Exploiting Circuit Representations in QBF Solving

Alexandra Goultiaeva and Fahiem Bacchus

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

Abstract. Previous work has shown that circuit representations can be exploited in QBF solvers to obtain useful performance improvements. In this paper we examine some additional techniques for exploiting a circuit representations. We discuss the techniques of propagating a dual set of values through the circuit, conversion from simple negation normal form to a more optimized circuit representation, and adding phase memorization during search. We have implemented these techniques in a new QBF solver called CirQit2 and evaluated their impact experimentally. The solver has also displayed superior performance in the non-prenex non-CNF track of the QBFEval'10 competition.

1 Introduction

Quantified Boolean Formulas (QBF) are a PSPACE-complete extension of satisfiability (SAT) in which the propositional variables can be either universally or existentially quantified. The addition of quantifiers, and the arbitrary nesting of quantifiers, provides considerable additional representational power: QBFs can compactly represent a much wider range of problems than SAT. This can make QBF more effective than SAT for representing and solving some problems [1].

While traditionally QBF solvers have represented their input in prenex conjunctive normal form (CNF), some recent work has explored non-CNF solvers. For example, in [2] a QBF solver utilizing a circuit representation was presented and shown to have some advantages over CNF based solvers.

In this paper we present some additional ways of exploiting a circuit representation to obtain better performance. We present an overview of dual propagation (explored more fully in [3]) which provides superior detection and learning of solutions; some techniques for converting negation normal form to a more optimized circuit representation; and extend the phase memorization technique utilized in SAT solvers [4] to QBF. We have implemented all of these techniques as an extension of the CirQit solver [2], and we present empirical evidence supporting their usefulness.

2 Background

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 $\exists x$) one for each variable in ϕ (i.e., the formula

O. Strichman and S. Szeider (Eds.): SAT 2010, LNCS 6175, pp. 333–339, 2010.

has no free variables).[1] The truth value of a QBF is defined recursively: $\exists x \boldsymbol{Q}.\phi$ is true iff there is at least one value v of x for which $\boldsymbol{Q}.\phi|_{x=v}$ is true, and $\forall x \boldsymbol{Q}.\phi$ is true iff $\boldsymbol{Q}.\phi|_{x=v}$ is true for both values v of x.

Typically QBF solvers represent the body of the QBF, ϕ, in CNF. However, ϕ can be represented in other ways. In circuit-based solvers, e.g., the CirQit solver of [2], ϕ is represented as a **logical circuit** consisting of AND, OR and NOT gates along with lines running from gate outputs to the inputs of other gates. The quantified variables of $\boldsymbol{Q}$ are the inputs to the circuit and the output line of each gate is labeled with a new variable. Formally, these variables can be treated as new existentially quantified auxiliary variables scoped by all of the input variables with a path to them in the circuit.

The CirQit solver explores different settings of the input variables during a backtracking search and propagates **primal values** through the circuit to determine logical consequences of these settings. During search the aim is to determine when the QBF body ϕ is satisfied. This occurs when the circuit inputs force the circuit output to be true. Hence, CirQit initially sets the circuit output to 1 and propagates this **primal** value back through the circuit. Whenever a circuit line is forced to be both 0 (false) and 1 (true) (by propagation from the inputs and from the output's primal value) a contradiction is detected: the circuit cannot be satisfied under the current input settings. Similarly, a contradiction is detected whenever the value of a universally quantified input is forced.

As shown in [2] clausal reasons for each forced line can be extracted from the circuit, and clause learning can be performed when conflicts are detected. Further, when ϕ is satisfied **cubes** can be extracted by finding a subset of the input lines sufficient to propagate 1 to the circuit output. Finally, the circuit representation naturally supports don't care reasoning. In particular, certain lines of the circuit, can be marked as don't care during a don't care propagation process. Input lines that become don't care do not need to be branched on during search, and do not need to appear in the extracted cubes. This can yield better (smaller) cubes than the technique employed in CNF solvers.[2]

3 New Techniques for Exploiting the Circuit Representation

3.1 Dual Propagation

While clauses are used in QBF solvers to learn settings that falsify the QBF formula, cubes are used to learn settings that satisfy the formula. However, cubes are usually not as effective as clauses in improving the efficiency of QBF solvers. In particular, the solver starts off with no cubes, and the cubes it learns initally are usually quite large and thus only useful quite deep in the search tree. On the other hand, the solver starts off with many short clauses in its input (in the circuit representation these clauses are implicit in the logical relationships between the gate inputs and its output), and even initially learnt clauses can be quite short.

[1] This is the prenex form where the quantifiers precede the formula body. Non-prenex representations allow quantifiers to appear in front of any sub-formula of the body. We do not address the possible advantages of non-prenex representations in this paper.

[2] In CNF a cube is extracted by finding a set of true literals sufficient to satisfy every clause. Even clauses from the don't care part of the circuit must be satisfied.

In [3] a new technique is presented that allows the identical technique of clauses and clause learning to be used to detect both satisfying and falsifying input settings. This new technique is called **dual propagation**, and we present a brief overview of its main ideas here, leaving the details and comparison with prior methods to [3].

Consider the negation of the QBF being solved $\neg \boldsymbol{Q}.\phi$. This formula is false iff the original QBF $\boldsymbol{Q}.\phi$ is true. Taking the negation in we obtain $(\neg \boldsymbol{Q}).(\neg \phi)$, where $\neg \boldsymbol{Q}$ is the same as $\boldsymbol{Q}$ except that its 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, by simply passing the output of C_ϕ through a NOT gate.

If we want to solve $\neg \boldsymbol{Q}.\phi$ with a circuit based solver we would take the circuit NOT(C_ϕ) and set its output to 1/true. 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 search various settings of the circuit input lines. These are identical to the input lines of C_ϕ but have reversed quantifiers. Propagation and clause learning operate just as before.

Conflicts discovered while solving $\neg \boldsymbol{Q}.\phi$ are actually solutions for $\boldsymbol{Q}.\phi$; if $\neg \boldsymbol{Q}.\phi$ cannot be made true under the current input settings, $\boldsymbol{Q}.\phi$ cannot be made false. Propagation on $\neg \boldsymbol{Q}.\phi$ can force variables that in $\boldsymbol{Q}.\phi$ are universal. This indicates that in $\boldsymbol{Q}.\phi$ the other value is guaranteed to lead to a solution and need not be explored. Similarly, clause learning on $\neg \boldsymbol{Q}.\phi$ yields clauses that if falsified indicate that $\boldsymbol{Q}.\phi$ is satisfied. Thus, propagation and clause learning on $\neg \boldsymbol{Q}.\phi$ detects settings that satisfy $\boldsymbol{Q}.\phi$ using the same techniques used on $\boldsymbol{Q}.\phi$ to detect settings that falsify $\boldsymbol{Q}.\phi$.

The circuit representation can be exploited to implement this idea by simply propagating two sets of values through the circuit: the original **primal** values generated by setting the circuit output to 1, and a new set of **dual** values generated by setting the circuit output to 0. Since both $\neg \boldsymbol{Q}.\phi$ and $\boldsymbol{Q}.\phi$ have the same input lines (with flipped quantifiers) the input lines always have identical primal and dual values. Thus values can be transferred between the primal and dual channels via the input lines.

As before, backtracking search sets input lines, and both primal and dual values are propagated through the circuit. Thus propagation might set either or both values of the auxiliary variables. Contradictions are detected from both the primal and dual values, and in either case a clause is learnt and the solver backtracks. Primal and dual clauses are put in separate databases: unit propagating primal clauses forces primal values while unit propagating dual clauses forces dual values.

It can be seen that a primal conflict causes backtrack and an existential of $\boldsymbol{Q}.\phi$ to be forced (the opposite value must falsify $\boldsymbol{Q}.\phi$), while a dual conflict causes backtrack and a universal of $\boldsymbol{Q}.\phi$ to be forced (the opposite value must truthify $\boldsymbol{Q}.\phi$). 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 a conflict with the other value. The solver terminates on learning either an empty primal or an empty dual clause indicating that the input QBF is false or true respectively. Finally, don't care propagation can be extended to work with dual propagation.

3.2 Better Circuits from NNF

Negation normal form (NNF) has recently been adopted as the standard non-clausal input format for the QBF evaluation. However, because NNF involves pushing all negations down to the level of the propositional variables, it can obscure structure in the input formula. For example, consider the propositional formula $(c \vee F(\boldsymbol{x})) \wedge (d \vee \neg F(\boldsymbol{x}))$, where $F(\boldsymbol{x})$ is some sub-formula over the variables $\boldsymbol{x}$. In a circuit representation a single sub-circuit, $C_{F(\boldsymbol{x})}$, could be used to represent $F(x)$, its output feed into the gate OR$(c, F(\boldsymbol{x}))$, and the negation of its output feed into the gate OR$(d,$ NOT$(F(\boldsymbol{x}))$. Initially, if the formula must be true, the output of both OR gates is forced to 1. Then, e.g., if during search c was set to 0, propagation would set the output line of $C_{F(\boldsymbol{x})}$ to 1, the output line of NOT$(F(\boldsymbol{x}))$ to 0, and thus force d to 1. If this formula was first converted to NNF, then a separate sub-circuit, $C_{\neg F(\boldsymbol{x})}$, would have to be constructed for $\neg F(\boldsymbol{x})$ which would share only inputs (and negated inputs) with $C_{F(\boldsymbol{x})}$. Depending on the complexity of $C_{F(\boldsymbol{x})}$, it is likely that forcing the output of $C_{F(\boldsymbol{x})}$ would have no effect on the output of $C_{\neg F(\boldsymbol{x})}$—e.g., if the output of $C_{F(\boldsymbol{x})}$ failed to propagate down to any of its inputs.

Here we suggest a technique for recovering structure that might be lost in an NNF representation. We define an inductive relation of structural equivalence $\overset{i}{\equiv}$ and structural negated equivalence $\overset{ni}{\equiv}$ between two propositional formulas represented in NNF:

- For every literal l, $l \overset{i}{\equiv} l$ and $l \overset{ni}{\equiv} \neg l$
- $G_1 \circ \cdots \circ G_n \overset{i}{\equiv} F_1 \circ \cdots \circ F_n$ for $\circ \in \{\wedge, \vee\}$ if $G_i \overset{i}{\equiv} F_{f(i)}$ for every $i \in \{1, \ldots, n\}$ under some bijection $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$.
- $G_1 \circ \cdots \circ G_n \overset{ni}{\equiv} F_1 \bullet \cdots \bullet F_n$ for $\{\circ = \wedge, \bullet = \vee\}$ or $\{\circ = \vee, \bullet = \wedge\}$ if $G_i \overset{ni}{\equiv} F_{f(i)}$ for every $i \in \{1, \ldots, n\}$, under some bijection $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$.

It can proved by a simple induction that if $G \overset{i}{\equiv} F$ then G and F represent the same boolean function, and that if $G \overset{ni}{\equiv} F$ then G and $\neg F$ represent the same boolean function. A simple example of this definition is that $(a \wedge (b \vee c)) \overset{ni}{\equiv} (\neg a \vee (\neg b \wedge \neg c))$.

To simplify an NNF we first generate for every subformula F a function identifier (id) $f_{id}(F)$ and a negated function id $nf_{id}(F)$ so that for any two subformulas G and F, $G \overset{i}{\equiv} F$ iff $f_{id}(G) = f_{id}(F)$, and $G \overset{ni}{\equiv} F$ iff $f_{id}(G) = nf_{id}(F)$.

This accomplished by using hashing techniques to identify structurally equivalent sub-formulas (similar to how BDDs are constructed). Ids for the subformulas are assigned bottom up so that a subformula obtains an id only after its children subformulas have obtained theirs. For each variable v unique ids are created for v and for $\neg v$. In addition we set $nf_{id}(v) = f_{id}(\neg v)$ and $nf_{id}(\neg v) = f_{id}(v)$. Each function id assigned is kept in a hash table. The subformula $F_1 \circ \cdots \circ F_k$ is assigned an id by first sorting its children F_i by their ids. Then an id is computed that is a function of the operator $\circ$ ($\vee$ or $\wedge$) and the ids of the sorted children. Similarly a negated id is computed by repeating the computation using the dual operator of $\circ$ and the (already computed) negated ids of the children (resorted by their negated ids). Both ids are stored in the hash table. It is not hard to show that this procedure generates ids that satisfy the conditions stated above, i.e., $G \overset{i}{\equiv} F$ iff $f_{id}(G) = f_{id}(F)$, and $G \overset{ni}{\equiv} F$ iff $f_{id}(G) = nf_{id}(F)$.

Using the subformula ids an optimized circuit can now be constructed to represent the NNF. Again we build up the circuit bottom up, first creating input lines corresponding

to $f_{id}(v)$ and $nf_{id}(\neg v)$ for each variable. To construct the circuit for the subformula $F = F_1 \circ \cdots \circ F_k$ we first check to see if subcircuits for $f_{id}(F)$ or $nf_{id}(F)$ have already been constructed. If so we simply reuse the output line (or output line run through a NOT gate) as the line representing F. Otherwise we create a new gate representing $\circ$ whose inputs are the outputs of the subcircuits corresponding to the F_i, and mark this as the circuit that has already been constructed for $f_{id}(F)$. This allows the subcircuit constructed for F to be reused to represent future subformulas.

This process creates a more compact circuit. In particular, the circuit needs only one sub-circuit (and perhaps an additional NOT gate) to represent an entire set of subformulas $\{F_1, \ldots, F_k\}$ of the NNF with either $F_i \overset{i}{\equiv} F_j$ or $F_i \overset{ni}{\equiv} F_j$ $(1 \leq i, j \leq k)$.

3.3 Simplifying the NNF

Using the ids constructed as described above we can perform some further simplifications of the NNF. This utilizes two easy to verify logical rules: $\phi \wedge F(\phi) \equiv \phi \wedge F(\text{TRUE})$ and $\phi \vee F(\phi) \equiv \phi \vee F(\text{FALSE})$, where ϕ is any propositional formula.

The algorithm is simple and is applied before the circuit representation is generated: using the f_{id} and nf_{id} identifiers, any repeated occurrences of a subformula in its sibling subformula can be replaced with a constant according to the above logical rules. The NNF is then simplified, function ids regenerated, and the rules are applied again. This process is repeated until no further simplifications are possible, after which the circuit representation can be generated.

3.4 Phase Memorization

Phase memorization in SAT involves setting variables to their previously set values on the new descent after backtrack [4]. The main idea rests on the intuition that often a problem decomposes into different subproblems. Reusing the previous values after backtrack allows the solver to retain some of the work done on other subproblems.

In QBF, we have two types of backtrack—success and failure. Also, universal and existential variables play very different roles in QBF. We consider this interplay in light of the above idea and in light of the now completely symmetric processes of failure and success backtracking under dual propagation.

In a conjunction of different subproblems, the universal variables in one subproblem cannot affect the solution of another one—all possible settings need to be checked anyway. This leads to the idea that remembering existential variables might be beneficial after a conflict is generated on one of the subproblems, but that the universal settings are irrelevant. Symmetrically, if a solution is found, only the settings of the universal variables are important.

Based on this intuition, we implemented the following phase memorization schema: we restore phase values only for those variables that are existential with respect to the primal or dual conflict that occurs. This means that phases are restored for the existential variables when backtracking from a conflict, and for universal variables when backtracking from a solution.

4 Experimental Results

We have modified the solver CirQit2 [3], which includes dual propagation. The new solver, CirQit2.1, is slightly more optimized, reads NNF input and includes all techniques discussed here. The experiments were run on all the non-Prenex, non-CNF benchmarks currently available from QBFLIB [5]. All tests were run on a 2.8GHz machine with 12GB of RAM under a time limit of 1200 CPU seconds per instance.

Table 1 shows the comparison between CirQit2.1, CirQit2 [3], CirQit [2] and some state-of-the-art CNF-based QBF solvers: quantor (version 3.0, with the recommended picosat back end) [6], Qube (version 6.5) [7], nenofex [8] and depqbf [9]—the latter two are the versions submitted to the main track of QBFEVAL'10. The non-prenex non-CNF instances are converted to prenex CNF using a conversion tool available on the QBFEval site. Internally, CirQit converts input instances to prenex form using a naive algorithm that places the variables in the same order in which they are encountered.

On the domains from the set *BMC_QBF_1.0* ("assertion", "consistency" and "possibility"), the bottom-up solver quantor outperforms CirQit2.1, as well as any other search-based solver. With this exception, CirQit2.1 proves to be superior. The drastic difference between CirQit and CirQit2 shows the effectiveness of dual propagation, while the improvement of CitQit2.1 shows the effectiveness of the other techniques. Each technique contributes to the improvement. Turning off any one of them reduces the performance of CitQit2.1. Without generating better circuits, it can solve only 344 problems (out of 492); without simplification, 340; without phase memorization, 344. Overall the preprocessing techniques reduced the number of variables by 0.99%, but only because they were less useful on the more numerous benchmark families.

Table 1. Comparison between CirQit2.1 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. Percent decrease in the number of variables (from CirQit2.1 to CirQit2) is also shown.

	CirQit2.1			CirQit2		CirQit		quantor		Qube6.5		nenofex		depqbf	
	Solved	Time	% Dec	Sol.	Time	Sol.	Time	Sol.	Time	Sol.	Time	Sol.	Time	Sol.	Time
Seidl (150)	**150**	43	12.63	150	318	147	2,281	42	3,272	149	2,485	82	1,160	150	557
assertion (120)	48	16,166	0.67	40	14,503	3	1	**119**	8,736	6	1,180	12	4,170	24	145
consistency (10)	7	2,562	0.51	4	1,283	0	0	**10**	720	0	0	1	306	0	0
counter (45)	**43**	1,368	7.77	40	492	39	1,315	28	414	31	540	29	1,727	31	70
dme (11)	**11**	11	27.59	10	5	10	15	0	0	7	88	8	94	11	901
possibility (120)	57	17,535	0.50	45	16,121	10	1,707	**111**	7,976	14	4,713	12	4,037	10	143
ring (20)	**20**	40	24.31	20	53	15	60	11	479	16	189	11	4	13	243
semaphore (16)	**16**	2	39.99	16	3	16	7	16	12	16	361	16	1,193	16	39
Total (492)	**352**	37,728	0.99	325	32,779	240	5,389	337	21,613	239	9,557	171	12,692	255	2,098

References

1. 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, pp. 240–245 (2007)

2. Goultiaeva, A., Iverson, V., Bacchus, F.: Beyond CNF: A circuit-based QBF solver. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 412–426. Springer, Heidelberg (2009)
3. Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: Proceedings of the AAAI National Conference (2010) (accepted for publication)
4. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
5. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001), `www.qbflib.org`
6. Biere, A.: Resolve and expand. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 238–246. Springer, Heidelberg (2005)
7. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A system for deciding Quantified Boolean Formulas satisfiability. In: Proceedings of the International Joint Conference on Automated Reasoning, pp. 364–369 (2001)
8. Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF solving. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 196–210. Springer, Heidelberg (2008)
9. Lonsing, F., Biere, A.: A compact representation for syntactic dependencies in QBFs. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 398–411. Springer, Heidelberg (2009)