Preprocessing QBF

Horst Samulowitz, Jessica Davies, and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada.
[horst| jdavies | fbacchus]@cs.toronto.edu

Abstract. In this paper we investigate the use of preprocessing when solving Quantified Boolean Formulas (QBF). Many different problems can be efficiently encoded as QBF instances, and there has been a great deal of recent interest and progress in solving such instances efficiently. Ideas from QBF have also started to migrate to CSP with the exploration of Quantified CSPs which offer an intriguing increase in representational power over traditional CSPs. Here we show that QBF instances can be simplified using techniques related to those used for preprocessing SAT. These simplifications can be performed in polynomial time, and are used to preprocess the instance prior to invoking a worst case exponential algorithm to solve it. We develop a method for preprocessing QBF instances that is empirically very effective. That is, the preprocessed formulas can be solved significantly faster, even when we account for the time required to perform the preprocessing. Our method significantly improves the efficiency of a range of state-of-the-art QBF solvers. Furthermore, our method is able to completely solve some instances just by preprocessing, including some instances that to our knowledge have never been solved before by any QBF solver.

1 Introduction

QBF is a powerful generalization of SAT in which the variables can be universally or existentially quantified (in SAT all variables are implicitly existentially quantified). While any NP problem can be encoded in SAT, QBF allows us to encode any PSPACE problem: QBF is PSPACE-complete. This increase in representational power also holds for finite domain CSPs, with quantified CSPs being able to represent PSPACE-complete problems not expressible as standard CSP problems [13, 19].

This opens a much wider range of potential application areas for a QBF or QCSP solver, including problems from areas like automated planning (particularly conditional planning), non-monotonic reasoning, electronic design automation, scheduling, model checking and verification, see, e.g., [9, 12, 17]. However, the difficulty is that QBF and QCSP are in practice much harder problems to solve.

Current QBF solvers are typically limited to problems that are about 1-2 orders of magnitude smaller than the instances solvable by current SAT solvers (1000's of variables rather than 100,000's). A similar difference holds between current QCSP solvers and CSP solvers. Nevertheless, both QBF and QCSP solvers continue to improve. Furthermore, many problems have a much more compact encoding when quantifiers are available, so a quantified solver can still be useful even if it can only deal with much smaller instances than a traditional solver.

In this paper we present a new technique for improving QBF solvers. Like many techniques used for QBF, ours is a modification of techniques already used in SAT. Namely we preprocess the input formula, without changing its meaning, so that it becomes easier to solve. As we demonstrate below our technique can be extremely effective, sometimes reducing the time it takes to solve a QBF instance by orders of magnitude. Although our technique is not immediately applicable to QCSP, it does provide insights into preprocessing that in future work could have a positive impact on the efficiency of QCSP solvers. We discuss some of the connections to QCSP in our future work section.

In the sequel we first present some necessary background, setting the context for our methods. We then present further details of our approach and state some results about its correctness. Then we provide empirical evidence of the effectiveness of our approach, and close with a discussion of future work and some conclusions.

2 QBF

A quantified boolean formula has the form $\boldsymbol{Q}.F$, where F is a propositional formula expressed in CNF and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\boldsymbol{Q}$ and that the set of variables in F and $\boldsymbol{Q}$ be identical (i.e., F contains no free variables, and $\boldsymbol{Q}$ contains no extra or redundant variables).

A **quantifier block** qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable x in F appears in some quantifier block $qb(x)$, and the ordering of the quantifier blocks imposes a partial order on the variables. For two variables x and y we say that $x \leq_q y$ iff $qb(x) \leq qb(y)$. Note that the variables in the same quantifier block are unordered. We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is $\forall$ ($\exists$).

For example, $\exists e_1 e_2 . \forall u_1 u_2 . \exists e_3 e_4 . (e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with $\boldsymbol{Q} = \exists e_1 e_2 . \forall u_1 u_2 . \exists e_3 e_4$ and F equal to the two clauses $(e_1, \neg e_2, u_2, e_4)$ and $(\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, and we have, e.g., that, $e_1 <_q e_3$, $u_1 <_q e_4$, u_1 is universal, and e_4 is existential.

A QBF instance can be reduced by assigning values to some of its variables. The **reduction** of a formula $\boldsymbol{Q}.F$ by a literal ℓ (denoted by $\boldsymbol{Q}.F\big|_{\ell}$) is the new formula $\boldsymbol{Q}'.F'$ where F' is F with all clauses containing ℓ removed and the negation of ℓ, $\neg\ell$, removed from all remaining clauses, and $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with the variable of ℓ and its quantifier removed. For example, $\forall xz.\exists y.(\neg y, x, z) \wedge (\neg x, y)\big|_{\neg x} = \forall z.\exists y(\neg y, z)$.

Semantics. A SAT model $\mathcal{M}_s$ of a CNF formula F is a truth assignment π to the variables of F that satisfies every clause in F. In contrast a QBF model (**Q-model**) $\mathcal{M}_q$ of a quantified formula $\boldsymbol{Q}.F$ is a **tree** of truth assignments in which the root is the empty truth assignment, and every node n assigns a truth value to a variable of F not yet assigned by one of n's ancestors. The tree $\mathcal{M}_q$ is subject to the following conditions:

1. For every node n in $\mathcal{M}_q$, n has a sibling if and only if it assigns a truth value to a universal variable x. In this case it has exactly one sibling that assigns the opposite truth value to x. Nodes assigning existentials have no siblings.

2. Every path π in $\mathcal{M}_q$ (π is the sequence of truth assignments made from the root to a leaf of $\mathcal{M}_q$) must assign the variables in an order that respects $<_q$. That is, if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_q x$.
3. Every path π in $\mathcal{M}_q$ must be a SAT model of F. That is π must satisfy the body of $\boldsymbol{Q}.F$.

Thus a Q-model has a path for every possible setting of the universal variables of $\boldsymbol{Q}$, and each of these paths is a SAT model of F. We say that a QBF $\boldsymbol{Q}.F$ is QSAT iff it has a Q-model. The QBF problem is to determine whether or not $\boldsymbol{Q}.F$ is QSAT.

A more standard way of defining QSAT is the recursive definition: (1) $\forall x\boldsymbol{Q}.F$ is QSAT iff both $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\neg x}$ are QSAT, and (2) $\exists x\boldsymbol{Q}.F$ is QSAT iff at least one of $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\neg x}$ is QSAT. By removing the quantified variables one by one we arrive at either a QBF with an empty clause in its body F (which is not QSAT) or a QBF with an empty body F (which is QSAT). It is not difficult to prove by induction on the length of the quantifier sequence that the definition we provided above is equivalent to this definition.

The advantage of our "tree-of-models" definition is that it makes two key observations more apparent. These observations can be used to prove the correctness of our preprocessing technique.

A. If F' has the same satisfying assignments (SAT models) as F then $\boldsymbol{Q}.F$ will have the same satisfying models (Q-models) as $\boldsymbol{Q}.F'$.
Proof: $\mathcal{M}_q$ is a Q-model of $\boldsymbol{Q}.F$ iff each path in $\mathcal{M}_q$ is a SAT model of F iff each path is a SAT model of F' iff $\mathcal{M}_q$ is a Q-model of $\boldsymbol{Q}.F'$.
This observation allows us to transform F with any model preserving SAT transformation. Note that the transformation must be model preserving, i.e., it must preserve all SAT models of F. Simply preserving whether or not F is satisfiable is not sufficient.

B. A Q-model preserving (but not SAT model preserving) transformation that can be performed on $\boldsymbol{Q}.F$ is **universal reduction**. A universal variable u is called a *tailing universal* in a clause c if for every existential variable $e \in c$ we have that $e <_q u$. The universal reduction of a clause c is the process of removing all tailing universals from c [10]. Universal reduction preserves the set of Q-models.
Proof: Say that $v \in c$ is a tailing universal, then along any path π in any Q-Model $\mathcal{M}_q$ of $\boldsymbol{Q}.F$, c must be satisfied by π prior to v being assigned a value. Say not, then since v is universal, the prefix of π that leads to the assignment of v must also be the prefix of another path π' that sets v to false: but then π' will falsify c because at this point c is a unit clause containing only the universal variable v. Therefore $\mathcal{M}_q$ cannot be a Q-model of $\boldsymbol{Q}.F$. Hence every path π satisfies the universal reduction of c (and all other clauses in F), and thus $\mathcal{M}_q$ is also Q-model of $\boldsymbol{Q}.F'$ where F' is F with the tailing universal v removed from c. This process can be repeated to remove all tailing universals from all clauses of F.

3 HyperBinary resolution

The foundation of our polynomial time preprocessing technique is the SAT method of reasoning with binary clauses using hyper-resolution developed in [2, 3]. This method reasons with CNF SAT theories using the following "HypBinRes" rule of inference:

Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the new clause $b = (c - D) \cup \{\ell\}$ if b is either binary or unary.

For example, from (a, b, c, d), $(h, \neg a)$, $(h, \neg c)$ and $(h, \neg d)$, we infer the new binary clause (h, b), similarly from (a, b, c) and $(b, \neg a)$ the rule generates (b, c). The HypBinRes rule covers the standard case of resolving two binary clauses (from (l_1, l_2) and $(\neg l_1, \ell)$ infer (ℓ, l_2)) and it can generate unit clauses (e.g., from (l_1, ℓ) and $(\neg l_1, \ell)$ we infer $(\ell, \ell) \equiv (\ell)$).

The advantage of HypBinRes inference is that it does not blow up the theory (it can only add binary or unary clauses to the theory) and it can discover a lot of new unit clauses. These unit clauses can then be used to simplify the formula by doing unit propagation which in turn might allow more applications of HypBinRes. Applying HypBinRes and unit propagation until closure (i.e., until nothing new can be inferred) uncovers *all* failed literals. That is, in the resulting reduced theory there will be no literal ℓ such that forcing ℓ to be true followed by unit propagation results in a contradiction. This and other results about HypBinRes are proved in the above references.

In addition to uncovering unit clauses we can use the binary clauses to perform equality reductions. In particular, if we have two clauses $(\neg x, y)$ and $(x, \neg y)$ we can replace all instances of y in the formula by x (and $\neg y$ by $\neg x$). This might result in some tautological clauses which can be removed, and some clauses which are reduced in length because of duplicate literals. This reduction might yield new binary or unary clauses which can then enable further HypBinRes inferences. Taken together HypBinRes and equality reduction can significantly reduce a SAT formula removing many of its variables and clauses [3].

4 Preprocessing QBF

Given a QBF $\boldsymbol{Q}.F$ we could apply HypBinRes, unit propagation, and equality reduction to F until closure. This would yield a new formula F', and the QBF $\boldsymbol{Q'}.F'$ where $\boldsymbol{Q'}$ is $\boldsymbol{Q}$ with all variables not in F' removed. Unfortunately, there are two problems with this approach. One is that the new QBF $\boldsymbol{Q'}.F'$ might not be Q-equivalent to $\boldsymbol{Q}.F$, so that this method of preprocessing is not sound. The other problem is that we miss out on some important additional inferences that can be achieved through universal reduction. We elaborate on these two issues and show how they can be overcome.

The reason why the straightforward application of HypBinRes, unit propagation and equality reduction to the body of a QBF is unsound, is that the resulting formula F' does not have exactly the same SAT models as F, as is required by condition **A** above. In particular, the models of F' do not make assignments to variables that have been removed by unit propagation and equality reduction. Hence, a Q-model of $\boldsymbol{Q'}.F'$ might not extendable to a Q-model of $\boldsymbol{Q}.F$. For example, if unit propagation forced a universal variable in F, then $\boldsymbol{Q'}.F'$ might be QSAT, but $\boldsymbol{Q}.F$ is not (no Q-model of $\boldsymbol{Q}.F$ can exist since the paths that set the forced universal to its opposite value will not be SAT models of F). This situation occurs in the following example. Consider the QBF $\boldsymbol{Q}.F = \exists abc \forall x \exists yz (x, \neg y)(x, z)(\neg z, y)(a, b, c)$. We can see that $\boldsymbol{Q}.F$ is not QSAT since when x is false, $\neg y$ and z must be true, falsifying the clause $(\neg z, y)$. If we apply HypBinRes and unit propagation to F, we obtain $F' = (a, b, c)$. Note that the universal variable x has been unit propagated. As anticipated, $\boldsymbol{Q}'.F' = \exists abc(a, b, c)$ is QSAT, so this reduction of F has not preserved the QSAT status of the original formula. However, it is easy to fix this problem. Making unit propagation sound for QBF simply requires that we regard the unit propagation of a universal variable as equivalent to the derivation of the empty clause. This fact is well known and applied in all search-based QBF solvers.

Ensuring that equality reduction is sound for QBF is a bit more subtle. Consider a formula F in which we have the two clauses $(x, \neg y)$ and $(\neg x, y)$. Since every path in any Q-model satisfies F, this means that along any path x and y must have the same truth value. However, in order to soundly replace all instances of one of these variables by the other in F, we must respect the quantifier ordering. In particular, if $x <_q y$ then we must replace y by x. It would be unsound to do the replacement in the other direction. For example, say that x appears in quantifier block 3 while y appears in quantifier block 5 with both x and y being existentially quantified. The above binary clauses will enforce the constraint that along any path of any Q-model once x is assigned y must get the same value. In particular, y will be invariant as we change the assignments to the universal variables in quantifier block 4. This constraint will continue to hold if we replace y by x in all of the clauses of F. However, if we perform the opposite replacement, we would be able to make y vary as we vary the assignments to the universal variables of quantifier block 4: i.e., the opposite replacement would weaken the theory perhaps changing its QSAT status. The same reasoning holds if x is universal and y is existential. However, if y is universal, the two binary clauses imply that we will never have the freedom to assign y its two different truth values. That is, in this case the QBF is UNQSAT, and we can again treat this case as if the empty clause has been derived.

Therefore a sound version of equality reduction must respect the variable ordering. We call this ($<_q$ preferred) equality reduction. That is, if we detect that x and y are equivalent and $x <_q y$ then we always remove y from the theory replacing it by x. With this restriction on equality reduction we have the following result:

Proposition 1 *Let F' be the result of applying HypBinRes, unit propagation, and ($<_q$ preferred) equality reduction to F until closure. If F' has the same set of universal variables as F (i.e., no universal variable was removed by unit propagation or equality reduction), then the Q-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the Q-models of $\boldsymbol{Q}.F$. In particular, $\boldsymbol{Q}.F$ is QSAT iff $\boldsymbol{Q}'.F'$ is QSAT. On the other hand, if F' has fewer universal variables than F then $\boldsymbol{Q}.F$ is UNQSAT.*

The idea behind the proof is that we can map any SAT model of F' to a SAT model of F by assigning all forced variables their forced value, and assigning all equality reduced variables a value derived from the variable they are equivalent to. That is, if x was removed because it was equivalent to $\neg y/y$, we assign x the opposite/same value assigned to y in F''s SAT model. In the other direction any SAT model of F can be mapped to a SAT model of F' by simply omitting the assignments of variables not in F'. Given this relationship between the SAT models, we can then show that any Q-model of F can be transformed to a unique Q-model of F' (by splicing out the nodes that assign variables not in F') and vice versa (by splicing in nodes to assign the variables not in

F'). This gives us that the number of Q-models of each formula are equal and that the transformation must be a 1-1 mapping.

This proposition tells us that we can use a SAT based reduction of F as a way of preprocessing a QBF, as long as we ensure that equality reduction respects the quantifier ordering and check for the removal of universals. This approach, however, does not fully utilize the power of universal reduction (condition **B** above). So instead we use a more powerful approach that is based on the following modification of HypBinRes that "folds" universal reduction into the inference rule. We call this rule "HypBinRes+UR":

> Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the universal reduction of the clause $(c \setminus D) \cup \{\ell\}$ if this reduction is either binary or unary.

For example, from $c = (u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$ we infer the new binary clause (u_1, e_2) when $u_1 \leq_q e_2 \leq_q e_3 \leq_q u_4 \leq_q e_5 \leq_q u_6 \leq_q e_7$. Note that without universal reduction, HypBinRes would need 5 binary clauses in order to reduce c, while with universal reduction, 2 fewer binary clauses are required. This example also shows that HypBinRes+UR is able to derive clauses that HypBinRes cannot. Since clearly HypBinRes+UR can derive anything HypBinRes can, HypBinRes+UR is a more powerful rule of inference.

In addition to using universal reduction inside of HypBinRes we must also use it when unit propagation is used. For example, from the two clauses $(e_1, u_2, u_3, u_4, \neg e_5)$ and (e_5) (with $e_1 <_q u_i$) unit propagation by itself can only derive (e_1, u_2, u_3, u_4), but unit propagation with universal reduction can derive (e_1).

It turns out that in addition to gaining more inferential power, universal reduction also allows us to obtain the unconditionally sound preprocessing we would like to have.

Proposition 2 *Let F' be the result of applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to F until closure, where we always apply universal reduction before unit propagation. Then the Q-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the Q-models of $\boldsymbol{Q}.F$.*

This result can be proved by showing that universal reduction generates the empty clause whenever a universal variable is to be unit propagated or removed via equality reduction. For example, for a universal u to be forced it must first appear in a unit clause (u), but then universal reduction would generate the empty clause (given that we apply universal reduction before unit propagation). Similarly, to make a universal variable u equivalent to an existential variable e with $e \leq_q u$ we would first have to generate the two binary clauses $(e, \neg u)$ and $(\neg e, u)$ which after universal reduction would yield (e) and $(\neg e)$ which after unit propagation would yield the empty clause. Thus the cases where Proposition 1 fails to preserve Q-models are directly detected through the generation of an UNQSAT $\boldsymbol{Q}'.F'$. In this case we still preserve the Q-models—neither formula has any.

Proposition 3 *Applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to $\boldsymbol{Q}.F$ until we reach closure can be done in time polynomial in the size of F.*

This result can be proved by making three observations: (1) F can never become larger than $|F|^2$ since we are only adding binary clauses, (2) there are at most a polynomial number of rule applications possible before closure since each rule either reduces a clause, removes a variable or adds a binary clause, and (3) at each stage detecting if another rule can be applied requires only time polynomial in the current size of the theory.

Our QBF preprocessor modifies $\boldsymbol{Q}.F$ exactly as described in Proposition 2. It applies HypBinRes+UR, unit propagation, universal reduction, and ($<_q$ preferred) equality reduction to F until it reaches closure. It then outputs the new formula $\boldsymbol{Q}'.F'$. Proposition 2 shows that this modification of the formula is sound. In particular, this preprocessing does not change the QSAT status of the formula.

To implement the preprocessor we adapted the algorithm presented in [3] which exploits a close connection between HypBinRes and unit propagation. In particular, this algorithm uses trial unit propagations to detect new HypBinRes inferences. The main changes required to make this algorithm work for QBF were adding universal reduction, modifying the unit propagator so that it performs universal reduction prior to any unit propagation step, and modifying equality reduction to ensure it respects the quantifier ordering.

To understand how trial unit propagation is used to detect HypBinRes+UR inferences, consider the example above of inferring (u_1, e_2) from $(u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$. If we perform a trial unit propagation of $\neg e_2$, dynamically performing universal reduction we obtain the unit clause (u_1). Because the trial propagation started with $\neg e_2$ this unit clause actually corresponds to the binary clause (u_1, e_2) (i.e., $\neg e_2 \rightarrow u_1$). The trial unit propagation has to keep track of the "root" of the propagation so that it does not erroneously apply universal reduction (every clause reduced during this process implicitly contains e_2).

5 Empirical Results

We considered all of the non-random benchmark instances from QBFLib (2005) [14] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these are all very quickly solved by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the instances coming from the benchmark families Jmc, and Jmc-squaring. None of these instances (with or without preprocessing) can be solved within our time bounds by any of the QBF solvers we tested. This left us with 468 remaining instances from 19 different benchmark families. We tested our approach on all of these instances.

All tests were run on a Pentium 4 3.60GHz CPU with 6GB of memory. The time limit for each run of any of the solvers or the preprocessor was set to 5,000 seconds.

5.1 Performance of the Preprocessor

We first examine the time required to preprocess the QBF formulas by looking at the runtime behaviour of the preprocessor on the given set of benchmark families. On the vast majority of benchmarks the preprocessing time is negligible. In particular, the preprocessing time for even the largest instances in the benchmarks Adder, Chain, Connect, Counter, FlipFlop, Lut, Mutex, Qshifter, Toilet, Tree, and Uclid is less than one second.

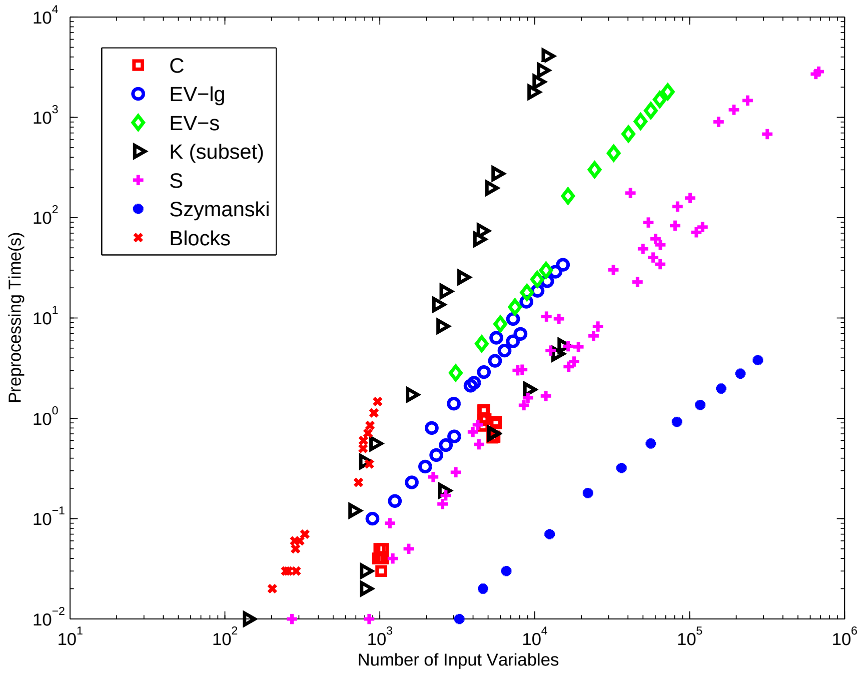

Fig. 1. Logarithmic scale comparison between the number of input variables and the preprocessing time in seconds on a selected set of benchmark families.

For example, the instance Adder-16-s with $\approx$ 22,000 variables and $\approx$ 25,000 clauses is preprocessed in 0.3 seconds.

The benchmarks that require more effort to preprocess are C, EVPursade, S, Szymanski, and Blocks and a subset of the K benchmark:[1] k-branch-n, k-branch-p, k-lin-n, k-ph-n, and k-ph-p. To examine the runtime behaviour on the these benchmark families we plot the number of input variables of each instance against the time required for preprocessing (Figure 1), clustering all of the K-subfamilies into one group. Both axis of the plot are drawn in logarithmic scale.

Figure 1 shows that for all of these harder benchmarks the relationship between the number of input variables and preprocessing time is approximately linear on the loglog-plot. This is not surprising since Proposition 3 showed that the preprocessor runs in worst-case polynomial time. Any polynomial function is linear in a loglog scale with the slope increasing with the degree of the polynomial. Fitting a linear function to each benchmark family enables a more detailed estimate of the runtime, since the slope of the fitted linear function determines the relationship between the number of input variables and preprocessing time. For instance, a slope of one indicates a linear runtime, a slope of two indicates quadratic behaviour, etc. Except for the benchmarks 'k-ph-n' and 'k-

[1] This benchmark family is divided into sub-families.

Solver	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
# Instances	311	**351**	262	**312**	226	**238**	213	**243**	205	**239**
Time on common instances	9,748	**9,595**	10,384	**2,244**	36,382	**20,188**	41,107	**23,196**	46,147	**25,554**
Time on new instances	-	12,756	-	16,829	-	9,579	-	9,707	-	2,421

Table 1. Summary of results reported in Tables 2 and 3. For each solver we show its number of solved instances among all tested benchmark families with and without preprocessing, the total CPU time (in seconds) required to solve the preprocessed and un-preprocessed instances taken over the "common" instances (instances solved in both preprocessed and un-preprocessed form), and the total CPU time required by the solvers to solve the "new" instances (instances that can only be solved in preprocessed form).

ph-p' the slope of the fitted linear function ranges between 1.3 (Szymanski) and 2.3 (Blocks) which indicates a linear to quadratic behaviour of the preprocessor. The two K-subfamilies 'k-ph-n' and 'k-ph-p' display worse behaviour, on them preprocessing time is almost cubic (slope of 2.9).

The graph also shows that on some of the larger problems the preprocessor can take thousands of seconds. However, this is not a practical limitation. In particular out of the 468 instances only 23 took more than 100 seconds to preprocess. Of these 18 could not be solved by any of our solvers, either in preprocessed form or unpreprocessed form. That is, these problems are so hard that we have no way of evaluating the effect of preprocessing them. Of the other 5 instances that were solved by some solver there exist in total 25 pairs of instances and solvers. Among these there exist only 13 pairs where some solver succeeded either on the preprocessed instance only or on both versions of the instance. On 62% of these 13 successful runs, the preprocessor yielded a net speedup. Furthermore, despite being a net slowdown on the other 38% of runs, another 38% of the runs were cases where the solver was only able to solve the preprocessed instance. So our conclusion is that except for a few instances, preprocessing is not a significant added computational burden.

5.2 Impact of Preprocessing

Now we examine how effective the preprocessor is. Is it able to improve the performance of state of the art QBF solvers, even when we consider the time it takes to run? To answer this question we studied the effect preprocessing has on the performance of five state of the art QBF solvers **Quaffle** [20] (version as of Feb. 2005), **Quantor** [8] (version as of 2004), **Qube** (release 1.3) [15], **Skizzo** (v0.82, r355) [4] and **SQBF** [18]. Quaffle, Qube and SQBF are based on search, whereas Quantor is based on variable elimination. Skizzo uses mainly a combination of variable elimination and search, but it also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances.

A summary of our results is presented in Table 1. The second row of the table shows the total time required by each solver to solve the instances that could be solved in both preprocessed and unpreprocessed form (the "common instances"). The data demonstrates that preprocessing provides a speedup for every solver. Note that the times for the preprocessed instances *include* the time taken by the preprocessor. On these common instances Quantor was 4.6 times faster with preprocessing, while Quaffle, Qube and SQBF were all approximately 1.8 times faster with preprocessing. Skizzo is only slightly faster on the preprocessed benchmarks (that it could already solve).

The first row of Table 1 shows the number of instances that can be solved within the 5000 sec. time bound. It demonstrates that in addition to speeding up the solvers on problems they can already solve, preprocessing also extends the reach of each solver, allowing it to solve problems that it could not solve before (within our time and memory bounds). In particular, the first row shows that the number of solved instances for each solver is significantly larger when preprocessing is applied. The increase in the number of solved instances is 13% for Skizzo, 19% for Quantor, 5% for Quaffle, 14% for Qube and 17% for SQBF.

The time required by the solvers on these new instances is shown in row 3. For example, we see that SQBF was able to solve 34 new instances. None of these instances could previously be solved in 5,000 sec. each. That is, 170,000 CPU seconds were expended in 34 failed attempts. With preprocessing all of these instances could be solved in 2,421 sec. Similarly, Quantor expended 250,000 sec. in 50 failed attempts, which with preprocessing could all solved in 16,829 sec. Skizzo expended 200,000 sec. in 40 failed attempts which with preprocessing could all be solved in 12,756 seconds. Quaffle expended 60,000 sec. in 12 failed attempts, which with preprocessing could all be solved in 9,579 sec. And Qube expended 150,000 sec. in 30 failed attempts, which with preprocessing could all be solved in 9,707 seconds.

These results demonstrate quite convincingly that our preprocessor technique offers robust improvements to all of these different solvers, even though some of them are utilizing completely different solving techniques.

Tables 2 and 3 provide a more detailed breakdown of the data. Table 2 gives a family by family breakdown of the common instances (instances that can be solved in both preprocessed and unpreprocessed form). Specifically, the table shows for each benchmark family and solver (a) the percentage of instances that are solvable in both preprocessed and unpreprocessed form, (b) the total time required by the solvable instances when no preprocessing is used, and (c) the total time required with preprocessing (i.e., solving as well as preprocessing time).

Table 2 shows that the benefit of preprocessing varies among the benchmark families and, to a lesser extent, among the solvers. Nevertheless, the data demonstrates that among these benchmarks, preprocessing almost never causes a significant increase in the total time required to solve a set of instances. On the other hand, each solver has at least 2 benchmark families in which preprocessing yields more than an order of magnitude improvement in solving time. There are only two cases (Skizzo on Mutex, SQBF on the toilet benchmark) where preprocessing causes a slowdown that is as much as an order of magnitude (from 0 to 102 seconds and from 395 to 3,060 seconds).

Table 3 provides more information about the instances that were solvable only after preprocessing. In particular, it shows the percentage of each benchmark family that can be solved by each solver before and after preprocessing (for those families where this percentage changes). From this table we can see that for each solver there exist benchmark families where preprocessing increases the number of instances that can be

Benchmark (# instances)	*Skizzo*			*Quantor*			*Quaffle*			*Qube*			*SQBF*		
	Succ. %	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*
ADDER (16)	50%	954	**792**	25%	**24**	25	25%	1	1	13%	72	**27**	13%	3	**1**
adder (16)	44%	**455**	550	25%	29	**27**	42%	5	**4**	44%	**0**	1	38%	2,678	**2,229**
Blocks (16)	56%	108	**11**	100%	308	**79**	75%	1,284	**762**	69%	1774	**242**	75%	7,042	**1,486**
C (24)	25%	**1,070**	1,272	21%	140	**32**	21%	5,356	**14**	8%	3	**5**	17%	4	**0**
Chain (12)	100%	1	**0**	100%	0	0	67%	6,075	**0**	83%	4,990	**0**	58%	4,192	**0**
***Connect** (60)*	68%	802	**5**	67%	14	**7**	70%	253	**5**	75%	7,013	**7**	67%	**0**	5
Counter (24)	54%	1,036	**731**	50%	217	**141**	38%	5	5	33%	2	**1**	38	**9**	20
EVPursade (38)	29%	**1,450**	1,765	3%	**73**	82	26%	1,962	**1,960**	18%	4,402	**2,537**	32%	4,759	**4,508**
FlipFlop (10)	100%	6	**4**	100%	**3**	4	100%	**0**	4	100%	**1**	4	80%	5,027	**1**
K (107)	88%	**1,972**	2,228	63%	3,839	**39**	35%	21,675	**17,083**	37%	21,801	**19,203**	33%	5,563	**5,197**
Lut (5)	100%	9	9	100%	3	3	100%	1	1	100%	**3**	6	100%	1,247	**66**
Mutex (7)	100%	**0**	102	43%	**0**	1	29%	**43**	49	43%	**64**	71	43%	**1**	6
Qshifter (6)	100%	**8**	9	100%	**26**	29	17%	0	0	33%	29	29	33	**1,107**	2,103
S (52)	27%	**644**	1,886	25%	**910**	1,530	2%	0	0	4%	**401**	451	2%	0	0
Szymanski (12)	42%	1,147	**179**	25%	7	0	0%	0	0	8%	**0**	200	0%	0	0
TOILET (8)	100%	**1**	25	100%	4,135	**3**	75%	**61**	84	63%	496	**325**	100%	1,307	**621**
toilet (38)	100%	84	**50**	100%	684	**243**	97%	**115**	207	100%	**58**	90	97%	**395**	3,060
Tree (14)	100%	0	0	100%	0	0	100%	37	**9**	100%	**0**	1	93%	**1,051**	1,251

Table 2. Benchmark family specific information about commonly solved instances. Shown are the percentage of instances that are solved in both preprocessed and unpreprocessed form and the total time in CPU seconds taken to solve these instances within each family with and without preprocessing. Best times shown in **bold**.

solved. It is interesting to note that preprocessing improves different solvers on different families. That is, the effect of preprocessing is solver-specific. Nevertheless, preprocessing allows every solver to solve more instances. It can also be noted that the different solvers have distinct coverage, with or without preprocessing. That is, even when a solver is solving a larger percentage of a benchmark it can still be the case that it is failing to solve particular instances that are solved by another solver with a much lower success percentage on that benchmark. Preprocessing does not eliminate this variability.

Some instances are actually solved by the preprocessor itself. There are two benchmark families that are completely solved by preprocessing: FlipFlop and Connect. While the first family is rather easy to solve the second one is considered to be hard. In fact, $\approx 25\%$ of the Connect benchmarks could not be solved by any QBF solver in the 2005 QBF evaluation [16]. Our preprocessor solves the complete benchmark family in less than 10 seconds. In addition, a few benchmarks from the hard S benchmark family can be solved by the preprocessor. Again these instances could not be solved by any of the QBF solvers we tested within our time bounds. In total, the preprocessor can completely solve 18 instances that were unsolvable by any of the solvers we tested (in our time bounds). The Chain benchmark is another interesting case (its instances have 2 quantifier alternations $\exists\forall\exists$). The instances in this family are reduced to ordinary SAT instances by preprocessing. The preprocessor was able to eliminate all existential

Benchmark	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
Blocks	69%	**88%**	100%	100%	75%	**88%**	69%	69%	75%	**81%**
C	25%	**29%**	21%	**30%**	21%	**25%**	8%	**21%**	17%	**25%**
Chain	100%	100%	100%	100%	67%	**100%**	83%	**100%**	58%	**100%**
Connect	68%	**100%**	67%	**100%**	70%	**100%**	75%	**100%**	58%	**100%**
FlipFlop	100%	100%	100%	100%	100%	100%	100%	100%	80%	**100%**
K	89%	**91%**	63%	**83%**	35%	**36%**	37%	**42%**	33%	**35%**
S	27%	**37%**	25%	**31%**	2%	**8%**	4%	**8%**	2%	**8%**
Szymanski	42%	**75%**	25%	**50%**	0%	0%	8%	**25%**	**8%**	0%
toilet	100%	100%	100%	100%	97%	**100%**	100%	100%	97%	97%
Uclid	0%	**67%**	0%	0%	0%	0%	0%	0%	0%	0%

Table 3. Benchmark families where preprocessing changes the percentage of solved instances (within our 5,000 sec. time bound). The table shows the percentage of each families' instances that can be solved with and without preprocessing.

variables from the innermost quantifier block and consequently remove all universals by universal reduction. The resulting SAT instance is trivial to solve (it is smaller than the original QBF instance). In all of these cases the extended reasoning applied in the preprocessor exploits the structure of the instances very effectively. Note that the preprocessing cannot blow up the body of the QBF since it can only add binary clauses to the body. Thus, any time the preprocessor converts a QBF instance to a SAT instance, the SAT instance cannot be much larger that the original QBF.

There were only five cases where for a particular solver preprocessing changed a solvable instance to be unsolvable (Quaffle on one instance in the K benchmarks, SQBF on one instance in the Szymanski benchmarks, Skizzo on two instances in the Blocks benchmark and on one instance in the K benchmark). This is not apparent from Table 3 since both Quaffle and Skizzo can still solve more instances of the K and Blocks benchmarks respectively, with preprocessing than without. However, we can see that the percentage of solved instances for SQBF on the Szymanski benchmark falls to 0% after preprocessing. This simply represents the fact that SQBF can solve one instance of Szymanski before preprocessing and none after. That is, we have found very few cases when preprocessing is detrimental.

In total, these results indicate that preprocessing is very effective for each of the tested solvers across almost all of the benchmark families.

6 Related Work

In this section we review the similarities between our preprocessor and the methods applied in existing QBF solvers. We conclude that HypBinRes has not been previously lifted to the QBF setting, although equality reduction and binary clause reasoning have been used in some state-of-the-art QBF solvers. Our experimental results support this, since our preprocessor aids the performance and reach of even the solvers that employ binary clause reasoning and equality reduction.

Skizzo applies equality reduction as part of its symbolic reasoning phase [4]. [4] makes the claim that Skizzo's SHBR rule performs a symbolic version of hyper binary

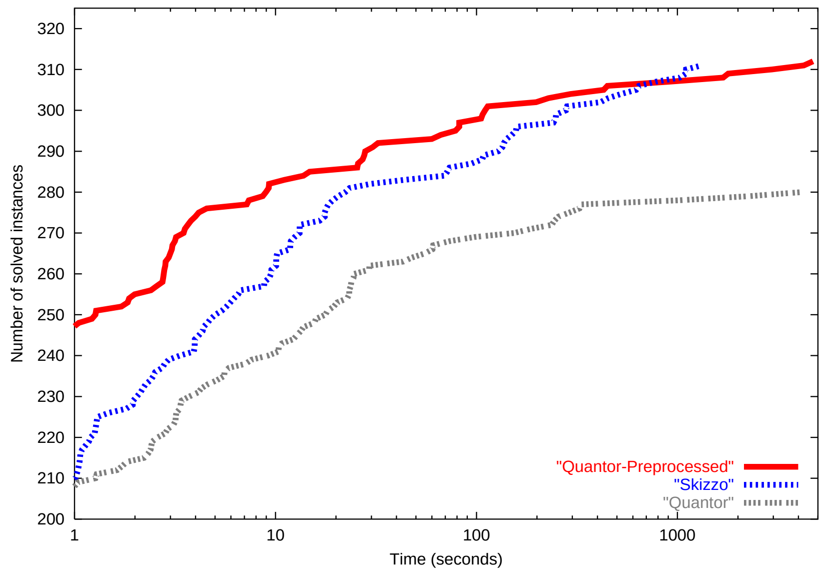

Fig. 2. Logarithmic time scale comparison of Quantor and Skizzo on the original and Quantor on the preprocessed benchmarks. Shown is the time in seconds versus the number of instances solved.

resolution. However, a close reading of the papers [4, 6, 5, 7] suggests that in fact the SHBR rule is a strictly weaker form of inference than HypBinRes. SHBR traverses the binary implication graph of the theory, where each binary clause (x, y) corresponds to an edge $\neg x \rightarrow y$ in the graph. It detects when there is a path from a literal l to its negation $\neg l$, and in this case, unit propagates $\neg l$. This process will not achieve HypBinRes. Consider the following example where HypBinRes is applied to the theory $\{(a, b, c, d), (x, \neg a), (x, \neg b), (x, \neg c)\}$. HypBinRes is able to infer the binary clause (x, d). Yet the binary implication graph does not contain any path from a literal to its negation, so Skizzo's method will not infer any new clauses. In fact, the process of searching the implication graph is well known to be equivalent to ordinary resolution over binary clauses [1]. On the other hand, HypBinRes can infer anything that SHBR is able to since it captures binary clause resolution as a special case. Therefore SHBR is strictly weaker than HypBinRes. This conclusion is also supported by our experimental results, which show, e.g., that our preprocessor is able to completely solve the Connect Benchmark where as Skizzo is only able to solve 68% of these instances.

The variable elimination algorithm of Quantor also bears some resemblance to hyper binary resolution, in that variables are eliminated by performing all resolutions involving that variable in order to remove it from the theory. General resolution among n-ary clauses is a stronger rule of inference than HypBinRes, but it is difficult to use as a preprocessing technique due to its time and space complexity (however see [11]).

7 Future Work

Additional techniques for preprocessing remain to be investigated. Based on the data we have gathered with our preprocessor, we can conclude that a very effective technique would be to run our preprocessor followed by running Quantor for a short period of time (10-20 seconds). This technique is capable of solving a surprising number of instances. As shown in Figure 2 the combination of the preprocessor and Quantor is in fact able to solve more instances than Skizzo [4]. Hence, by simply employing hyper resolution and variable elimination it is possible to gain an advantage over such sophisticated QBF solvers as Skizzo. Furthermore, this technique solves a number of instances that are particularly problematic for search based solvers. Figure 2 shows that this technique (the "Quantor-Preprocessed" line) can solve approximately 285 instances within 10 seconds. Yet if we continue to run Quantor for another 5000 seconds very few additional problems are solved (about 25 more instances). We have also found that search based solvers can solve a larger number of these "left-over" instances than Quantor.

This suggests the strategy of first running the preprocessor, then running Quantor, and then a search based solver if Quantor is unable to solve the instance quickly. Even more interesting would be to investigate obtaining the partially eliminated theory from Quantor after it has run for a few seconds, and then seeing if it could be further preprocessed or fed directly into a search based solver. The Skizzo solver [4] attempts to mix variable elimination with search in a related way, but it does not employ the extended preprocessing reasoning we have suggested here.

Another important direction for future work is to investigate how some of these ideas can be used to preprocess QCSP problems. Unfortunately although preprocessing is common in CSPs (achieving some level of local consistency prior to search), our particular technique of HypBinRes has no immediate analog in CSP. HypBinRes takes advantage of the fact that binary clauses form a tractable subtheory of SAT, however binary constraints are not a tractable subtheory in CSPs unless we also have binary valued domains. Nevertheless, what our work does indicate is that it might be worth investigating the usefulness of achieving higher levels of local consistency prior to search in QCSPs than might be sensible for standard CSPs. This is because QCSPs require more extensive search: all values of every universal variable have to be solved. So effort expended prior to search can be amortized over a larger search space. Note that HypBinRes+UR is a more powerful form of inference than what most of the QBF solvers are applying during search.

8 Conclusions

We have shown that preprocessing can be very effective for QBF and have presented substantial and significant empirical results to verify this claim. Nearly all of the publicly available instances are taken into account, and five different state of the art solvers are compared. The proposed method of preprocessing offers robust improvements across the different solvers among all tested benchmark families. The achieved improvement also includes almost 20 instances that to our knowledge have never been solved before.

References

1. B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithms for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
2. Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, 2002.
3. Fahiem Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Lecture Notes in Computer Science 2919*, pages 341–355, 2003.
4. M. Benedetti. sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03, 2004.
5. M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR04)*, number 3452 in LNCS. Springer, 2005.
6. M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proc. of 9th International Joint Conference on Artificial Intelligence (IJCAI05)*, 2005.
7. M. Benedetti. Quantifier Trees for QBFs. In *Proc. of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT05)*, 2005.
8. A. Biere. Resolve and expand. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 238–246, 2004.
9. R. Bryant, S. Lahiri, and S. Seshia. Convergence testing in term-level bounded model checking. Technical Report CMU-CS-03-156, Carnegie Mellon University, 2003.
10. H. K. Büning, M. Karpinski, and A. Flügel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.
11. N. Een and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *In Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, number 3569 in LNCS. Springer, 2005.
12. Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *AAAI/IAAI*, pages 417–422, 2000.
13. Ian P. Gent, Peter Nightingale, and Kostas Stergiou. Qcsp-solve: A solver for quantified constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 138–143, 2005.
14. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.
15. E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 364–369, 2001.
16. M. Narizzano and A. Tacchella. QBF evaluation, 2005. http://www.qbflib.org/qbfeval/2005.
17. Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
18. H. Samulowitz and F. Bacchus. Using SAT in QBF. In *Principles and Practice of Constraint Programming*. Springer-Verlag, New York, 2005.
19. Kostas Stergiou. Repair-based methods for quantified csps. In *Principles and Practice of Constraint Programming*, pages 652–666, 2005.
20. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Principles and Practice of Constraint Programming (CP2002)*, pages 185–199, 2002.