Blocksolve: A bottom-up approach for solving quantified CSPs

Guillaume Verger, Christian Bessiere

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

Guillaume Verger and 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 that are difficult to express in the standard CSP formalism. This is only recently that the constraint community got interested in QCSP and proposed algorithms to solve it. 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 (several) values of universal 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 examples of problems that can nicely be modeled with QCSP. But such an expressiveness has a cost. Whereas CSP is in NP, QCSP is PSPACE-complete.

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*. Biere uses expansion of universal variables into clauses to eliminate them, and Pan and Vardi use symbolic techniques. The bottom-up approach is motived by the fact that the efficiency of heuristics that are used in SAT is lost when following the ordering of the sequence of quantifiers. The drawback of bottom-up approaches is the cost in space.

The interest of the community in 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 like arc-consistency [4, 5], intelligent backtracking, and some QBF techniques like the Pure Literal rule. This state-of-the-art solver is faster than previous approaches that transform the QCSP into a QBF problem before calling a QBF solver. Repair-based methods seem to be quite helpful as well, as shown by Stergiou in [6].

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 search. On the other hand, `BlockSolve` only uses standard 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. Hence, `BlockSolve` uses the constraint propagation techniques of a standard CSP solver as long as it enforces at least forward checking (FC) [7]. The factorization technique used in `BlockSolve` is very close to that used by Fargier *et al.* for Mixed CSPs [8]. Mixed CSPs are QCSPs in which the sequence of variables is only composed of two consecutive sets, one universally quantified and the other existentially quantified. Fargier *et al.* decomposed Mixed CSPs to solve them using subproblem extraction as in [9]. `BlockSolve` uses this kind of technique, but extends it to deal with any number of alternations of existential and universal variables. Like QBF bottom-up algorithms, `BlockSolve` requires an exponential space to store combinations of values for universal variables that have been proved to extend to inner existential variables. However, storing them in a careful way dramatically decreases this space, as we observed in the experiments.

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`, starting by an example and discusses its space complexity. 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 $\mathcal{QC}$ in which:*

- *$\mathcal{Q}$ 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 $D(x_i)$,*
- *$\mathcal{C}$ is a conjunction of constraints $(c_1 \wedge ... \wedge c_m)$ where each c_i involves some variables among $x_1, \ldots, x_n$.*

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

Definition 2 (Solution tree). *The* solution tree *of a quantified constraint network $\mathcal{QC}$ 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 $x_1, \ldots, x_n$ defined by the labels of nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.*

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 $\mathcal{Q}$ gives a different network.

Example 1. The network $\exists x_1 \forall x_2, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ is inconsistent, there is no value for x_1 in $D(x_1)$ that is compatible with all values in $D(x_2)$ for x_2.

Example 2. The network $\forall x_2 \exists x_1, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ has a solution: whatever the value of x_2 in $D(x_2)$, x_1 can be instantiated.

Notice that if all variables are existentially quantified, a solution to the quantified network is a classical instantiation. Hence, 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 3. 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 tree for this network.

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*.*

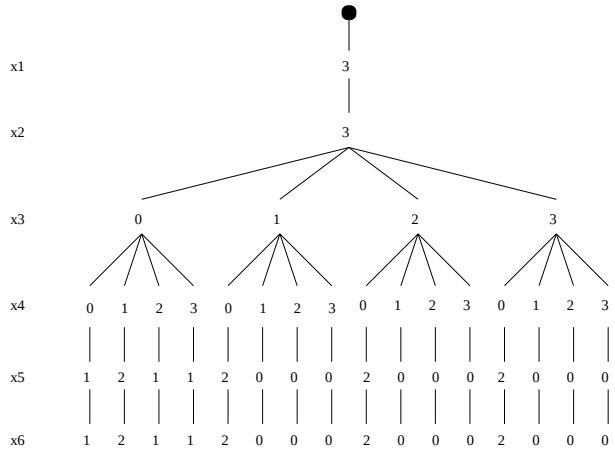


Fig. 1. A solution tree for Example 3

Inverting two variables of a same block does not change the problem, whereas inverting variables of two different blocks changes the problem. 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}. Nevertheless, `BlockSolve` can handle non-binary constraints if they overlap at most two blocks and there is at most one variable in the outer block (if there are two blocks).

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

Definition 5 (Solution block-tree). *The solution of a quantified constraint network $\mathcal{QC}$ is a tree such that:*

- *the root node r has no label,*
- *every node s at distance i from the root represents the ith block in $\mathcal{Q}$,*
- *every node s at distance i from the root r is labelled by an instantiation of the variables in the ith block if it is an existential block, or by a union of Cartesian products of sub-domains of its variables if it is a universal block,*
- *for every node s at depth i, the number of successors of s in the tree is 1 if the i+1th block is existential, or possibly more than 1 if the i+1th block is universal. When the i+1th block is universal, every combination of values for its variables appears in the label of one of the successors of s,*
- *for any leaf, an instantiation on $x_1, \ldots, x_n$ defined by the labels of the existential nodes from r to the leaf and any of the labels of the universal nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.*

Note that the root node is present only for having a tree and not a forest in cases where the first block is universal. Figure 2 is the block-based version of

the solution tree in Fig. 1. The root node is not shown because the first block is existential. 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 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. The block-tree in Figure 2 is a compressed version of the tree in Figure 1.

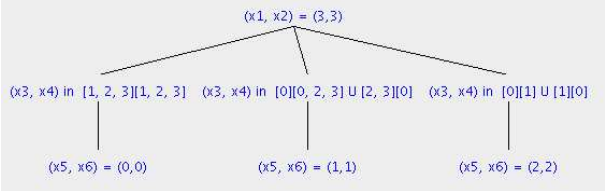


Fig. 2. Solution block-tree of example 3

`BlockSolve` uses this concept of blocks for generating a solution and for solving the problem. Blocks divides the problem in levels.

Definition 6 (Level). *A network $\mathcal{P} = \mathcal{QC}$ is divided in p* levels *from 1 to p. Each level $k, 1 \leq k \leq p$, is composed of a universal block $block_\forall(k)$, and the following existential block in $\mathcal{Q}$, noted $block_\exists(k)$. If the first block in $\mathcal{Q}$ is existential, the first level contains only this block, and if the last block is universal, the last level contains only this block.*

We call $\mathcal{P}_k$ the subproblem that contains variables in levels k to p and constraints that are defined on those variables. $\mathcal{P}_1$ 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 describe `BlockSolve`, our QCSP solving algorithm. First of all we run the algorithm on Example 3. Afterwards, we provide the general algorithm.

As done in QCSP-Solve, we start by a preprocessing that permanently removes constraints $\forall x_i \forall x_j\ c_{ij}$ and $\exists x_i \forall x_j\ c_{ij}$. Let us explain why these constraints can be completely removed. For constraints of type $\forall x_i \forall x_j\ c_{ij}$, if there exists a couple (v_i, v_j) of values for x_i and x_j that is forbidden by c_{ij}, then the whole problem is inconsistent. If not, the constraint will ever be satisfied, so we can

remove it. For constraints of type $\exists x_i \forall x_j\ c_{ij}$, if there exists a couple (v_i, v_j) of values for x_i and x_j that is forbidden by c_{ij}, then x_i cannot take value v_i. So, we can remove it from the domain of x_i. If $D(x_i)$ becomes empty, the problem is inconsistent. Once all values in $D(x_i)$ have been checked, we can remove c_{ij}. Once the network has been preprocessed this way, the main algorithm can start. `BlockSolve` uses classical propagation techniques, and thus can be integrated into a CSP solver (like Choco [10]). It then inherits all propagation algorithms implemented in the solver. Let us illustrate the behavior of `BlockSolve` on the network of Example 3 before describing how it works.

3.1 Running `BlockSolve` on an example

In this section we run the algorithm on the network of Example 3 whose solution is presented in Figure 2. The following pictures are an execution of `BlockSolve` on this example.

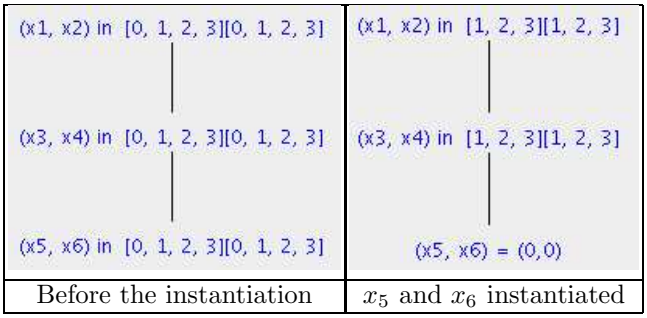
The main idea in `BlockSolve` is to instantiate existential variables of the last block, and to go up to the root instantiating all existential variables. Each assignment v_i of an existential variable x_i can lead to the deletion of inconsistent values of outer variables by propagation. (We illustrate here with FC).

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 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.

At level k, `BlockSolve` looks for a solution to $\mathcal{P}_{k+1}$, and then tries to solve $\mathcal{P}_k$. The first subproblem `BlockSolve` tries to solve is the innermost subproblem. In the example, `BlockSolve` will instantiate variables of the last block (x_5 and x_6) as if the problem was a classical CSP.

First step.



Before the instantiation	x_5 and x_6 instantiated

`BlockSolve` has found an instantiation ($(x_5, x_6) = (0, 0)$) which is consistent with all remaining values of the other variables (thanks to FC filtering). Thus, if there is a consistent assignment for (x_1, x_2) with their remaining values, it is consistent with values that we assigned to (x_5, x_6).

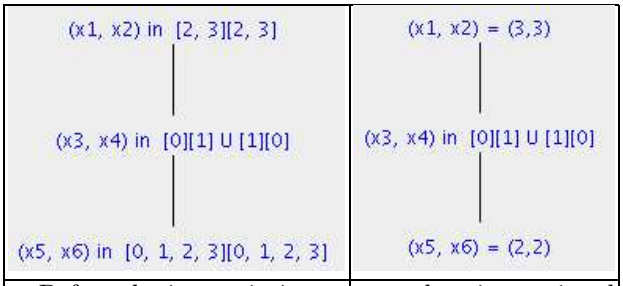
Here, FC removed value 0 for x_3 and x_4, and for x_1 and x_2. It means that `BlockSolve` has not found an instantiation for (x_5, x_6) that is consistent with tuples in $D(x_3) \times D(x_4)$ that contain 0 for x_3 or for x_4. So, in the next step, `BlockSolve` tries to find a partial solution on x_5 and x_6 that is consistent with some of the tuples in $\{0\} \times \{0, 1, 2, 3\} \cup \{1, 2, 3\} \times \{0\}$ for x_3 and x_4 (i.e., x_3 or x_4 is forced to take 0).

Second step.

Before the instantiation	x_5 and x_6 instantiated

In the second step, `BlockSolve` has found the instantiation $(1, 1)$ for (x_5, x_6), which is consistent with some of the remaining tuples of x_3, x_4. This partial solution $(x_5, x_6) = (1, 1)$ is inconsistent with $(x_3, x_4) = (1, 0)$ and $(x_3, x_4) = (0, 1)$. Note that domains of x_1 and x_2 have been reduced as well.

Last step.



Before the instantiation	x_5 and x_6 instantiated

Finally, in the last step, we find the instantiation $(2, 2)$ for (x_5, x_6), which is consistent with the last remaining combinations for x_3, x_4 (namely, $(0, 1)$ and $(1, 0)$). At this point we know that any combination of values on (x_3, x_4) can be extended to x_5, x_6. The subproblem $\mathcal{P}_2$ is solved. During this process the domains of x_1 and x_2 have been successively reduced until they both reached the singleton $\{3\}$. These are the only values consistent with all instantiations found for x_5, x_6 in the three previous steps. These values 3 for x_1 and 3 for x_2 being compatible (there is no constraint between x_1 and x_2), we know that $\mathcal{P}_1$ $(= \mathcal{P})$ is satisfiable. The solution generated by `BlockSolve` is the one depicted in Figure 2.

3.2 Description of `BlockSolve`

In this section, we describe `BlockSolve`, presented as Algorithm 1. This is a recursive algorithm. `BlockSolve`(k) is the call of the algorithm at level k, which itself calls `BlockSolve`$(k + 1)$. In the section above, we saw that it is necessary to keep in memory the tuples of each block. This is saved in two tables: $T_\forall[1..p]$ and $T_\exists[1..p]$ where p is the number of levels. `BlockSolve`(k) modifies the global tables $T_\forall[]$ and $T_\exists[]$ as side-effects. Local tables A and B are used to restore T adequately depending on success or failure in inner subproblems.

`BlockSolve` works as follows: for a level k starting from level 1, we try to solve the subproblem $\mathcal{P}_{k+1}$, keeping in mind that it must be compatible with all constraints in $\mathcal{P}$. If there is no solution for $\mathcal{P}_{k+1}$, it means that current values of existential variables in $block_\exists(k)$ do not lead to a solution. But it may be the case that previous choices in $\mathcal{P}_{k+1}$ provoked the removal of those values in $block_\exists(k)$ that lead to a solution with other values in $\mathcal{P}_{k+1}$. So we try to solve $\mathcal{P}_{k+1}$ again, after having removed tuples on $block_\exists(k)$ that 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 $block_\forall(k)$, exactly as if it was a classical CSP. If success, we remove from $T_\forall[k]$ 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 $block_\forall(k)$. The first call is made with these parameters: $\mathcal{P}_1$ which is the whole problem, and for each level k, the Cartesian products $T_\exists[k]$ and $T_\forall[k]$ of domains of variables in the blocks of level k.

Here we describe the main lines of the algorithm `BlockSolve`.

At line 1, `BlockSolve` returns *true* for the empty problem.

At line 2, we test if there remain tuples in $T_\forall[k]$, that is, tuples of $block_\forall(k)$ for which we have not yet found a partial solution in $\mathcal{P}_k$. If empty, it means we have found a partial solution tree for $\mathcal{P}_k$ and we can go up to level $k - 1$ (line 13). We also test if we have tried all tuples in $T_\exists[k]$. If yes (i.e., $T_\exists[k] = \emptyset$), it means that $\mathcal{P}_k$ cannot be solved. In this case, we will go up to level $k - 1$ (line 13), and we will have to try other values for variables of $block_\exists(k - 1)$ (line 12).

At line 4, a call is made to solve $\mathcal{P}_{k+1}$. If the result is true, $T_\forall[i], \forall i \leq k$ and $T_\exists[i], \forall i \leq k$ are tables of tuples that are compatible with the partial solution of

Algorithm 1: `BlockSolve`

```
   in: P, k, T∀[k], T∃[k]
   in/out: T∀[1..k − 1], T∃[1..k − 1]
   Result: true if there exists a solution, false otherwise
   begin
1      if k > number of levels then return true;
2      while T∀[k] ≠ ∅ ∧ T∃[k] ≠ ∅ do
3          A∀[1..k] ← T∀[1..k]; A∃[1..k] ← T∃[1..k];
4          solved ← BlockSolve(k + 1);
5          B∃ ← T∃[k]; B∀ ← T∀[k];
           if solved then
6              repeat
7                  (inst, T∀^Inc) ←solve-level(k);
8                  if inst then T∀[k] ← T∀^Inc;
9              until T∀[k] = ∅ ∨ ¬inst ;
10             solved ← (B∀ ≠ T∀[k]);
           if solved then
11             T∃[k] ← A∃[k]; T∀[k] ← (A∀[k] \ B∀) ∪ T∀[k];
           else
12             T∃[k] ← A∃[k] \ B∃ ;
               T∃[1..k − 1] ← A∃[1..k − 1]; T∀[1..k] ← A∀[1..k];
13     return (T∀[k] = ∅);
   end
```

$\mathcal{P}_{k+1}$. If the result is false, there is no solution for $\mathcal{P}_{k+1}$ consistent with tuples in $T_\exists[k]$ for existential variables at level k.

At line 5, tuples on $block_\forall(k)$ and $block_\exists(k)$ compatible with $\mathcal{P}_{k+1}$ are saved in $B_\forall$ and $B_\exists$.

From line 6 to line 9, `BlockSolve` tries to instantiate variables of $block_\exists(k)$ consistently with tuples of $T_\forall[k]$, i.e., tuples of values of universal variables for which it has found a partial solution of $\mathcal{P}_{k+1}$. At line 7, `BlockSolve` calls `solve-level(`k`)` which is presented in Algorithm 2. This is a classical CSP solver that instantiates only existential variables at level k ($block_\exists(k)$) so that the instantiation is compatible with all constraints. This CSP solver has to propagate at least FC to ensure that values of outer variables that are inconsistent with the instantiation are removed. This is due to the fact that we limit constraints to constraints on two blocks, with only one variable in the outermost block. Hence we ensure that all variables but the outermost are instantiated when propagating a constraint. Each time it finds an instantiation, `solve-level(`k`)` removes from $T_\forall[k]$ the tuples not consistent with the instantiation of $block_\exists(k)$, and returns the table $T_\forall^{Inc}$ containing these tuples. This is the new value for $T_\forall[k]$ (line 8). `BlockSolve` will indeed try to find another instantiation on $block_\exists(k)$ as long as not all tuples in $block_\forall(k)$ compatible with $\mathcal{P}_{k+1}$ (those in $B_\forall$) have found

Algorithm 2: `solve-level`(k)

in: $\mathcal{P}$,k, $T_\forall[k]$,$T_\exists[k]$
in/out: $T_\exists[1..k-1]$, $T_\forall[1..k-1]$
Result: a couple $(inst, T_\forall^{Inc})$:
begin
  Instantiate variables of $block_\exists(k)$ consistently with $T_\forall[]$ and $T_\exists[]$ and propagate the constraints;
  if *success* **then**
    $T_\forall^{Inc} \leftarrow$ tuples in $T_\forall[k]$ that are inconsistent with the instantiation;
    return *(true,* $T_\forall^{Inc}$*)*;
  else return *(false, —)*;
end

extension to $block_\exists(k)$ or there is no more instantiation which is compatible with $T_\forall[k]$ (i.e., `solve-level` returns *false*).

If we have extended some new tuples in $block_\forall(k)$ since line 5 (test in line 10), then line 11 updates $T_\forall[k]$: there remains to consider all tuples that have been removed by `BlockSolve`$(k+1)$ or `solve-level(`k`)` since the beginning of the loop (line 3). For all these tuples of $block_\forall(k)$, `BlockSolve` has not yet found any partial solution of inner variables consistent with them. Remark that existential variables in $block_\exists(k)$ are restored to their state at the beginning of the loop (line 3).

At line 12, two cases: either *solved* has taken the value $false$ from the call to `BlockSolve`$(k+1)$ or it has taken the value $false$ because `BlockSolve`$(k+1)$ has found a solution but there was no possible instantiation of variables in $block_\exists(k)$ with a tuple of $B_\exists$ compatible with tuples of $B_\forall$. In both cases no tuple in $B_\exists$ can lead to a partial solution while universal variables of $block_\forall(k)$ have their values in $B_\forall$. But there might exist a solution on $B_\forall$ consistent with some other tuples of $A_\exists[k]$ (tuples that have been removed because of choices in `BlockSolve`$(k+1)$). We update $T_\exists[k]$ to contain them.

We should bear in mind that function `solve-level` (Algorithm 2) is a standard CSP solving algorithm that tries to instantiate existential variables of $block_\exists(k)$. If it is possible to instantiate them, *inst* is *true* and $T_\forall^{Inc}$ contains all tuples of $T_\forall[k]$ that are in conflict with the instantiation. Maintaining consistency with outer variables is done as side-effects on tables $T_\exists[]$ and $T_\forall[]$.

`BlockSolve` can give the solution block-tree as in Definition 5. Figure 2 shows the result given. In order to build such a tree, `BlockSolve` takes as parameter a node that corresponds to the existential block of the previous level (or the root for the first level). When solving $\mathcal{P}_{k+1}$, `BlockSolve` produces a tree that is plugged to the current existential node. Plugging the current sub-tree can be done after the call to `solve-level` (line 7), using the instantiation of the variables in $T_\exists[k]$, and the compatible tuples in $T_\forall[k]$. If solving $\mathcal{P}_{k+1}$ fails, nothing is plugged.

3.3 Spatial complexity

`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. But when solving a QCSP, the user usually prefers to obtain a solution tree than an answer: *"yes, it is satisfiable"*. Since a solution tree takes exponential space, any algorithm that returns a solution tree of a quantified network requires exponential space.[1]

`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.

4.1 Coding `BlockSolve`

`BlockSolve` is developed in Java using Choco as constraint library [10]. This library provides different propagation algorithms and a CSP solver. After loading the data of a problem, `BlockSolve` creates tables of sets of tuples for each block and finally launches the main function.

Heuristics The algorithm uses a value ordering heuristic to increase efficiency. Because `BlockSolve` is able to factorize subtrees, the most efficient way to solve a QCSP is to minimize the number of subtrees during the search. One way to accomplish this is to select the value v of variable x that is compatible with the largest set of tuples of outer blocks. In order to determine which value is the best according to this criterion, the solver instantiates x to all its values successively. For each value it propagates the instantiation to others domains and computes the number of branches it is in (i.e., the number of compatible tuples in outer blocks). The larger the better.

4.2 The random problem generator

Instances of QCSP presented in these experiments have been created with a generator based on that used in [3]. In this model, problems are composed of

[1] We can point out that a solution can be returned in polynomial space if we allow interactive computation: the values of the first existential block are returned, then, based on the values chosen adversarially for the first universal block, values of the second existential block are returned, and so on.

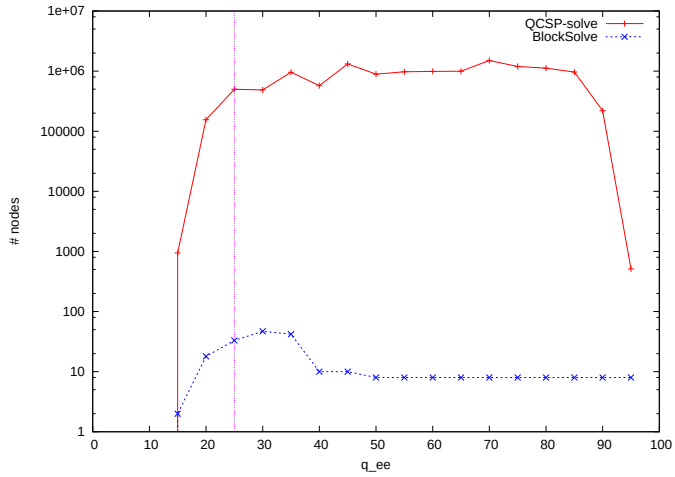


Fig. 3. Number of nodes for $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = .50$ where $q_{\exists\exists}$ grows. (cross-over point at $q_{\exists\exists} = .25$)

three blocks, the first is an existential block. It takes seven parameters as input $< n, n_\forall, n_\exists, d, p, q_{\forall\exists}, q_{\exists\exists} >$ where n is the total number of variables, $n_\forall$ is the number of universal variables, $n_\exists$ is the number of existential variables in the first block, d is the size of domains of variables (the same for each variable), p is the number of binary constraints as a fraction of all possible constraints. All constraints are $\forall x_i \exists x_j \ c_{ij}$ constraints or $\exists x_i \exists x_j \ c_{ij}$ constraints, other type of constraints that can be removed during the preprocessing are not generated. $q_{\forall\exists}$ and $q_{\exists\exists}$ are the looseness of constraints, i.e., the number of *goods* as a fraction of all possible couples of values. To avoid the flaw with which almost all problems are inconsistent, constraints $\forall x_i \exists x_j \ c_{ij}$ have very few forbidden couples.

We extended this generator to allow more than 3 blocks. We added an eighth parameter, $b_\forall$, that is the number of universal blocks. Variables are sequenced as follow: $n_\exists$ existential variables followed by $n_\forall$ universal variables, then again $n_\exists$ existential variables followed by $n_\forall$ universal variables.

4.3 Results

Now we present some results on randomly generated instances created by the generator. For each experiment, 100 instances were generated for each value of $q_{\exists\exists}$, from 0.05 to 0.95.

The first three figures are from experiments on problems with these characteristics: each instance contains 15 variables, one block of 7 universal variables. Figure 3 shows the results in terms of number of nodes explored. In the leftmost part of the figure, both algorithms detect inconsistency before exploring any node. As we can see, `BlockSolve` traverses far less nodes than QCSP-Solve. No-

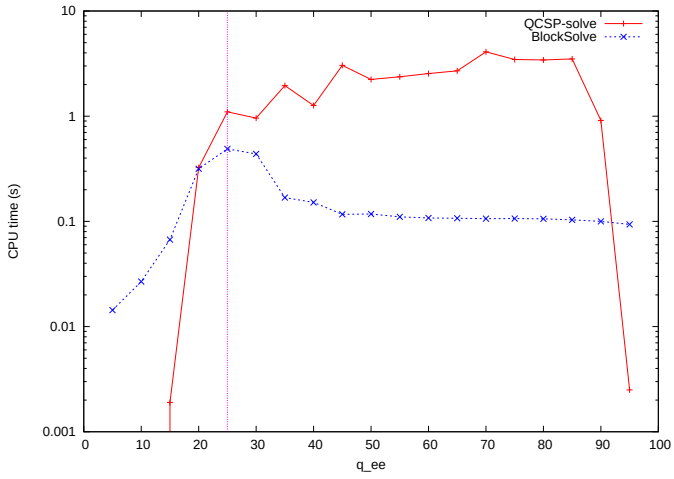


Fig. 4. cpu time for $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = .50$ where $q_{\exists\exists}$ grows. (cross-over point at $q_{\exists\exists} = .25$)

tice that `BlockSolve` explores only existential nodes, not universal ones. QCSP-Solve seems to have difficulties to solve problems that have solutions. On under-constrained problems, `BlockSolve` finds a solution without any backtrack. This means that for easy problems, there exists an instantiation of the innermost existential block that is consistent with all tuples of the universal block.

Figure 4 shows the results in term of CPU time. Comparing it to Figure 3, it is clear that `BlockSolve` takes a lot of time for exploring one node. This is because at each node `BlockSolve` looks for the best value for matching more tuples in outer universal blocks. This heuristic is quite long to compute compared to what QCSP-Solve does at a node. Note that QCSP-Solve determines faster than `BlockSolve` that a problem is inconsistent ($q_{\exists\exists} < .25$), but `BlockSolve` finds a solution much faster when it exists ($q_{\exists\exists} > .25$). For very high values of $q_{\exists\exists}$ ($> .90$), QCSP-Solve is more efficient than `BlockSolve`.

It is interesting to see that `BlockSolve` is more stable than QCSP-Solve, as shown in Figure 5. In this figure, each point represents an instance of problem. We see that in the satisfiable region, it is hard to predict how much time will take QCSP-Solve to solve an instance.

We ran both algorithms on instances that have more than three blocks. Figure 6 presents results for instances that have five blocks ($\exists\forall\exists\forall\exists$) of five variables each (left graph), and instances that have seven blocks of four variables each (right graph). These experiments show that the general behavior of both algorithms looks similar whatever the number of levels in the problem. In unsolvable problems, QCSP-Solve detects inconsistency before `BlockSolve`, whereas for solvable instances QCSP-Solve takes as much time as at the cross-over point (i.e., where there are as many satisfiable instances as unsatisfiable ones).

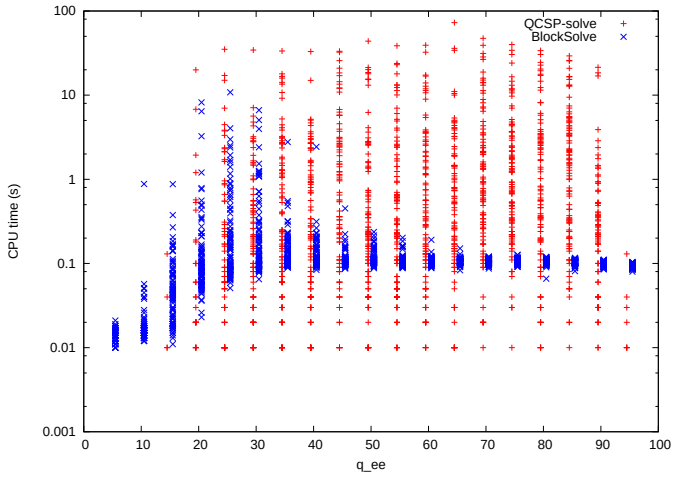


Fig. 5. Scattered plot for cpu time on $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30$, $q_{\forall\exists} = .50$

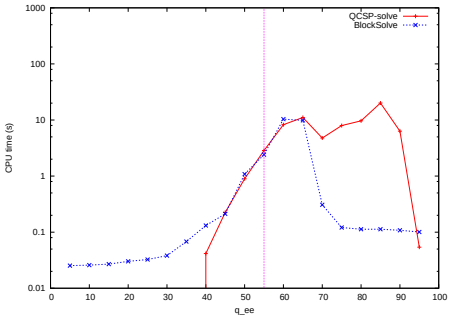
5 Conclusions

In this paper we presented `BlockSolve`, a bottom-up QCSP solver that uses standard 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.

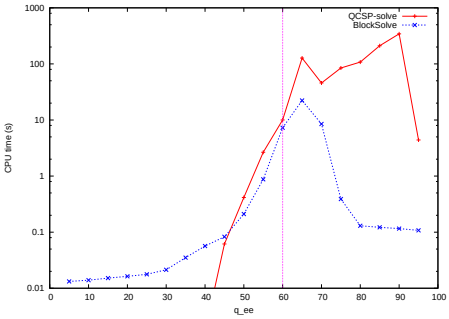
Future work will focus on improving time efficiency of `BlockSolve`. Great improvements can probably be obtained by designing heuristics to efficiently prune subtrees that are inconsistent. Furthermore, most of the cpu time is spent updating and propagating tables of tuples on blocks. Finding better ways to represent them could significantly decrease the cpu time of `BlockSolve`. The current implementation of `BlockSolve` being far from being optimized, this leaves a lot of space for significant improvements. Finally, we plan to generalize `BlockSolve` to global constraints.

Acknowledgements

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



cpu time for $n = 25, n_\forall = 5, n_\exists = 5$, $b_\forall = 2, d = 8, p = 20, q_{\forall\exists} = .50$ (cross-over point at $q_{\exists\exists} = .55$)



cpu time for $n = 28, n_\forall = 4, n_\exists = 4$, $b_\forall = 3, d = 8, p = 20, q_{\forall\exists} = .50$ (cross-over point at $q_{\exists\exists} = .60$)

Fig. 6. Problems with 25 variables and 5 blocks (left) and 28 variables and 7 blocks (right).

References

1. Biere, A.: Resolve and expand. In: Proceedings SAT'04, Vancouver BC (2004)
2. Pan, G., Vardi, M.: Symbolic decision procedures for QBF. In: Proceedings CP'04, Toronto, Canada (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. Bordeaux, L., Montfroy, E.: Beyond NP: Arc-consistency for quantified constraints. In: Proceedings CP'02, Ithaca NY (2002) 371–386
5. Mamoulis, N., Stergiou, K.: Algorithms for quantified constraint satisfaction problems. In: Proceedings CP'04, Toronto, Canada (2004) 752–756
6. Stergiou, K.: Repair-based methods for quantified csps. In: Proceedings CP'05, Sitges, Spain (2005) 652–666
7. Haralick, R., Elliott, G.: Increasing tree seach efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
8. Fargier, H., Lang, J., Schiex, T.: Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. In: Proceedings AAAI'96, Portland OR (1996) 175–180
9. Freuder, E., Hubbe, P.: Extracting constraint satisfaction subproblems. In: Proceedings IJCAI'95, Montréal, Canada (1995) 548–557
10. Choco: A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. URL: http://choco-solver.net (2005)