Quantified Boolean Formula Solver

Solving QBF by converting to EPR

David Green
BSc(Hons) Computer Science and Mathematics
Project report for third year project under the supervision of Konstantin Korovin done in the School of Computer Science at the University of Manchester

May, 2016

Abstract

This paper details the tool *qbftoepr* that converts quantified boolean formulas (QBFs) to effectively propositional logic (EPR) through a process of skolemization. This opens new techniques for solving QBFs using automated theorem provers for first order logic such as iProver [2] over traditional techniques for solving QBFs such as (a variation of) the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3]. Other techniques are discussed for improving the efficiency of conversion and reducing the complexity of the EPR result. In particular, an implementation of dependency schemes is detailed and the concept of anti-prenexing is outlined. The tool is evaluated against the traditional QBF solver DepQBF [4] to compare the efficiency of converting and solving to solving the QBF directly. It is also evaluated against another tool called qbf2epr [5] that also converts QBFs to EPR to compare it against a different implentation of the same conversion.

Contents

Chapter 1

Background

First we must introduce the terminology and concepts of boolean logic including propositional logic, first order logic (including EPR) and quantified propositional logic. After the terminology has been introduced the complexity classes of the satisfiability problem of each logic will be discussed. Finally the concept of automated reasoning of both first order logic and QBF will be detailed.

1.1 Boolean Logic and Satisfiability

Boolean logics and satisfiability (SAT) form a calculus which can be used to reason about statements. There are different formal systems of logic that can be used for different purposes, we shall look at propositional logic, QBF and first order logic.

1.1.1 Propositional Logic

A propositional variable p can take one of two values; either *true* or $false$. A variable can be negated using the *negation* ($\neg$) symbol which reverses its value. If p was *true* then $\neg p$ is $false$ and vice versa. We will call a variable or its negation a *literal* and denote the positive literal by l and the negative literal by $\bar{l}$. Boolean formulas are constructed from propositional variables built using the logical connectives *disjunction* ($\vee$), *conjunction* ($\wedge$) and *implies* ($\rightarrow$). A typical boolean formula might look like $(x \wedge y) \rightarrow z$.

The satisfiability of a boolean formula is a decision problem that asks if an assignment of truth values to propositional variables can make the boolean formula true. The aforementioned logical connectives tell us how to combine the truth values of two variables. With a disjunction, $x \vee y$ is true if either (or both) x or y is true. In a conjunction, $x \wedge y$ is true if both x and y are true. An implication $x \rightarrow y$ can be read as "if x is true then y is true." with the case of x being false defined as being vacuously true. In the previous example of $(x \wedge y) \rightarrow z$ we can see that it is satisfiable as the assignment $x := true; y := true; z := true$ makes it true.

We require a more standard form of boolean formula that is easier to describe as an input format. For this we will use conjunctive normal form (CNF). CNF is a conjunction of clauses and a clause is a disjunction of literals. For example, a clause might be $(p \vee \neg q)$ and a full formula in CNF might look like $(p \vee r) \wedge (\neg r \vee q) \wedge (q)$. Using CNF allows us to more easily work with boolean formulas algorithmically.

1.1.2 Quantified Boolean Formulas

QBF extends propositional logic with the *universal* ($\forall$) and *existential* ($\exists$) quantifiers. The statement $\forall x \exists y (x \vee y)$ states that for every assignment of x there is at least one assignment of y such that the formula $(x \vee y)$ is true. We can see that this is true; if $x := true$ then the formula is true but if $x := false$ then the assignment $y := false$ doesn't work but that $y := true$ does make the formula satisfiable. Therefore, for any assignment of x there exists an assignment of y such that the formula is true.

In the most general case quantifiers can appear anywhere in a QBF. Again we need a more standard form of QBF that we can deal with algorthmically. This form is called *prenex* CNF (however we may refer to it by just CNF assuming that the formula is prenexed). Any QBF is logically equivalent to a CNF formula and the process for transforming the QBF into CNF is called *prenexing*. This process uses rewriting rules to move all the quantifiers in the formula to the leftmost part of the formula resulting in a *quantifier prefix*. For example, $(\neg(\exists x A) \wedge B$ is equivalent to $\forall x(\neg A \wedge B)$. Because all QBFs are equivalent to some QBF in CNF we shall assume that any QBF we are dealing with is already in CNF.

quant level?

1.1.3 First Order Logic

First order logic uses propositions that take variables or functions as arguments to form its formulas. These variables range over a specified problem domain such as the natural numbers. For example in thedomain of the natural numbers the formula $\forall n \exists m P(n, m)$ where $P(n, m) = m > n$ is true; for any natural number n there is a number m that is larger than n. This differs from our previous definition of QBF in that QBF deals with only variables in a two valued domain (i.e. boolean) and does not have propositions.

Our notions of prenexed CNF also extend to first order logic.

As with propositional logic and QBF we require a way to write our formulas that is convenient to work with. In this case we will use EPR, formally known as the *Bernays-Schönfinkel class* of formulas. A formula is in EPR form if when it is written in CNF it has the quantifier prefix $\exists * \forall *$ and contains no functions. This format will be useful because we can solve these problems using first order logic theorem provers.

1.2 Complexity of Satisfiability

Boolean Logics are significant in complexity theory as they are standard embeddings for other problems in their complexity classes.

1.2.1 SAT is NP-complete

An NP problem is one that can be solved by a non-deterministic algorithm that runs in time relative to a polynomial of the size of the input. SAT is one such problem. Stephen Cook proved in 1971 that the SAT problem is NP-complete [6] meaning that any other NP problem can be reduced to the SAT problem. This spurs much of the interest into SAT solvers, if we can solve the SAT problem in polynomial time then we can solve any NP problem in polynomial time too. This is known as the P=NP problem.

1.2.2 QBF is PSPACE-complete

A PSPACE problem is one that can be solved by a deterministic algorithm that runs using space relative to a polynomial of the size of the input. QBF belongs to this complexity class and can be shown to be PSPACE-complete using Savitch's theorem [7]. NP problems are a subset of PSPACE problems and it is not yet known if the two classes are equal or not. Because these algorithms run much slower than general SAT algorithms there has been much less interest in QBF solvers outside academia compared to SAT solvers.

1.2.3 EPR is NEXPTIME-complete

Similarly to NP problems, NEXPTIME problems are solved by non-deterministics algorithms running in time relative to an exponential of the size of the input. EPR was shown to be NEXPTIME-complete by Harry Lewis in 1980 [8]. While this does mean that algorithms for proving satisfiability of EPR are in general slower than algorithms for propositional satisfiability we are still interested to see how EPR solvers fare when given inputs derived from QBFs.

1.3 Automated Reasoning

Automated reasoning tools use algorithms to solve these SAT problems as well as other more general logic based problems based around deduction to find a result that isn't necessarily satisfiability. This brings artificial intelligence interests into the research in an effort to create artificial intelligences that can perform deductive reasoning. This project makes heavy use of the automated reasoning tool iProver [2] to solve the SAT problems generated by *qbftoepr*.

Chapter 2

Converting QBF to EPR

Detailed conversion goes here

Now that we have introduced the concepts behind QBF and EPR we can look at the procedure that *qbftoepr* implements in greater depth. The algorithm is composed of three main steps detailed below. We shall follow an example through the process from input to output. The example QBF we will work with is the following formula

$$\begin{aligned} &\forall w \forall x \forall y \exists z \\ &(y \lor z) && \land \\ &(x \lor \neg z) && \land \\ &(\neg x \lor w) \end{aligned} \tag{2.1}$$

2.1 Raising QBF to First Order Logic

The input to *qbftoepr* is in QBF form so it has propositional variables with no notion of predicates. We require an output in first order logic so the inputted QBF must be 'raised' to first order logic before the algorithm continues.

This 'raising' is relatively straightforward as most of the symbols used in the QBF are also used in first order logic with the only difference being the variables and predicates. For example, the conjunction symbol used in QBF can be used in first order logic with the same meaning. To raise the propositional variables used in the clauses to first order logic we introduce a predicate of arity 1 that takes the variable as an argument. For example the propositional variable x would be raised to the predicate $p(x)$. Finally, the predicate p must be defined. This is achieved by adding two clauses $(p(true)) \land (\neg p(false))$ to define how p handles truth values in the new domain. Repeating this process recursively over an inputted QBF gives us an *equisatisfiable* formula in first order logic. This means that the QBF is satisfiable if and only if the version translated into first order logic is satisfiable.

In the case of the example QBF 2.1 raising it to first order logic gives the following formula

$$\begin{array}{ll} \forall w \forall x \forall y \exists z & \\ (p(y) \vee p(z)) & \wedge \\ (p(x) \vee \neg p(z)) & \wedge \\ (\neg p(x) \vee p(w)) & \wedge \\ (p(true)) & \wedge \\ (\neg p(false)) & \end{array} \tag{2.2}$$

2.2 Removing Existential Quantifiers by Skolemization

Once we have embedded our QBF in first order logic we can begin to turn it into EPR. This process is called Skolemization [9]. The first step in this process is to remove the existential quantifiers and replace the variables they quantify with functions that take as arguments the variables that the existential variable 'depends on'. What we mean by 'depends on' will be covered in greater depth in section 2.4. Strictly speaking since EPR does allow existentially quantified variales at the start of the prefix replacing every existentially quantified variable is not completely necessary but we do require it for the output format. These outermost existential variables will be replaced by constants. Similarly to raising the formula to first order logic this process of Skolemization gives an equisatisfiable formula.

We shall apply skolemization to our example QBF 2.2 after it has been raised to first order logic assuming naïvely that our existential variables depend on every universal variable. This gives the following formula

$$\begin{array}{ll} \forall w \forall x \forall y & \\ (p(y) \vee p(f_z(w,x,y))) & \wedge \\ (p(x) \vee \neg p(f_z(w,x,y))) & \wedge \\ (\neg p(x) \vee p(w)) & \wedge \\ (p(true)) & \wedge \\ (\neg p(false)) & \end{array} \tag{2.3}$$

2.3 Removing Function Symbols Introduced by Skolemization

The final step in converting our QBF to EPR is to remove the function symbols that were introduced by skolemization. This is done by 'lifting' the functions to predicate. Lifting the functions to predicates means creating a new predicate whose arguments are the variables in the function being lifted. For example $p(f_z(x, y))$ would become the predicate $p_z(x, y)$. Once again this process produces a new formula that is equisatisfiable to the previous formula.

Once all the functions have been lifted to predicates we have our EPR result. The definition of EPR required the prefix to be in the form $\exists * \forall *$ which was achieved by skolemization to remove all the existentially quantified variables and it also required there to be no function symbols which was achieved by lifting the functions to predicates. This result can then be used as the input to an EPR solver to determine its satisfiability. Because each step in the process preserved satisfiability proving (or disproving) the satisfiability of the EPR output gives the satisfiability of the original QBF input.

This is our example QBF 2.4 after lifting its functions to predicates

$$\begin{array}{ll} \forall w \forall x \forall y & \\ (p(y) \vee p_z(w, x, y)) & \wedge \\ (p(x) \vee \neg p_z(w, x, y)) & \wedge \\ (\neg p(x) \vee p(w)) & \wedge \\ (p(true)) & \wedge \\ (\neg p(false)) & \end{array} \tag{2.4}$$

2.4 Dependency Schemes

In section 2.2 existentially quantified variables were replaced by a function whose arguments were the universally quantified variables that the existentially quantified variable 'depended on'.

Chapter 3

Technical Details

probably give it a better name

3.1 Language Choice

3.2 Input and Output Formats

3.2.1 QDIMACS

3.2.2 TPTP

3.3 Data Structures

3.4 Algorithms

3.4.1 Skolemization

3.4.2 Removing Functions

3.4.3 Dependency Scheme Construction

3.4.4 Complexity

3.5 Testing

Chapter 4

Future work

Future work goes here

4.1 Dependency Scheme Optimisations

4.2 Anti-prenexing

Chapter 5

Evaluation

evaluation goes here

5.1 Comparison Against Direct QBF Solvers

5.2 Comparison Against EPR Converters

Chapter 6

Project plan

project plan goes here

Chapter 7

Conclusion

conclusion goes here

Chapter 8

Bibliography

[1] University of Manchester logo from Wikipedia by source, fair use
https://en.wikipedia.org/w/index.php?curid=43485475
2014

[2] Korovin, Konstantin. "iProver-an instantiation-based theorem prover for first-order logic (system description)." Automated Reasoning. Springer Berlin Heidelberg, 2008. 292-298.

[3] Davis, Martin, George Logemann, and Donald Loveland. "A machine program for theorem-proving." Communications of the ACM 5.7 (1962): 394-397.

[4] Lonsing, Florian, and Armin Biere. "DepQBF: A dependency-aware QBF solver." Journal on Satisfiability, Boolean Modeling and Computation 7 (2010): 71-76.

[5] Seidl, Martina, Florian Lonsing and Armin Biere. "qbf2epr: A Tool for Generating EPR Formulas from QBF." PAAR@ IJCAR. 2012.

[6] Cook, Stephen A. "The complexity of theorem-proving procedures." Proceedings of the third annual ACM symposium on Theory of Computing. ACM, 1971.

[7] Savitch, Walter J. "Relationships between nondeterministic and deterministic tape complexities." Journal of computer and system sciences 4.2 (1970): 177-192.

[8] Lewis, Harry R. "Complexity results for classes of quantificational formulas." Journal of Computer and System Sciences 21.3 (1980): 317-353.

[9] Skolem, Thoralf. “Logisch-Kombinatorische Untersuchungen über die Erf’üllbarkeit oder Beweisbarkeit Mathematischen Sätze nebst einem Theoreme über Dichte Mengen.” Translated from: Van Heijenoort, Jean. “From Frege to Gödel: a source book in mathematical logic, 1879-1931.” Vol. 9. Harvard University Press, 1967.

Acronyms