Beyond CNF: A Circuit-Based QBF Solver

Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus

Department of Computer Science
University of Toronto
{alexia,viverson,fbacchus}@cs.toronto.edu

Abstract. State-of-the-art solvers for Quantified Boolean Formulas (QBF) have employed many techniques from the field of Boolean Satisfiability (SAT) including the use of Conjunctive Normal Form (CNF) in representing the QBF formula. Although CNF has worked well for SAT solvers, recent work has pointed out some inherent problems with using CNF in QBF solvers.

In this paper, we describe a QBF solver, called CirQit (Cir-Q-it) that utilizes a circuit representation rather than CNF. The solver can exploit its circuit representation to avoid many of the problems of CNF. For example, we show how this approach generalizes some previously proposed techniques for overcoming the disadvantages of CNF for QBF solvers. We also show how important techniques like clause and cube learning can be made to work with a circuit representation. Finally, we empirically compare the resulting solver against other state-of-the-art QBF solvers, demonstrating that our approach can often outperform these solvers.

1 Introduction

QBF is a powerful generalization of SAT in which the variables can be universally or existentially quantified. While any problem in NP can be encoded in SAT, QBF allows us to encode any problem in PSPACE. This opens a much wider range of potential application areas for a QBF solver, including problems from areas like automated planning (particularly conformant and conditional planning), non-monotonic reasoning, electronic design automation, scheduling, model checking and verification, strategic decision making, and multi-agent scenarios, see for, e.g., [1,2,3].

State-of-the-art QBF solvers have utilized a number of techniques inherited from SAT solving technology. This has included the use of DPLL search augmented with clause learning along with additional QBF-specific techniques like solution backtracking and cube learning. Besides DPLL the original Davis-Putnam SAT solving technique [4] of ordered resolution has also been utilized [5], as well as methods involving the use of Skolemization to convert the QBF formula to SAT [6]. One constant in almost all of this work, however, has been the utilization of conjunctive normal form (CNF) in representing the QBF formula.

It has long been noted that conversion to CNF can lead to losing structure that could potentially be exploited computationally. As a result there has been some work on non-CNF SAT solvers, e.g., [7,8]. This work has shown that non-clausal representations can be effective for solving SAT. Nevertheless, the allure of CNF is that it can lead to very high performance implementations since it is a very simple and uniform representation.

Hence, the extra structure that can be exploited in a non-clausal representation has not been able to significantly outweigh the practical advantages of CNF in SAT solvers, and most SAT solvers continue to utilize CNF.

In QBF however the situation is different. In particular, for a similarly sized problem the search space explored by a QBF solver tends to be much larger than that explored by a SAT solver. Hence, there is much more potential for savings from exploiting the extra structure contained in non-clausal representations. In fact, there have been a number of papers that have identified various inadequacies of the CNF representation for QBF and proposed alternate representations aimed at addressing these problems, e.g., [9,10].

One of the most general and structure laden non-clausal representations is a circuit representation. Circuit representations have been used before in SAT solvers, e.g., [7,8], and in this paper we explore the use of this representation in a QBF solver.

One advantage of circuits is that they are more compatible with real problems—typically CNFs are generated from more structured representations like circuits. We also investigate ways of exploiting within our solver some of the extra structural information contained in the circuit. One particular example is the exploitation of don't care reasoning. We explain why don't care reasoning has more potential for efficiency gains when solving a QBF than when solving SAT. We also demonstrate how the essential techniques of unit propagation, clause learning, and cube learning used in CNF solvers can be adapted to a circuit representation. Finally, we explain how a circuit representation generalizes some of the key previously proposed techniques for addressing the inadequacies of CNF in the context of QBF.

We have implemented a solver we call **CirQit** (pronounced Cir-Q-it) that is based on our approach of utilizing a circuit representation. We are able to show empirically that it is very competitive with current state-of-the-art QBF solvers, and that on some problem suites it exhibits superior performance.

In the rest of the paper we first provide some essential background on QBF and the circuit representation of a QBF. We present some of the details of our circuit-based solver. Our solver utilizes a DPLL search procedure running on a circuit representation rather than on a CNF representation. We describe how propagation can be performed, and how clause and cube learning can be implemented. We discuss related work on QBF solvers based on non-clausal representations. Finally, we present various experimental results demonstrating the merit of our approach, and close with some conclusions.

2 Background

2.1 QBF

A QBF has the form $\boldsymbol{Q}.\phi$, where ϕ is an arbitrary propositional formula and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that the set of variables in ϕ be contained in $\boldsymbol{Q}$ so that $\boldsymbol{Q}.\phi$ has no free variables, and that ϕ contain only the connectives AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).

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. The quantifier blocks are ordered by their 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 ϕ appears in some quantifier block $qb(x)$. For two variables x and y we say that y is **downstream** of x (x is **upstream** of y) if $qb(y) > qb(x)$ ($qb(x) < qb(y)$). We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is $\forall$ ($\exists$).

A QBF instance can be reduced by a literal ℓ (i.e., an assignment to one of its variables). The **reduction** of a formula $\boldsymbol{Q}.\phi$ by ℓ is denoted by $\boldsymbol{Q}.\phi|_\ell$. The reduction is the new formula $\boldsymbol{Q}.\phi'$ where ϕ' is ϕ with v replaced by the constant TRUE (if $\ell = v$) or FALSE (if $\ell = \neg v$), and optionally simplified according to standard logical rules: e.g., for any formula ψ, FALSE $\wedge \psi$ is equivalent to FALSE and $\forall x.\psi$ is equivalent to ψ if the variable x does not appear in ψ. A specific example is $\forall xz.\exists y.(\neg y \vee (x \wedge z)) \wedge \neg(x \vee z)|_{\neg x}$ which is equal to $\forall xz.\exists y.(\neg y \vee (\text{FALSE} \wedge z)) \wedge \neg(\text{FALSE} \vee z)$ which simplifies to $\forall z.\exists y.\neg y \wedge \neg z$.

Semantically, the truth or falsity of a QBF formula (with no free variables) can be defined recursively: (1) $\forall x \boldsymbol{Q}.\phi$ is true iff both $\boldsymbol{Q}.\phi|_x$ and $\boldsymbol{Q}.\phi|_{\neg x}$ are true, and (2) $\exists x \boldsymbol{Q}.\phi$ is true iff at least one of $\boldsymbol{Q}.\phi|_x$ or $\boldsymbol{Q}.\phi|_{\neg x}$ is true. By instantiating the quantified variables one by one, following the quantifier ordering, and substituting true or false into ϕ we arrive at either a QBF where ϕ simplifies to FALSE (which is a false QBF) or a QBF where ϕ simplifies to TRUE (which is a true QBF).

A circuit is a directed acyclic graph with a single sink where the nodes are logical gates and the edges are signal lines connecting the gates. Each gate is either an AND, OR, or NOT gate, has a single outgoing output line, and one or more incoming input lines. The output line of the sink gate is the circuit output, and the lines that are not outputs of any gate are the circuit inputs. A circuit representation $\boldsymbol{Q}.C$ for the QBF formula $\boldsymbol{Q}.\phi$ is a circuit C where the variables in $\boldsymbol{Q}$ are in 1-1 correspondence with the circuit inputs. C can be constructed recursively as follows. If ϕ is a variable x, then C has only one line labeled by the variable x and no gates. If $\phi = \neg\psi$, then C consists of the circuit representing ψ with the output of this circuit connected to the input of a NOT gate. If $\phi = \psi_1 \wedge \cdots \wedge \psi_i$ ($\psi_1 \vee \cdots \vee \psi_i$), then C consists of the outputs of the circuits representing ψ_1 to ψ_i connected as inputs of an AND (OR) gate. One key feature of the circuit representation is that duplicated sub-formulas in ϕ can be represented by a single subcircuit—the output line of that subcircuit can be used as an input in all places the sub-formula appears.

The lines of a circuit can take on the values TRUE or FALSE, and these values can be propagated to other lines of the circuit using standard rules of Boolean logic. For example, if an input line of an AND gate has value FALSE then FALSE can be propagated to the output line of the gate. A circuit $\boldsymbol{Q}.C$ **represents a formula $\boldsymbol{Q}.\phi$** when for any setting of the variables in ϕ, ϕ will simplify to TRUE (FALSE) if and only if TRUE (FALSE) is propagated to the output of C given the same setting for its corresponding input lines. The construction described above yields a circuit C that represents ϕ.

Hence, we can evaluate a QBF formula $\boldsymbol{Q}.\phi$ by constructing a circuit $\boldsymbol{Q}.C$ representing it, and then evaluating the previously given definition of truth for a QBF formula by propagating values in C. That is, we can detect when ϕ simplifies to TRUE or FALSE by detecting when TRUE or FALSE is propagated to the output line of C.

3   A Circuit-Based Solver

Similar to previous circuit based SAT solvers, e.g., [7,8], our solver utilizes DPLL search to determine the truth or falsity of the QBF $Q.\phi$. Specifically, ϕ is represented as a circuit C with the variables in Q being the inputs to C. During DPLL search, these variables are branched on in an order respecting the quantifier ordering (i.e., if x is upstream of y then the search must branch on x before y). Each branch sets a variable of ϕ and hence a corresponding input line of C. The input line values are propagated through C, and the search verifies that at least one side each existential branch and both sides of each universal branch lead to a true circuit output (i.e., satisfies ϕ).

However, to make this process more efficient, e.g., to detect when certain input lines must take on a particular value for the circuit output to be TRUE, the solver initially sets the circuit output line to TRUE and propagates values backwards in the circuit as well as forwards. Backwards propagation, like forward propagation, follows the rules of Boolean logic, e.g., if an OR gate's output line is set to FALSE then FALSE can be propagated to all of its input lines. With backward propagation from a TRUE output we can detect when ϕ is falsified (i.e., when a setting of the input lines would lead to FALSE being propagated to the output line) by the occurrence of a conflict where both TRUE and FALSE is propagated to some line in the circuit. Such conflicts also allow us to employ 1UIP clause learning techniques on the circuit representation.

One negative aspect of fixing the circuit output line to TRUE, however, is that we can no longer use the propagation of TRUE to the circuit output to detect when the formula ϕ has become satisfied by the current setting of its variables—the output line always has that value. We discuss below how this problem is resolved in our solver by utilizing information gathered during don't care propagation.

3.1   Propagation

Our implementation of forward and backward propagation in the circuit is based on a previous circuit based SAT solver described in [7]. In that paper Thiffault et al. showed that this kind of propagation in the circuit corresponds in a precise way to Unit Propagation (UP) on an equivalent CNF encoding of the formula. In particular, if the circuit was converted to CNF using the standard Tseitin encoding [11], then corresponding to each circuit line l there would be a new variable v in the CNF encoding such that a value would be propagated to the line l in the circuit if and only if UP in the CNF forces v to take the same value. Besides this basic mechanism, however, our QBF solver differs from previous circuit based SAT solvers in a number of ways.

Representing internal lines. The CNF encoding of a formula introduces additional variables that correspond to the sub-formulas of the formula. These additional variables are very useful in a SAT solver as they can be branched on in a DPLL search (implicitly positing a truth value for an entire sub-formula), and they can be included in learnt clauses increasing the effectiveness of clause learning.

A key feature of our circuit based QBF solver is that it also utilizes the learning techniques common in DPLL based QBF solvers that employ CNF [12] i.e., clause and cube learning. Hence, to facilitate the power of the learnt clauses we introduce additional variables to label the internal lines of the circuit, as is done in circuit based SAT solvers. (The

input lines are all labeled with a variable of the original formula $Q.\phi$). Formally, all of these new variables are existential, and we place them as early in the quantifier ordering as possible. Specifically, each internal line l in the circuit is the output line of a sub-circuit c that has some set of input lines representing a set of variables V of ϕ. We place the new variable representing l in the quantifier prefix immediately after the last variable of V in the prefix. By placing these new variables as early as possible in the quantifier prefix we enable more effective universal reduction during clause learning and propagation.

Note however that in QBF, unlike SAT, these new variables are never branched on during the DPLL search. The DPLL search must respect the quantification order when selecting variables to branch on, so by the time it can select a variable v representing the output of sub-circuit c all of the inputs to c must have already been assigned, and hence v would already be assigned by propagation.

Universal Reduction. In CNF represented QBF formulas universal reduction is a powerful additional rule of inference that enables further unit propagation and conflict detection. We say that a universal variable is **tailing** in a clause if it is downstream of all existentials in the clause. Universal reduction is the rule of inference where all tailing universals can be removed from a clause. It can be applied during search: when an existential in a clause is falsified and thus removed from the clause some universal in the clause might become tailing and thus removable by universal reduction.

There are two cases where universal reduction can reduce DPLL search. First, it can be used to infer a conflict when a clause contains only universal variables: by universal reduction we can reduce any such clause to the empty clause. Second, it can be used to infer unit clauses when a clause becomes unit after universal reduction. In this case it must be that the clause contains a single existential variable e with all other variables in the clause being universal and downstream of e.

Our solver can detect the same set of conflicts and unit propagants arising from universal reduction as would be detected in CNF representation. Two additional propagation rules are utilized to achieve this. The first rule is triggered whenever there is a gate g such that (a) the output line of g has been assigned some value TF, (b) TF is not entailed by g's assigned input lines (e.g., if g is an OR gate, TF = TRUE, and none of g's assigned inputs are TRUE), and (c) all of g's unassigned input lines are universally quantified. In this case we have a conflict corresponding to the generation of a clause containing only universals. The second rule is triggered whenever there is a gate g such that conditions (a) and (b) as above hold, and (c) g's unassigned input lines contain only a single existential line e and all of the other unassigned lines, which are hence universal, are downstream of e. In this case we can force e to take on a value that entails TF. This corresponds to the generation of a unit clause after universal reduction. For example, if g is an AND gate, TF = FALSE, and all of g's other assigned inputs are TRUE, then e is forced to be FALSE.

3.2   Don't Care Propagation

An important way in which the circuit structure can be exploited is via don't care reasoning. For example, when one input of an OR gate is set to TRUE, the other inputs become irrelevant to its output value. By detecting the variables that have become irrelevant to all the gates they feed into, DPLL can avoid branching on them. Don't care
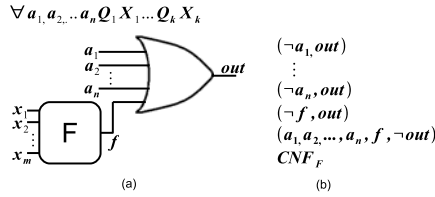
$$\forall\, a_{1,}\, a_{2,..}\, a_n Q_1 X_1 ... Q_k X_k$$

Fig. 1. Circuit and Equivalent CNF Encoding

propagation detects such variables and we implement don't care propagation in our solver using the techniques developed in [7].

Don't care propagation can be useful in SAT, but it has even more potential to be helpful in QBF due to repetitions caused by universal variables. To illustrate, consider the circuit in Figure 1, where $Q_1 X_1 ... Q_k X_k$ represents an arbitrary set of quantifiers over the variables $x_1, ..., x_m$, and F is an arbitrarily complicated boolean circuit. It can be seen that any variable assignment with at least one of $a_1, ..., a_n$ set to TRUE makes the circuit output TRUE. In our solver, as soon as one of the variables a_i is set to TRUE, all x_i variables can be recognized as irrelevant, so there is only one setting of $a_1, ... a_n$ for which the solver actually branches on any x_i variables.

A CNF based solver, on the other hand, would have the CNF representation shown in Figure 1b, where CNF_F represents the clausal encoding of F. If any of the a_i variables are set TRUE, then *out* is also set to TRUE, and all clauses disappear except for those in CNF_F. The solver will then have to continue branching on the x_i variables until a solution is found that satisfies all clauses in CNF_F, a potentially difficult task. Furthermore, we see that the solver can unnecessarily try to satisfy the clauses in CNF_F $2^n - 1$ times. A solver exploiting learning might solve these repetitions more efficiently, but can still perform many unnecessary branching operations. It is this repetition from universal variables that makes don't care reasoning more effective in QBF.

Don't care propagation is achieved by detecting when gate outputs are **justified**. A gate in the circuit is justified when its assigned inputs are sufficient to imply its output. Once a gate is justified, its unassigned inputs have no effect, so they become *irrelevant* with respect to that gate. If a line becomes irrelevant with respect to all of the gates it is an input of, it becomes a **don't care**, meaning its value has no effect on the circuit output. Further, if the don't care line is a gate output all of its unset inputs can in turn be marked as irrelevant with respect to it, which might generate another round of don't care propagation. Since these don't care variables have no effect on the circuit the DPLL search engine need never branch on them. For example, in Figure 1 once one of the a_i inputs is set to TRUE, all remaining unassigned inputs to the final OR gate will be marked as don't care: they have all become irrelevant with respect to that gate and this is the only gate they are an input to. Don't cares can then be propagated back through all of the sub-circuit F until all of the x_i are marked as don't care. After this, DPLL can detect that it need not branch on any other variables as all remaining unassigned variables (input lines and internal lines) have become don't care.

3.3   Clause Learning

Following [7] we implement clause learning in our solver by computing a clausal rea-
son from the circuit structure for every line that is assigned by propagation. As DPLL
branches on variables that correspond to input lines of the circuit, propagation is used
to set other lines in the circuit. Since each circuit line is represented by an existential
variable, propagating a value to these lines corresponds to forcing a literal representing
the assignment of this value to the line's corresponding variable. The logical structure
that allowed the value to be propagated can then be used to construct a clausal rea-
son for that forced literal. For example, if g is an AND gate with its output o set to
FALSE, all of its assigned inputs a_1, ..., a_k, set to TRUE, and with unassigned inputs
$e, u_1, \ldots, u_m$ where e is existential, the u_i are universal, and e is upstream of all of the
u_i, then propagation will set e to FALSE. In this case we can extract from the circuit
the clause $(\neg e, \neg u_1, \ldots, \neg u_m, \neg a_1, \ldots, \neg a_k, o)$ as the clausal reason for $\neg e$. Hence,
on the trail of the DPLL search engine every forced literal can be given an associated
clausal reason. Note that these clausal reasons are like the clauses that a QBF solver
using a CNF representation would use to label its unit propagated literals.

   In a similar way when conflicts are detected in the circuit a conflict clause can be con-
structed and returned to the DPLL search engine. For example, if g is an OR gate with
its output o set to TRUE, with assigned inputs $a_1, \ldots, a_k$ all set to FALSE, and unas-
signed inputs $u_1, \ldots, u_m$ all of which are universal, then a conflict corresponding to
the detection of an all universal clause is detected. From this conflict the conflict clause
$(\neg o, a_1, \ldots, a_k, u_1, \ldots, u_m)$ can be constructed and returned to the DPLL search en-
gine. With a CNF representation this is the clause that the current assignments would
have reduced to an all universal clause. The case where a line has both TRUE and FALSE
propagated to it can be handled in a similar fashion.

   With conflict clauses to seed the process, and all forced literals on the trail la-
beled with clausal reasons, our solver can proceed to perform 1-UIP clause learning
in the manner standard to DPLL-based QBF solvers and to use these clauses to non-
chronologically backtrack the DPLL search. Finally, the solver can employ unit prop-
agation over the learnt clauses in conjunction with propagation in the circuit, using
literals forced by unit propagation to set lines and do further propagation in the circuit,
and using lines set in the circuit to initiate further unit propagation in the learnt clauses.

3.4   Cube Learning

As mentioned above, because the circuit output O is initially set to TRUE we cannot use
the propagation of TRUE to O to detect that the formula has become satisfied by the cur-
rent set of variable assignments. Nevertheless, we can employ don't care propagation to
detect circuit (formula) satisfaction. In particular, when all variables that are not marked
as being don't care have been assigned and no conflicts have been generated, we know
that the circuit is satisfied by the current set of assignments. Say we had not initially set
the circuit output to TRUE. It can then be observed that whenever the assigned circuit
input lines suffice to propagate TRUE to the circuit output, all remaining unset lines in
the circuit (both internal and input lines) become don't care. It can be further observed
that the don't care propagation mechanism outlined above will successfully label these
unset lines as don't care.

Once the formula has been satisfied by the current variable assignments, we would like to perform cube learning. This involves finding a **subset** of the current variable assignments that are sufficient to satisfy the formula. Such a subset forms a base cube that can then be stored in a cube database, triggered in other parts of the DPLL search, and resolved with other cubes during search to generate more powerful cubes. A key element in making cube learning effective is to be able to generate small base cubes.[1]

With CNF representations base cubes must contain at least one true literal from each clause in the theory. With a circuit representation, however, finding a subset of variable assignments sufficient to satisfy the formula corresponds to finding a subset of the circuit inputs whose assigned values suffice to propagate TRUE to the circuit output. Don't care propagation helps in constructing small base cubes, as it eliminates from consideration all circuit inputs marked as don't care.

The algorithm we use in our solver is specified in Algorithm 1. The algorithm is initially called with the circuit's output as its input argument, and it involves sweeping through the circuit from the output to inputs picking a set of lines whose assigned values suffices to support the circuit output. Starting with the output gate, the algorithm selects a set of input lines that support the gate output. Then it continues on to find supports for the selected input lines. For example, if the gate is an AND gate with output set to FALSE then only one false input line is needed as support.

We note that we need not consider any don't care lines, all of the gate output lines the algorithm encounters have to be justified. To guide the selection of a supporting input, for each gate output line we memorize the input line that was responsible for it first becoming justified. For gate output line l we use $l.justfiedReason$ to denote this input line.

This approach for selecting a supporting input for each gate output has two advantages. First, it is very efficient to implement: the cube can be recovered in a single pass of the circuit. Second, it favours adding the earliest-set variables to the base cube which sometimes allows the solver to backtrack further.

In the algorithm specification we also use $l.gateType$ to denote the gate type that l is an output for. If l is an input line (and hence not associated with a gate) we let $l.gateType$ be equal to INPUT. Finally, let $l.inputs$ denote input lines of the gate that l is an output, and let $l.val$ denote the value assigned to l.

4   Related Work

4.1   CCDNF

In [10] Zhang proposed adding to the CNF encoding of the QBF a redundant DNF encoding, creating a Combined Conjunctive and Disjunctive Normal Form (CCDNF). The aim of the DNF encoding was to overcome the inability of CNF to easily detect when the formula becomes satisfied. The DNF allowed the resulting solver to detect solutions earlier, without needing to assign all variables in the formula.

In some cases, our circuit based solver achieves similar early solution detection through its don't care propagation. In particular, once a partial assignment is sufficient to imply the circuit output, all remaining variables will be marked as don't care and

---

[1] Other heuristic considerations come into play, but space precludes discussing them here.

Algorithm 1: RecoverCube—Construct a Base Cube from a Circuit

---

1 RecoverCube (l)
  // Return a set of supporting input lines
2 begin
3   **if** $l.gateType = INPUT$ **then**
4   **return** $\{l\}$
5   **else if** $l.gateType = NOT$ **then**
6   **return RecoverCube** ($l.inputs$)
7   **else if** $\left(\begin{array}{c} (l.gateType = AND \text{ and } l.val = \text{TRUE}) \\ or\ (l.gateType = OR \text{ and } l.val = \text{FALSE}) \end{array} \right)$ **then**
8 $S = \emptyset$ // Consider all children
9   **foreach** $c \in l.inputs$ **do**
10 $S = S \cup$ **RecoverCube** (c)
11   **end**
12   **return** S
13   **else**
             // root is a False AND gate or a True OR gate
             // Can select one child that is assigned the same
                value
14   **return RecoverCube** ($l.justfiedReason$)
15 end

---

the search engine can immediately backtrack. However, when a solution is detected, our solver must execute Algorithm 1 to extract a base cube—with the DNF encoding this computation is not needed, the information contained in the base cube is already encoded in the triggered DNF. Also the DNF encoding contains auxiliary variables that can make the cubes more compact, and perhaps more powerful. In our solver all base cubes contain input variables only.

However, the circuit representation has some advantages over IQTest. It preserves more problem structure than the CCDNF encoding. Potentially, additional ways can be discovered for further exploiting this structure. Also don't care propagation allows us to avoid branching on irrelevant variables. IQTest, on the other hand, has no way of determining when a variable is irrelevant, and can still branch on such variables prior to finding a solution. Thus, our solver can sometimes make fewer decisions during search.

Also, while making its conversion, IQTest creates two different sets of auxiliary variables: one set for the CNF, and another one for DNF representations. This limits the amount of knowledge sharing between the two representations. The circuit representation has only one set of auxiliary variables (variables representing the internal lines), so that the different modes of reasoning share the same representation.

Nevertheless, given that the circuit representation contains all of the information used to generate the DNF encoding, it is possible that the computational advantages of the DNF encoding can be captured directly from the circuit representation. We plan to investigate this possibility in future work.

The empirical results in the next section show that while IQTest outperforms our solver on some benchmarks, there are domains where the advantages of the circuit representation are evident.

4.2   Don't Care Literals

In [13], the authors augment the CNF encoding by adding *don't care literals* to clauses so they can be marked as redundant when the don't care literals become true. This is effectively the same as our solver marking a circuit line as redundant, but as they point out, they are unable to encode all don't care conditions. By dynamically detecting don't care conditions as they occur, we are able to detect more don't care variables during search than their static method.
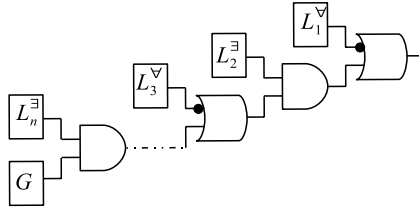

Fig. 2. Adversarial game encoding

4.3   Dual CNF and DNF

[9] created a dual CNF-DNF encoding geared towards addressing the inadequacies of CNF when encoding adversarial games. Their approach is to encode the rules for the universal player in a DNF. Then, if the universal player ever violates the rules, the DNF portion is detected to be true, and the existential player is declared the winner.

The main benefit of their approach—determining when the universal player cheats—is easily achieved in our solver by exploiting a circuit representation. Figure 2 shows an example of a circuit encoding an arbitrary two player game with n turns [14]. A box labeled $L_i^\exists$ represents a sub-circuit encoding the rules for the existential player in move i, while $L_j^\forall$ encodes the rules for the universal player in move j. At any move, if the universal player violates their rules, all remaining moves in the game become don't care, and the existential player is declared the winner (i.e., TRUE is propagated to the circuit output).

4.4   Negation Normal Form

In [15] the authors discuss a solver qpro for formulas in Negation Normal Form (NNF). The main focus of qpro is relaxing the restriction of a prenex form. This is orthogonal to our approach, and the circuit solver can be extended in a similar manner.

However, even without explicitly dealing with non-prenex formulas, don't care propagation together with clause and cube learning can often achieve similar results, and our solver is quite competitive with qpro even in the domains with very short but wide quantifier trees. This is demonstrated in the experimental results.

The backtracking technique of qpro—relevance sets—involves the solver computing the set of variables whose values determined the truth or falsity of the formula, and allows the solver to backtrack non-chronologically over irrelevant variables. The idea of identifying relevant sets underlies the notions of clause and cube learning. Learnt

cubes and clauses also allow a solver to backtrack non-chronologically over unrelated variables, with the added benefit that the learnt cubes and clauses can be utilized in the rest of the search. Since our solver implements cube and clause learning it does not need to compute relevance sets.

5   Experimental Results

Our solver CirQit implements the ideas described in this paper. Its input is a circuit description in ISCAS-85 format using AND, OR and NOT gates, along with the quantifier prefix. The solver first simplifies the circuit by merging identical subformulas. It then solves the circuit using DPLL search running on the circuit representation as described above.

Table 1. Comparison between CirQit and other state-of-the-art non-CNF non-Prenex solvers. The largest number of instances solved is shown in **bold**, with ties broken by the time taken to solve those instances.

Benchmark Families	CirQit		qpro		pQBF	
(number of instances)	Solved	Time	Solved	Time	Solved	Time
Seidl (150)	147	2,281	**150**	7	13	3,326
assertion (120)	**3**	1	1	0	0	0
consistency (10)	0	0	0	0	0	0
counter (45)	**39**	1,315	31	126	31	161
dme (11)	**10**	15	10	1,193	5	287
possibility (120)	**10**	1,707	0	0	0	0
ring (20)	**15**	60	9	397	9	158
semaphore (16)	**16**	7	16	91	16	726
Total (492)	**240**	5,389	217	1,816	74	4,660

We compared CirQit with state-of-the art CNF and non-CNF solvers on all the non-Prenex, non-CNF benchmarks currently available from QBFLIB [16]. Unless otherwise stated, all tests were run on a 2.8GHz machine with 12GB of RAM. The results display the number of problems that each solver was able to solve within the time limit of 1200 CPU seconds per instance, and the total time taken for all the **solved** instances, rounded down to the nearest second.

Table 1 shows the comparison against the two top solvers from the non-prenex non-CNF track of the QBFEVAL'08 competition. One of the solvers is qpro (discussed above), version of 29.02.08 available from the authors' site. The other one is pQBF [17].[2] The benchmarks, originally in QBF1.0 format, were converted into ISCAS-85 format for CirQit and into *pro* format for qpro. Conversion time was negligible and was not included in the results.

---

[2] On some instances pQBF gave a *parser stack overflow* error. In a few cases, it proceeded to return an answer. This happened on large instances in benchmark families for which pQBF timed out on smaller problems. The answer returned under these circumstances was always FALSE, and on at least one instance it was confirmed to be incorrect by multiple other solvers. This led us to believe that this answer was returned in error. The results presented here consider such instances as failure cases for pQBF.

Table 2. Comparison between CirQit 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.

Benchmark Families (number of instances)	CirQit		sKizzo		2clsQ		yquaffle		quantor		Qube	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
Seidl (150)	**147**	2281	37	6,301	0	0	0	0	42	3,272	144	4,688
assertion (120)	3	1	14	796	49	7,035	23	114	**119**	8,736	3	0
consistency (10)	0	0	1	40	0	0	0	0	**10**	720	0	0
counter (45)	**39**	1,315	34	1,185	30	89	31	1,077	28	414	29	1225
dme (11)	**10**	15	0	0	0	0	0	0	0	0	6	75
possibility (120)	10	1,707	13	700	13	1,666	10	505	**111**	7,976	10	25
ring (20)	**15**	60	12	752	11	1,048	12	607	11	479	15	1,781
semaphore (16)	**16**	7	14	68	13	47	7	261	16	12	14	1,833
Total (492)	240	5,389	125	9,844	116	9,888	83	2,566	**337**	21,613	212	9,629

qpro outperforms CirQit on only the Seidl dataset (shown in Table 1). This dataset contains problem instances that typically have short but wide quantifier trees, a structure that qpro is particularly well suited to exploit. Although CirQit performs worse than qpro on this dataset, we can see that it outperforms all the other solvers (including the CNF-based solvers discussed below), and is one of only two solvers that come close to qpro on this dataset.

Other than the Seidl dataset, CirQit dominates the other two non-CNF solvers: it was able to solve all the problems that they solved, and also some additional ones.

Table 2 compares CirQit against a number of CNF-based solvers. In order to apply the CNF-based solvers, the benchmarks were converted from QBF1.0 to qdimacs format using the translator available from QBFLIB webpage. Again, the conversion times were not included.

The solvers tested were sKizzo (v0.8.2) [18], 2clsQ [19], yQuaffle (version 21006) [12], quantor (version 3.0, with the recommended picosat back end) [5] and Qube (version 6.1) [20]. These solvers are state-of-the-art QBF solvers as shown by QBFEVAL competition results. The predecessors of solvers sKizzo and Quantor were the best two solvers at QBFEVAL'05; 2clsQ and sKizzo took first and second places at QBFE-VAL'06; Qube won QBFEVAL'07 with yQuaffle being the next-best standalone solver (disregarding the solvers based on a portfolio approach), and Qube6.1 was the best standalone solver at the QBFEVAL'08 competition– second only to a solver based on a portfolio approach.

Comparing with the CNF solvers, we see that CirQit is quite competitive with them. It outperforms all of the CNF solvers on a number of domains. For domains counter and dme, CirQit is able to solve a number of problems that no CNF-based solver could solve; for each of ring and semaphore domains, CirQit is tied with one CNF-based solver (Qube and quantor, respectively) on the number of solved instances but wins based on the time taken to solve them, and does notably better than the other solvers.

The domains on which CirQit does not outperform the CNF solvers are the assertion, consistency and possibility domains, which are all part of the set *BMC_QBF_1.0*. Note that all of the non-CNF solvers perform badly on this benchmark set. On these problems, Quantor is the clear winner. It also far outperforms all the other solvers on these benchmarks. Quantor does not employ DPLL search, using instead a combination
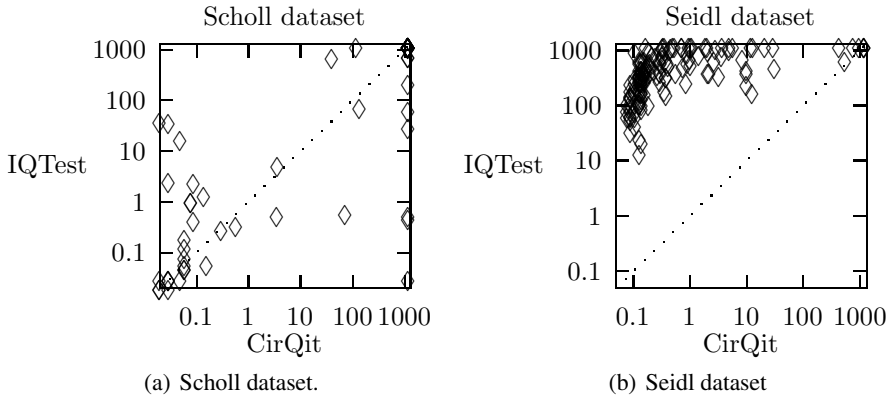
(a) Scholl dataset.　　　　　　　　　(b) Seidl dataset

Fig. 3. Time comparison between CirQit and IQTest on two benchmark sets

of resolution and universal quantification to reduce the formula. Clearly this approach is better for these problems than any form of DPLL search.

Finally, we compared our solver with the previously discussed IQTest [10], which uses both a CNF and an DNF encoding of the problem. We were not able to perform a comprehensive comparison with IQTest due to the fact that IQTest is available only as a Windows executable. We were, however, able to experiment with two datasets. The first dataset is the Scholl dataset that was used to demonstrate IQTest in the paper [10]. The second dataset we tested is the Seidl dataset from QBFLIB.

A plot comparing the runtimes for the Scholl and Seidl datasets is shown in Figure 3. The experiments were run on a 2.41GHz machines with 2GB of RAM. An instance is plotted with the time CirQit took to solve it on the x-axis and IQTest on the y-axis. So, an instance above the bisecting line is one on which our solver exhibited superior performance, and an instance below the line is one where IQTest was superior. Timed out instances are placed at the 1200 second mark on the graph.

On the first dataset, IQTest outperforms CirQit. There are eight problems that IQTest was able to solve, sometimes fairly quickly, but CirQit was unable solve in the time allotted. However, there were also a number of problems that CirQit was able to solve a few orders of magnitude faster than IQTest. On the Seidl dataset, on the other hand, CirQit confidently outperforms IQTest. The detailed results were that on Scholl, containing 63 problems, CirQit solved 38 problems in 382 seconds, while IQTest solved 46 problems in 3,887 seconds. On the other hand, on Seidl, containing 150 problems, CirQit solved 147 problems in 2,969 seconds, while IQTest solved 126 problems in 53,110 seconds.

Although this is not a complete analysis, these sets show that while IQTest is better than CirQit on some problems, there are problem suites for which CirQit is better suited.

Finally, although we don't show any results, we did experiment with CirQit turning don't care propagation on and off. Over a large number of problems we found that don't care propagation yielded on average almost a 40% speedup.

6   Conclusions and Future Work

This paper demonstrates the effectiveness of exploiting structural information in QBF solving. By skipping the last step of encoding QBF problems into CNF, the structure of the problem can be maintained and used by a DPLL search engine. While other work has been done in the past to overcome some of the limitations of the CNF representation, our circuit based solver includes many of the benefits realized by these partial solutions.

We demonstrated that a solver using a circuit representation can be highly competitive with state of the art solvers using both non-CNF and CNF representations.

The circuit representation is compact, and allows more powerful propagation. Many more benefits could potentially be reaped from the circuit representation. The circuit representation allows the solver to generate CNF clauses for clause learning on-the-fly. We believe that it is possible to use the circuit in a similar way to extract DNF cubes on the fly. We are investigating this approach.

Many orthogonal improvements, such as exploiting non-prenex structure or using problem decomposition, can also be applied to the circuit solver.

In sum it seems that using a circuit representation is a very fruitful direction for obtaining further advances in QBF solving.

References

1. Rintanen, J.: Asymptotically optimal encodings of conformant planning in QBF. In: Proceedings of the AAAI National Conference (AAAI), pp. 1045–1050 (2007)
2. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified boolean formulas. In: Proceedings of the AAAI National Conference (AAAI), pp. 417–422. AAAI Press, Menlo Park (2000)
3. Mangassarian, H., Veneris, A.G., Safarpour, S., Benedetti, M., Smith, D.: A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In: International Conference on Computer-Aided Design (ICCAD), pp. 240–245 (2007)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
5. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)
6. Benedetti, M.: sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03 (2004)
7. Thiffault, C., Bacchus, F., Walsh, T.: Solving non-clausal formulas with DPLL search. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT) (2004)
8. Wu, C.A., Lin, T.H., Lee, C.C., Huang, C.Y.: QuteSAT: a robust circuit-based SAT solver for complex circuit structure. In: Design, Automation and Test in Europe Conference and Exposition (DATE), pp. 1313–1318 (2007)
9. Sabharwal, A., Ansótegui, C., Gomes, C.P., Hart, J.W., Selman, B.: QBF modeling: Exploiting player symmetry for simplicity and efficiency. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 382–395. Springer, Heidelberg (2006)
10. Zhang, L.: Solving QBF with combined conjunctive and disjunctive normal form. In: Proceedings of the AAAI National Conference (AAAI) (2006)

11. Tseitin, G.: On the complexity of proofs in poropositional logics. In: Siekmann, J., Wright-son, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, vol. 2. Springer, Heidelberg (1983); Originally published (1970)

12. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quanti-fied boolean formula evaluation. In: Van Hentenryck, P. (ed.) CP 2002, vol. 2470, pp. 200–215. Springer, Heidelberg (2002)

13. Tang, D., Malik, S.: Solving quantified boolean formulas with circuit observability don't cares. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 368–381. Springer, Heidelberg (2006)

14. Benedetti, M., Lallouet, A., Vautard, J.: QCSP made practical by virtue of restricted quantifi-cation. In: Veloso, M.M. (ed.) Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 38–43 (2007)

15. Egly, U., Seidl, M., Woltran, S.: A solver for QBFs in negation normal form. Con-straints 14(1), 38–79 (2009)

16. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001), www.qbflib.org

17. Stéphan, I.: Boolean propagation based on literals for quantified boolean formulae. In: 17th European Conference on Artificial Intelligence (2006)

18. Benedetti, M.: skizzo: A suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 369–376. Springer, Heidelberg (2005)

19. Samulowitz, H., Bacchus, F.: Dynamically partitioning for solving QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 215–229. Springer, Heidelberg (2007)

20. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A system for deciding Quantified Boolean Formulas satisfiability. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 364–369. Springer, Heidelberg (2001)