Solution Directed Backjumping for QCSP

Fahiem Bacchus[1] and Kostas Stergiou[2]

[1] Department of Computer Science, University of Toronto, Canada.
fbacchus@cs.toronto.edu
[2] Department of Information and Communication Systems Engineering,
University of the Aegean, Greece.
konsterg@aegean.gr

Abstract. In this paper we present new techniques for improving backtracking based Quantified Constraint Satisfaction Problem (QCSP) solvers. QCSP is a generalization of CSP in which variables are either universally or existentially quantified and these quantifiers can be alternated in arbitrary ways. Our main new technique is solution directed backjumping (SBJ). In analogue to conflict directed backjumping, SBJ allows the solver to backtrack out of solved sub-trees without having to find all of the distinct solutions normally required to validate the universal variables. Experiments with the solver QCSP-Solve demonstrate that SBJ can improve its performance on random instances by orders of magnitude. In addition to this contribution, we demonstrate that performing varying levels of propagation for universal vs. existential variables can also be useful for enhancing performance. Finally, we discuss some techniques that are technically interesting but do not yet yield empirical improvements.

1 Introduction

In this paper we present new techniques for improving the performance of solvers for Quantified Constraint Satisfaction Problems (QCSPs). QCSPs are an extension of standard constraint satisfaction problems (CSPs) that can compactly represent a wider range of problems than the standard CSP formalism. Whereas all of the variables of a CSP are implicitly existentially quantified, QCSPs also allow variables to be universally quantified. Furthermore, universal and existential variables can be alternated in arbitrary ways in a QCSP. From a theoretical point of view, these added features make QCSPs PSPACE complete; any problem in PSPACE can be encoded as a polynomially sized (poly-sized) QCSP. CSPs, on the other hand, are NP complete; any problem in NP can be encoded as a poly-sized CSP. It is known that both NP $\subseteq$ PSPACE and co-NP $\subseteq$ PSPACE, and it is widely believed that these containments are proper, i.e., that there are problems outside of NP and co-NP that still lie in PSPACE. Such problems will not have a poly-sized CSP representation, but will have a poly-sized QCSP representation.

From a practical point of view, this difference means that if effective QCSP solvers can be developed they would enable a wide range of practical applications that lie beyond the reach of CSP solvers. Even though CSP and QCSP solvers both have the same exponential worst case complexity, experience has shown that solvers can often achieve reasonable run times in practice. The real issue separating CSP and QCSP solvers lies in size of the problem representation, i.e., the size of the input. If NP $\neq$ PSPACE, then there will exist problems for which a CSP formalization will always be exponentially

larger than the equivalent QCSP representation. Thus a CSP solver could not even get started on the problem—the problem would contain too many variables and constraints. The QCSP representation, on the other hand, would still be polynomial in size and thus potentially solvable by a QCSP solver if that solver was able to achieve reasonable run times in practice.

A good illustration of this point comes from the area of circuit diagnosis. In [1] an innovative application of quantified boolean formulas (QBF is a restricted form of QCSP), for diagnosing hardware circuits is given. The key feature of the approach is that the QBF encoding is many times smaller than the equivalent SAT encoding (SAT is a restricted form of CSP). In fact, in experiments the SAT encoding grew so large that it could no longer be solved by existent SAT solvers, whereas the QBF encoding remained compact and was solvable by existent QBF solvers. Other applications of QCSP come from areas like conditional planning and planning under incomplete knowledge [14], non-monotonic reasoning [9], and hardware verification and design [1, 7].

In this paper we present new techniques for improving backtracking based QCSP solvers. Our main contribution is to demonstrate how the technique of cubes, utilized in QBF solvers [12, 16], can be extended to QCSPs in a technique we call solution directed backjumping (SBJ). SBJ allows the solver to backtrack intelligently after having encountered a solution. SBJ subsumes and improves on the technique of solution directed pruning (SDP) [11], in an manner analogous to how conflict directed backjumping (CBJ) extends ordinary backjumping. In particular, SBJ computes information that can be used at internal nodes of the tree rather than just at the leaf nodes. Experiments demonstrate that SBJ can improve performance by orders of magnitude.

In addition to SBJ, we demonstrate that performing varying levels of propagation for universal vs. existential variables can enhance performance. In particular, we show that enforcing very high levels of consistency on universal variables can pay off, as detecting a locally inconsistent value of a universal variable immediately forces a backtrack. We also discuss validity pruning, a technique that can be used to prune the domains of universally quantified variables. Our current empirical investigations with random problems indicate that validity pruning does not yield significant improvements. Nevertheless, it has the potential to be useful in other types of problems.

This paper is structured as follows. Section 2 gives the necessary definitions and background on QCSPs. Section 3 describes methods to enhance propagation in QCSPs, while in Section 4 we present SBJ, a method to enhance intelligent backtracking in QCSPs. Section 5 gives experiments results demonstrating the efficiency of the proposed methods. Finally, in Section 6 we conclude.

2 Background

We are concerned here with QCSPs defined over finite valued variables. Let $V = \{v_1, \ldots, v_n\}$ be a set of variables. Each variable v_i has an associated finite domain of values $dom[v_i]$. We write $v = d$ if the variable v has been assigned the value d always requiring that $d \in dom[v]$, i.e., a variable can only be assigned a value from its domain. A constraint c is a function from a subset of the variables in V to $\{\textbf{true}, \textbf{false}\}$ (1/0). This subset of variables is called the **scope** of c, $scope(c)$, and the cardinality of this set is the **arity** of the constraint. Any tuple τ of values for the variables in $scope(c)$

will be mapped by c to **true** or **false**. If $c(\tau) =$ **true**, τ is said to be a **satisfying** tuple of c, else $c(\tau) =$ **false** and τ is a falsifying tuple of c. We will consider τ to be a set, and write $v = d \in \tau$ if v has the value d in τ.

A conjunction of constraints $C = c_1 \wedge \cdots \wedge c_m$, their associated variables $V = \bigcup_{j=1}^{m} scope(c_j)$, and domains for these variables $D = \{dom[v] : v \in V\}$ define a standard CSP, $C[D]$, which forms the **body** of a QCSP. Let τ be a tuple of values for all of the variables in V, and for each constraint $c_i \in C$ let τ_i be the subset of τ containing the values assigned to variables in $scope(c_i)$. τ is a **solution** of the body $C[D]$ if each τ_i is a satisfying tuple of c_i.

Definition 1 (QCSP) *Given a body $C[D]$ a Quantified Constraint Satisfaction Problem (**QCSP**) is a formula of the form $Q.C[D]$ where Q is a quantifier prefix consisting of the variables of $\bigcup_{c_j \in C} scope(c_j)$ arranged in some sequence and each proceeded by either an existential ($\exists$) or universal quantifier ($\forall$).*

For example

$$\forall v_1, \exists v_2, \forall v_3, \exists v_4 . c_1(v_1, v_2) \wedge c_2(v_1, v_3, v_4) \wedge c_3(v_2, v_3, v_4)$$
$$[\{dom[v_1] = \{a, b\}, dom[v_2] = \{a, b, c\}, dom[v_3] = \{a\}]$$

is a QCSP with the quantifier prefix $\forall v_1, \exists v_2, \forall v_3, \exists v_4$. This QCSP asserts that for all values of v_1 there exists a value of v_2 (perhaps dependent on the particular value of v_1) such that for all values of v_3 there exists a value of v_4 (perhaps dependent on the particular values of v_1, v_2 and v_3) such that all the constraints c_1, c_2 and c_3 are satisfied.

A **quantifier block** qb of Q is a maximal contiguous subsequence of Q such that every variable in qb has the same quantifier type. For two quantifier blocks qb_1 and qb_2 we say that $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in Q. Each variable v in Q appears in some quantifier block $qb(v)$ and we say that $v_1 \leq_q v_2$ if $qb(v_1) \leq qb(v_2)$ and $v_1 <_q v_2$ if $qb(v_1) < qb(v_2)$. We also say that v is **universal** (**existential**) if its quantifier in Q is $\forall$ ($\exists$).

A QCSP makes an assertion that is either true or false. The assertion made by $Q.C[D]$ is true iff $Q.C[D]$ has a Q-Model. A Q-Model for a $Q.C[D]$ is a tree in which each node except for the root is labeled by a variable assignment and that is subject to the following conditions. Let n and m be any two nodes in the tree such that n's label is $x = a$ while m's label is $y = b$.

1. If n is an ancestor of m, then it must be the case that $x \leq_q y$. That is, the sequence of assignments along any path from the root to a leaf must respect the ordering of the quantifier blocks.
2. If x is universally quantified, then n must have $k - 1$ siblings where k is the size of $dom[x]$. For each value $d \in dom[x]$, n or one of its $k - 1$ siblings must be labeled by $x = d$. On the other hand if x is existentially quantified, then n has no siblings.
3. The tuple of assignments along any path from the root to a leaf node must be a solution to $C[D]$.

Hence in a Q-Model there is a path for every possible setting of the universal variables in Q each of which is a CSP solution to the body of the QCSP. Thus a Q-Model of a QCSP containing k universal variables will contain $2^{O(k)}$ solutions to the body. From this definition it can be seen that any CSP can be viewed as a QCSP with all its variable

existentially quantified. The Q-Models of such existential only QCSPs contain only a single path, and determining the truth of such QCSPs (the existence of a Q-Model) is equivalent to determining if the CSP has a solution.

The **reduction** of $C[D]$ by an assignment $v = d$, $C[D]\big|_{v=d}$ is the new body obtained by removing from the domain of v all values not equal to d (i.e., reducing $dom[v]$ set to the singleton set $\{d\}$). We can also reduce the body by pruning a value from the domain of a variable: $C[D]\big|_{v\neq d} = C[(D\big(dom[v])/(dom[v]-d)\big)]$. The reduction by a set of assignments or value prunings is defined as the sequential application of these reductions. Note that if any variable domain is reduced to the empty set, then the QCSP is false. It cannot have a Q-Model as every Q-Model must assign every variable a value from its domain along each path.

Proposition 1 *Let v be a variable and d be some value in its domain. If v is universal then $Q.C[D] \Rightarrow Q.C[D]\big|_{v\neq d}$. If v is existential then $Q.C[D]\big|_{v\neq d} \Rightarrow Q.C[D]$.*

Proof: If v is universal and $Q.C[D]$ has a Q-Model then so does $Q.C[D]\big|_{v\neq d}$: we simply remove all subtree rooted by nodes labeled $v = d$ from $Q.C[D]$'s Q-Model. If v is existential then any Q-Model of $Q.C[D]\big|_{v\neq d}$ is a Q-Model of $Q.C[D]$.

A common way of solving a QCSP is via backtracking search. In its most basic form such a search works much like CSP backtracking search except for two additional conditions: (1) the variable ordering along any branch must respect the ordering of the quantifier blocks (although it is free to dynamically reorder the variables within each block), and (2) for every universal variable v the search needs to solve for every value in $dom[v]$.

The search tries to find a Q-Model: a successful run verifies that a Q-Model exists by traversing a Q-Model during its search while a failed run has tried to traverse all possible Q-Models thus verifying that one does not exist. In particular, at any node n the search tree that has been reached by making the sequence of assignments $\pi_k = \langle v_1 = d_1, \ldots, v_k = d_k\rangle$, the search in the subtree below n attempts to find a Q-Model for $Q.C[D]\big|_{\pi_k}$. Thus at the root the search attempts to find a Q-Model for the original problem $Q.C[D]$.

The key to making backtracking search for QCSPs effective is by developing techniques that allow unsuccessful subtrees to be refuted more efficiently, and successful subtrees to be verified more efficiently. Efficient refutation of unsuccessful subtrees is also the goal in backtracking CSP solvers, but here we aim to exploit the additional structure of QCSPs to develop better methods for achieving this goal. Efficient verification of successful subtrees, on the other hand, has no analogue in CSP solvers which typically can stop as soon as a single solution is found. With a QCSP however, a successful subtree has an exponential number of solutions and finding each of these would be very slow. Here again our aim is to exploit the additional structure of QCSPs to develop methods for verifying that all of these solutions exist without having to actually find each one.

In the sequel we report on some new methods for achieving these two goals as well as on our empirical evaluation of their effectiveness. From here on we will confine our attention to QCSPs with constraints of arity at most two. It can be noted that any QCSP with non-binary constraints can be converted to an equivalent QCSP containing

only binary constraints by applying the hidden variable transformation (see e.g., [2]) to convert the body to a binary CSP and then adding all of the newly introduced hidden variables as new existential variables to the end of the quantifier prefix. Whether or not this is a effective way of dealing with non-binary constraints is a topic for future work. The alternative of dealing directly with non-binary constraints poses some considerable additional formal and practical challenges and is also a topic for future work.

3 Propagation

Our first techniques arise from the standard idea of constraint propagation. These techniques use the constraints of the QCSP body to provide additional information that can simplify the task of searching the subtree below the current node.

3.1 Detecting Inconsistent Values

An assignment $v = d$ is **inconsistent** for $Q.C[D]$ if it does not appear in *any* Q-Model of $Q.C[D]$. If $v = d$ is inconsistent and v is existential then $Q.C[D] \equiv Q.C[D]\big|_{v \neq d}$: any Q-model for $Q.C[D]$ must also be a Q-Model for $Q.C[D]\big|_{v \neq d}$ since it cannot contain $v = d$, while Prop. 1 supplies the opposite direction. On the other hand if v is universal then $Q.C[D]$ is **false**: any Q-Model must contain $v = d$.

Of course it is in general hard to detect inconsistent values, but as with CSPs various local checks can be performed that detect some but not all inconsistent values. Such checks can be done at every node n of the search (including prior to search at the root). In particular, if an inconsistent existential value is detected it can be pruned before searching the subtree below n, and if an inconsistent universal value is detected the search can immediately backtrack from n.

Since every path in a Q-Model is a standard CSP solution to the body, any standard CSP technique for detecting inconsistent values can be used: any value inconsistent for the body cannot appear in any Q-Model. Additionally, we can do better than this by exploiting the additional structure of QCSPs. In particular, as shown in [6, 13], arc consistency (AC) can be extended to QCSPs to support the detection of values that are inconsistent for the QCSP even though they are not inconsistent for the CSP body. AC for binary QCSPs has been implemented in the QCSP-Solve system that we employ in our empirical evaluations. QCSP-Solve uses AC only as a preprocessing step (i.e., at the root), as FC (forward checking) seems to be more cost effective during search [11].

The key feature of AC for QCSPs is that it allows many of the constraints of the body to be removed at the root. In particular, the only constraints $c(x, y)$ that remain in the problem after AC preprocessing are those where both x and y are existential, and those where x is universal, y is existential, and $x <_q y$ (see [11] for more details).

Pruning inconsistent values improves the efficiency of search in the subtree below, but local consistency checking has its greatest impact when it allows us to avoid that search altogether. This happens when either all values of an existential are pruned or a single value of a universal is pruned. It is more likely that local propagation can prune a single universal value than all values for some existential. Hence, it can be worth while to expend more effort checking for inconsistency universal values. This intuition already appears to some extent in the QCSP-Solve system via its FC1 and MAC1 propagation. In these propagation methods, whenever a universal variable x is to be branched on, before descending deeper in the search tree all of its possible

values are tried and FC or AC performed after each trial assignment. If any of these assignments yield a contradiction the algorithm can immediately backtrack. This extra work on universals was shown to be cost effective in the experiments of [11].

Our first new technique is to further investigate the technique of doing more work on the consistency checking of universals. In particular, we investigate applying a different and stronger level of consistency checking on universals and a weaker, and thus cheaper, level of consistency on existentials, in addition to the technique of checking all universal values prior to descending deeper, used in [11].

3.2 Strong Levels of Consistency on Universals

Like QCSP-Solve after any existential is assigned we perform FC. But further to QCSP-Solve we also check all future universal variables to ensure that they are arc consistent in all of the constraints they participate in. Like QCSP-Solve if a universal is about to be assigned we check each of its values first. But further to QCSP-Solve we check each value with a much higher level of local consistency than FC. The particular form of local consistency we found to be effective is a mixture of path consistency (PC) and max restricted path consistency (maxRPC). If any value of the universal fails this local consistency test we backtrack. If they all pass this test, we then assign the universal a value and then perform FC followed by enforcing AC on all constraints involving a future universal. Hence, we have two changes from QCSP-Solve: (1) after each instantiation we check that the future universals are AC in their constraints, and (2) checking a higher level of consistency on all values of a universal prior to assigning it a specific value.

Now we specify more precisely the local consistency test we employ on the values of an about to be assigned universal. A pair of values (d_i, d_j), $d_i \in dom[v_i]$ and $d_j \in dom[v_j]$, is *path consistent* (PC) iff the two values are compatible and for any third variable v_k there exists a value $d_k \in dom[v_k]$ that is compatible with both d_i and d_j. A value $d_i \in dom[v_i]$ is *max Restricted Path Consistent* (maxRPC) [8] iff for any variable v_j constrained with v_i there exists a value $d_j \in dom[v_j]$ that is compatible with d_i and has the following property: for any third variable v_k, there exists a value $d_k \in dom[v_k]$ that is compatible with both d_i and d_j. In this case we say that d_j is a maxRPC-support of d_i. In other words d_i is maxRPC if it is a member of *some* path consistent pair in every constraint it participates in while path consistency ensures that every pair d_i of is path consistent. When during search we are about the assign the universal v_i, after having some set of assignments π, the local consistency test we employ is specified in Figure 1.

In Figure 1 when a universal v_i is reached during search we check that each of its values d_i has a maxRPC support in the domain of each existential it is constrained with, and that d_i is path consistent with all future universals. AC preprocessing ensures there are no constraints over two universals, thus to check the consistency of pairs of universal values we must consider the existentials they are jointly constrained with; hence our use of path consistency.

The following example demonstrates how the application of PC and maxRPC prunes the search space upon reaching a universal variable.

function `maxRPC+PC_Propagation` $(Q.C[D], \pi, v_i)$
1: **for** each value $d_i \in dom[v_i]$
2: **for** each unassigned existential variable v_j constrained with v_i
3: **if** d_i has no maxRPC-support in $dom[v_j]$
4: **then return** `FAIL`
5: **for** each unassigned universal variable v_j
6: **for** each value $d_j \in dom[v_j]$
7: **if** (d_i, d_j) is not path consistent
8: **then return** `FAIL`

Fig. 1. Strong propagation on universal variables.

Example 1 Consider the QCSP

$$\exists v_1, \forall v_2, \exists v_3, \forall v_4, \exists v_5.(v_1 \neq v_5 - 2 \wedge v_2 = v_3 \wedge v_2 \neq v_5 \wedge v_3 \neq v_5 - 1 \wedge v_4 = v_5)$$
$$[dom[v_1] = dom[v_2] = dom[v_3] = dom[v_4] = \{0, 1\}, dom[v_5] = \{0, 1, 2\}]$$

A chronological backtracking algorithm that applies PC and maxRPC upon reaching a universal will solve the problem as follows. Variable v_1 is assigned value 0. Forward checking removes 2 from $dom[v_5]$. The next variable v_2 is a universal. We will now call the function of Figure 1 to apply PC and maxRPC on v_2's values. v_3 is existential and is constrained with v_2. Therefore, we check if value 0 of v_2 has a maxRPC-support in $dom[v_3]$ (line 3). The only value compatible with 0 in $dom[v_3]$ is 0 and there is no value in $dom[v_5]$ that is compatible with both $0 \in dom[v_2]$ and $0 \in dom[v_3]$. Therefore, value 0 of v_2 is not maxRPC and the algorithm immediately backtracks and assigns 1 to v_1. Again the function of Figure 1 is called. Value 0 of v_2 now has a maxRPC-support in $dom[v_3]$ (value 0), because 2 has been restored to $dom[v_5]$ and it is compatible with both $0 \in dom[v_2]$ and $0 \in dom[v_3]$. v_4 is a universal so we now apply PC on its values. That is, we check if the values of v_4 have a support in v_5 that is also a support for value 0 of v_2 (lines 6-8). This is not the case for value 0 of v_4 and therefore the algorithm backtracks and determines that the problem is false.

3.3 Detecting Valid Values

In QCSPs a duality exists between universal and existential variables that manifests itself in various aspects of the processing that can be done when solving a QCSP. With respect to detecting inconsistent values the dual notion is detecting valid values.

An assignment $v = d$ is **valid** for a constraint c if for every tuple τ of assignments to the variables in $scope(d)$ with $v = d \in \tau$ we have $c(\tau) =$ **true**. In other words the assignment $v = d$ renders c vacuous. We say that an assignment $v = d$ is valid for a conjunction of constraints C if it is valid for every constraint in C that has v in its scope.

This notion of validity corresponds both to that defined in [3] and to the notion of **purity** defined in [11]. It is also related to notions described in [5]. A useful fact from [3] is that validity is the dual of inconsistency with respect to GAC. That is, $v = d$ is valid for a constraint c if and only if $v = d$ is GAC inconsistent for $\neg c$, where $\neg c$ is the negation of c. That is, $scope(\neg c) = scope(c)$ and for any tuple of assignments τ, $\neg c(\tau) =$ **true** iff $c(\tau) =$ **false**.

If $v = d$ is valid for C in the QCSP $Q.C[D]$ and v is existential then $Q.C[D] \equiv Q.C[D]\big|_{v=d}$: we can assign v the value d. In particular, if $Q.C[D]$ has a Q-Model, then

we can replace every assignment to v in that Q-Model by $v = d$. Since $v = d$ is valid this change cannot cause any constraint to be violated, hence the modified Q-Model is still a Q-Model for $Q.C[D]\big|_{v=d}$. Prop.1 provides the opposite direction. On the other hand if v is universal then $Q.C[D] \equiv Q.C[D]\big|_{v\neq d}$: we can prune d from $dom[v]$. In this case if $Q.C[D]\big|_{v\neq d}$ has a Q-Model we can add the node $v = d$ as a new sibling to all sets of siblings labeled by the other assignments to v and then simply copy the subtree below one of these other assignments to create subtree below $v = d$. Since $v = d$ is valid the other assignment's subtree will continue to be a tree of solutions under $v = d$. This modified Q-Model is a Q-Model of $Q.C[D]$. Prop.1 provides the opposite direction.

Validity was previously utilized in QCSP-Solve by waiting until a variable was about to be assigned. At that point the values of the variable would be checked to see if any of them were valid (pure). For an existential the valid value would immediately be assigned, and for a universal the valid values would be pruned, in accord with the above observations.

An alternative to the approach of QCSP-Solve is to do validity propagation. In particular, instead of waiting until a variable is about to be assigned one could detect valid values of future variables and prune or assign them dependent on their type. Validity propagation can be achieved by exploiting the relationship cited above between GAC (AC) on the negation of a constraint and validity. That is, it is fairly easy to alter AC lookahead to detect valid values of future variables by running AC on the negations of the constraints.

It should be noted that validity propagation does not affect the size of the search tree: a valid value of a future variable will still be exploited even if it is only detected at the time the variable is about to be assigned. Potentially, it can be more efficient to determine that a value is valid once near the top of the search tree, rather than each time the variable is to be assigned. On the other hand, one could waste time detecting valid values for variables that are never reached because an inconsistency is found before they are instantiated. The main potential benefit of validity propagation over future variables lies in the fact that it dynamically alters the size of the variable domains; potentially differently along different branches of the search tree. As noted above, although the order in which variables are instantiated is restricted by the ordering of the quantifier blocks, within a quantifier block the variable ordering can be selected heuristically. Hence, validity propagation could potentially provide "within a block" dynamic ordering with useful information about varying domains sizes.

We implemented validity propagation and used it in conjunction with dynamic variable ordering within quantifier blocks. Our experimental results were disappointing, but we only tested random problems. Potentially this technique could be useful on other QCSPs.

4 Intelligent Backtracking

Our second set of techniques arises from the idea of keeping track of the reasons a path failed or succeeded so that irrelevant variables can be backtracked over. QCSP-Solve already utilizes conflict directed backjumping (CBJ), as described in [11]. Hence, when backtracking from a failure node irrelevant variables can be skipped over. However, CBJ

does not support intelligent backtracking from successful nodes. Extending intelligent backtracking so that it can be applied after success is achieved by our new technique of solution directed backjumping (SBJ).

4.1 Solution Directed Backjumping (SBJ)

In QBF solvers cube learning is an technique used to backtrack from successful nodes [12, 16]. Cubes are computed at solution leaves of the search tree by identifying a subset of the assigned literals sufficient to satisfy all clauses of the QBF. The aim is to identify universal variables whose setting was irrelevant to the discovered solution. Potentially those variables can be backtracked over without having to test if their other value is solvable.

In a QBF solver the leaf cubes (cubes computed as solution leaf nodes) support backtrack to the deepest universal they contain, and at internal nodes cubes computed for each setting of a universal can be combined to support further non-chronological backtracking. Since a successful subtree in QBF (or QCSP) can contain an exponential number of solutions, backtracking out of such subtrees by using cubes can provide a considerable performance improvement. For example, if at a any node a cube consisting entirely of existential literals is computed, then the search can immediately terminate.

In QCSPs however a straight forward application of this idea is not effective. In particular it is hardly ever the case that a universal variable is completely irrelevant to the solution found. Rather, the solution found at a leaf node might continue to be a solution under some other settings of the universal variable, but not under other settings. Hence the idea behind SBJ is to keep track of the values of the universals that are verified by the current solution so that on backtrack these values need not be verified again. It is however slightly easier to formalize SBJ as keeping track of the complement of the verified values.

Definition 2 (QCSP Cube) *Let qbe be a set containing (a) a set of existential assignments $(v = a)$ and (b) for each universal variable v a set of values $uncovered[v] \subset dom[v]$. Let $C[D]\big|_{qbe}$ be the reduction of $C[D]$ by $v = a$ for each existential assignment $(v = a) \in qbe$ and by $v \neq d$ for each $d \in uncovered[v]$ for each universal variable v. The set qbe is a* ***cube*** *iff $Q.C[D]\big|_{qbe}$ is* ***true*** *(i.e., has a Q-Model).*

We use the convention of omitting mention of the set $uncovered[v]$ from a cube if it is empty, and we say that the universal variable v_u **is in a cube**, qbe, if $uncovered[v_u] \in qbe$ (i.e., $uncovered[v_u] \neq \emptyset$). An existential assignment $v_e = a \in qbe$ is called **tailing** if for all universal variables $v_u \in qbe$ we have $v_u <_q v_e$.

Observation 1 *If $v_e = a$ is a tailing existential in a cube qbe then $qbe - \{v_e = a\}$ is also a cube. That is, tailing existential assignments can be removed from a cube.*

Proof: $Q.C[D]\big|_{qbe}$ is **true** (by definition) and $Q.C[D]\big|_{qbe} \Rightarrow Q.C[D]\big|_{qbe-\{v_e=a\}}$ (Prop 1). Hence $Q.C[D]\big|_{qbe-\{v_e=a\}}$ is also **true** and $qbe - \{v_e = a\}$ is a cube.

Figure 2 gives the algorithm for computing a QCSP cube at a solution leaf. Let π be the sequence of assignments made on the path to this leaf node. (π satisfies all of the constraints of the body). In this algorithm $\pi(v_i{=}d)$ denotes the set of assignments π modified so that v_i is now assigned the value d. The algorithm computes $uncovered[v_i]$ for each universal variable v_i; this is the set of values of v_i that are incompatible with

function `ComputeLeafCube` $(Q.C[D], \pi)$
1: qbe = the assignments to the existential variables in π
2: **for** each universal variable v_i
3: $\quad uncovered[v_i] = \{\}$
4: $\quad$ **for** each $d \in dom[v_i]$
5: $\quad\quad$ **if** $\pi(v_i{=}d)$ does not satisfy C
6: $\quad\quad\quad uncovered[v_i] = uncovered[v_i] \cup \{d\}$
7: $\quad qbe$ = $qbe \cup \{uncovered[v_i]\}$
8: qbe = remove tailing existentials from qbe

Fig. 2. Computing a QCSP cube at a solution leaf node.

the current solution. Note that $uncovered[v_i]$ can never contain v_i's current value (the condition on line 5 cannot be satisfied since π satisfies all constraints).

Proposition 2 *The set qbe returned by* `ComputeLeafCube` *is a cube.*

Proof: Consider qbe before tailing existentials are removed (line 8). At this point qbe contains an assignment for every existential variable. We must show that $Q.C[D]\big|_{qbe}$ has a Q-Model. Such a Q-Model will be a tree with paths for every combination of assignments to the universal variables not in the $uncovered$ sets. Construct such a tree by assigning the existential variables along every path its value in qbe. Due to our restriction to binary constraints and preprocessing of the problem $C[D]$ only contains constraints $c(v_u, v_e)$ between a universal and an existential and constraints $c(v_{e_1}, v_{e_2})$ between two existentials. Since the existential assignments in qbe came from a solution π all constraints between two existentials are satisfied. Furthermore, line 5 ensures that all constraints between a universal and an existential are satisfied by any universal value not in the $uncovered$ sets. Hence each path in this tree is a solution to $C[D]$, and $Q.C[D]\big|_{qbe}$ has a Q-model (is **true**). By the previous observation qbe remains a cube after its tailing existentials have been removed.

Let v be the deepest universal in qbe, i.e., the universal assigned at the deepest level along the path to the current solution leaf with $uncovered[v] \neq \emptyset$. Let n be the node assigning v its current value. The fact that qbe is a cube tells us that the subtree under n has been solved: this subtree is attempting to solve $Q.C[D]\big|_{\pi_v}$ where π_v is set of assignments in the path to n. π_v agrees with qbe on the assignment to its existential variables but further restricts its universal variables to assignments that lie in the domains of $C[D]\big|_{qbe}$. By Prop. 1 $Q.C[D]\big|_{qbe} \Rightarrow Q.C[D]\big|_{\pi_v}$, and thus $Q.C[D]\big|_{\pi_v}$ must be **true** since qbe is a cube. Furthermore, qbe also verifies that the subtrees of the other assignments to v not in $uncovered[v]$ are also solved: by the same reasoning all of these subproblems are also **true**. Hence, the search can backtrack to the node that assigned v, and from that point only attempt to solve the values for v in $uncovered[v]$ that have not been previously verified.

Each time the search backtracks to a universal variable v a new cube is returned, and at least one more value from v's domain has been verified (v's current assignment must be verified by the cube). Say that the search backtracks to v a total of k times before all of v's domain has been verified, in the process returning k cubes $qbe_1, \ldots, qbe_k$. At that point, the function in Figure 3 is invoked to compute a new cube (where π_v is the sequence of assignments made before v was selected to be assigned).

function `ComputeInternalCube` $(Q.C[D], v, \pi_v, qbe_1, \ldots, qbe_k)$
1: qbe = the assignments to the existential variables in π_v
2: **for** each universal variable $v_i \neq v$
3: $uncovered[v_i] = \bigcup_{j=1}^{k} uncovered[v_i] \in qbe_j$
4: qbe = $qbe \cup \{uncovered[v_i]\}$
5: qbe = remove tailing existentials from qbe

Fig. 3. Computing a QCSP cube at an internal node where the universal variable v was assigned.

In `ComputeInternalCube` each universal's uncovered values is the union of its uncovered values in the k cubes $qbe_1, \ldots, qbe_k$. Note also that $uncovered[v]$ is omitted from the new cube (i.e., $uncovered[v]$ is implicitly empty). Since v was the deepest universal in each of the cubes qbe_i, we see that the newly computed cube also contains no universals deeper that v. Nor does it contain any existential assignments deeper than v due to line 5 and the fact that each $cube_i$ also previously had their tailing existentials removed.

Once qbe is has been computed the search can once again backtrack to the node assigning the deepest universal v' in qbe, and at that point continue by solving all values of v' in $uncovered[v'] \in qbe$ that have not be previously verified. If all of these values were previously verified `ComputeInternalCube` will be invoked again on the set of cubes that were returned to v' (i.e., qbe and any other cubes returned by earlier backtracks to v'). The new cube it returns will then generate yet another backtrack.

Proposition 3 *Assume that $qbe_1, \ldots, qbe_k$ are all cubes, have been existentially reduced, agree on all existential assignments, have v as their deepest universal, and together verify all of the values in $dom[v]$ (i.e., for each $a \in dom[v]$ there exists j such that $a \notin uncovered[v] \in qbe_j$). Then the set qbe returned by `ComputeInternalCube` is a cube.*

This proposition can be proved by constructing a Q-Model for $Q.C[D]\big|_{qbe}$ using parts of the k Q-Models known to exist for $Q.C[D]\big|_{qbe_i}$. Subject to the assumed conditions these Q-Models are sufficiently compatible that these parts can be put together to cover all values for the universal variable v and all values in the universal domains of $C[D]\big|_{qbe}$.

The above propositions demonstrate that SBJ computes correct cubes and that these cubes verify that the backtracking described above is sound. In particular, SBJ will backtrack to the root of the search tree if and only if it has verified that the empty set is a cube. That is, $Q.C[D]\big|_{\emptyset} = Q.C[D]$ is **true**. Finally, two more observations about SBJ can be made. First, SBJ's space requirements are bounded by $O(dn^2)$, where d is the maximal sized variable domain and n is the number of variables. In particular, at each node along the current path (max n nodes) we need only store the union of the cubes that have been returned to that node so far. Furthermore, this set can be deleted when we backtrack from the node. Second, at each node the cubes contain all of the previously assigned existentials, so we need not explicitly store these in cube. These existentials would be needed however if we wanted to store the cubes to use along future paths of the search tree (i.e., if we were to perform cube learning).

5 Empirical Study

The random QCSP instances used in our empirical study were generated following the generation model introduced in [11]. As in [15] we added an extra parameter that denotes the number of universal blocks. The generator takes 8 parameters: $\langle n, n_\exists, n_\forall, d, p, q_{\forall\exists}, q_{\exists\exists}, b_\forall \rangle >$ where n is the number of variables, $n_\exists$ is the number of existentials in each block, $n_\forall$ is the number of universals in each block, d is the uniform domain size, p is the number of binary constraints as a fraction of all possible constraints, and $b_\forall$ is the number of universal blocks. $q_{\exists\exists}$ is the fraction of satisfying tuples in constraints between existentials. The satisfying tuples in a constraint between a universal and an existential later in the variable sequence are specified as follows. A random total bijection is generated from the domain of the universal to the domain of the existential. All 2-tuples not in the bijection satisfy the constraint. Parameter $q_{\forall\exists}$ is the fraction of satisfying tuples from the d tuples in the bijection.

Constraints between universals or an existential and a universal later in the variable sequence are not generated as these can be removed by preprocessing [11]. With certain parameter settings the randomly generated instances are free from the flaw described in [10]. Variables are quantified in blocks with alternating quantification starting with a block of $n_\exists$ existentials.

5.1 SBJ and Strong Consistency on Universals

To evaluate the effects of SBJ and the constraint propagation methods for universals, we have compared QCSP-Solve against three solvers obtained by extending QCSP-Solve with these features. The first solver (QCSP-Solve prop) augments QCSP-Solve with strong propagation on universals. The second one (QCSP-Solve+) augments QCSP-Solve with SBJ. The third one (QCSP-Solve++) applies both SBJ and strong propagation on universals. We used random problems with a variety of parameter settings. The results presented hereafter are averages over 100 instances generated at each data point. In each figure the value of $q_{\exists\exists}$ is varied in steps of 0.05. For the experiments of this section variables were instantiated according to the quantifier sequence. Values were always ordered lexicographically.

Figure 4 shows cpu times and node visits from problems where $n = 24$, $n_\exists = n_\forall = 8$, $b_\forall = 1$, $d = 9$, $p = 0.15$, $q_{\forall\exists} = 0.44$. Under these parameter settings all instances are guaranteed to be flaw-free.

The results given in Figure 5 are from problems generated using similar parameter settings as in [15]. The left plot in Figure 5 shows cpu times from problems where $n = 25$, $n_\exists = n_\forall = 5$, $b_\forall = 2$, $d = 8$, $p = 0.20$, $q_{\forall\exists} = 0.50$. The right plot in Figure 5 shows cpu times from problems where $n = 28$, $n_\exists = n_\forall = 4$, $b_\forall = 3$, $d = 8$, $p = 0.20$, $q_{\forall\exists} = 0.50$. Note that neither of these parameter settings guarantees flaw-free instances.

In all sets of problems QCSP-Solve++ is considerably faster than QCSP-Solve. For high values of $q_{\exists\exists}$, where most instances are soluble, the speed-up obtained can be up to two orders of magnitude. This is because, through the use of SBJ, the solver avoids repeatedly searching for solutions involving all sequences of assignments to universals. For low values of $q_{\exists\exists}$, where most problems are insoluble, SBJ has little effect and the computation/maintainance of solution cubes is an overhead that slows down search.

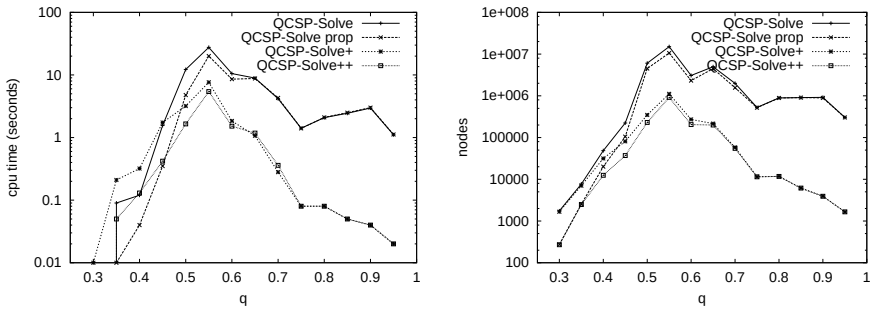

Fig. 4. Cpu times (left) and node visits (right).

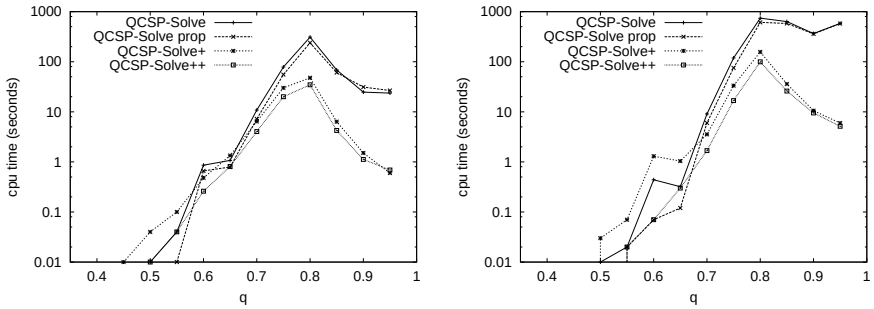

Fig. 5. Cpu times on problems with 25 variables (left) and 28 variables (right).

However, the early failure detection offered by the strong consistencies applied on universals outweighs this and speed-ups compared to QCSP-Solve are obtained. Comparing QCSP-Solve to QCSP-Solve prop and QCSP-Solve+ to QCSP-Solve++ shows that the effects of SBJ and strong propagation on universals are more or less orthogonal.

As is evident from the results shown in the figures, a small increase in the number of variables and quantifier alternations can have a significant impact on the difficulty of the problem.

Other Approaches to QCSP Solving. Apart from QCSP-Solve, two direct solvers for QCSPs have been developed and a number of encodings of QCSP into QBF have been proposed. The two solvers are BlockSolve [15] and QeCode [4]. BlockSolve is a bottom-up solver that displays very good performance on soluble instances, but as a downside requires exponential space. QeCode is built on top of Gecode and hence is equipped with many advanced CSP techniques. However, it lacks specialized features for QCSPs, such as pure value handling.

Although we have not directly compared our work to these solvers, we can make some conjectures by observing the performance of the solvers on instances generated with similar parameters. SBJ makes QCSP-Solve far more competitive with BlockSolve than before on soluble instances. However, BlockSolve still holds an advantage, as it can achieve a speed-up of up to four orders of magnitude over QCSP-Solve; albeit with an exponential memory cost. At the phase transition and to its left, where problems are insoluble, BlockSolve is outperformed by our techniques. This conjecture is based

on the observation that BlockSolve displays roughly the same performance as QCSP-Solve at the phase transition while it is slower in the insoluble region [15]. Experiments with QeCode showed that it displays roughly similar performance as QCSP-Solve [4]. Therefore, we conjecture that SBJ makes QCSP-Solve considerably more efficient than QeCode on soluble instances. QBF solvers that run on the efficient adapted and enhanced log encodings are typically slower than QCSP-Solve on insoluble instances and faster on soluble ones [10, 11]. We conjecture that SBJ makes QCSP-Solve at least competitive with the encodings on soluble instances.

5.2 Validity Pruning and Dynamic Variable Ordering

We now study the effect of validity pruning and dynamic variable ordering (DVO) within blocks. In Figure 6 we compare three variations of QCSP-Solve++ augmented with validity pruning. The first one (QCSP-Solve++.1) applies validity pruning to achieve early detection of valid values and uses a static variable ordering. Its performance is very close to that of QCSP-Solve++ (it is negligibly slower). The second variation (QCSP-Solve++.2) dynamically reorders variables within both existential and universal blocks. The third variation (QCSP-Solve++.3) applies DVO only within existential blocks and orders the universals statically. The heuristic used is dom/deg. The left plot in Figure 6 gives results from the problems of Figure 4 while the right plot gives results from the problems of Figure 5 (the ones with 25 variables).

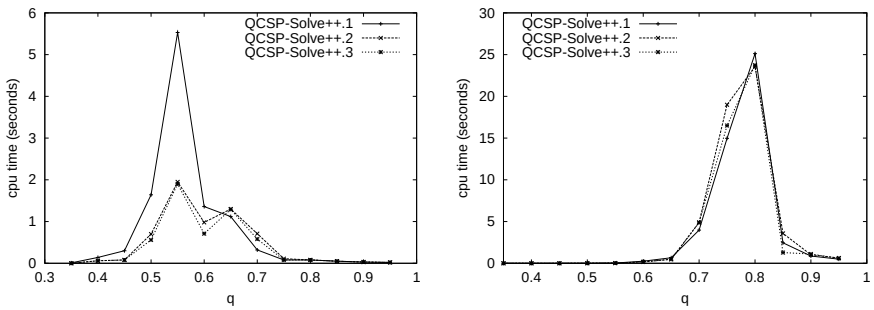

Fig. 6. Cpu times on problems with 25 variables (left) and 28 variables (right).

Not surprisingly, DVO is effective on insoluble problems with large existential blocks. However, is has little effect on soluble problems, and even slows down search in some cases. Again unsurprisingly, problems with blocks of small size do not benefit from DVO. Finally, since QCSP-Solve++.2 and QCSP-Solve++.3 yield similar results, it seems that the reordering of universals does not improve the performance of the solver. However, fail-first heuristics like dom/deg may not be ideal for universal variables, so it is possible that better heuristics, which exploit the information offered by validity pruning, will be designed in the future.

6 Conclusions

We have presented new techniques for improving the performance of backtracking based QCSP solvers. Our main contribution is the development of solution directed

backjumping for QCSPs. In analogue to conflict directed backjumping, SBJ allows the solver to backtrack out of solved sub-trees without having to find all of the distinct solutions normally required to validate that all sequences of assignments to the universal variables lead to solutions. We also demonstrated that performing varying levels of propagation for universal vs. existential variables can be useful for enhancing performance. Experiments with the solver QCSP-Solve demonstrate that both these techniques, and especially SBJ, can significantly improve the performance of backtracking solvers. Finally, we discussed validity pruning, a potentially useful technique that can be used to prune the domains of universally quantified variables during search.

References

1. M.F. Ali, S. Safarpour, A. Veneris, M.S. Abadir, and R. Drechsler. Post-verification debugging of hierarchical designs. In *International Conf. on Computer Aided Design (ICCAD)*, pages 871–876, 2005.
2. F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of AAAI-98*, pages 311–318, 1998.
3. F. Bacchus and T. Walsh. Propagating logical combinations of constraints. In *Proc. of 19th IJCAI*, pages 35–40, 2005.
4. M. Benedetti, A. Lallouet, and J. Vautard. Reusing CSP propagators for QCSPs. In *Proceedings of CSCLP-2006*, 2006.
5. L. Bordeaux, M. Cadoli, and T. Mancini. CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of AAAI-2005*, pages 360–365, 2005.
6. L. Bordeaux and E. Monfroy. Beyond NP: Arc-consistency for Quantified Constraints. In *Proceedings of CP-2002*, pages 371–386, 2002.
7. R. Bryant, S. Lahiri, and S. Seshia. Convergence testing in term-level bounded model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 348–362. Springer-Verlag, 2003.
8. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings of CP-97*, pages 312–326, 1997.
9. Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *Proceedings of AAAI-2000*, pages 417–422, 2000.
10. I. Gent, P. Nightingale, and A. Rowley. Encoding Quantified CSPs as Quantified Boolean Formulae. In *Proceedings of ECAI-2004*, pages 176–180, 2004.
11. I. Gent, P. Nightingale, and K. Stergiou. QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems. In *Proceedings of IJCAI-2005*, 2005.
12. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Eighteenth national conference on Artificial intelligence*, pages 649–654, 2002.
13. N. Mamoulis and K. Stergiou. Algorithms for Quantified Constraint Satisfaction Problems. In *Proceedings of CP-2004*, pages 752–756, 2004.
14. Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
15. G. Verger and C. Bessière. Blocksolve: a Bottom-Up Approach for Solving Quantified CSPs. In *Proceedings of CP-2006*, pages 635–649. Springer, 2006.
16. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Proceedings of CP2002*, pages 185–199, 2002.