Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)

Exploiting QBF Duality on a Circuit Representation

Alexandra Goultiaeva and **Fahiem Bacchus**
Department of Computer Science
University of Toronto
{alexia, fbacchus}@cs.toronto.edu

Abstract

Search based solvers for Quantified Boolean Formulas (QBF) have adapted the SAT solver techniques of unit propagation and clause learning to prune falsifying assignments. The technique of cube learning has been developed to help them prune satisfying assignments. Cubes, however, have not been able to achieve the same degree of effectiveness as clauses.

In this paper we demonstrate how a circuit representation for QBF can support the propagation of dual truth values. The dual values support the identical techniques of unit propagation and clause learning, except now it is satisfying assignments rather than falsifying assignments that are pruned. Dual value propagation thus exploits the circuit representation and the duality of QBF formulas so that the same effective SAT techniques can now be used to prune both falsifying and satisfyingly assignments. We show empirically that dual propagation yields significant performance improvements and advances the state of the art in QBF solving.

Introduction

Quantified Boolean Formulas (QBF) are an extension of SAT in which universally quantified variables are added to the (implicitly) existentially quantified variables of SAT. Determining the truth of a QBF is a PSPACE-Complete problem, and thus a much wider range of problems can be polynomially encoded in QBF than in SAT. This means that an effective QBF solver could have a very wide range of practical applications, and makes developing such a solver a research problem with great potential for practical impact. Although more research still needs to be done, modern solvers have already reached the point where some problems can be more efficiently solved when encoded as QBF than as SAT (Mangassarian et al. 2007). In this paper we make further progress on this research goal by presenting a technique that improves the performance of state of the art QBF solvers.

A QBF formula has the form $\vec{Q}.\phi$ where $\vec{Q}$ is a sequence of universally ($\forall$) and existentially ($\exists$) quantified variables, and ϕ is a propositional formula over those variables. Let $\phi|_{x=0}$ ($\phi|_{x=1}$) be the reduction of ϕ by the assignment $x = 0$ ($x = 1$) computed by replacing the variable x with the constant 0/false (1/true) followed by simplifying. The truth of a QBF formula is defined recursively: $\exists x\vec{Q}.\phi$ is true iff either $\vec{Q}.\phi|_{x=0}$ or $\vec{Q}.\phi|_{x=1}$ is true, and $\forall x\vec{Q}.\phi$ is true iff both $\vec{Q}.\phi|_{x=0}$ and $\vec{Q}.\phi|_{x=1}$ are true. We assume that all of the variables in ϕ are contained in $\vec{Q}$. Hence ϕ will eventually be reduced to the constant 1 or 0: $\vec{Q}.1$ is always true and $\vec{Q}.0$ is always false. A SAT formula, which asks whether or not there exists a setting of the variables that makes the formula true, is thus a QBF formula in which $\vec{Q}$ consists only of existential variables.

Many QBF solvers are based on the same technique of backtracking search (DPLL) used for solving SAT. Search based QBF solvers have successfully adapted the powerful techniques of unit propagation and clause learning that are at the core of modern SAT solvers. These techniques are very successful at identifying and steering the solver away from assignments that force ϕ to be false, and at learning new clauses when ϕ is falsified so as to detect similar assignments that must also falsify ϕ.

QBF solvers, however, face the equally important task of not wanting to waste time exploring assignments that force the formula to be true. For example, if $\vec{Q}.\phi$ has a prefix of k universally quantified variables, $\forall u_1, \ldots, \forall u_k$, the above definition of truth implies that each of the 2^k different assignments to the u_i must yield a reduction of $\vec{Q}.\phi$ that is true. Clearly, it is intractable to check each of these assignments. Instead a QBF solver must have mechanisms that allow it to identify and avoid assignments that force the QBF to be true, and that learn new information when the QBF is truthified so as to detect similar assignments that must also truthify the QBF. The technique of Cube learning has been developed to accomplish this (Zhang and Malik 2002; Giunchiglia, Narizzano, and Tacchella 2002). However, cube learning is not as effective at pruning the truthifying assignments as clause learning is at pruning the falsifying assignments.

Although most QBF solvers have employed a clausal representation (CNF) for ϕ, the body of the QBF, Goultiaeva, Iverson, and Bacchus (2009) have presented a solver that uses a circuit representation for ϕ and shown that it has a number of advantages over CNF. They have also shown that unit propagation and clause learning can be fully realized even though ϕ is represented as a circuit and not as a set of clauses. Cube learning can also be performed. However, although the circuit representation yields better cubes than

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

produced with a CNF representation, in their system cube learning is still not as effective as clause learning.

In this paper we show how the circuit representation can be further exploited by propagating a dual set of values through it. These dual values allow us to prune truthifying assignments with the same techniques used to prune falsifying assignments, allowing the solver to prune both types of assignments equally effectively. The result is a significant improvement in performance.

We have modified the circuit based QBF solver of (Goultiaeva, Iverson, and Bacchus 2009) to include the propagation of dual values. Our experiments verify its improved performance. We have also tested our new solver against other state of the art QBF solvers, and our experiments demonstrate that our technique improves the state of the art in QBF solving on an number of different benchmarks.

Background

A **circuit representation** of a QBF $\vec{Q}.\phi$ involves representing the propositional formula ϕ as a boolean circuit utilizing AND, OR and NOT gates and lines from gate outputs to gate inputs. The quantified variables of $\vec{Q}$ are the inputs to the circuit. The output line of each gate is labeled with a new variable. Formally, these variables behave as existentially quantified variables scoped by all of the input variables with a path to them in the circuit. For example, the QBF $\exists e_1 \forall u_1 \exists e_2 \forall u_2.(e_1 \wedge \neg u_1) \vee (e_2 \wedge u_2)$ would be represented by a circuit with: (1) x_1 = NOT(u_1), i.e., a NOT gate with input u_1 and output x_1; (2) x_2 = AND(e_1, x_1); (3) x_3 = AND(e_2, u_2); and (4) out = OR(x_2, x_3). The x_i are the new existential auxiliary variables which if put into the quantifier yield $\exists e_1 \forall u_1 \exists x_1 x_2 e_2 \forall u_2 \exists x_3$. We see that x_1 and x_2 are only scoped by e_1 and u_1 while the x_3 has narrowest scope. Note that the circuit output out does not appear in the "augmented" quantifier. This is because out will be assigned a fixed value so it will be replaced by a constant.

The **backtracking search** solver CirQit of Goultiaeva, Iverson, and Bacchus (2009) executes a tree-search in which variables are instantiated and the consequences of those decisions propagated. However, with QBF the solver has to verify both settings of the universal variables and the order in which variables are instantiated must respect the quantifier ordering in $\vec{Q}$ (all variables scoping v must be instantiated before v can be chosen). This implies that the solver will never branch on any of the auxiliary variables. All inputs they depend on scope them and must be instantiated first; after which their value will be forced. For example, if u_1 is set to true, the outputs x_1 of NOT(u_1) and x_2=AND(e_1, x_1) are both set to false. A clausal reason for every literal (v or $\neg v$) that is forced can be extracted from the circuit. For example, $\neg x_2$ is forced because the clause $(x_1, \neg x_2) \equiv \neg x_1 \rightarrow \neg x_2$ has become unit. This clause is extracted from the logic of the AND gate that relates x_1 and x_2.

It can be shown (Thiffault, Bacchus, and Walsh 2004) that the auxiliary variables labeling the gate outputs are the same as the new variables that would be introduced when the circuit is encoded in CNF. Furthermore, propagation on the circuit forces exactly the same literals as would be forced by unit propagation on the CNF encoding, and labels them with exactly the same clausal reasons.

CirQit propagates **primal values (1-values)**. In the CNF encoding, the circuit output line out is always set to 1 and the CNF simplified prior to search. This is motivated by SAT where the aim is to find a setting of the variables that makes the formula true (i.e., inputs which that the circuit output 1). Similarly, CirQit sets the output of the circuit to 1 and propagates this value back through the circuit. This can interact with the settings of the input lines to cause further propagation. For example, if u_1 is set forcing $\neg x_2$, then x_3 will also be forced. The final OR gate OR(x_2,x_3) must output 1, and thus $\neg x_2$ forces the other input to be 1. In turn x_3 forces e_2 and u_2. A clausal reason can be extracted from the circuit for each forced literal; e.g., the reason for u_2 is the clause $(\neg x_3, u_2)$.

A **contradiction** occurs when either both 1 and 0 are propagated to the same line, or when a disjunction of universal literals is forced to be true. In the example above the universal literal u_2 is forced so propagation detects a contradiction. A conflict clause can be extracted from the circuit representation and used to initiate clause learning. In QBF conflict clauses are untruthified clauses all of whose existential literals have been falsified. Starting with the conflict clause and the fact that all forced literals have clausal reasons, a clause learning process similar to that used in SAT solvers can then be executed. The only difference is that any universal literal that scopes no existential in the learnt clause can be removed (these are called tailing universals). The new clause can then be used to backtrack the solver just as in SAT solvers.

Search based QBF solvers also employ **Cube Learning**. With a circuit representation cube learning is initiated once a sufficient number of input lines are set (i.e., input literals made true) by the search to cause 1 to be propagated to the circuit output. The solver knows that the body of the QBF has now been reduced to 1, and hence that the QBF has been truthified. A cube can be extracted from the true input literals. Any subset of these input literals that suffices to propagate 1 to the output forms a cube, and the solver will typically employ a greedy approach to finding such a subset. (Search might have set many other input literals that ultimately were not needed to force the circuit output).

All tailing existentials can then be removed from the cube (existentials that don't scope a universal in the cube), and the solver can then backtrack to the most recently set universal u in the cube and try u's other value (the cube verifies that the u's current value truthifies the QBF). If u's other value has already been verified, then the solver must have already backtracked to u with a cube K_0 verifying u's other value. Now on this backtrack it has found a new cube K_1 verifying u's current value. K_0 and K_1 can be resolved removing u and $\neg u$ and all existentials that become tailing to obtain a new cube K. The solver can then use K to backtrack further.

Cube learning with a CNF representation is similar. The solver can stop when all clauses have been truthified, and a cube consisting of a subset of the true literals sufficient to truthify all clauses can be extracted.

Cubes as currently implemented in QBF solvers have **lim-**

ited effectiveness. The first limitation arises from that fact that the solver starts off with no cubes—cubes have to be learnt during search. This means that for the early phases of the search the solver must delve very deep in its search tree before it can recognize that the QBF has been made true. On the other hand the solver starts off with many short clauses (in the circuit representation these clauses are implicit in the logical relationships between the gate inputs and its output).

The second limitation is that the cubes learnt when a solution is identified tend to be very large—they have to contain enough input literals to truthify the circuit, or to satisfy every clause in the CNF. Cubes learnt from a circuit tend to be shorter than those learnt from a CNF, but they are still not small enough. Once we start with long cubes it will require a lot of resolutions (and a lot of search time) to generate short ones. Hence, even when cubes are learnt, they tend to be useful only quite deep in the search tree. On the other hand, learnt clauses, although they can be large, can also often be short. A SAT solver, for example, often learns new unit clauses.

Finally, a third limitation is that cubes constructed from a circuit representation include only literals corresponding to set input lines (cubes constructed from a CNF representation can contain auxiliary literals but will typically also contain just as many input literals). Learnt clauses, on the other hand can often contain only auxiliary literals, corresponding to settings of various internal lines. Since many different settings of the inputs can generate the same settings of various internal lines, we see that a single clause with auxiliary variables can represent many different sets of input literals that falsify the formula. Cubes with only input literals, on the other hand, capture only one specific setting of the input variables that truthifies the formula.

Dual Propagation

How can we address the weaknesses of current cube learning? First, we note that cubes and clauses are duals of each other. If all of the literals in a clause become false the formula is falsified; if all of the literals in a cube become true the formula is truthified. However, in current cube learning clauses and cubes are generated by quite different processes. Intuitively, the duality between cubes and clauses should mean that there exists a dual version of clause learning that can make the solver's treatment of cubes more like its treatment of clauses.

Consider the negation of the QBF being solved $\neg\vec{Q}.\phi$. This formula is false iff the original QBF $\vec{Q}.\phi$ is true. Taking the negation in we obtain $(\neg\vec{Q}).(\neg\phi)$, where $\neg\vec{Q}$ is the same as $\vec{Q}$ except that the quantifiers are flipped. For $\neg\phi$ we can exploit the circuit representation. This formula can be represented by the same circuit used to represent ϕ, C_ϕ, then passing the output of C_ϕ through a NOT gate.

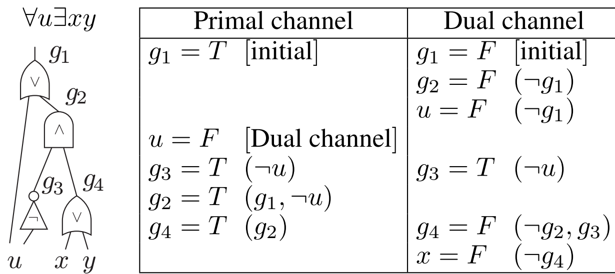
If we want to solve $\neg\vec{Q}.\phi$ with a circuit based solver we would take the circuit NOT(C_ϕ) and set its output to 1. The 1 would propagate back through the final NOT gate, and set the output of C_ϕ to 0. So we see that the final NOT gate can be discarded, it suffices to set the output of C_ϕ to 0. Now the solver would start to set the input lines of the circuit. These are identical to the input lines of C_ϕ but have reversed quantifiers. The set input lines are propagated and contradictions are detected. Propagation exploits the local logical properties of the gates and labels each propagated value with a clausal reason. Contradictions generate falsified clauses (conflict clauses) consisting of local sets of inputs and outputs of a single gate. Thus clause learning operates just as before.

Logically, if propagation forces a input literal ℓ, then $\neg\ell$ must make $\neg\vec{Q}.\phi$ false and $\vec{Q}.\phi$ true. So when solving $\vec{Q}.\phi$ unit propagation on $\neg\vec{Q}.\phi$ tells us that we need not explore the subtree below $\neg\ell$: the formula is provably truthified under that setting. Note that, if a clause c becomes unit (ℓ) when solving $\neg\vec{Q}.\phi$, then ℓ must be an existential in $\neg\vec{Q}.\phi$; otherwise c would be a conflict all-universal clause. Thus, unit propagation in $\neg\vec{Q}.\phi$ can only force input universals in $\vec{Q}.\phi$, while unit propagation in $\vec{Q}.\phi$ can only force input existentials in $\vec{Q}.\phi$. Unit propagation on $\neg\vec{Q}.\phi$ can be just as powerful as unit propagation of $\vec{Q}.\phi$. So unit propagation on $\neg\vec{Q}.\phi$ allows us to identify and steer the solver away from truthifying assignments just as effectively as unit propagation on $\vec{Q}.\phi$ allows us to identify and steer the solver away from falsifying assignments.

Similarly we see conflicts in $\neg\vec{Q}.\phi$ identify when $\vec{Q}.\phi$ must be true. And clauses learnt from these conflicts allow us to detect similar assignments that must also truthify $\vec{Q}.\phi$. Furthermore, clauses learnt this way can be just as effective on truthifying assignments as clauses learnt from $\vec{Q}.\phi$ are on falsifying assignments.

To **implement** this idea we do not actually need to do propagation on $\neg\vec{Q}.\phi$. Rather we can use the same circuit representation for ϕ and propagate two values in this circuit along a primal and dual channel. Formally, every line of the circuit is given two truth values: a primal or 1-value and a dual or 0-value. The primal inputs of a gate are logically connected to the gate's primal output, and its dual inputs are logically connected to its dual output. Initially, the circuit output's primal value is set to 1 and its dual value to 0. These values propagate back through the primal and dual channels of the circuit affecting the primal and dual values of other lines respectively. The input lines always have an equal primal and dual value, hence the solver need only keep track of one value for the input lines. Whenever their value is set, by decision or by values being propagated back along one channel, the same value goes out along both channels. Thus, the channels can affect each other via the input lines.

As before, search sets input lines and propagation sets the auxiliary variables. (Both $\vec{Q}.\phi$ and $\neg\vec{Q}.\phi$ have the same existential auxiliary variables scoped by the same set of input variables). Now, however, both primal and dual values are propagated through the circuit to the auxiliary variables. Thus propagation might set either or both values of the auxiliary variables. Contradictions are detected from both the primal and dual values. In both cases a clause is learnt and the solver backtracks. Primal clauses and dual clauses are put in separate databases: unit propagating primal clauses forces primal values while unit propagating dual clauses forces dual values. As more clauses are learnt these



Primal channel	Dual channel
$g_1 = T$ [initial]	$g_1 = F$ [initial]
	$g_2 = F$ $(\neg g_1)$
	$u = F$ $(\neg g_1)$
$u = F$ [Dual channel]	
$g_3 = T$ $(\neg u)$	$g_3 = T$ $(\neg u)$
$g_2 = T$ $(g_1, \neg u)$	
$g_4 = T$ (g_2)	$g_4 = F$ $(\neg g_2, g_3)$
	$x = F$ $(\neg g_4)$

Figure 1: Example circuit solved by dual propagation

databases can considerably increase the power of primal and dual unit propagation, as well as the ability to detect primal and dual conflicts.

It can be seen that a primal conflict causes backtrack and an existential of $\vec{Q}.\phi$ to be forced (the opposite value must falsify $\vec{Q}.\phi$), while a dual conflict causes backtrack and a universal of $\vec{Q}.\phi$ to be forced (the opposite value must truthify $\vec{Q}.\phi$). Furthermore, the solver always encounters either a primal conflict or a dual conflict along each path it explores—once a sufficient number of input lines have been set either a 0 or a 1 must be propagated to the circuit output causing either a primal or a dual conflict. Finally, the solver terminates on learning either an empty primal or an empty dual clause indicating that the input QBF is false or true respectively.

Figure 1 illustrates dual propagation with a simple example. It shows the sequence of values forced by propagation on the two channels, together with their reasons, starting with the circuit output g_1. The last assignment on the dual channel causes a conflict, since the variable x is universal along that channel. So, the formula is found to be true. Note that primal propagation alone cannot solve this problem.

Don't-care propagation

One of the advantages of a circuit representation over CNF is that it supports unrestricted detection of don't care. This can yield significant performance improvements (Goultiaeva, Iverson, and Bacchus 2009).

This raises the question of how dual propagation affects don't cares. Since propagation has the same logic along both the primal and dual channels (albeit with potentially different values being propagated), it is not difficult to extend the notion of don't cares to the dual channel values. We define DC-1 variables to be those variables whose value is irrelevant assuming the 1-value of the output, and DC-0 variables as those variables whose value is irrelevant assuming the 0-value of the output. The techniques used to detect DC-1 variables can be extended to detect DC-0 variables, and the following proposition can be proved.

Proposition 1. *If all the input variables are set, except for those that are also DC-1 or DC-0, then propagation will cause a conflict to be reached on one of the channels.*

This result says that the solver need never branch on any variable that is don't care on either channel: the truth of the formula will still be resolved along every path of its search tree. It is possible that some variables are recognized as DC only on one of the channels. For example, in a formula of the form $\alpha_1 \vee (\alpha_1 \wedge \alpha_2)$, the variables belonging exclusively to α_2 will be found DC-0 but not DC-1. Thus, doing dual propagation can increase the power of don't care propagation.

Related work

It has previously been recognized that the weakness of cubes (Ansótegui, Gomes, and Selman 2005) and the assymmetrical treatment of variables (Bordeaux and Zhang 2007) seriously harm the performance of search based QBF solvers.

An early work in this direction (Otwell, Remshagen, and Truemper 2004) dealt only with a heavily restricted formalization (limited to two quantifier levels and a particular formula structure).

The solver Duaffle (Sabharwal et al. 2006) requires that the QBF be formulated in two parts, a CNF and a DNF, either conjoined or disjoined. The argument is made that such a formulation is natural for many types of problems, although clearly it is not always natural. The idea here is that the DNF terms, when they become true, can be used to more quickly detect that the formula has been truthified. With dual propagation nothing special is required of the formulation, and clause learning allows truth to be detected in more cases.

The other, and most closely related work, was the approach of converting a circuit representation into both a CNF and a DNF (CCDNF) (Zhang 2006). Thus the formula is represented twice. The DNF allows for more powerful detection of truth, while the CNF is used in the standard way to detect falsity. The resulting solver, IQTest, however, had to implement a dual version of unit propagation and DNF term learning in addition to a standard unit propagation and clause learning. In our approach the same module can be used to deal with both the dual and the primal channels. Furthermore, by preserving the original circuit representation our solver can also implement don't care propagation.

One disadvantage of our approach is that it needs the input to be represented as a propositional formula or circuit to perform optimally. It can operate on CNF, and dual propagation will still offer some advantages, but the more time expensive circuit data structures will be a drag on performance. This disadvantage is shared by the other approaches just mentioned.

Experimental results

We have implemented our ideas (including don't cares on the dual channel) in the solver CirQit (Goultiaeva, Iverson, and Bacchus 2009). We compared the resulting solver, called CirQit2, with the original, and with other state-of-the-art QBF solvers.

The experiments were run on all the non-Prenex, non-CNF benchmarks currently available from QBFLIB (Giunchiglia, Narizzano, and Tacchella 2001a). The tests were run on a 2.8GHz machine with 12GB of RAM under a time limit of 1200 CPU seconds per instance.

Figure 2 shows the drastic effect that dual propagation had on the performance of the solver. For each instance, the time CirQit2 took to solve it is on the y-axis, and CirQit on the x-axis. We can see that most of the instances are

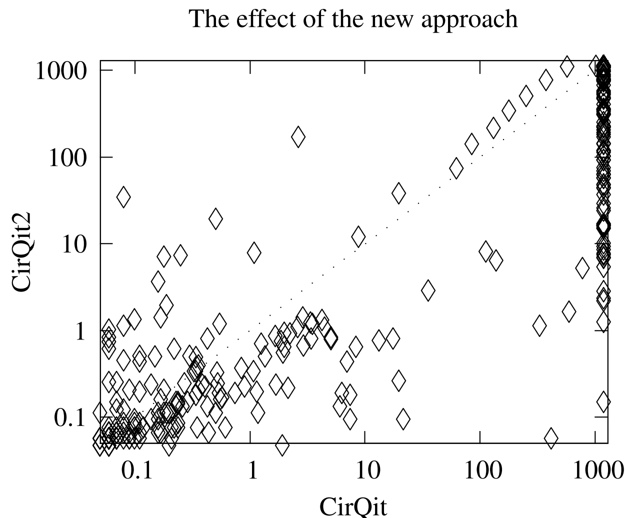


Figure 2: Time comparison between CirQit and CirQit2

strongly below the bisecting line, which means that CirQit2 took considerably less time to solve them. Instances that timed out are plotted on the 1200 second mark on the graph. We can see a lot of instances on the right border. Those are the problems that CirQit2 solved but on which CirQit timed out. Note that there are no problems along the top border: CirQit2 solved all the problems CirQit did.

For the problems that are above the bisecting line, dual propagation increased solving time. Note that the scale is logarithmic, and most of the cloud in the lower left corner is solved by both solvers within a second. The line of problems just above the bisecting line are those where the solution did not benefit from the dual propagation, but was slowed down by the associated overhead. These rare problems now take about twice the time it originally took to solve them: dual propagation costs no more than twice the original. A small handful of problems in the upper left show CirQit2 taking considerably more time than its counterpart. These cases are very rare, and are caused by heuristic reasons.

Table 1 shows the comparison between CirQit2, CirQit and some state-of-the-art CNF-based QBF solvers: sKizzo (v0.8.2) (Benedetti 2005), quantor (version 3.0, with picosat back end) (Biere 2004), Qube (version 6.5) (Giunchiglia, Narizzano, and Tacchella 2001b), nenofex (Lonsing and Biere 2008) and depqbf (Lonsing and Biere 2009)—the latter two are the versions submitted to the main track of QBFEVAL'10 (QBFEVAL is an annual international QBF solver competition). Predecessors of sKizzo and Quantor took first and second place at QBFEVAL'05. Qube6.5 is an updated version of the best standalone solver of QBFEVAL'08, and of the winner of QBFEVAL'07. The predecessor of Nenofex was the second best standalone solver of the QBFEVAL'08. The non-prenex non-CNF instances are converted to prenex CNF using a conversion tool available on the QBFEval site.

Quantor is a bottom up solver based on resolution rather than on top-down search. It is well known that the bottom up approach works much better than top-down search on some QBFs, but is outperformed by search on other formulas. Dual propagation improves search based solvers, but it is still outperformed by quantor on the domains from the set *BMC_QBF_1.0*. These domains are "assertion", "consistency" and "possibility", for which bottom up solvers are better suited. Together, they comprise over half of the total problems, which is why the total number of problems solved by quantor is higher. However, CirQit2 outperforms quantor in all the other domains. It is also worth noting that CirQit2 significantly out performs all of the other search based solvers on these three "hard for search" domains. Except for these three domains, and for "dme", where it is unable to solve a problem that depqbf is able to solve, CirQit2 proves to be superior.

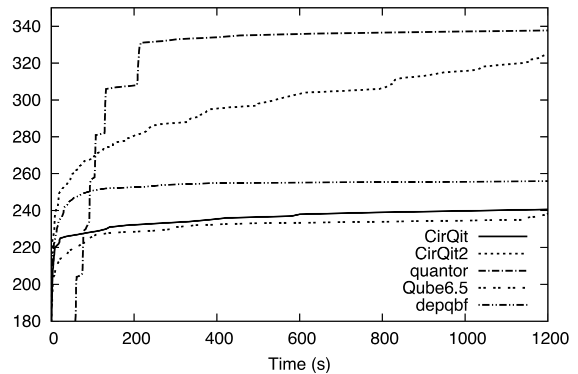


Figure 3: Problems solved vs time allotted per problem

Comparison with the non-CNF QBF solvers currently available was omitted. As shown in (Goultiaeva, Iverson, and Bacchus 2009), CirQit already outperforms these solvers, and since CirQit2 outperforms CirQit (Figure 2) it handly outperforms these solvers.

Due to technical limitations we were not able to do many experiments with IQTest (Zhang 2006) (only a Windows executable of IQTest is available). However, we ran two problem suites. The first was the Seidl benchmarks with 150 instances. Using a Windows machine and a 1200 sec. timeout, CirQit2 required 1009 sec. to solve all 150 problems, while IQTest could only solve 126 problems, taking 53,000 seconds (not counting the time consumed by the timed out instances). On the Scholl benchmarks, distributed with IQTest, however, IQTest outperforms CirQit2. Of these 63 problems, CirQit2 solved 38 while IQTest solved 46 (within the timeout). We do not have a good explanation for the differences. It could be that IQTest's heuristics were developed with input from the Scholl benchmarks only.

Figure 3 shows the number of problems solved by the solvers as time goes on. For compactness, only those solvers that solved over 200 problems were included. We can see that all the other solvers, except for CirQit2, essentially plateau: after a timeout of 250 seconds, giving them more time does not noticeably increase the number of problems they could solve. The line representing the result of CirQit2, on the other hand, keeps climbing. This indicates that technical issues, such as code optimization and better data struc-

Benchmark Families	CirQit2				CirQit				sKizzo		quantor		Qube6.5		nenofex		depqbf	
(number of instances)	Solved	Time	avg Cube	#Cubes	Solved	Time	avg Cube	#Cubes	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
Seidl (150)	**150**	318	91.31	170526	147	2281	57.41	1016738	37	6301	42	3272	149	2485	82	1160	150	557
assertion (120)	40	14503	-	0	3	1	-	0	14	796	**119**	8736	6	1180	12	4170	24	145
consistency (10)	4	1283	-	None	0	0	-	None	1	40	**10**	720	0	0	1	306	0	0
counter (45)	**40**	492	107.55	5932	39	1315	334.27	68126	34	1185	28	414	31	540	29	1727	31	70
dme (11)	10	5	10.83	179	10	15	373.64	2997	0	0	0	0	7	88	8	94	**11**	901
possibility (120)	45	16121	-	0	10	1707	-	0	13	700	**111**	7976	14	4713	12	4037	10	143.376
ring (20)	**20**	53.55	37.91	2692	15	60	190.96	165591	12	752	11	479	16	189	11	4	13	243
semaphore (16)	**16**	3	15.22	435	16	7	63.72	21045	14	68	16	12	16	361	16	1193	16	39
Total (492)	325	32779	90.78	179764	240	5389	90.41	1274497	125	9844	**337**	21613	239	9557	171	12692	255	2098

Table 1: Comparison between CirQit2 and other state-of-the-art CNF-based solvers. The largest number of instances solved is shown in **bold**, with ties broken by the time taken to solve those instances. Average size of dual clauses (cubes) and the total number of cubes learned is provided for the instances which were solved by *both* CirQit and CirQit2

tures, can bring considerable benefits to this solver. In fact, a better optimized version of CirQit2 with some other additions not discussed here was able to solve 352 of the problems in this benchmark set, beating Quantor.

Comparing the results of CirQit and CirQit2 in Table 1 we see that dramatic speedups are evident in all domains. Probably the most obvious improvement happened in the benchmark sets “assertion”, “consistency” and “possibility”, where CirQit was unable to make much progress. The addition of dual propagation put these benchmarks firmly within its grasp, with solving time increasing roughly linearly for each next problem. In the benchmark sets “Seidl” and “ring”, CirQit2 is able to solve all the problems in less time than its previous version took to solving only some. The remaining three domains also displayed a noticeable reduction in solving time. Finally, cube size is noticeably reduced in all domains except for “Seidl”, and the number of cubes needed to solve the same problems is an order of magnitude smaller in all domains.

Conclusion

In this paper, we presented dual propagation—a simple approach that reconciles clauses and cubes and greatly increases the power and simplicity of the solver. We implemented the idea, and have demonstrated the drastic effect it has on performance.

There have been some previous attempts at overcoming the weaknesses of cubes, but these attempts have required more algorithmic changes and in some cases restricted the problem representation.

Due to its simplicity, many additional techniques can be used with dual propagation. We showed, for example, how it can be combined with Don’t Care Propagation.

References

Ansótegui, C.; Gomes, C. P.; and Selman, B. 2005. The achilles’ heel of QBF. In *Proceedings of the AAAI National Conference*, 275–281.

Benedetti, M. 2005. skizzo: A suite to evaluate and certify QBFs. In *Proceedings of the International Conference on Automated Deduction*, 369–376.

Biere, A. 2004. Resolve and expand. In *Theory and Applications of Satisfiability Testing*, 238–246. Springer.

Bordeaux, L., and Zhang, L. 2007. A solver for quantified boolean and linear constraints. In *Proceedings of the 2007 ACM symposium on Applied computing*, 321–325.

Giunchiglia, E.; Narizzano, M.; and Tacchella, A. 2001a. Quantified Boolean Formulas satisfiability library (QBFLIB). `www.qbflib.org`.

Giunchiglia, E.; Narizzano, M.; and Tacchella, A. 2001b. QUBE: A system for deciding Quantified Boolean Formulas satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning*, 364–369.

Giunchiglia, E.; Narizzano, M.; and Tacchella, A. 2002. Learning for quantified boolean logic satisfiability. In *Proceedings of the AAAI National Conference*, 649–654.

Goultiaeva, A.; Iverson, V.; and Bacchus, F. 2009. Beyond CNF: A circuit-based QBF solver. In *Theory and Applications of Satisfiability Testing*, 412–426.

Lonsing, F., and Biere, A. 2008. Nenofex: Expanding NNF for QBF solving. In *Theory and Applications of Satisfiability Testing*, 196–210.

Lonsing, F., and Biere, A. 2009. A compact representation for syntactic dependencies in QBFs. In *Theory and Applications of Satisfiability Testing*, 398–411.

Mangassarian, H.; Veneris, A. G.; Safarpour, S.; Benedetti, M.; and Smith, D. 2007. A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In *International Conference on Computer-Aided Design*, 240–245.

Otwell, C.; Remshagen, A.; and Truemper, K. 2004. An effective qbf solver for planning problems. In *MSV/AMCS, CSREA Press*, 311–316.

Sabharwal, A.; Ansótegui, C.; Gomes, C. P.; Hart, J. W.; and Selman, B. 2006. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *Theory and Applications of Satisfiability Testing*, 382–395.

Thiffault, C.; Bacchus, F.; and Walsh, T. 2004. Solving non-clausal formulas with DPLL search. In *Theory and Applications of Satisfiability Testing*.

Zhang, L., and Malik, S. 2002. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Proceedings of Principles and Practice of Constraint Programming*, 200–215.

Zhang, L. 2006. Solving QBF with combined conjunctive and disjunctive normal form. In *Proceedings of the AAAI National Conference*.