Quantified Boolean Formula Solver

Solving QBF by converting to EPR

David Green
BSc(Hons) Computer Science and Mathematics
Project report for the third year project under the supervision of Konstantin Korovin 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 implementation of the same conversion.

Acknowledgments

Thanks to Konstantin Korovin for his help, advice and guidance during the project.

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.

more formal definitions

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

of x or y or both are 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 QBF is satisfiable. If $x := true$ then the formula is true and any assignment of y will work. If $x := false$ then the assignment $y := false$ makes the formula false but the assignment $y := true$ does make the formula true. Therefore, for any assignment of x there exists an assignment of y such that the formula is true.

In the most general case a quantified variable can appear in front of any sub-formula of 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 lots of interest into SAT solvers because 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 in memory 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 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 formula 2.1.

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

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 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 formula 2.2.

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

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 (called Skolem 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 formula 2.3.

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

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.

Formula 2.4 is formula 2.3 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', this will now be defined. 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

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 linearly 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.

Figure 2.1 is the incidence graph of formula 2.1. 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 doesn't depend on w 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

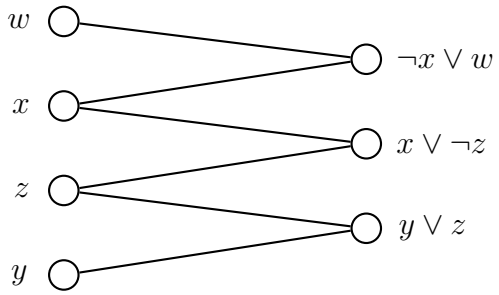


Figure 2.1: Bipartite graph for formula 2.1

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 formula 2.5.

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

The incidence graph for formula 2.5 is figure 2.2.

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 (full proof by Samer and Szeider [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.

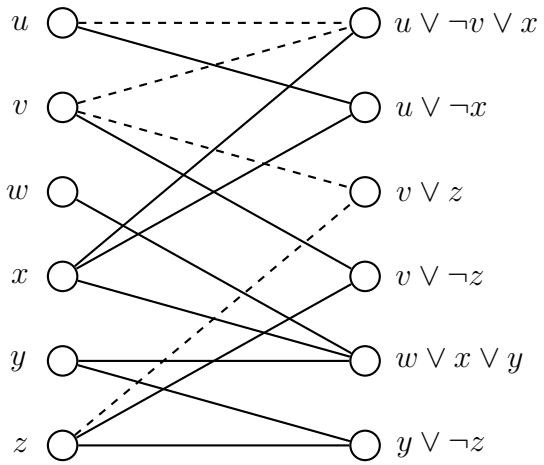


Figure 2.2: Bipartite graph for formula 2.5

Chapter 3

Development

This chapter will describe the design and implementation of *qbftoepr* 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 motivation 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 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.

This chapter will contain snippets of (simplified) OCaml code so some understanding of the syntax is required. Most of the syntax is straightforward with the exception that function arguments don't require brackets and functions always return the result of their last command. The `match` feature is used to handle custom types by comparing a variable against a list of types and defining actions to perform in the case of each of the types. OCaml variables are also immutable by default so code is written in a way that creates new data structures and populates them with the changes to the data from the original data structures rather than modifying the original data structures directly.

Figure 3.1: Factorial example

```
let rec factorial n =
  if n <= 1 then 1
  else n * factorial (n - 1)
```

The factorial example 3.1 illustrates some of these features. In the if statement the last command executed is `1` so the function returns `1` in that case. If the else statement is reached then the result of the multiplication is returned. The call to `factorial` in the multiplication is passed one argument; the result of `(n-1)`.

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 supported by *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.

Figure 3.2: QDIMACS example

```
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
```

Figure 3.2 is the example from chapter 2 in QDIMACS form. 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 and it only requires a small subset of the format's features to output the result of the EPR conversion process.

Figure 3.3: TPTP example

```
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))).
```

Figure 3.3 is the example from chapter 2 after being converted to EPR in the TPTP output format. 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

The design of the data structures is very important because it determines the functions that can be used to manipulate them. OCaml has excellent built in support for lists so *qbftoepr* makes extensive use of lists in its data structures. It is also important to avoid too much nesting of types to minimize the extra code required to delve into the type structure.

A `qbf` and a `fol_qbf` (first order logic QBF) have the same basic structure. They are formed of a `record` type which can be though of as similar to a `struct` in C like languages; they have several typed fields. These records have a `prefix` and a `matrix` which are defined as a list of the `quantified_variable` and `clause` types respectively. The `quantified_variable` type is another record which has information about the quantification level, the quantifier and the variable. The `clause` type is defined as a list of `literal`s and the `literal` level is where the regular `qbf` and `fol_qbf` start to differ. A `literal` is a record with the sign (positive or negative) and either an `atom` (another name for a variable) in the case of a regular `qbf` or a `predicate` in the case of a `fol_qbf`.

A `predicate` has a name and a list of arguments which are either an `atom`, a `func`, `true` or `false`. A `func` also has a name and a list of arguments but it only takes `atom` arguments. This extra complexity in the `literal`s of `fol_qbf`s drives some important optimization decisions that will be discussed in section 3.6.

3.4 Implementation

This section will tie together the theoretical background from chapter 2 with the practicalities of implementation.

3.4.1 Input & Output

Input is handled by the OCaml variants of the Lex and Yacc tools for C called OCamllex and OCamlyacc. Tokens are defined with regular expressions and the input text is matched to the regular expressions and parsed into tokens.

The tokens are given to code that builds the `qbf` data structure out of the input which is then passed to the main body of the code. The simplicity of the QDIMACS grammar was helpful here to keep the regular expressions and token structure simple.

Outputting the result in TPTP is fairly straightforward. The EPR result is just a list of clauses so given the `fol_qbf` the `prefix` can be discarded and each `clause` in the `matrix` list can simply be printed to the output file.

3.4.2 Raising to First Order Logic

Raising a `qbf` to a `fol_qbf` is a relatively simple procedure. The idea is to delve into the `prefix` and `matrix` lists of the `qbf` and copy the values into the `fol_qbf` record.

The `quantified_variable` types can be copied verbatim as they do not change between `qbf` and `fol_qbf` but the `literal`s require more care. When a `literal` is found the sign is copied and the `atom` is added as the argument to a new `predicate` which is copied into the new `literal`.

The simplified OCaml code snippet 3.4 shows the raising process. The `matrix` is iterated through using `List.map` which applies the given function to each element in the list and puts the result into a new list. When a `literal` is found either a positive or negative `literal` is created with the `atom` from the old `literal`. The raising process introduces two new `clause`s which are appended to the `matrix`.

Figure 3.4: Raising to first order logic

```
let convert_qbf_to_fol qbf =
  {prefix = convert_qbf_prefix_to_fol qbf.qbf_prefix;
   matrix =
     convert_matrix_to_fol qbf.qbf_matrix
     @ clauses_for_introduced_predicate}

let convert_matrix_to_fol qbf_matrix =
  List.map convert_clause_to_fol qbf_matrix

let convert_clause_to_fol clause =
  List.map convert_literal_to_fol clause

let convert_literal_to_fol literal =
  match literal.sign with
    Pos -> make_pos_literal
      (make_predicate fol_pred_name [Atom(literal.atom)])
  | Neg -> make_neg_literal
      (make_predicate fol_pred_name [Atom(literal.atom)])
```

3.4.3 Skolemization

Skolemization is the more algorithmically challenging stage of the QBF to EPR conversion process. Section 2.2 discussed the background; replace each existential variable with a Skolem function. The implementation first builds an association list (a list of key value pairs) with the existential variables for keys and the Skolem function that will replace them as the values (this is also when the Skolem functions are built). The original `matrix` is then linearly scanned with its values being copied into a new `matrix` but with the instances of the variable keys in the association list replaced by the Skolem function value that corresponds to that key. This new `matrix` and a new `prefix` with all the existential variables removed are returned as the Skolemized `fol_qbf`.

Code snippet 3.5 shows some of the implementation of Skolemization. Of note are the functions `build_skolem_func_list` and `get_skolemized_predicate`. The first recursively builds the association list with the `prefix` reduced by one existential variable at each point, returning the current association list or an empty list in the base case where the prefix has no existential variables. The latter makes the assumption that the predicate has only one argument that is an `atom`, this assumption and others like it will be discussed in the optimizations section 3.6.

Figure 3.5: Skolemization

```
let skolemization fol_qbf dep_scheme =
  let skolem_list = build_skolem_func_list fol_qbf.prefix dep_scheme in
  build_fol_qbf
  (prefix_without_existential_variables fol_qbf.prefix)
  (get_skolemized_matrix fol_qbf.matrix skolem_list)

let rec build_skolem_func_list prefix dep_scheme =
  try
    let outermost = List.find (fun var -> var.quantifier = E) prefix in
    let func_args = List.assoc outermost.atom dep_scheme in
    [outermost.atom, build_skolem_function outermost func_args]
    @ (build_skolem_func_list
        (prefix_without_quantifier prefix outermost)
        dep_scheme)
  with Not_found -> []

let get_skolemized_matrix matrix skolem_list =
  List.map (get_skolemized_clause skolem_list) matrix

let get_skolemized_clause skolem_list clause =
  List.map (get_skolemized_literal skolem_list) clause

let get_skolemized_literal skolem_list literal =
  {sign = literal.sign;
   pred = get_skolemized_predicate skolem_list literal.pred}

let get_skolemized_predicate skolem_list predicate =
  let atom = List.hd predicate.pred_arguments in
  match atom with
    Atom(n) ->
      make_predicate
      predicate.pred_name
      (replace_atom_with_skolem_func n skolem_list)
  | _ -> predicate
```

3.4.4 Removing Functions

Removing the `funcs` from the `fol_qbf` is straightforward. the `matrix` is iterated over once and when a `func` is found a new `predicate` is created

with the `func`'s arguments to be used in the new `matrix`. This is similar to the process of replacing existential variables with a Skolem function during Skolemization. The simplified OCaml listing 3.6 shows the replacement process. When a `literal` is reached if its predicate matches a `func` the new `predicate` is generated otherwise the original `predicate` is returned.

Figure 3.6: Removing Functions

```
let remove_functions_from_fol_qbf fol_qbf =
  let new_matrix = remove_functions_from_matrix fol_qbf.matrix in
  build_fol_qbf fol_qbf.prefix new_matrix

let remove_functions_from_matrix matrix =
  List.map remove_functions_from_clause matrix

let remove_functions_from_clause clause =
  List.map remove_function_from_literal clause

let remove_function_from_literal literal =
  {sign = literal.sign;
   predicate = remove_function_from_predicate literal.predicate}

let remove_function_from_predicate predicate =
  match List.hd predicate.predicate_arguments with
    Func(f) -> raise_function_to_predicate f
  | _ -> predicate
```

3.4.5 Dependency Scheme Construction

A dependency scheme is implemented as an association list where the key is the variable and the value is a list of the variables that the key depends on.

The trivial dependency scheme is a matter of finding all `quantified_variable`s quantified universally before a given existential `quantified_variable`. This is implemented by filtering the `prefix` to just the `quantified_variable`s with both the universal quantifier and a lower `quant_level` than the given variable. The implementation of this filtering is shown in the code snippet 3.7. This is performed on every existential `quantified_variable` and compiled into the association list.

Figure 3.7: Trivial Dependency Scheme

```
let universally_quantified_below_quant_level prefix quant_level =
  List.filter (universally_quantified_before quant_level) prefix

let universally_quantified_before quant_level var =
  var.quantifier = A && var.quant_level < quant_level
```

The optimal implementation for the standard dependency scheme involves a graph search as described in section 2.4.2 however time constraints necessitated a more straightforward but less efficient implementation. The trade-offs are discussed more in section 5.1.1, this section concentrates on the current implementation.

The current implementation of the standard dependency scheme searches through the `matrix` for clauses containing the given variable. This creates a new `matrix`-like structure that must be flattened to give the list of variables in the same clauses as the given variable. This list is then further pruned to just those variables that were universally quantified before the given variable which gives a list of the variables it depends on. However this list must be filtered once more to remove the duplicate variables which is accomplished using the List module's `sort_uniq` function. An extract of this implementation can be seen in the code snippet 3.8 which omits some boilerplate code around converting between `atom`s and `quantified_variables`.

This implementation is not correct; it misses some dependencies. The example formula 2.5 in section 2.4.2 shows a non-obvious dependency that this implementation would miss. A fix for this is discussed in section 5.1.1.

Figure 3.8: Standard Dependency Scheme

```
let std_dep_scheme_for_variable qbf var =
  let all_potential_atoms =
   List.sort_uniq
     compare_atoms
     (atoms_of_clauses (clauses_containing_atom var.variable qbf.matrix)) in
  [var.variable,
   List.filter
     (universally_quantified_before_variable qbf.prefix var)
     all_potential_atoms]

let universally_quantified_before_variable prefix var_one atom_two =
  let quant_var_two = quantified_variable_for_atom atom_two prefix in
  universally_quantified_before var_one.quant_level quant_var_two

let clause_contains_atom atom clause =
  List.exists (fun literal -> literal.atom = atom) clause

let clauses_containing_atom atom matrix =
  List.filter (clause_contains_atom atom) matrix
```

3.5 Complexity

Realistic formulas can easily run into tens or hundreds of thousands of variables so reducing the complexity is a priority in order to keep the runtime tractable.

3.5.1 Raising to First Order Logic

Raising the `qbf` to `fol_qbf` runs in time linear in the length of the formula. The `prefix` can be copied directly and the `matrix` is scanned once building a new `matrix` with `predicate`s for the new `matrix` created from the `atom`s of the old `matrix` in a one to one correspondence. Two new clauses are added but this is a constant time factor so can be ignored.

3.5.2 Skolemization

The Skolemization algorithm first builds the Skolem variable association list. This runs in time linear to the length of the prefix because one function is

created per existential variable. The function creation takes constant time, the name is created from the predicate's name and the variable's name and the function's arguments are given by the dependency scheme's association list.

Replacing the variables in the `matrix` runs in time proportional to the length of the `matrix` multiplied by the length of the Skolem list created in the first step which is itself proportional to the length of the `prefix`. The `matrix` is only scanned once but every `literal` must be checked to see if its variable is in the Skolem list which must be scanned once to find it.

3.5.3 Removing Functions

Removing the functions from a `matrix` runs in time linear in the length of the formula. The original `matrix` is iterated over once linearly and when a `func` is found a new `predicate` is created and used in its place in the new `matrix`.

3.5.4 Dependency Schemes

The trivial dependency scheme is computed in linear time. The `prefix` is scanned once for variables matching the criteria of being universally quantified before the given variable.

The current implementation of the standard dependency scheme is not very well optimized. An optimization is discussed in section 5.1.1. The current implementation performs one pass over the `matrix` to build the list of variables in the same clause as the given variable and has to repeat this for every existential variable. Therefore it runs in time proportional to the length of the `matrix` multiplied by the length of the `prefix`.

3.6 Optimizations

The first working implementation had many inefficiencies which had to be optimized to properly evaluate the project. The performance gains from these optimizations will be examined in detail in chapter 4. This section will outline the optimizations that were implemented.

3.6.1 Skolemization

The original implementation of Skolemization was closer to the theoretical idea of Skolemization wherein the instances of an existential variable are

replaced by Skolem functions and this process is repeated for each existential variable. However this approach rebuilds the entire `matrix` at each step which is an incredibly costly operation. The current implementation performs only one rebuilding of the `matrix` after computing the variables to be replaced and the Skolem functions replacing them up front rather than when needed.

Another smaller optimization is the way that a `predicate`'s arguments are scanned. Because the arguments must be implemented as a list the original implementation scanned this list of arguments to replace them individually with a Skolem function or to check if they are a function that must be removed. However at this stage in the conversion process the `predicate` can only ever have one argument so it suffices to look only at the head of the list. Despite the list only having one element it is actually faster to look only at the head rather than scan the list to find only the single element. This technique is used in the two aforementioned places and can be seen in the relevant code snippets 3.5 and 3.6.

3.6.2 Printing TPTP

To print a `clause` in TPTP format the literals must be combined into a string. This was originally implemented with a string builder using the List module's `fold_left` function that scans a list and accumulates a result along the way which in this case was the concatenation of the `literal`s in the `clause`. The disadvantage of folding operations is that they can be memory intensive and on some larger problems the memory usage exceeded reasonable limits. The implementation was changed to use a regular concatenation method which gave a significant performance increase alongside the reduced memory usage.

3.6.3 Dependency Schemes

At first only the trivial dependency scheme was implemented and it was calculated when needed during Skolemization. Refactoring was required to implement the standard dependency scheme and to make the dependency scheme computation general enough to be able to add more dependency schemes in future. This meant moving the dependency scheme into the current association list that is created before Skolemization takes place and is passed to the Skolemization procedure as an argument. The dependencies are also computed on the `qbf` data structure rather than the `fol_qbf` data structure because the `qbf`'s `literal`s are much more simple, just a record type with a `sign` and an `atom` which saves operations to extract data from the `predicate`s of the `fol_qbf`.

Chapter 4

Evaluation

This chapter discusses the evaluation of *qbftoepr* starting with the automated testing then benchmarks against other QBF solvers and against *qbf2epr*, a tool that also converts QBF to EPR.

4.1 Automated Testing

Automated testing was implemented using the OUnit library which is an OCaml analogue of JUnit for Java. Each function was tested individually against valid and invalid inputs to determine whether the code behaved correctly then linked together in integration tests to test the sections of the algorithm (such as Skolemization) from the top level. Each update to the code also updated the tests if required and ensured that the tests passed before the code was committed.

4.2 Testing Methodology

Two test sets were selected from the QBFLIB 2010 problem library. The first was an easy set consisting of four easier problems with fewer clauses and the second was a more difficult set of sixteen harder problems with orders of magnitude more clauses. The exact problems are listed in appendix A.

The easy set was used to test *qbftoepr* against the QBF solver DepQBF. The problems generated by the harder set were not practical to solve using iProver due to time constraints. This will become evident when comparing the iProver solving time of results generated using the trivial dependency scheme versus the standard in subsection 4.3.3. The easy set of benchmarks runs quickly enough to be able to benchmark efficiently but is complex enough to still provide a useful benchmark.

The hard set is used when evaluating the different dependency schemes and when comparing *qbftoepr* to *qbf2epr*. The solving time of iProver doesn't matter when comparing the two EPR converters because they produce the same result (ignoring differences in naming of variables and predicates) so a harder data set can be used to really challenge the converters.

The tests were run on a mid-2014 Macbook Pro with an Intel Core i7-4578U processor and 16GB of 1600MHz RAM running OS X El Capitan version 10.11.4. Each benchmark was ran three times to take an average.

4.3 Comparison Between Trivial and Standard Dependency Schemes

Because the implementation of the standard scheme is inefficient it is expected to take much longer to convert the QBF than the trivial scheme. The real benefits of the standard scheme are shown in the size of the outputted EPR result as a smaller output from the same input indicates the variable dependencies that were removed. This small result then follows into less time taken for iProver to solve the result. The exact results of this testing can be found in appendix B.

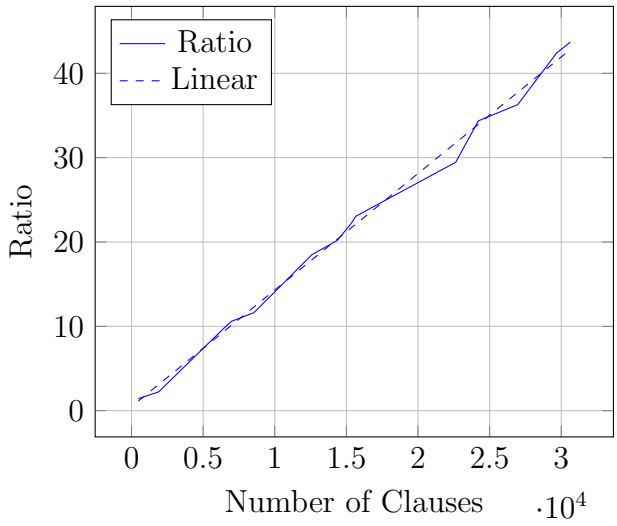
4.3.1 Time Taken to Convert

As expected using the standard dependency scheme means the conversion takes significantly more time on larger problem instances. The difference is much larger for larger problem sets. This is most clear by looking at the ratio of time taken to convert using the standard scheme over the time taken to convert using the trivial scheme as the number of clauses in the problem increases. In the smallest cases the trivial scheme might only be twice as fast but on the largest problems it can be forty times as fast. There is a linear relationship between this ratio and the number of clauses in the problem. Roughly doubling the number of clauses roughly doubles the ratio of trivial time over standard time as can be seen in the graph 4.1.

4.3.2 Size of Result

The standard scheme does produce significantly smaller results than the trivial scheme. The relationship is similar to the ratio discussed in the time taken to convert. We look at the ratio of the size of the trivial scheme result over the size of the standard scheme result. In the smallest cases the standard

Figure 4.1: Ratio of standard over trivial duration
Standard Duration over Trivial Duration (seconds)



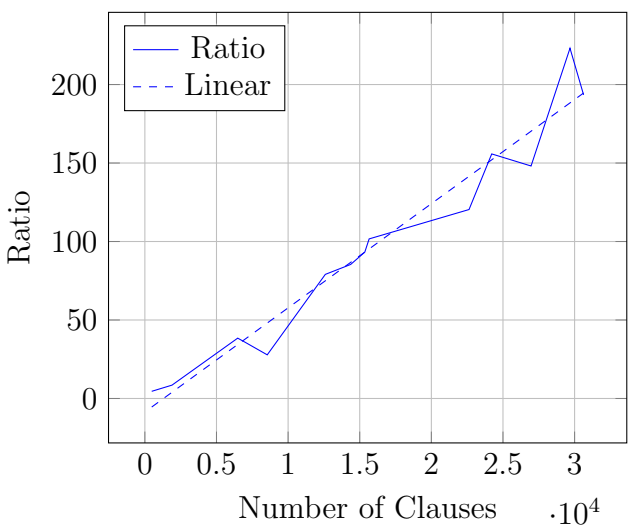
scheme is at least four to five times smaller but can be as much as two hundred times smaller for larger problems. Again there is a linear relationship between the number of clauses and this ratio. As the number of clauses doubles, the ratio of trivial size over standard size doubles too, shown in the graph 4.2.

Given that this implementation of the standard dependency scheme misses some dependencies the correct file-size is slightly higher but not enough to break this relationship.

4.3.3 Time to Solve

The hope is that the smaller problems are faster for iProver to solve and that this speed improvement outweighs the deficit introduced by the longer conversion time. This proved to be harder to show but can be loosely inferred from a single result. The problem `s27_d5_u-shuffled` is the smallest problem in the harder data set with 478 clauses. Using the trivial scheme took 0.015 seconds to convert to EPR with an output file-size of 117 Kilobytes and using the standard scheme took 0.021 seconds with an output of 26 Kilobytes. However, iProver took only 147.12 seconds to solve the standard result but had not solved the trivial result after an hour of processing. This shows that the extra time taken to convert using the standard dependency scheme is vastly outweighed by the time taken to solve in iProver on sufficiently large problems. To test this inference further the same test was carried out using

Figure 4.2: Ratio of trivial over standard file-size



the easy set of benchmarks but the difference here was negligible on the easy problems with only a hundred clauses.

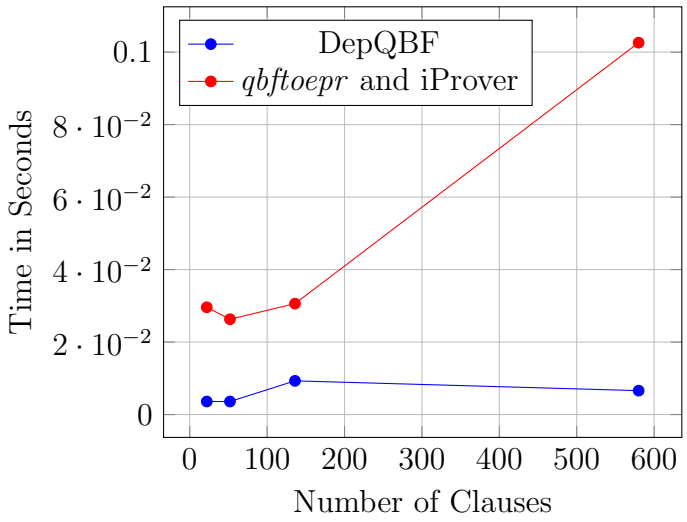
4.4 Comparison Against a Direct QBF Solver

As mentioned to in section 1.2.3, in theory solving EPR is slower than solving the QBF directly. The results agreed with this theory. Solving via converting with *qbftoepr* and solving the EPR with iProver was approximately an order of magnitude slower than solving the problem directly with DepQBF. This is shown in the graph 4.3 with the full results in appendix C.

4.5 Comparison Against an EPR Converter

The trivial dependency scheme is used throughout these tests as it is the only scheme implemented in qbf2epr so it is used when testing *qbftoepr* to give a better comparison. Since the two converters give identical output (aside from differences in variable names) only the conversion time is examined as the differences in time taken for iProver to solve the output would be negligible. The results show in the graph 4.4 that the conversion time of qbf2epr increases linearly with the number of clauses whereas *qbftoepr* increases with the square of the number of clauses. This means that as the problem size

Figure 4.3: DepQBF vs. *qbftoepr* on the easy benchmark



increases the algorithmic inefficiencies in *qbftoepr* lead to a significant time difference, up to 20 times slower in the largest problem. However on the smaller problem sets *qbftoepr* is almost an order of magnitude faster than qbf2epr (shown in graph 4.5) suggesting that if it were improved it could be significantly faster on larger problem sets.

Figure 4.4: *qbftoepr* vs. qbf2epr on the hard benchmark

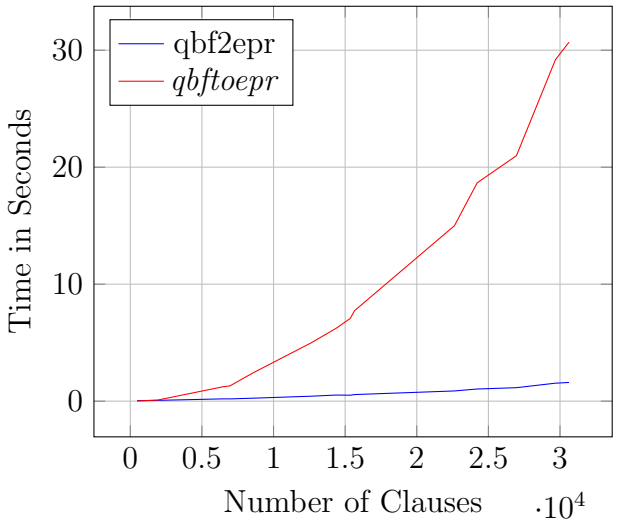
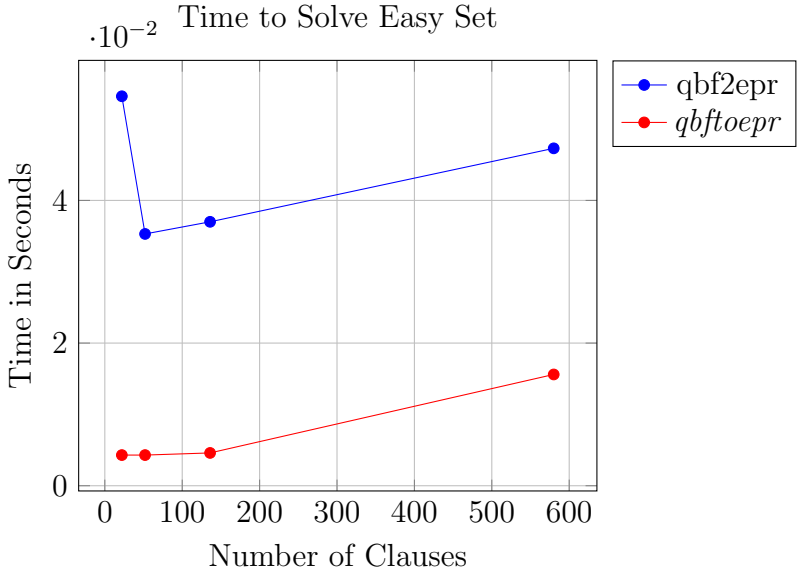


Figure 4.5: *qbftoepr* vs. qbf2epr on the easy benchmark



Chapter 5

Future Work

The core implementation of *qbftoepr* is complete in that it can take in a QDIMACS input and produce a TPTP output. Nonetheless there are still areas where optimizations can be made. This chapter will discuss three optimizations that were not implemented due to time constraints.

5.1 Dependency Scheme Optimizations

The construction of dependency schemes was discussed in section 3.4.5. The current implementation of the standard dependency scheme iterates over the matrix to find variables in the same clauses as a given existential variable but section 2.4.2 showed that the tractable algorithm for finding the standard dependency scheme uses a bipartite graph.

5.1.1 Tractable Standard Dependency Scheme

As seen in section 4.3 the standard dependency scheme produces TPTP outputs that are significantly smaller than the results from the trivial dependency scheme but the former takes much more time to execute. This is due to the iterative implementation of the standard dependency scheme (the complexity of which was discussed in section 3.5.4). Changing this implementation to build the incidence graph of the formula as in figure 2.1 then perform the linear search described in section 2.4.2 would fix the problem with the current implementation described in section 3.4.5 and would make the dependency scheme tractable and potentially return a significant saving in execution time.

5.1.2 Triangle Dependency Scheme

Samer and Szeider [10] detailed one more dependency scheme in their paper called the triangle dependency scheme that will only be summarised here. It is constructed in a similar way to the standard dependency scheme by creating a graph of literals and the clauses they are in. However it is not a bipartite graph this time because literals are also connected to their negation. The dependencies are paths between a variable x and y that avoid its negation $\neg x$ and paths between its negation $\neg x$ and y that avoid x while only using clauses with variables in the set $R(x)$ (as in 2.4.2) but not including y. This scheme is also tractable by finding paths in the graph between the two literals and further reduces the number of dependencies of a variable.

5.2 Anti-prenexing

Section 1.1.2 discussed prenexing where a series of rewrite rules are performed on a formula to bring the quantifiers to the front of the formula to form a prefix. This prenexing process is not deterministic though. A different choice of rewrite rules could lead to a different order of quantifiers in the prefix which could in turn lead to fewer dependencies if existential variables are moved further up the prefix. Anti-prenexing is the process of applying the rewrite rules in reverse to backtrack to a choice of rewrite rules to try another choice in the hope that it gives a better prenexing.

Section 3.3 discussed the `qbf` and `fol_qbf` data structures. It is hard to implement anti-prenexing on these data structures as the `clause` type only allows for `literal`s so introducing `quantified_variable`s via anti-prenexing (which would split the prefix and add quantifiers into the clauses) is not possible. These data structures were designed with the QDIMACS input format in mind which is already prenexed so anti-prenexing was not anticipated. This makes them very fit-to-purpose and they dispose of the generalities of the construction of QBFs that would allow quantified variables inside a sub-formula. A more tree like structure with sub-formulas as nodes would permit anti-prenexing.

An implementation of anti-prenexing alone is not enough though. Once the rewriting rules can be applied in reverse they must be applied in the right way so as to be advantageous whereas it is possible to arrive back at the same prenexing or potentially even a worse prenexing. Therefore heuristics are required to determine which rules to use in which order to aim for an advantageous prenexing.

Chapter 6

Conclusion

Finally, this chapter will summarize the project first looking at the progress then some closing analysis.

6.1 Progress

The project progressed slowly at first. The mix of multiple new technologies to learn at once delayed the start of the project until it was roughly six weeks behind. However from the implementation of Skolemization the project proceeded according to plan until the project demo. At this stage the project was extremely inefficient and not in a suitable state for providing evaluation results for the project demo. This meant that time had to be spent on improving the basic functionality rather than implementing some of the extensions that were originally planned beyond just the standard dependency scheme. The gantt chart in appendix E shows the planned time for each stage of the development.

6.2 Conclusion

The aim of the project was to produce a tool that would take a QBF as input and produce an EPR output to be used in iProver. This was completed in the estimated time frame with extensions to the original functionality to use the standard dependency scheme over just the trivial dependency scheme. Some extensions such as the triangle dependency scheme and anti-prenexing discussed in chapter 5 were not implemented due to time constraints. It does not perform as well as competing tools but with more time the implementation could be improved to the point where it would be faster than the competing tools.

Overall the project was a valuable learning experience from the first experiences with a functional language to researching beyond the undergraduate course to understand the background of the project.

Chapter 7

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*. 5

$\forall$ *universal*. 5

$\wedge$ *conjunction*. 4

$\vee$ *disjunction*. 4

$\neg$ *negation*. 4

$\rightarrow$ *implies*. 4

CNF conjunctive normal form. 5, 6, 16

DPLL Davis-Putnam-Logemann-Loveland. 1

EPR effectively propositional logic. 1, 4, 6, 7, 8, 9, 10, 16, 18, 19, 26, 27, 28, 29, 34

QBF quantified boolean formula. 1, 4, 5, 6, 7, 8, 9, 10, 15, 16, 17, 19, 26, 27, 29, 33, 34

SAT satisfiability. 4, 6, 7

Appendices

Appendix A

QBF Problem Sets

Append `-shuffled.qimacs` for the filename in the QBFLIB 2010 set.

A.1 Easy Set

Name	No. Variables	No. Clauses
toilet_g_02_01.2	11	22
toilet_g_04_01.2	22	52
toilet_g_08_01.2	43	136
toilet_g_20_01.2	105	580

A.2 Hard Set

Name	No. Variables	No. Clauses
s1196_d3_u	7758	14 367
s1269_d3_s	8353	15 354
s27_d5_u	403	478
s298_d11_s	22 003	15 656
s298_d2_s	853	1895
s298_d5_s	4744	6482
s298_d9_s	14 846	12 598
s386_d12_u	36 462	24 215
s499_d15_s	56 827	30 628
s499_d4_s	4368	6967
s510_d12_s	45 786	29 679
s510_d4_s	5506	8543
s820_d6_s	18 299	22 630
s820_d7_s	24 757	26 960

Appendix B

Trivial vs. Standard Dependency Scheme

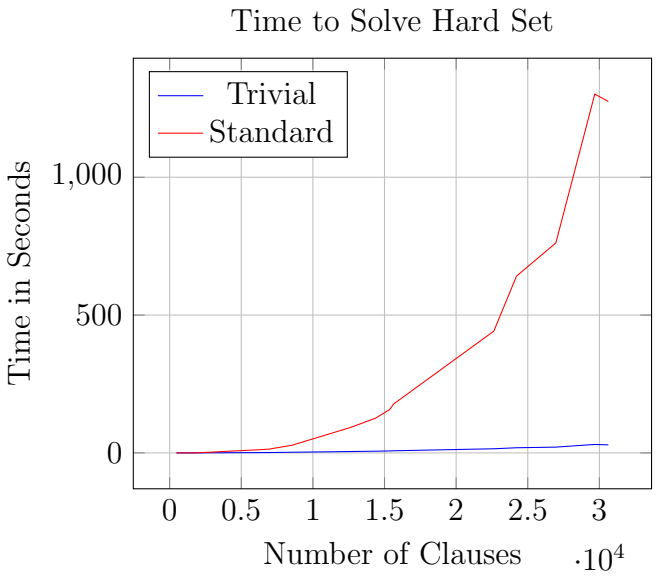
B.1 Trivial Results

Name	Average Time in Seconds	Result Filesize in Bytes
s1196_d3_u	6.2243	75 655 431
s1269_d3_s	7.0686	87 889 943
s27_d5_u	0.0150	120 032
s298_d11_s	7.7326	105 460 769
s298_d2_s	0.0873	867 681
s298_d5_s	1.2313	15 304 926
s298_d9_s	4.9640	64 671 307
s386_d12_u	18.6606	256 703 397
s499_d15_s	29.1823	407 056 522
s499_d4_s	1.3090	15 289 855
s510_d12_s	30.6883	447 528 018
s510_d4_s	2.4000	28 416 567
s820_d6_s	14.9966	180 183 099
s820_d7_s	20.9876	268 467 131

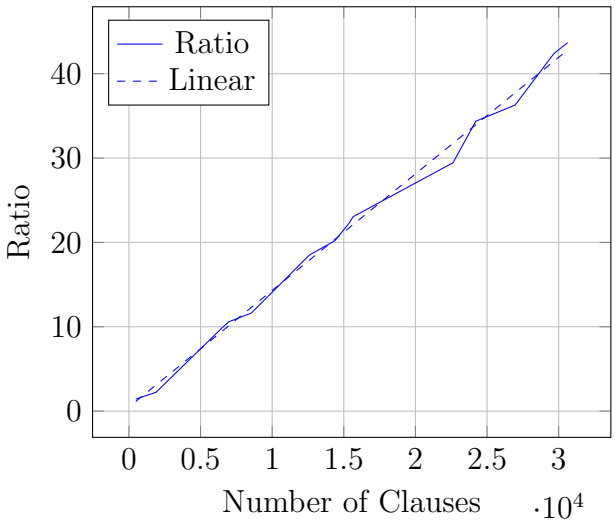
B.2 Standard Results

Name	Average Time in Seconds	Result Filesize in Bytes
s1196_d3_u	125.6636	886 023
s1269_d3_s	157.2360	942 497
s27_d5_u	0.0213	26 293
s298_d11_s	178.2380	1 037 383
s298_d2_s	0.1947	101 987
s298_d5_s	12.1103	398 125
s298_d9_s	91.8376	817 978
s386_d12_u	641.3433	1 647 641
s499_d15_s	1274.8680	2 104 483
s499_d4_s	13.8606	424 915
s510_d12_s	1301.6493	2 005 687
s510_d4_s	27.8870	519 597
s820_d6_s	441.6953	1 496 930
s820_d7_s	761.5803	1 812 656

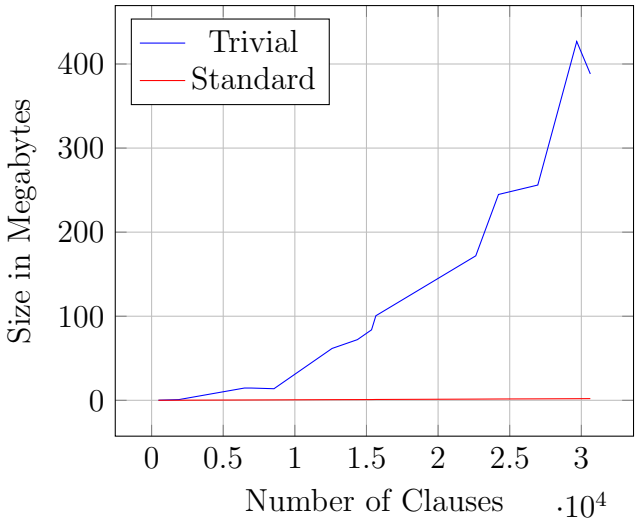
B.3 Graphs of Time and Size

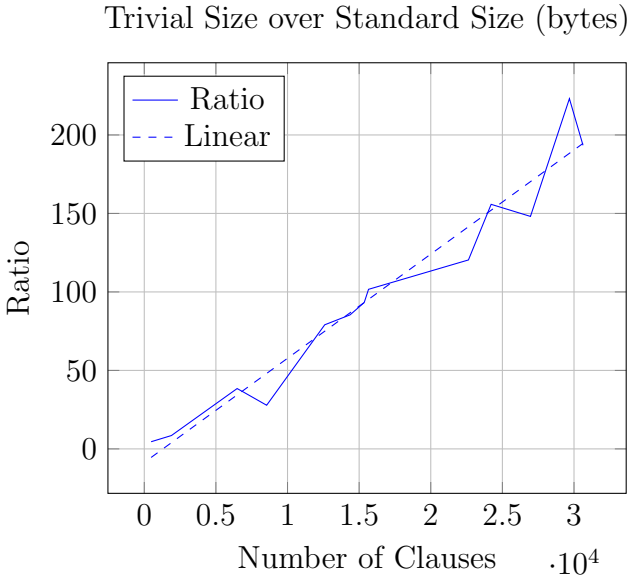


Standard Duration over Trivial Duration (seconds)



Output File-Size




Trivial Size over Standard Size (bytes)
Ratio
Linear
200
150
100
50
0
Ratio
0
0.5
1
1.5
2
2.5
3
Number of Clauses
·10^4

Appendix C

DepQBF vs. *qbftoepr* and iProver

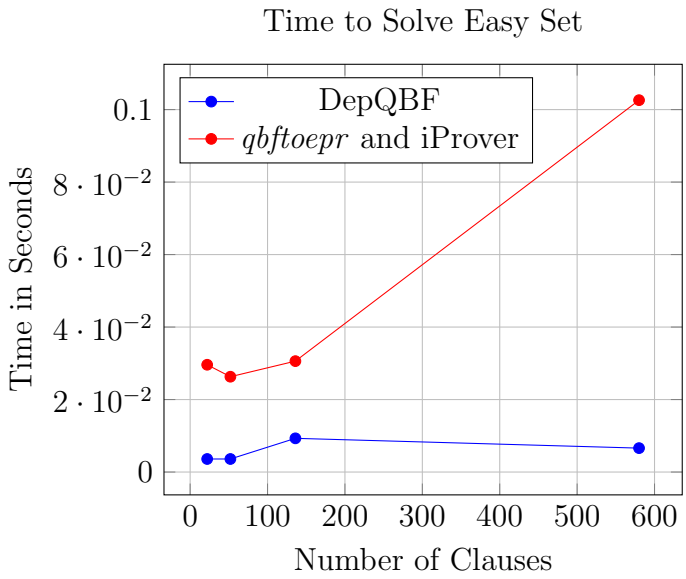
C.1 *qbftoepr* and iProver Results

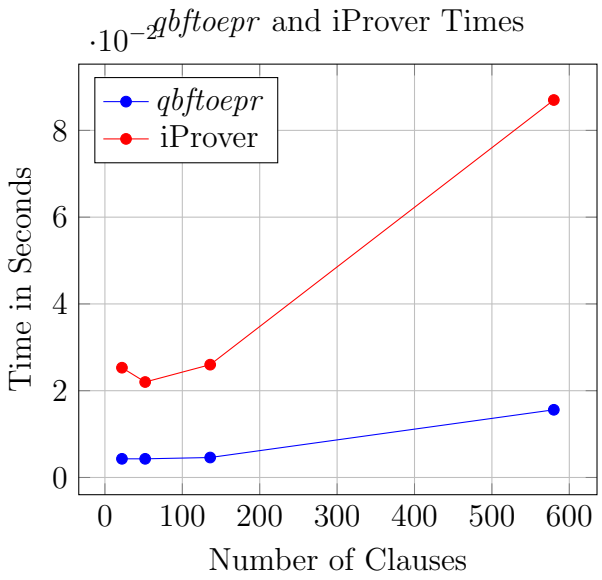
Name	Average Time in Seconds		
	qbftoepr	**iProver**	**Total**
`toilet_g_02_01.2`	0.0043	0.0253	0.0296
`toilet_g_04_01.2`	0.0043	0.0220	0.0263
`toilet_g_08_01.2`	0.0046	0.0260	0.0306
`toilet_g_20_01.2`	0.0156	0.0870	0.1026

C.2 DepQBF Results

Name	Average Time in Seconds
`toilet_g_02_01.2`	0.0036
`toilet_g_04_01.2`	0.0036
`toilet_g_08_01.2`	0.0093
`toilet_g_20_01.2`	0.0066

C.3 Graphs of Time





Appendix D

qbftoepr vs. qbf2epr

D.1 *qbfoepr* Results

Name	Average Time in Seconds
s1196_d3_u	6.2243
s1269_d3_s	7.0686
s27_d5_u	0.0150
s298_d11_s	7.7326
s298_d2_s	0.0873
s298_d5_s	1.2313
s298_d9_s	4.9640
s386_d12_u	18.6606
s499_d15_s	29.1823
s499_d4_s	1.3090
s510_d12_s	30.6883
s510_d4_s	2.4000
s820_d6_s	14.9966
s820_d7_s	20.9876

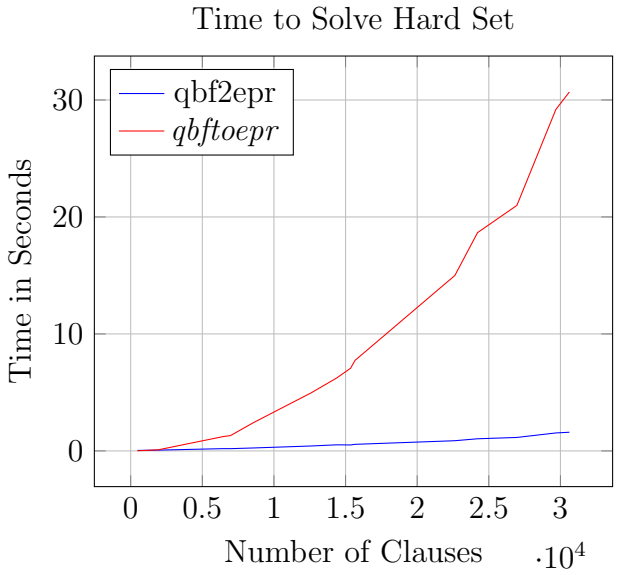
Name	Average Time in Seconds
toilet_g_02_01.2	0.0043
toilet_g_04_01.2	0.0043
toilet_g_08_01.2	0.0046
toilet_g_20_01.2	0.0156

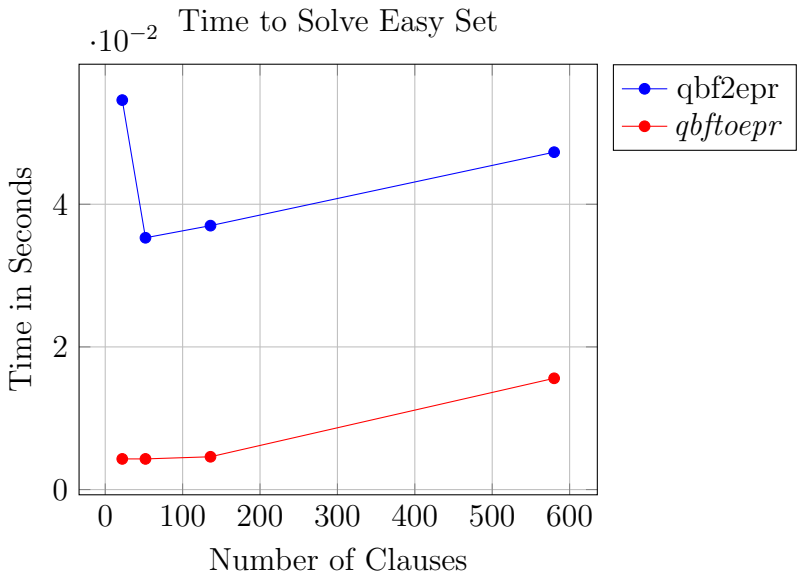
D.2 qbf2epr Results

Name	Average Time in Seconds
s1196_d3_u	0.5206
s1269_d3_s	0.5090
s27_d5_u	0.0410
s298_d11_s	0.5616
s298_d2_s	0.0676
s298_d5_s	0.1940
s298_d9_s	0.4213
s386_d12_u	1.0376
s499_d15_s	1.5356
s499_d4_s	0.1923
s510_d12_s	1.5966
s510_d4_s	0.2506
s820_d6_s	0.8716
s820_d7_s	1.1506

Name	Average Time in Seconds
toilet_g_02_01.2	0.0546
toilet_g_04_01.2	0.0353
toilet_g_08_01.2	0.0370
toilet_g_20_01.2	0.0473

D.3 Graphs of Time




Time to Solve Easy Set
$\cdot 10^{-2}$
Time in Seconds
0
2
4
0
100
200
300
400
500
600
Number of Clauses
qbf2epr
qbftoepr

Appendix E

Gantt Chart