

# BlockSolve: a Bottom-Up Approach for Solving Quantified CSPs

Student: Guillaume Verger  
Supervisor: Christian Bessiere

LIRMM, CNRS/University of Montpellier, France  
{verger,bessiere}@lirmm.fr

**Abstract.** Thanks to its extended expressiveness, the quantified constraint satisfaction problem (QCSP) can be used to model problems in model checking or planning under uncertainty that cannot be expressed in the standard CSP formalism. This is only recently that the constraint community got interested in QCSPs and that algorithms to solve it were proposed. In this paper we propose **BlockSolve**, an algorithm for solving QCSPs that factorizes computations made in branches of the search tree. Instead of following the order of the variables in the quantification sequence, our technique searches for combinations of values for existential variables at the bottom of the tree that will work for values of variables earlier in the sequence. An experimental study shows the good performance of **BlockSolve** compared to a state of the art QCSP solver.

## 1 Introduction

The quantified constraint satisfaction problem (QCSP) is an extension of the constraint satisfaction problem (CSP) in which variables are totally ordered and quantified either existentially or universally. This generalization provides a better expressiveness for modelling problems. Model Checking and planning under uncertainty are problems that can be modeled with QCSP. But such an expressiveness implies a high complexity. Whereas CSP is in NP, QCSP is PSPACE-complete, which is considered harder.

The SAT community has also done a similar generalization from the problem of satisfying a Boolean formula into the quantified Boolean formula problem (QBF). The most natural way to solve instances of QBF or QCSP is to instantiate variables from the outermost quantifier to the innermost. This approach is called *top-down*. Most QBF solvers implement top-down techniques. Those solvers lift SAT techniques to QBF. Nevertheless, Biere [1], or Pan and Vardi [2] proposed different techniques to solve QBF instances. Both try to eliminate variables from the innermost quantifier to the outermost quantifier, an approach called *bottom-up*. The bottom-up approach is motivated by the fact that the efficiency of heuristics that are used in the SAT domain seems to be greatly reduced in QBF. The drawback of such approaches is the cost in space.

The problem of solving a QCSP is more recent than QBF, so there are few QCSP solvers. Gent, Nightingale and Stergiou [3] developed QCSP-Solve, a top-down solver that uses generalizations of well-known techniques in CSP. Repair-based methods seem to be quite helpful as well, as shown by Stergiou in [4].

In this paper we introduce `BlockSolve`, the first bottom-up algorithm to solve QCSPs. `BlockSolve` instantiates variables from the innermost to the outermost. On the one hand, this permits to factorize equivalent subtrees during the search. On the other hand, `BlockSolve` only uses classical CSP techniques, no need for generalizing them into QCSP techniques. The algorithm processes a problem as if it were composed of pieces of classical CSPs.

The rest of the paper is organized as follows. Section 2 defines the concepts that we will use during the paper. Section 3 describes `BlockSolve`. Finally, Section 4 experimentally compares `BlockSolve` to the state-of-the-art QCSP solver QCSP-Solve and Section 5 contains a summary of this work and details for future work.

## 2 Preliminaries

In this section we define the basic concepts that we will use.

**Definition 1 (Quantified Constraint Network).** A quantified constraint network is a formula  $QC$  in which:

- $Q$  is a sequence of quantified variables  $Q_i x_i, i \in [1..n]$ , with  $Q_i \in \{\exists, \forall\}$  and  $x_i$  a variable with a domain of values  $D(x_i)$ ,
- $C$  is a conjunction of constraints  $(c_1 \wedge \dots \wedge c_m)$  where each  $c_i$  involves some variables among  $x_1, \dots, x_n$ .

Now we define what is a solution of a quantified constraint network.

**Definition 2 (Solution).** The solution of a quantified constraint network  $QC$  is a tree such that:

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

It is important to notice that contrary to classical CSPs, variables are ordered as an input of the network. A different order in the sequence  $Q$  gives a different network.

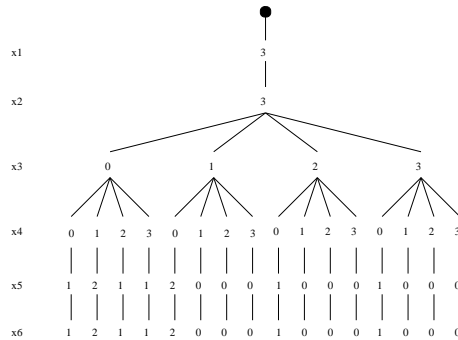
If all variables are existentially quantified, a solution to the quantified network is a classical instantiation, so the network is a classical constraint network.

Definition 2 leads to the concept of quantified constraint satisfaction problem.

**Definition 3 (QCSP).** A quantified constraint satisfaction problem (QCSP) is the problem of the existence of a solution to a quantified constraint network.

We point out that this original definition of QCSP, though different in presentation, is equivalent to previous recursive definitions. The advantage of ours is that it formally specifies what a solution of a QCSP is.

*Example 1.* Consider the quantified network  $\exists x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 \exists x_6, (x_1 \neq x_5) \wedge (x_1 \neq x_6) \wedge (x_2 \neq x_6) \wedge (x_3 \neq x_5) \wedge (x_4 \neq x_6) \wedge (x_3 \neq x_6), D(x_i) = \{0, 1, 2, 3\}, \forall i$ . Figure 1 shows a solution of this network.



**Fig. 1.** A solution tree of Example 1

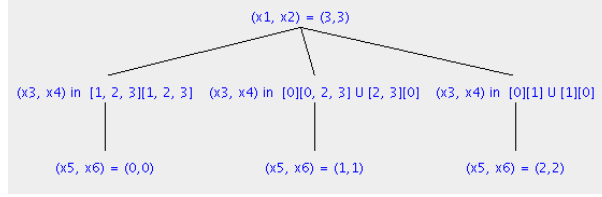
We define the concept of block, which is the main concept handled by our algorithm `BlockSolve`.

**Definition 4 (Block).** A block in a network  $\mathcal{QC}$  is a maximal subsequence of variables in  $\mathcal{Q}$  that have the same quantifier. We call a block that contains universal variables a universal block, and a block that contains existential variables an existential block.

If  $x$  and  $y$  are variables in two different blocks, with  $x$  earlier in the sequence than  $y$ , we say that  $x$  is the *outer* variable and  $y$  is the *inner* variable. In this paper, we limit ourselves to binary constraints for simplicity of presentation: A constraint involving  $x_i$  and  $x_j$  is noted  $c_{ij}$ .

The concept of block can be used to define a solution tree of a QCSP. This is a compressed version of the solution tree defined above.

Figure 2 is the block-based version of the solution tree in Fig. 1. The problem is divided in three blocks, the first and the third blocks are existential whereas the second block is universal. Existential nodes are completely instantiated, it means that all variables of those blocks have a single value. The universal block is in three nodes, each one composed of the name of the variables and a union



**Fig. 2.** Solution of example 1

of Cartesian products of sub-domains. Each of the universal nodes represents as many nodes in the solution tree of Fig. 1 as there are tuples in the product.

**BlockSolve** will use this concept of blocks for generating a solution and for solving the problem.

Blocks will divide the problem in levels. Each couple (universal block, existential block) is a level. If the first block is existential, the first level is only composed by this block. We note  $p$  the number of levels in a problem.

We call  $\mathcal{P}_k$  the subproblem that contains variables in levels  $k$  to  $p$  and constraints that are defined on those variables. The universal block at level  $k$  is noted  $block_{\forall}(k)$  and the existential block is  $block_{\exists}(k)$ .  $\mathcal{P}_1$  is the whole problem  $\mathcal{P}$ . The principle of **BlockSolve** is to solve  $\mathcal{P}_p$  first, then using the result to solve  $\mathcal{P}_{p-1}$ , and so on until it solves  $\mathcal{P}_1 = \mathcal{P}$ .

### 3 The BlockSolve Algorithm

In this section we present the algorithm, and describe how it works on QCSP instances.

The main idea in **BlockSolve** is to instantiate existential variables of the last block, and to go up to the root instantiating all existential variables. Each assignment  $v_i$  of an existential variable  $x_i$  can lead to the deletion of inconsistent values of outer variables by propagation.

Removing a value of an outer *existential* variable is similar to the CSP case. While the domains of variables are non empty, it is possible to continue instantiating variables. But if a domain is reduced to the empty set, it will be necessary to backtrack on previous choices on inner variables and to restore domains.

Removing a value of an outer *universal* variable implies that we will have to find also another instantiation of inner variables that supports this value, because all tuples in universal blocks have to match to a partial solution of inner subproblem. But the instantiation that removes a value in the domain of an universal variable must not be rejected: it can be compatible with a subset of tuples of the universal block. The bigger the size of the subset, the better the grouping. Factorizing tuples of values for a universal block in large groups is a way for minimizing the number of times the algorithm has to solve subproblems. Each time an instantiation of inner variables is found consistent with a subset of

tuples for a universal block, we must store this subset and solve again the inner subproblem wrt remaining tuples for the universal variables.

For a level  $k$  starting from level 1, we try to solve the subproblem  $\mathcal{P}_{k+1}$ , without forgetting that it must be compatible with all constraints in  $\mathcal{P}$ . If there is no solution for  $\mathcal{P}_{k+1}$ , we try to solve  $\mathcal{P}_k$  with the tuples on  $block_{\exists}(k)$  not yet tried when solving  $\mathcal{P}_{k+1}$  (they were removed by instantiations in  $\mathcal{P}_{k+1}$  that finally led to failure). If there exists a solution for  $\mathcal{P}_{k+1}$ , we try to instantiate  $block_{\exists}(k)$  with values consistent with some of the tuples on  $block_{\forall}(k)$ . If success, we remove the tuples on  $block_{\forall}(k)$  that are known to extend on inner variables, and we start again the process on the not yet supported tuples of  $block_{\forall}(k)$ .

**BlockSolve** needs more space than a top-down algorithm like **QCSP-Solve**. It keeps in memory all tuples of existential and universal blocks for which a solution has not yet been found. The size of such sets can be exponential in the number of variables of the block. **BlockSolve** keeps sets of tuples as unions of Cartesian products, which uses far less space than tuples in extension. In addition, computing the difference between two unions of Cartesian products is much faster than with tuples in extension.

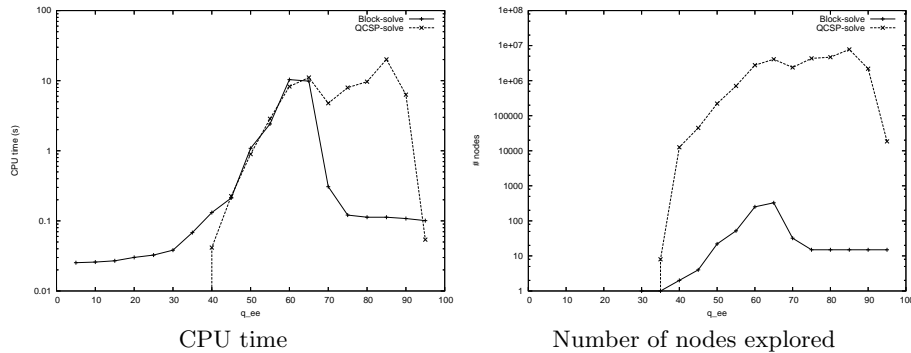
## 4 Experiments

In this section we compare **QCSP-Solve** and **BlockSolve** on random problems. The experiments show the differences between these two algorithms in CPU time and number of visited nodes.

**BlockSolve** is developed in Java using Choco as constraint library [5]. This library provides different propagation algorithms and a CSP solver. After loading the data of a problem, **BlockSolve** creates tables that fit to set of tuples of each block and finally launches the main function.

Instances of **QCSP** presented in these experiments have been created with a generator based on that used in [3]. The generator we use allow us to set the number of variables, the number of blocks, the number of variables in existential and universal blocks, the number of constraints, the looseness of binary constraints between a universal variable and an inner existential variable ( $q_{\forall\exists}$ ), and the looseness of constraints between two existential variables ( $q_{\exists\exists}$ ). We fix all parameters except  $q_{\exists\exists}$  during the experimentations. Results that are presented here in figure 3 are for instances of **QCSP** that have 25 variables with domains of 8 values, 5 variables by block. The first block of each instance is an existential one. For these instances, the transition phase is at  $q_{\exists\exists} = 55$

We note that results are comparables for instances that are unsatisfiable (on the left of the transition phase), even if **QCSP-Solve** detects the inconsistency a little bit faster than **BlockSolve** when problems are far from the transition phase. On the right, for problems that are satisfiable, **BlockSolve** is more efficient than **QCSP-Solve**. In almost all instances, **BlockSolve** visits far less nodes than **QCSP-Solve**.



**Fig. 3.** Problems with 25 variables and 5 blocks.

## 5 Conclusions

In this paper we presented **BlockSolve**, a bottom-up QCSP solver that uses classical CSP techniques. Its specificity is that it treats variables from leaves to root in the search tree, and factorizes lower branches avoiding the search in subtrees that are equivalent. The larger this factorization, the better the algorithm, thus minimizing the number of nodes visited. Experiments show that grouping branches gives **BlockSolve** a great stability in time spent and in number of nodes visited. The number of nodes **BlockSolve** visits is much smaller than the number of nodes visited by QCSP-Solve in almost all instances.

Future work will focus on improving time efficiency of **BlockSolve**. Great improvements can probably be obtained by designing heuristics to efficiently prune subtrees that are inconsistent. Furthermore, most of the cpu time is spent updating and propagating tables of tuples on blocks. Finding better ways to represent them could significantly decrease the cpu time of **BlockSolve**.

We are very grateful to Kostas Stergiou, Peter Nightingale and Ian Gent, who kindly provided us with the code of QCSP-Solve.

## References

1. Biere, A.: Resolve and expand. In: SAT. (2004)
2. Pan, G., Vardi, M.Y.: Symbolic decision procedures for qbf. In Wallace, M., ed.: CP. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 453–467
3. Gent, I., Nightingale, P., Stergiou, K.: QCSP-solve: A solver for quantified constraint satisfaction problems. In: Proceedings IJCAI’05, Edinburgh, Scotland (2005) 138–143
4. Stergiou, K.: Repair-based methods for quantified csp. In: Proceedings CP’05, Sitges, Spain (2005) 652–666
5. Choco: A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving, <http://choco.sourceforge.net>. (2005)