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

1 Background **3**
- 1.1 Boolean Logic and Satisfiability . . . 3
  - 1.1.1 Propositional Logic . . . 3
  - 1.1.2 Quantified Boolean Formulas . . . 4
  - 1.1.3 First Order Logic . . . 5
- 1.2 Complexity of Satisfiability . . . 5
  - 1.2.1 SAT is NP-complete . . . 5
  - 1.2.2 QBF is PSPACE-complete . . . 5
  - 1.2.3 EPR is NEXPTIME-complete . . . 6
- 1.3 Automated Reasoning . . . 6

2 Converting QBF to EPR **7**
- 2.1 Raising QBF to First Order Logic . . . 7
- 2.2 Removing Existential Quantifiers by Skolemization . . . 8
- 2.3 Removing Function Symbols Introduced by Skolemization . . . 9
- 2.4 Dependency Schemes . . . 9
  - 2.4.1 Trivial Dependency Scheme . . . 10
  - 2.4.2 Standard Dependency Scheme . . . 10

3 Development **13**
- 3.1 Language Choice . . . 13
- 3.2 Input and Output Formats . . . 14
  - 3.2.1 QDIMACS . . . 14
  - 3.2.2 TPTP . . . 15
- 3.3 Data Structures . . . 16
- 3.4 Implementation . . . 16
  - 3.4.1 Skolemization . . . 16
  - 3.4.2 Removing Functions . . . 16
  - 3.4.3 Dependency Scheme Construction . . . 16
  - 3.4.4 Complexity . . . 16
- 3.5 Optimizations . . . 16

3.6 Future Work 16
3.6.1 Dependency Scheme Optimizations 16
3.6.2 Anti-prenexing 16

4 Evaluation **17**
4.1 Testing 17
4.2 Comparison Against Direct QBF Solvers 17
4.3 Comparison Against EPR Converters 17

5 Reflection & Conclusion **18**
5.1 Milestones 18
5.2 Experiences 18
5.3 Conclusion 18

6 Bibliography **19**

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 algorithmically. 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.

Another useful notion will be the idea of an order to a prenexed QBF's prefix. If a variable x is quantified to the left of another variable y in the prefix such as $\exists x \forall y$ we say that x is quantified before y. Using this idea we can assign a quantification level to the variables so that x has a quantification level of 1 and y has a quantification level of 2. The variable with the lowest quantification level is called the outermost quantified variable and the variable with the highest quantification level is the innermost.

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 the domain 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-deterministic 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

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

$$\begin{aligned} &\forall w \forall x \forall y \exists z \\ &(y \vee z) \quad \wedge \\ &(x \vee \neg z) \quad \wedge \\ &(\neg x \vee w) \end{aligned} \tag{2.1}$$

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 $(p(true)) \wedge (\neg p(false))$ 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 variables 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'. A dependency scheme maps a variable to the variables that it depends on but this map must be computed. A dependency scheme is called tractable if it can be computed in polynomial time (proportional to the length of the formula). However, the problem of deciding whether a given dependency scheme is the optimal dependency scheme is P-SPACE complete (proved by Marko Samer and Stefan Szeider [10]) so it is impractical to compute the optimal dependencies every time. Dependency schemes are important because they can vastly affect the time

to solve a given formula. If the number of variables a variable depends on is reduced the solver does not have to consider all of those extra dependencies and so can solve the formula more quickly.

2.4.1 Trivial Dependency Scheme

The simplest method of assigning dependencies to a variable is to say that it depends on everything that was quantified before it with a different quantifier. For the existentially quantified variables being considered for Skolemization this means that they depend on any variable universally quantified before them. For example in the prefix $\exists w \forall x \forall y \exists z$ the variable z depends on x and y because they were universally quantified before it but not w because it was universally quantified before it. This is called the trivial dependency scheme and is clearly tractable by searching the prefix for universally quantified variables.

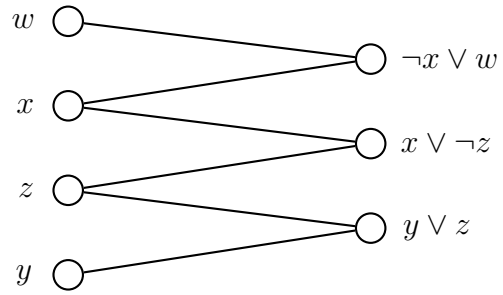
2.4.2 Standard Dependency Scheme

The standard dependency scheme is harder to define so this description will not cover it in great depth. A full description can be found in the aforementioned paper from Samer and Szeider [10].

We will look at the idea of dependency from the opposite perspective; given a variable, say x, what variables depend on x? We must first define $R(x)$ to be the all the existential variables quantified to the right of x. Then we have a notion of dependency pairs in which two variables x and y (with y quantified to the right of x) form a dependency pair (x, y) if they are 'connected' with respect to $R(x)$. The two variables x and y are connected with respect to $R(x)$ if in the incidence graph of the formula there exists a path through the graph from a clause containing x to a clause containing y that only travels through clauses that contain at least one variable in $R(x)$. The incidence graph is a bipartite graph with a variable joined to a clause if the variable is in the clause. A path starts at one clause, travels to a variable in $R(x)$ and from that variable travels to a clause and so on until it reaches a clause containing y. Trivially, if x and y are in the same clause then y depends on x.

The incidence graph of our example formula is below. Trivially we can see that z depends on both x and y because it appears in clauses with both of them but compared to the trivial dependency scheme it hasn't given w as a dependency of z because there is no path from w to z in the graph traveling only through clauses containing variables in the set $R(w) = \{z\}$, the only

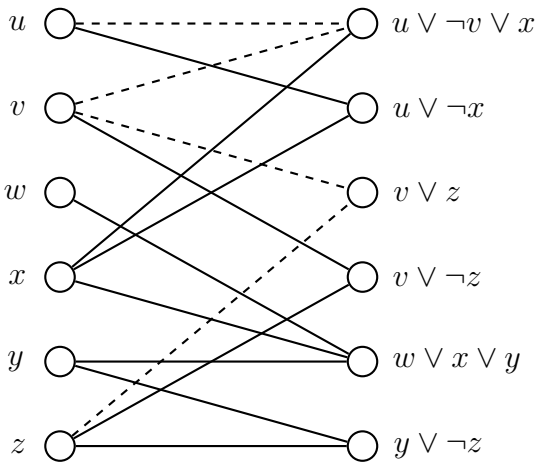
way to do so is to go via x which is not in $R(w)$. Despite being quantified to the right of w, x was quantified universally so it is not included.



To see less obvious dependencies with the standard dependency scheme we need a slightly more complex example. Consider the following formula.

$$\begin{array}{ll} \forall u \exists v \forall w \exists x \forall y \exists z & \\ (u \vee \neg v \vee x) & \wedge \\ (u \vee \neg x) & \wedge \\ (v \vee z) & \wedge \\ (v \vee \neg z) & \wedge \\ (w \vee x \vee y) & \wedge \\ (y \vee \neg z) & \end{array} \tag{2.5}$$

This formula has the following bipartite graph.



It is not obvious from the formula that under the standard dependency scheme that z depends on u. However in this case $R(u) = \{v, x, z\}$ and a path starting at the clause $(v \vee z)$ can travel through v (because it is in $R(u)$) and thus reach the clause $(u \vee \neg v \vee x)$ which contains u. This is the path labeled in dashed lines on the graph. We can also see that z does not depend on w as a path to w must travel through x or y, neither of which are in $R(w)$.

The standard dependency scheme is tractable [10] because we can work backwards from a given variable x to see what variables depend on it doing a linear search across the graph and upon finding a variable that matches the criteria it is added to the list of variables that depend on x.

Chapter 3

Development

Development goes here

This chapter will describe the design and implementation of *qbftoepr* at a high level with some of the technical choices and why they were made as well as some of the optimizations and compromises that were taken to improve performance.

3.1 Language Choice

The project was implemented in the language OCaml [11] which is a functional language notable for the language extensions that give it object oriented and imperative functionality. The main motivator for implementing *qbftoepr* in OCaml is the fact that iProver is implemented in OCaml which provides three advantages:

- Code from iProver can be re-used easily in *qbftoepr*
- Using the same language in *qbftoepr* as iProver makes post-project maintenance easier
- Using the same language would allow tighter integration when *qbftoepr* passes its output to iProver (though this wasn't done in practice)

Even without these advantages OCaml would be a suitable language given that it has been designed with performance as a priority and has excellent built-in functions for manipulating lists of which the project makes extensive use.

3.2 Input and Output Formats

The input and output formats were an important choice but one that was relatively easy to make. The decision was to go with the standards already in use by other theorem provers to be able to compare *qbftoepr* to them on exactly the same inputs. Another consideration was that the input format had to have a relatively simple grammar to simplify the input handling and the output format had to be simple enough to make printing the output efficient. The output format must also be accepted as an input format by iProver.

3.2.1 QDIMACS

QDIMACS [12] is the input format decided on for *qbftoepr*. It is the format used by the QBFLIB [13] which is a library of QBF problem instances. QBFLIB also holds a competition called QBFEVAL which tests QBF solvers against each other using the QDIMACS format. This makes it the standard of the QBF research community and a clear choice of input format for *qbftoepr*. It also opens access to the QBFLIB problem library providing test cases of all difficulty scales and the results of other solvers on these test cases. It also has a very simple grammar meaning that parsing a QDIMACS input is simple.

Below is our example from chapter 2 in QDIMACS form.

```
c this is a comment
p cnf 4 3
a 1 2 3 0
e 4 0
3 4 0
2 -4 0
-2 1 0
```

Lines beginning with c are comments, p denotes the problem line which tells us the problem is in CNF and has four variables and three clauses. Lines prefixed with a are universally quantified variables, e are existential. Lines after the prefix form the matrix which is the list of clauses. Numbers represent variables and a negative number represents the negative literal of that variable, i.e. $\bar{l}$. The prefix and matrix lines are appended with a zero as a line terminator.

3.2.2 TPTP

The output format chosen for *qbftoepr* is TPTP [14]. Similarly to QBFLIB, the TPTP is a large problem library for automated theorem proving and as such it is one of the input formats that iProver accepts. The popularity of the TPTP library means it is also used by other automated theorem provers. Because the TPTP is a general problem library (whereas QBFLIB is specifically QBF problems) its grammar is very complicated. However *qbftoepr* does not need to parse it to output it and only requires a small subset of the format's features to output the result of the EPR conversion process.

Below is the example from chapter 2 after being converted to EPR in the TPTP output format.

```
cnf(cl_0,plain,(p(X3) | p_f_4(X1,X2,X3))).
cnf(cl_1,plain,(p(X2) | ~p_f_4(X1,X2,X3))).
cnf(cl_2,plain,(~p(X2) | p(X1))).
cnf(cl_3,plain,(p(true))).
cnf(cl_4,plain,(~p(false))).
```

Each line is a clause, named *cl_x* where x is an identifier. Following is the *plain* keyword which says there are no user defined semantics then the list of literals where ~ denotes the negation of a literal.

3.3 Data Structures

3.4 Implementation

3.4.1 Skolemization

3.4.2 Removing Functions

3.4.3 Dependency Scheme Construction

3.4.4 Complexity

3.5 Optimizations

3.6 Future Work

3.6.1 Dependency Scheme Optimizations

3.6.2 Anti-prenexing

Chapter 4

Evaluation

evaluation goes here

4.1 Testing

4.2 Comparison Against Direct QBF Solvers

4.3 Comparison Against EPR Converters

Chapter 5

Reflection & Conclusion

conclusion goes here

5.1 Milestones

5.2 Experiences

maybe better name

5.3 Conclusion

Chapter 6

Bibliography

[1] University of Manchester logo from Wikipedia by source, fair use
`https://en.wikipedia.org/w/index.php?curid=43485475`
Uploaded 6th August 2014, Accessed 11th April 2016

[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." Translation sourced from: Van Heijenoort, Jean. *From Frege to Gödel: a source book in mathematical logic, 1879-1931.* Vol. 9. Harvard University Press, 1967.

[10] Samer, Marko, and Stefan Szeider. "Backdoor sets of Quantified Boolean Formulas." *Journal of Automated Reasoning* 42.1 (2009): 77-97.

[11] OCaml homepage
`https://ocaml.org`
Accessed 18th April 2016.

[12] QDIMACS grammar definition
`http://www.qbflib.org/qdimacs.html`
Released 21st December 2005, accessed 18th April 2016.

[13] QBFLIB homepage
`http://qbflib.org`
Accessed 18th April 2016.

[14] TPTP homepage
`http://www.cs.miami.edu/\~tptp/`
Accessed 18th April 2016.

Acronyms

$\exists$ *existential*. 4

$\forall$ *universal*. 4

$\wedge$ *conjunction*. 3

$\vee$ *disjunction*. 3

$\neg$ *negation*. 3

$\rightarrow$ *implies*. 3

CNF conjunctive normal form. 4, 5, 14

DPLL Davis-Putnam-Logemann-Loveland. 1

EPR effectively propositional logic. 1, 3, 5, 7, 8, 9, 14, 15

QBF quantified boolean formula. 1, 3, 4, 5, 7, 8, 9, 14

SAT satisfiability. 3, 5, 6