a$ and $\forall x_j \in \mathcal{X}_C : t_j \in D_j$.*

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



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

Fig. 1. Connect-4 endgame

Definition 2. *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 $p = \{\langle x, a\rangle, \langle x_i, b\rangle | b \in D_i, \langle x_j, c\rangle | c \in D_j, \ldots]\}$ 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.

Definition 3. *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 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).*

For the example in figure 1, an algorithm enforcing SQGAC would be able to prune from all three existential variables, in contrast to WQGAC. An SQGAC algorithm would infer $grey2 \in \{4, 5\}$, and $grey3 \in \{2, 4, 5, 6\}$.

3 A general algorithm for enforcing WQGAC

This section describes the proposed WQGAC-Schema algorithm, derived from GAC-Schema[4], 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 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. This change does not alter the time complexity. The space complexity increases to $O(nd^n)$. The modified data structure S_C is described below (S and $last_C$ are modified likewise [8]), to be compared with [4].

$S_C(p)$ 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.

In all cases the full description of a procedure is given in [8]. To propagate a pruned value (x, a), the procedure is summarized here. For all tuples t in S_C which contain (x, a), t is removed from S_C. If this leaves a partial assignment p unsupported, a new support is sought by calling seekNextSupport(). This procedure is specific to the constraint type.

Action	Tuples tested	Total tested %	Values pruned	CPU time
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

Table 1. Connect-4 results

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. The only change from the GAC-Schema version in [4] is that the variable y and value b have been replaced everywhere with partial assignment p. The basic idea is that supporting tuples are tested in lexicographic order against f_C, skipping forward whenever possible.

Positive constraints Here the set of allowed tuples (C_S) is given explicitly. This is slightly altered from the algorithm given by Bessière and Régin [4], with the data structure from Mohr and Masini [7]. 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 simply returns the next valid tuple in the relevant list.

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 modification [4].

Testing WQGAC-Schema on a Connect-4 endgame To illustrate the strength of WQGAC and the efficiency of WQGAC-Schema, I use the running example (figure 1). The predicate instantiation of WQGAC-Schema is used. To my knowledge, there is no way of representing the quintary constraint with shorter constraints without losing propagation, hence there is no direct comparison to be made. Grey can win in three moves if black defends perfectly, and in two moves if black makes a mistake. There are five winning 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 1 shows three consecutive actions on the greywins constraint. (Asserting a value includes calling propagate to exhaustion.) 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. Only 15.2% of the tuples were tested against the predicate, showing that WQGAC-Schema is effective in jumping over irrelevant tuples.

Testing WQGAC-Schema with noughts and crosses WQGAC-Schema is embedded in a simple backtracking search for QCSP. An experiment based on noughts and crosses (tic-tac-toe) is described in the technical report [8]. First the game is modelled with 9 move variables (with alternating quantification) each with domain size 9, and board state variables for each move. The pure value rule [2] is dynamically applied to universal variables, so that values representing cheating moves are not explored. The question encoded is: can the first player win under any circumstances, to which the answer is no. All constraints are implemented using WQGAC-Schema+predicate. The longest constraints are arity 10, for detecting the winning condition. The search explored 4107 internal nodes in 26.205s, taking on average 6.38ms per node. This is compared to a similar model, with the final 3 moves eliminated and replaced with one large constraint of arity 12. Three constraints of arity 10 are removed. The number of internal nodes is reduced to 3403, explored in 13.782s, on average 4.05ms per node. 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.

4 Conclusion and acknowledgments

Generalized arc-consistency has been well studied and is very important in CSP. I have defined for QCSP two new levels of consistency based on GAC, and have developed an algorithm for one. This was briefly tested on game problems.

This work is funded by EPSRC, and I would like to thank my supervisor Ian Gent, the anonymous reviewers and Ian Miguel, who made many helpful comments on this paper.

References

1. Nikos Mamoulis and Kostas Stergiou, Algorithms for Quantified Constraint Satisfaction Problems, in *Proc. of the 10th CP*, pages 752-756, 2004.
2. Ian P. Gent, Peter Nightingale and Kostas Stergiou, QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems, to appear in *Proc. of the 19th IJCAI*, 2005.
3. Lucas Bordeaux and Eric Monfroy, Beyond NP: Arc-Consistency for Quantified Constraints, in *Proc. of the 8th CP*, pages 371-386, 2002.
4. Christian Bessière and Jean-Charles Régin, Arc consistency for general constraint networks: preliminary results, in *Proc. of the 15th IJCAI*, pages 398-404, 1997.
5. Ian Gent and Andrew Rowley, Encoding Connect-4 using Quantified Boolean Formulae, APES Technical Report APES-68-2003, 2003.
6. 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. of the 18th IJCAI*, pages 306-319, 2003.
7. Roger Mohr and Gérald Masini, Good Old Discrete Relaxation, in *Proc. of the 8th ECAI*, pages 651-656, 1988.
8. Peter Nightingale, Consistency for Quantified Constraint Satisfaction Problems, CP-Pod Technical Report CPPOD-11-2005, 2005. Available from http://www.dcs.st-and.ac.uk/~cppod/publications/reports/