Algorithms for Quantified Boolean Formulas

Ryan Williams*

Abstract

We present algorithms for solving quantified Boolean formulas (QBF, or sometimes QSAT) with worst case runtime asymptotically less than $O(2^n)$ when the clause-to-variable ratio is smaller or larger than some constant. We solve QBFs in conjunctive normal form (CNF) in $O(1.709^m)$ time and space, where m is the number of clauses. Extending the technique to a quantified version of constraint satisfaction problems (QCSP), we solve QCSP with domain size $d = 3$ in $O(1.953^m)$ time, and QCSPs with $d \geq 4$ in $O(d^{m/2+\epsilon})$ time and space for $\epsilon > 0$, where m is the number of constraints. For 3-CNF QBF, we describe an polynomial space algorithm with time complexity $O(1.619^n)$ when the number of 3-CNF clauses is equal to n; the bound approaches 2^n as the clause-to-variable ratio approaches 2. For 3-CNF Π_2-SAT (3-CNF QBFs of the form $\forall u_1 \cdots u_j \exists x_{j+1} \cdots x_n\, F$), an improved polyspace algorithm has runtime varying from $O(1.840^m)$ to $O(1.415^m)$, as a particular clause-to-variable ratio increases from 1.

1 Introduction

The recent past has seen the rise of an entire subfield of improved exponential time algorithms for NP-complete problems such as 3-coloring, satisfiability (SAT), and vertex cover (cf. [7], [18, 19, 5, 14, 10, 15], [4], respectively, to name a few). By "improved", we mean that the base of the exponent in the runtime is smaller than that of a brute-force search. For example, the deterministic local search method of [5], building upon the work of Schöning [19], solves 3-CNF SAT in $O(1.481^n)$ time, where n is the number of variables. This is a significant advance over $O(2^n)$; for example, instances with 60 variables can be solved in approximately 10^{10} steps, instead of 10^{18} (which may not be tractable). The study of improved algorithms is a practical way to work around the difficulty of solving NP-hard problems by solving small to medium-sized instances provably quickly.

Our work is an initial step in exploring improved algorithms for $PSPACE$-complete problems, such as quantified Boolean formulas (QBF). QBF is a generalization of the satisfiability problem. For this problem we inquire: given a prefix normal form first-order sentence in propositional logic, is the sentence true?

In this paper, we will give algorithms for determining the truth of sentences where the corresponding formula is in conjunctive normal form (CNF). The first algorithm solves arbitrary CNF QBF and executes in $O(1.709^m)$ time and space, where m is the number of clauses. This outperforms brute-force search when the clause-to-variable ratio is less than 1.294. Generalizing, we solve quantified constraint satisfaction problems efficiently as well. The other algorithms require only polynomial space, solving 3-CNF QBF and 3-CNF Π_2-SAT. Our interest in these special cases stems from the wide interest in 3-SAT algorithms, and the role of 3-CNF Π_2-SAT as a canonical problem studied in experimental QBF algorithms [16, 3, 8].

It has been proposed that an "easy-hard-easy" phase transition for 3-CNF Π_2-SAT occurs at a smaller clause-to-variable ratio than that for 3-SAT: either when $m/n \approx 1.4$, or $m/n \approx 2$, depending on the procedure used to select random 3-CNF formulas [8]. These small threshold values add significance to our bounds stated in terms of m.

2 Obstacles

All non-trivial algorithms we could find for $PSPACE$-complete subsets of QBF have only been verified experimentally in the literature [16, 3, 9]. Our research program was to study improved algorithms for SAT, and extract components of these algorithms that work for universally quantified variables. Several obstacles arose.

• **Lack of locality.** The technique of "local search" for a satisfying assignment, given a candidate assignment, has been very successful for finding improved SAT algorithms [18, 19, 5]. However, considering QBF as a two-player game, where competitors take turns setting variables of the formula, it appears difficult to model the opposing strategies of the players using a greedy local search method.

*Department of Computer Science, Cornell University, Ithaca, NY, 14850. Email: rrw9@cornell.edu. Supported by a NSF Graduate Research Fellowship.

• **Fixed ordering on variables.** Quantifying the variables of a formula forces some variables to be dependent on others. E.g. the value of an existential x_i that makes a formula true may depend directly on the value of a universal u_j quantified prior to x_i. In an algorithm solving this QBF, we would intuitively need to try values for u_j before we try those for x_i.

On the other hand, SAT algorithms such as those of Paturi, Pudlak, and Zane [13] (with Saks in [14]) work well because one can randomly choose which variable should be given a value next. In fact, most SAT algorithms we studied have improved running times because the "next variable" to try is chosen carefully. Our situation is somewhat alleviated by considering 3-CNF Π_2-SAT, because then we can choose among any of the universal variables.

• **Resolution versus Q-resolution.** Q-resolution [2] is a sound and complete proof system for QBF, and a simple extension of the well-known resolution proof system. Several improved algorithms for SAT, especially those with bounds in terms of the number of clauses [10, 11], depend crucially on resolution to ensure (among other things) that there are at least two positive and at least two negative occurrences of each variable in the formula. However, Q-resolution cannot provide such guarantees. Rintanen [17] demonstrates that the set of QBFs where each universal variable appears at most twice is still PSPACE-complete. Even with existential variables, Q-resolution does not enjoy the Davis-Putnam property [6]: replacing all clauses containing an existential variable x_i with all of the "q-resolvents" of these clauses can change the truth value of a QBF.

• **No autarkies.** The concept of autarkness has fueled much research in SAT algorithms, beginning with Monien and Speckenmeyer [12]. An assignment of some variables $V = \{v_{i_1}, \ldots, v_{i_k}\}$ of F is autark if every clause containing a v_{i_j} is satisfied by the assignment. Autarkness is nice because if an assignment of V is autark, then we may remove all of the clauses containing the v_{i_j}, yet preserve satisfiability. While this has been beneficial for SAT, it does not seem to help with QBF. We found generalizations of autarkness for use with QBF to be very messy.

What remains of the above? We employ two basic strategies in our work. One is to break the QBF into many small subformulas, and use dynamic programming to solve the subformulas efficiently. The second is to search over all possible satisfying assignments, but do so in conservative ways. The second is possible when considering k-CNF formulas.

3 Notation

We indicate universally (resp. existentially) quantified variables over the true/false symbols $\{\top, \bot\}$ by u_i (resp. x_i), for integers i. We require that for all $i = 1, \ldots, n$, either u_i or x_i is a variable in a formula, but not both. This permits us to assume a linear ordering on the variables in general. v_i refers to the ith variable (be it universal or existential). A literal of v_i (positive v_i or negative $\overline{v_i}$) is denoted by l_i. A literal is called existential (resp. universal) if it represents an existentially (resp. universally) quantified variable.

Let F be a Boolean formula in conjunctive normal form over the variables $v_1, \ldots, v_n$. We represent F as a family of subsets over $\{v_1, \overline{v_1}, \ldots, v_n, \overline{v_n}\}$, where v_i and $\overline{v_i}$ never appear in the same subset, for all i. As is standard, the sets of F are called clauses. The number of clauses in a formula is denoted by m, and the number of 3-CNF clauses is t. We designate the order of quantification by the index of the variable, i.e. the outermost quantified variable is either x_1 or u_1, and if $1 \leq i < j \leq n$, then v_i is quantified before v_j in the formula. This representation of QBFs is convenient because with it we can eliminate the quantification prefix that normally precedes a QBF.

4 Algorithm for CNF QBF

Our first algorithm has two stages. In the first, for $\delta > \frac{1}{2}$, a rule-based polyspace search over all possible assignments of the first δm variables occurs. The second stage is a dynamic programming method used on resulting formulas over the remaining $n - \delta m$ variables. The two phases can be performed independently of each other. To optimize the running time, we find a value for δ such that the two stages require approximately the same amount of time.

We start by describing the data structure of stage 2. Let F be a QBF over n variables, and $C \subseteq F$, where $v_1, \ldots, v_j$ are *monotone* in C; that is, the first j variables appear either only negatively in C, only positively, or they do not appear in any clause of C. The Stage 2 solves "subformulas" of F with the form $(C, j) \in \{\top, \bot\}$. (C, j) represents the value of the formula after the first j variables have been assigned values, and the clauses $C \subseteq F$ have not yet been satisfied as a result. Therefore, if (monotone) $v_1, \ldots, v_j$ appear in C, we can set these variables such that their literals are *false* (i.e. $\bot$), without loss of generality.

For simplicity, we will define the predicate

$$M(C, V) := (\forall v_i \in V)[v_i \text{ is monotone in } C],$$

where $V \subseteq \{v_1, \ldots, v_n\}$. We state the above rule for the sake of formality:

RULE 4.1. *(Subformula rule)*

For all (C, i), if a literal l_j appears in C, and $j < i$, then set $l_j := \bot$ in C.

Along with this rule, we provide additional simplification rules to be performed in stages 1 and 2. We apply them repeatedly, in the order that they are presented, until they are no longer applicable, or until (C, i) is set to a value. (They are being formulated in the notation of stage 2, but may also be used in stage 1 as well.)

RULE 4.2. *(Contradiction and truth rules)*

If some $c \in C$ is such that all $l_j \in c$ have $l_j := \bot$, then $(C, i) := \bot$. If $C = \varnothing$ then $(C, i) := \top$.

RULE 4.3. *(Monotone literal rule)*

For all (C, i), if $i \leq j$ and $M(C, x_j)$, then set $(C, i) := (C - \{c \in C : x_j \text{ or } \overline{x_j} \in c\}, i)$. If $M(C, u_j)$, then remove all occurrences of u_j from C while the rest of the rules are being applied.

I.e. if x_j occurs only positively (resp. negatively), then satisfy all clauses it occurs in by $x_j := \top$ (resp. $\bot$). If u_j occurs only positively (resp. negatively), then in the worst case $u_j := \bot$ (resp. $\top$). The monotone literal rule has been around since Davis-Putnam [6].

4.1 Algorithm

Let be a QBF formula F in CNF, and $\delta > \frac{1}{2}$ be a parameter to be fixed later.

Stage 1. Applying rules 4.1-4.3, perform the usual polyspace algorithm for QBF on the first $\lceil \delta m \rceil$ variables of F that are not automatically set by the rules. For each assignment of $\lceil \delta m \rceil$ variables, and resulting subformula C, find $(C, \delta m) \in \mathcal{D}$ from Stage 2 and return its value.

Stage 2. (*Construct initial subformulas for dynamic programming.*) For all $C \subseteq F$ with $|C| \leq m - (n-1)$ and $M(C, v_1, \ldots, v_{n-1})$, substitute into C the appropriate values for $v_1, \ldots, v_{n-1}$, and try $v_n := \bot$ in C, then try $v_n := \top$ in C. If v_n is universal (existential), and C is true for both trials (one of them), then $(C, n-1) := \top$. Otherwise, $(C, n-1) := \bot$.

(*Build table for the last $n - \lceil \delta m \rceil$ variables.*) Initialize $\mathcal{D}$ to contain all (C, n) pairs formed above; $i := n-1$.

Repeat:

$\mathcal{F} := \{C \subseteq F : (|C| \leq m - i) \wedge M(C, v_1, \ldots, v_i)\}$.

For all $C \in \mathcal{F}$,

Apply rules 4.1 – 4.3 to (C, i) if possible. If they do not yield a value for (C, i), then:

If v_i is existential (resp. universal), then by referencing $\mathcal{D}$, set $(C, i) :=$
$[(C - \{c \in C : v_i \in c\}, i+1) \vee$
(resp. $\wedge$)
$(C - \{c \in C : \overline{v_i} \in c\}, i+1)]$.

End for

$\mathcal{D} := \{(C, i) : C \in \mathcal{F}\}$; $i := i - 1$.

Until $i \leq \delta m$.

4.2 Analysis

The proof of correctness follows from the validity of the rules and the fact that for any subformula C resulting from stage 1, there is a corresponding pair $(C, \delta m) \in \mathcal{D}$ in stage 2 that contains the correct truth value for C. This is true since every subset of F that is a subformula after i variables have been set has at most $m - i$ clauses. (Each variable that is given a value removes at least one clause, because we do not set variables unless they appear both positively and negatively.) In the algorithm, every subformula C with size at most $m - i$ is considered as a pair (C, i).

Let h be the entropy function; that is, $h(\epsilon) = \epsilon \log \frac{1}{\epsilon} + (1-\epsilon) \log \frac{1}{1-\epsilon}$. The following describes an upper bound on the runtime of the algorithm.

THEOREM 4.1. *For $\frac{1}{2} < \delta < n/m$, the above algorithm runs in $O(2^{\delta m} + 2^{h(\delta)m})$ time and $O(2^{h(\delta)m} + \delta m)$ space within a polynomial factor.*

Proof. In the worst case, the first stage takes $2^{\delta m}$ steps and δm space.

For a fixed i, the second stage considers at most $\sum_{j=i}^{m} \binom{m}{m-j}$ different subformulas of type (C, i). Each (C, i) can be evaluated in polynomial time, using the rules and previously stored subformulas. Thus the total time of the second stage is bounded from above by a polynomial factor times

$$\sum_{i=\delta m}^{n-1} \sum_{j=i}^{m} \binom{m}{m-j} \quad \leq \quad \sum_{i=\delta m}^{n-1} \left[(m-i) \binom{m}{m-i} \right]$$

$$\leq \quad (n-\delta m)(m-\delta m)\binom{m}{m-\delta m},$$

when $n > \delta m > \frac{1}{2}$. The space complexity of the second stage is upper bounded by $|F|\cdot(n-\delta m)(m-\delta m)\binom{m}{m-\delta m}$, since at any given time we only save the subformulas (C, i) for the current i.

A well known result from coding theory says that when $\delta > \frac{1}{2}$, $2^{h(1-\delta)m} \geq p(n) \cdot \binom{m}{(1-\delta)m}$, where $p(n)$ is a polynomial. From this and the fact that $h(1-\delta) = h(\delta)$, the result follows. □

The time is minimized when $\delta = h(\delta)$, or $\delta \approx .7729$. Then, the time is $O(1.709^m)$ and it is an improvement when $m/n < 1.294$.

Since $h(\delta) > \delta$ for $\delta \leq .7729$, the above bound is optimal (for both time and space) when $\delta \geq .7729$. In this case, a tradeoff between time and space usage occurs. For example, when $\delta = .85$, the algorithm runs in $O(1.906^m)$ time (the time for stage 1) and $O(1.289^m)$ space.

While we are achieving a better runtime, the exponential space bound can be somewhat costly. Later, we will discuss polynomial space algorithms for 3-CNF quantified Boolean formulas.

4.3 Generalization

The dynamic programming technique does not rely heavily on specific properties of CNF Boolean formulas. It not surprising, then, that it can be used with quantified constraint satisfaction problems (QCSPs) in which the variables are quantified over a domain D of size d.

Our constraint notation follows that of [7, 19]. A constraint is a set of pairs of the form *(variable, value)*, and a CSP is a collection of constraints. A constraint $c = \{(v_{i_1}, d_1), \ldots, (v_{i_k}, d_k)\}$ is satisfied by an assignment a (a function from variables to domain D) if there is a pair $(v_{i_j}, d_j) \in c$ such that $a(v_{i_j}) \neq d_j$. A QCSP of domain d and constraint size k is defined as a first-order sentence in prefix normal form, where the predicate of the sentence is a CSP of domain d with k variables per constraint. QCSPs may be thought of as two-player versions of the usual CSPs, where one player tries to satisfy the constraints, and an adversary tries to foil his attempt by setting "bad" values to variables.

Stage 1 of our revised algorithm consists of $d^{\delta m}$ steps of trying possible solutions for the first δm variables that are not trivially set by the rules, and removing constraints that are satisfied by these variable settings.

Domain size	δ value	Time bound
$d = 2$	.7729	1.709^m
$d = 3$	.6091	1.953^m
$d \geq 4$	$.5 + \epsilon$	$d^{m/2+\epsilon m}$

Table 1: Worst case upper bounds for quantified CSPs in terms of number of constraints.

For Stage 2, we store subproblems here exactly how we stored subformulas in the original algorithm. It is not hard to see that for the subproblem (C, j), if some constraint $c \in C$ contains the pair (v_i, d_k), and $i < j$, then it must be that $a(v_i) = d_k$ in the (partial) assignment a that led to C. This is an analogue of the subformula rule, and it means that by specifying (C, j), we may quickly determine the values of any variables v_i (with $i < j$) that appear in C.

Notice that the previous running time bound of Stage 2 works here. When δm variables have been set to values, it is still true that at least δm constraints have been already been satisfied, so we still only need to choose subsets of size at most $m - \delta m$.

Thus the running time is $O(d^{\delta m} + 2^{h(\delta)m})$, within a polynomial factor. The following table gives times for several values of d, in comparison to brute-force search.

For $d \geq 4$, our algorithm does not work optimally; this is because the optimal value for δ in this case is less than or equal to $1/2$, and our results only hold for $\delta < 1/2$. Thus, for $d \geq 4$, the bound is $O(d^{(.5+\epsilon)m})$, for $\epsilon > 0$.

5 Algorithm for 3-CNF QBF

By restricting ourselves to 3-CNF formulas, we find an improved time bound for QBF that requires only polynomial space. It is well known that 2-CNF QBF is polynomial time solvable [1]. Our idea is to branch in a such way that either two 3-CNF clauses are "reduced" to 2-CNF (or removed entirely), or more than just one variable is removed in one of the branches.

A crucial observation we use has its origins with Monien and Speckenmeyer [12], and is essentially a reformulation of unit clause elimination. That is, given the QBF $F \wedge \{l_i, l_j\}$, with v_i as the outermost quantified variable, if v_i is existential/universal then $F \wedge \{l_i, l_j\}$ is true if and only if $F[l_i := \top]$ is true, or/and $F[l_i := \bot, l_j := \top]$ is true.

Additionally, we use another rule in this algorithm:

RULE 5.1. *(Unit clause rule)*

For $\{l_j\} \in F$, if l_i is universal, then $F := \bot$. If l_i is existential, then set $F := F - \{c \in C : l_a \in c\}$.

Like the monotone literal rule, unit clause was introduced with Davis-Putnam [6]. If a literal is universal and is the only literal of a clause, then any F containing this clause is false. If the literal is existential, it must be that l_i is true, if F is to be a true formula.

5.1 Algorithm

Given a QBF F,

Apply monotone literal rule (4.3) and unit clause (5.1). If some clause is false, return $\bot$. If all clauses are true, return $\top$.

Base cases: If there are no 3-CNF clauses in F, then solve the 2-CNF QBF and return its value. If $n = 1$, then try both values for the variable and return the value of F.

(A) If there are clauses of the form $\{l_i, l_j\}$, $\{\overline{l_i}, \overline{l_j}\}$, and $i < j$, then replace $\overline{v_j}$ (and v_j) with v_i ($\overline{v_i}$) in F.

(B) If there are clauses of the form $\{l_i, l_j\}$, $\{\overline{l_i}, l_j\}$, and $i < j$, then if v_j is universal, return $\bot$. Otherwise, set $l_j := \top$.

(C) If clauses of form $\{l_i, l_j\}$, $\{\overline{l_i}, l_k\}$, $\{\overline{l_j}, \overline{l_k}\}$, then replace $\overline{v_j}$ (v_j) and v_k ($\overline{v_k}$) with v_i ($\overline{v_i}$).

Let v_i be the variable with the smallest index in F. There are five cases:

(1) If there are clauses of the form $\{l_i, l_{j_1}, l_{k_1}\}$ and $\{\overline{l_i}, l_{j_2}, l_{k_2}\}$, then recursively call the algorithm on both:

F with $l_i := \bot$, and

F with $l_i := \top$.

(2.1) If clauses of the form $\{l_i, l_{j_1}, l_k\}$, $\{\overline{l_i}, l_{j_2}\}$, and $\{\overline{l_{j_2}}, l_a\}$, call the algorithm on:

F with $l_i := \bot$, and

F with $l_i := l_{j_2} := l_a := \top$.

(2.2) If clauses of the form $\{l_i, l_{j_1}, l_k\}$, $\{\overline{l_i}, l_{j_2}\}$, and $\{\overline{l_{j_2}}, l_a, l_b\}$, call the algorithm on:

F with $l_i := \bot$, and

F with $l_i := l_{j_2} := \top$.

(3.1) If clauses of form $\{l_i, l_j\}$, $\{\overline{l_i}, l_k\}$, $\{\overline{l_j}, l_a\}$, $\{\overline{l_k}, l_b\}$, then call the algorithm on:

F with $l_i := \bot$, $l_j := l_a := \top$ and

F with $l_i := \top$, $l_k := l_b := \top$.

(3.2) If clauses of form $\{l_i, l_j\}$, $\{\overline{l_i}, l_k\}$, $\{\overline{l_j}, \overline{l_k}, l_a\}$ (or $\overline{l_j}$,$\overline{l_k}$ appear in separate 3-CNF clauses), then call the algorithm on:

F with $l_i := \bot$, $l_j := \top$, and

F with $l_i := \top$, $l_k := \top$.

For each of these cases, if v_i is universal (resp. existential), then return $\top$ iff both (resp. one of the) calls return $\top$.

5.2 Analysis

The proof of correctness is straightforward; cases 3.1 and 3.2 are sufficient since (C) takes care of the case where $\{\overline{l_j}, \overline{l_k}\}$ is in F. Note that in cases 2.1 and 3.1, it is true that three distinct variables are set: if $i = a$ or $i = b$, then these cases would already been eliminated by either rules (A), (B), or (C), a contradiction. It is obvious that the above algorithm requires only polynomial space.

Let $T(n, t)$ be the worst case running time of the algorithm for F when the number of 3-CNF clauses is t, and the number of variables is n. When case (1) is taken, the recurrence is bounded by $2T(n-1, t-2)$; one 3-CNF clause is reduced to 2-CNF, the other is removed entirely. For case (2.1), the time is less than $T(n-1, t-1) + T(n-3, t-1)$. In case (2.2), it is $T(n-1, t-1) + T(n-2, t-2)$, and for cases (3.1) and (3.2), it is $2T(n-3, t)$ and $2T(n-2, t-1)$, respectively.

Then we have the following double recurrence representing an upper bound on T:

$T(n, 0) = O(n^2)$.
$T(1, t) = O(1)$.

$$T(n,t) = \max \left\{ \begin{array}{l} 2T(n-1,t-2), \\ T(n-1,t-1)+ \\ \quad T(n-3,t-1), \\ T(n-1,t-1)+ \\ \quad T(n-2,t-2), \\ 2T(n-3,t), \\ 2T(n-2,t-1) \end{array} \right\} + p(n),$$

where $p(n)$ is a polynomial representing the runtime of rule applications, (A)-(C), and case identification.

We solved for values of the recurrence assuming $t = cn$, for some interesting values of c; the results are shown in Table 2. Notice that as the ratio of 3-CNF clauses to variables approaches 2, the running time approaches 2^n. Due to space considerations, we only prove the $t = n$ case. To disregard distracting polynomial factors in the analysis, we work with a simplified recurrence T'. Note that here we set $T'(n, 1)$ as running in polynomial time,

which is true in the algorithm (if a formula has one 3-CNF clause, cases (1)-(3.2) are only considered once in the execution). This observation simplifies the proof.

$$T'(n,i) = 1 \text{ for } i \leq 1,\ T'(j,t) = 1 \text{ for } j \leq 1.$$
$$T'(n,t) = \max \left\{ \begin{array}{l} 2T'(n-1,t-2), \\ T'(n-1,t-1) + T'(n-3,t-1), \\ T'(n-1,t-1) + T'(n-2,t-2), \\ 2T'(n-3,t), \\ 2T'(n-2,t-1) \end{array} \right\}.$$

THEOREM 5.1. $T(n,n) = O(1.619^n)$.

Proof. It will help us to prove the facts

(1) If $n < t$, $T'(n-1,t) \leq T'(n,t-1)$, and
(2) For $n \geq 5$,

$$T'(n,n-1) = T'(n-1,n-2) + T'(n-2,n-3),$$

simultaneously with
(3) For $n \geq 3$,

$$T'(n,n) = T'(n-1,n-1) + T'(n-2,n-2),$$

the Fibonacci reccurrence; this will imply the theorem since $T'(n,n) = p(n) \cdot T(n,n)$ for some sufficiently large polynomial $p(n)$.

First, fact (1) follows by induction on $n+t$. For the first four values of $n+t$, we observe that

$T(1,2) = 1 \leq 1 = T(2,1)$,
$T(1,3) = 1 \leq 2 = T(2,2)$,
$T(1,4) = 1 \leq 2 = T(2,3)$.
$T(1,5) = 1 \leq 2 = T(2,4) \leq 3 = T(3,3)$.

Assume that (1) holds for n', t' with $6 \leq n' + t' < n + t$. Then by induction:

- $2T'(n-2,t-2) \leq 2T'(n-1,t-3)$,
- $T'(n-2,t-1) + T'(n-4,t-1) \leq T'(n-1,t-2) + T'(n-3,t-2)$,
- $T'(n-2,t-1) + T'(n-3,t-2) \leq T'(n-1,t-2) + T'(n-2,t-3)$,
- $2T'(n-4,t) \leq 2T'(n-3,t-1)$, and
- $2T'(n-3,t-1) \leq 2T'(n-2,t-2)$, which imply all together that $T'(n-1,t) \leq T'(n,t-1)$, by definition of T'.

Applying fact (1) to the terms in $T'(n,n)$, $T'(n-3,n-1) \leq T'(n-2,n-2)$ and $T'(n-3,n) \leq T'(n-2,n-1) \leq T'(n-1,n-2)$, so we find that

$$T'(n,n) = \max \left\{ \begin{array}{l} 2T'(n-1,n-2), \\ T'(n-1,n-1) + T'(n-2,n-2) \end{array} \right\}.$$

t/n ratio	Time bound
$c = 1$	$3 \cdot 1.619^n = O(1.619^n)$
$c = 1.294$	$2 \cdot 1.524^t = O(1.725^n)$
$c = 1.4$	$2 \cdot 1.4983^t = O(1.762^n)$
$c = 1.8$	$3 \cdot 1.436^t = O(1.919^n)$
$c = 2$	$2 \cdot 1.415^t = O(2^n)$

Table 2: Calculated bounds for the 3-CNF QBF algorithm.

Now we simultaneously prove (2) and (3) by induction. For $n = 5$, $T'(5,4) = T'(4,3) + T'(3,2)$, and for $n = 3$, $T'(2,2) + T'(1,1) = T(3,3)$.

For the inductive step, suppose facts (2) and (3) for any $k < n$. By induction, $T'(n-1,n-2) = T'(n-2,n-3) + T'(n-3,n-4)$ by (2), so

$$T'(n,n) = \max \left\{ \begin{array}{l} 2T'(n-2,n-3) + 2T'(n-3,n-4), \\ T'(n-1,n-1) + T'(n-2,n-2) \end{array} \right\}.$$

By definition of T', $2T'(n-2,n-3) \leq T'(n-1,n-1)$ and $2T'(n-3,n-4) \leq T'(n-2,n-2)$, therefore $T'(n,n) \leq T'(n-1,n-1) + T'(n-2,n-2)$, and (3) holds.

To prove (2), $T(n-1,n) =$

$$\max \left\{ \begin{array}{l} 2T'(n-2,n-2), \\ T'(n-1,n-2) + T'(n-2,n-3) \end{array} \right\},$$

and by induction on fact (3), $2T'(n-2,n-2) = 2T'(n-3,n-3) + 2T'(n-4,n-4)$.

Again, by definition of T', $T'(n-1,n-2) \geq 2T'(n-3,n-3)$ and $T'(n-2,n-3) \geq 2T'(n-4,n-4)$, so we conclude $T(n-1,n) \leq T'(n-1,n-2) + T'(n-2,n-3)$. □

6 Solving 3-CNF Π_2-SAT

By considering more restrictive quantifications of variables, we obtain a better bound. The formulas in 3-CNF Π_2-SAT consist of 3-CNF formulas F of the form: $\forall u_1 \cdots \forall u_j \exists x_{j+1} \cdots \exists x_n\ F$. More formally, for some j, if $i \leq j$ then u_i is a variable in F, and if $i > j$ then x_i is a variable.

We will say that a clause is *i-universal* if it contains exactly i universal literals. In solving 3-CNF Π_2-SAT, we may choose the order of what kind of clauses we remove: 0, 1, or 2-universal. Since the quantification consists of one set of universal variables followed by existential variables, the strategy is clear: we efficiently eliminate the clauses containing universal variables, then solve the remaining 3-SAT instance.

This algorithm uses another new rule:

RULE 6.1. *(Trivial falsity rule)*

If there is a clause $c \in F$ containing only universal variables, then set $F := \bot$.

I.e. if a clause has no existential literals, this clause cannot possibly be satisfied in every case, so the subformula is false. This rule appears in Cadoli, et. al. [3] and in Büning, et. al. [2].

6.1 Algorithm

Given a 3-CNF Π_2-SAT instance F,

(0) If some clause is false, return $\bot$. If all clauses are true, return $\top$. Aply trivial falsity (rule 6.1), unit clause elimination (rule 5.1), and monotone literal (rule 4.3).

(1) For each 2-universal 3-CNF clause $\{l_i, l_j, l_k\}$ with l_i and l_j as universal, recursively call the algorithm three separate times, with the following values substituted in F for each call:

$l_i := \top$, and

$l_i := \bot$, $l_j := \top$, and

$l_i := l_j := \bot$, $l_k := \top$.

Return true iff all three calls return true.

(2) [No 2-universal 3-CNF clauses for the rest of the computation.] For each 1-universal 2-CNF clause $\{l_i, l_j\}$ with l_i as universal, call the algorithm twice separately, with the following values substituted in F:

$l_i := \top$, and

$l_i := \bot$, $l_j := \top$.

Return true iff both calls return true.

(3) [No 2-universal clauses, and all 1-universal clauses are 3-CNF.] For each 1-universal clause $\{l_i, l_j, l_k\}$ with l_i as universal, try both possible values for l_i in the algorithm, and return true iff both values lead to true.

(4) Using Eppstein's $O(1.365^t)$ 3-SAT algorithm [7] and return true iff the remaining 3-SAT instance is satisfiable.

6.2 Analysis

The correctness of the above algorithm is easy to see, and so is the proof that it only uses polynomial space.

We use the concept of *branching tuples* [11] (also called *work factor*) [7] to aid analysis. Let $(t_1, \ldots, t_k)$ be a tuple of integers, and define $\lambda(t_1, \ldots, t_k)$ as the smallest positive root to the equation $1 - \sum_{i=1}^{k} \frac{1}{x^{t_i}} = 0$.

Suppose an algorithm has k separate recursive calls or "branches" within it (a la our algorithm above), and for the ith recursive call ($i = 1, \ldots, k$), a measure μ of the input (e.g. $\mu = m$ or $\mu = n$) is reduced to $\mu - t_i$. Assume these reductions take polynomial time each. From the work of Kullmann and Lockhardt [11], we find that the running time of this algorithm is at most

$$T(\mu) \leq \sum_{i=1}^{k} [T(\mu - t_i) + p(n)] \leq q(n)\lambda(t_1, \ldots, t_k)^{\mu},$$

where $p(n)$, $q(n)$ are polynomials. This bound is $O((\lambda(t_1, \ldots, t_k) + \epsilon)^{\mu})$, for any $\epsilon > 0$. For example, an algorithm consisting of only case (1) above reduces n to $n-1$ for its first recursive call, $n-2$ for its second, and $n-3$ for its third. Thus, this algorithm runs in time bounded by $(\lambda(1, 2, 3) + \epsilon)^n$.

Using branching tuples, we obtain the following result:

THEOREM 6.1. *Suppose $\overline{m_2}/\overline{n_2} \approx r$, where $\overline{m_2}$ is the number of 2-universal 3-CNF clauses and $\overline{n_2}$ is the number of variables in these clauses. Then the above algorithm runs in time $O(1.415^{m+(1.757/r-1)\overline{m_2}})$.*

Proof. First, we remark that the cases of the algorithm can be considered *independently*, in a sense. Once all 2-universal 3-CNF clauses have been considered and eliminated, case (1) no longer holds for the remainder of the algorithm's execution. The same holds for case (2) with respect to (3) and (4), and case (3) with respect to (4). Based on these observations, let n_i represent the number of variables of F *considered* during case i of the algorithm, and m_i be the number of 3-CNF clauses eliminated (by being satisfied, or by becoming 2-CNF) in case i (so for example, $m_1 = \overline{m_2}$, $m_2 = 0$). By considered, we mean that the variable appears in a clause where the algorithm sets values to some or all of the variables, depending on the branch.

Due to this case independence, multiplying the time for (1) given above (in terms of n_1 instead of n) with the time for (2), (3) and (4) gives a bound on the total time. Independently of the others, case (2) requires time $(\lambda(1, 2) + \epsilon)^{n_2} = O(1.619^{n_2})$.

Let m' be the number of 3-CNF clauses that have been satisfied in a branch after cases (1) and (2) are

m_2/n_2	Time bound
$r \geq 1.757$	1.415^m
$r \geq 1.4$	1.546^m
$r \geq 1.294$	1.603^m
$r \geq 1$	1.840^m

Table 3: Bounds for 3-CNF Π_2-SAT algorithm.

done, and n' be the number of variables considered in these clauses. Clearly $n' = n_1 + n_2$, but also we claim that $m' \geq m_1 = \overline{m_2}$: if some 2-universal 3-CNF clause c is not satisfied after the execution of case (1), then a literal in c was falsified before our algorithm considered c. Either its existential literal was falsified (and trivial falsity would imply that F is false), or two universal literals were falsified (and unit clause elimination would satisfy the clause), or one universal literal was falsified. In that case, c becomes 1-universal 2-CNF, so it is then satisfied by case (2).

Therefore, when considered independently of case (1) and (2), case (3) takes at most $(\lambda(2,2)+\epsilon)^{m-m'} \leq (\lambda(2,2)+\epsilon)^{m-\overline{m_2}}$ time; with either recursive step of case (3), two 3-CNF clauses are eliminated. Consequently, case (4) requires $O(1.365^{m-\overline{m_2}-m_3})$ time, due to Eppstein's 3-SAT algorithm based on the number of 3-CNF clauses [7].

In the worst situation, case (4) is never considered, and all of the clauses are removed by cases (1), (2), and (3), because $1.365 < 1.415 \approx \lambda(2,2) + \epsilon$, for sufficiently small $\epsilon > 0$. Initially, every clause of F is 3-CNF, so each 1-universal 2-CNF clause $\{l_j, l_k\}$ considered by case (2) was originally some 2-universal 3-CNF clause $\{l_i, l_j, l_k\}$ reduced by case (1). This implies that the number of variables in 2-universal 3-CNF clauses is at least the number considered in case (1) and (2), i.e. $\overline{n_2} \geq n_1 + n_2$, for all branches of the algorithm.

Then, the running time is bounded by $1.840^{n_2} \cdot 1.619^{n_1} \cdot 1.415^{m-\overline{m_2}} = O(1.840^{\overline{n_2}} \cdot 1.415^{m-\overline{m_2}})$. When $\overline{m_2}/\overline{n_2} \approx r$, then the time is at most $1.415^{\log(1.840)/\log(1.415)\overline{m_2}/r+m-\overline{m_2}}$, or more compactly, $O(1.415^{(1.757/r-1)\overline{m_2}+m})$. □

As the clause-to-variable ratio for the 2-universal clauses increases, the running time decreases. Note that this is an opposite effect from the 3-CNF QBF algorithm. Table 3 quantitatively demonstrates the tradeoff between clause-to-variable ratio and running time.

6.3 Relation with the QBF phase transition

We now consider our results in relation to the fraction of QBFs that are conjectured to be the hardest, as claimed by Gent and Walsh [8]. They give two models for choosing random QBFs. *Model A* consists of choosing a 3-CNF Π_2-SAT instance at random, then discarding all 3-universal and 2-universal clauses. With this model, Gent and Walsh observe a phase transition of the "easy-hard-easy" type at $m/n \approx 2$. That is, random instances generated by this model are true with high probability when either $m/n < 2$ and false with high probability when $m/n > 2$.

The second model, *Model B*, constructs random 3-CNF Π_2-SAT instances with m clauses and n variables by choosing two existential literals at random, then a universal variable, for each clause. The resulting formulas are of the same type as *Model A*, but the random choice produces an easy-hard-easy phase transition at a different point, $m/n \approx 1.4$.

Our algorithm for 3-CNF Π_2-SAT works well in the absence of 2-universal clauses: if the only 3-CNF clauses are 0 and 1-universal (so only cases (3) and (4) are considered), the running time is $O(2^{m_1/2} \cdot 1.365^{m-m_1/2})$, where m_1 is the number of 1-universal clauses. In the worst case, $m = m_1$ and the bound is 1.415^m. This implies that the algorithm runs in time better than 2^n when $m/n < 2$, for all formulas that may be chosen out of *Model A* or *Model B*. Hence the "easily true" instances from both models can be solved in time less than 2^n, using our Π_2-SAT algorithm. Notably, for $m/n \approx 1.4$ (the phase transition point for *Model A*), the algorithm runs in time $O(1.625^n)$.

7 Conclusions

We have seen various ways in which QBF algorithms can be devised: using dynamic programming, transformation rules, and pieces of existing SAT algorithms. The algorithms presented are simple yet effective, as to provide a solid starting point in research towards improved algorithms for QBF. Here are a few suggestions for further study.

- Recall that the proof for the time bound on the arbitrary CNF QBF algorithm only uses the first three transformation rules given (subformula, contradiction/truth, and monotone literal). This bound can probably be significantly improved by using more of these rules in our reasoning.

- It is interesting that in practice, 2-universal clauses in 3-CNF Π_2-SAT were not considered in evaluating "hard" formulas [8], yet the time bound for our algorithm is hampered by their presence. Perhaps an

interleaving of this algorithm with one closer to the experimental algorithms (which do well with 2-universal clauses) would give a solid $O(1.414^m)$ bound, or better.

• Our algorithms set the values for variables according to their order, except when a variable is forced to be a value (due to the rules). However, it is possible that some QBF solvers need not preserve the order of quantification when they set variables. For example, consider $\forall y_1 \cdots y_k \exists z\ F$. We could set $z := \top$, keeping a record of those y_i-values with which $z := \top$ works. Then when $z := \bot$ is considered, we ensure that if some value of y does not work in this case, then it worked when $z := \top$. Rintanen [16] describes a similar procedure he calls quantifier inversion. However, his randomized strategy for changing the order of the variables appears difficult to analyze rigorously.

• It seems plausible that a variation on local search could be implemented for Π_2-SAT. The search could have two phases: one in which a search for a "bad" assignment of universal variables ensues, and one in which a satisfying assignment of the existential variables is found, given the candidate universal assignment. One heuristic for local change towards a "bad" assignment is: if there exists a clause c such that exactly one of its literals l_i is true and l_i is universal, flip the variable value of u_i. (Such a clause and literal exist, otherwise the formula is true if it is satisfied by the current existential assignment.)

8 Acknowledgements

I am indebted to Dieter van Melkebeek and Eva Tardos for their generous comments and suggestions on drafts of this paper.

References

[1] B. Apswall, M. F. Plass and R. E. Tarjan, *A linear-time algorithm for testing the truth of certain quantified Boolean formulas*, Information Processing Letters, 8:121-123, 1979.

[2] H. K. Buning, M. Karpinski, and A. Flogel, *Resolution for quantified Boolean formulas*, Information and Computation, 117(1):12-18, 1995.

[3] M. Cadoli, A. Giovanardi, and M. Schaerf, *An algorithm to evaluate quantified Boolean formulae*, Proceedings of AAAI-98, pp. 262-267, 1998.

[4] J. Chen, I. A. Kanj, and W. Jia, *Vertex cover: Further observation and further improvements*, Lecture Notes in Computer Science 1665 (WG'99), Springer-Verlag, Berlin, pp. 313-324, 1999.

[5] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, U. Schöning, *A deterministic* $(2 - \frac{2}{k+1})^n$ *algorithm for k-SAT based on local search*, Accepted in Theoretical Computer Science, 2001.

[6] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM, 7(1):201-215, 1960.

[7] D. Eppstein, *Improved Algorithms for 3-Coloring, 3-Edge-Coloring, and Constraint Satisfaction*, Proceedings of 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 329-337, 2001.

[8] I. Gent and T. Walsh, *Beyond NP: the QSAT phase transition*, Proceedings of AAAI-99, 1999.

[9] E. Giunchiglia, M. Narizzano, and A. Tacchella. *QUBE: A system for deciding quantified Boolean formulas satisfiability*, Proceedings of the Int. Joint Conference on Automated Reasoning, 2001.

[10] E. A. Hirsch. *New worst-case upper bounds for SAT*, Journal of Automated Reasoning, 24(4): 397-420, 2000.

[11] O. Kullmann and H. Luckhardt, *Deciding propositional tautologies: algorithms and their complexity*, `http://www.cs.toronto.edu/~kullmann`.

[12] B. Monien and E. Speckenmeyer, *Solving satisfiability in less than* 2^n *steps*, Discrete Applied Mathematics, 10:287-295, 1985.

[13] R. Paturi, P. Pudlak, and F. Zane, *Satisfiability coding lemma*, Proceedings of 38th IEEE Symposium on Foundations of Comp. Sci., pp. 566-574, 1997.

[14] R. Paturi, P. Pudlak, M. E. Saks, and F. Zane. *An improved exponential-time algorithm for k-SAT*, Proceedings of 39th IEEE Symposium on Foundations of Comp. Sci., pp. 628-637, 1998.

[15] R. Rodosek, *A new approach on solving 3-satisfiability*, Proceedings of 3rd Int. Conference on AI and Symbolic Mathematical Computing, Springer-Verlag, LNCS 1138, pp. 197-212, 1996.

[16] J. Rintanen, *Improvements to the evaluation of quantified Boolean formulae*, 16th Int. Joint Conference on AI, pp. 1192-1197, 1999.

[17] J. Rintanen, *Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulas*, Workshop on Theory and Applications of QBF, Int. Joint Conference on Automated Reasoning, 2001.

[18] R. Schuler, U. Schöning, and O. Watanabe, *An improved randomized algorithm for 3-SAT*, Technical Report TR-C146, Dept. of Mathematical and Computing Sci., Tokyo Inst. of Tech., 2001.

[19] U. Schöning, *A probabilistic algorithm for k-SAT and constraint satisfaction problems*, Proceedings of 40th IEEE Symposoum on Foundations of Comp. Sci., pp. 410-414, 1999.