# Applying Monte Carlo Tree Search to Quantified Boolean Formulas

**Mathias N. Justesen**
Technical University of Denmark, July 2016

**Abstract**

In this paper, we apply an algorithm based on Uniform Confidence bounds applied to Trees, a bandit based Monte Carlo Tree Search algorithm, to the problem of evaluating Quantified Boolean Formulas, in order to determine the feasibility of such an approach. The performance of the algorithm is compared with a standard DPLL based algorithm based on the size of the search tree. Our results show that our algorithm is generally inferior to DPLL, and we discuss the reasons for this and possible future work to remedy this.

## 1 Introduction

The Monte Carlo Tree Search algorithm, UCT (Upper Confidence bounds applied to Trees), has shown great promise in adversarial planning domains, in particular the board game Go, in which computers were otherwise no match for the best human players. We investigate if UCT can be adopted to evaluate Quantifed Boolean Formulas (QBF). Just as SAT is the canonical NP-complete problem, so is QBF the canonical complete problem of PSPACE.[1] Hence, any breakthrough for this problem would be widely applicable for other problems involving knowledge representation, e.g., STRIPS-planning. A generalized version of Go has also been shown to be PSPACE-complete (Papadimitriou 1994), which gives us a hint that UCT might be a powerful enough to tackle QBF. Furthermore, the evaluation of a QBF can be viewed as two player game, just as Go. Finally, initial investigation of a UCT-based approach to SAT has shown promise (Previti et al. 2011).

Most current solvers of QBFs are based on the DPLL algorithm, originially proposed by (Davis, Logemann, and Loveland 1962) for SAT, and therefore is referred to as QDPLL in regard to QBF. The QDPLL algorithm explores the search space in a depth-first manner. In contrast, UCT tries to balance depth-first search and breadth-first search, by repeatedly exploring the search tree from the root and recursively choosing the most promising child, based on its utility computed on the previous iterations. When an unexplored node is chosen, the utility of the node is estimated using Monte Carlo methods.

[1]Therefore, the problem is sometimes referred to as QSAT in the literature on complexity theory.

In the next two sections we provide the necessary background for QBF and Monte Carlo Tree Search, UCT in particular. Then we describe how to extend UCT in order to make it complete for QBF in section 4. The results of the comparison with QDPLL are detailed in section 5, followed by a discussion. The paper is rounded off with a conclusion and proposals for future work.

## 2 Quantified Boolean Formulas

Quantified Boolean Formulas extend propositional logic with variables that are all bound by either the existential quantifier $\exists$ or the universal quantifier $\forall$. Since there are no free variables, a formula is always either true or false. For example, the formula

$$\forall x \exists y \, (x \vee y) \wedge (\neg x \vee \neg y) \tag{1}$$

is true, since no matter the value of $x$, we can assign $y := \neg x$. Note that if we switch the order of the quantifiers, i.e.,

$$\exists y \forall x \, (x \vee y) \wedge (\neg x \vee \neg y), \tag{2}$$

this is no longer true, since the truth value of $y$ is fixed first.

A formula $\varphi$ is said to be in *prenex normal form*, if it is written as a string of consecutive quantifiers followed by a quantifier free part, i.e.,

$$\varphi = Q_1 x_1 Q_2 x_2 \ldots Q_m x_m \, \psi,$$

where $Q_i$ is either $\exists$ or $\forall$, and $\varphi$ is free of quantifiers. Furthermore, if $\psi$ is in Conjunctive Normal Form (CNF), i.e., a conjunction of clauses (disjunctions of literals), then we say that $\varphi$ is in *clausal form*. Both (1) and (2) are in clausal form.

### 2.1 QDPLL

A straight forward way to evaluate a QBF in clausal form is given by Algorithm 1 (as found in, e.g., (Audemard and Saïs 2005)) is called QDPLL, since it is based on the DPLL algorithm for propositional logic. The algorithm first checks if (given the current model of assignments $M$) some clause is false, because then the entire formula is false. Otherwise, if all the clauses evaluate to true, then the formula is true. One of these must be the case when all variables have been assigned and potentially sooner. If these checks did not lead to termination, then we consider two cases:

1. If the outermost quantifier $Q_1$ is universal, then $\varphi$ is true, if and only if, $Q_2 x_2 \dots Q_m x_m \ \psi$ is true, no matter if we assign $x_1$ true or false. Hence, we call recursively for both these cases, and return true, if they are both true and false otherwise. Thus, the **and** can be *short-circuited*.
2. Otherwise, if $Q_1 = \exists$, then $\varphi$ is true, if either assigning $x_1$ true or false makes $\varphi$ true. Hence, the case is similar to the on above, only the **and** is replaced by an **or**, which can also be short-circuited.

**Algorithm 1** QDPLL algorithm

```
function QDPLL(φ = Q1 x1 Q2 x2 ... Qm xm ψ, M)
    SIMPLIFY(φ, M)
    if some clause is false then
        return false
    else if all clauses are true then
        return true
    end if
    if Q1 = ∀ then
        return QDPLL(Q2 x2 ... Qm xm ψ, M ∪ x1 := true)
           and QDPLL(Q2 x2 ... Qm xm ψ, M ∪ x1 := false)
    else (Q1 = ∃)
        return QDPLL(Q2 x2 ... Qm xm ψ, M ∪ x1 := true)
            or QDPLL(Q2 x2 ... Qm xm ψ, M ∪ x1 := false)
    end if
end function
```

Finally, QDPLL algorithms use various techniques for simplifying the formula, which often leads to faster termination. Many of these are described in (Cadoli et al. 2002). The implementation used in this paper uses the two of the most common techniques, *unit propragation* and *monotone literals*.

A clause is said to be a unit clause, if it consists of just one literal $l$. If this literal is universally quantified then we might as well consider $l$ to be false, thus, the clause is false and, hence, the entire formula is false. Otherwise, if $l$ is existentially quantified, then we might as well assign the corresponding variable, such that $l = true$. Note that this observation also holds, if $l$ is the only literal, which has not been assigned in a clause, hence, finding one unit clause and assigning the corresponding variable, can lead to new unit clauses. Hence, such assignments are said to propagate.

If a literal $l$ only occurs with one polarity in the formula, then it is said to be monotone. Hence, we might as well assign the corresponding variable $x$, such that $l = false$, if $x$ is bound by a universal quantifier, or if $x$ is bound by an existential quantifier, assign $x$ such that $l = true$.

## 3 Monte Carlo Tree Search

In recent years Monte Carlo Tree Search (MCTS) has received increased interest from the AI community, due to its success in Go. A comprehensive overview of the background, variations, and applications of MCTS can be found in (Browne et al. 2012).

MCTS can be applied to adversarial planning domains of two agents, usually referred to as MIN and MAX. For any given state $s$, a set $A = \text{ACTIONS}(s)$ of applicable actions is available, which leads to new states by a transaction function $T$, such that $T(s, a) = s'$ where $a \in A$. The state $s'$ is said to be the *successor* of $s$. If $A$ is empty, the state is *terminal* and the state $s$ has a utility value $Q = \text{UTILITY}(s) \in [-1, +1]$. MAX seeks terminal states with high utility and MIN seeks low utility states.

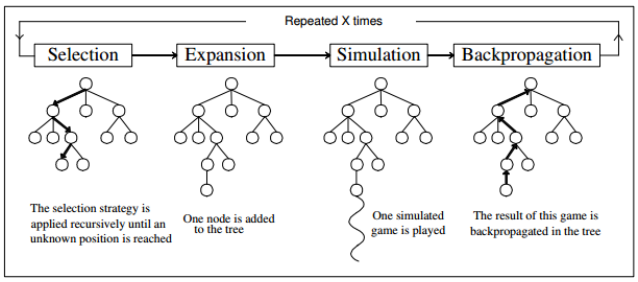


Figure 1: The four steps in an iteration of MCTS. Figure from (Chaslot 2010).

MCTS works by constructing a search tree in an iterative fashion. Starting with the root node of the initial state, the following four steps are performed repeatedly (as illustrated in Figure 1):

1. **Selection:** Starting from the root node $r$, select successive children according to a *tree policy* $\pi$, until a leaf node $n$ is reached.
2. **Expansion:** Unless $n$ is the node of a terminal state, add one (or more) child nodes to $n$ according to the set of available actions.
3. **Simulation:** From $n$ simulate the remaining actions according to a heuristic until a terminal state is reached.
4. **Backpropagation:** Use the result of the simulation to update the estimated utility of the nodes on the path from $n$ to $r$.

MCTS can proceed until the entire state space has been exhausted, but the strength of it is that it can be stopped at any time, still giving an estimated utility of the root node and, thus, the action which is currently believed to yield the best result.

### 3.1 UCT

At the heart of MCTS is the tree policy for selecting children, which determines the balance between exploration and exploitation. Originally proposed by (Kocsis and Szepesvári 2006), we consider the UCT variant of MCTS, which builds on the UCB1 algorithm for multi-armed bandits for its tree policy. An interesting property of UCT, is that given enought time (iterations) it converges to the Minimax-tree and, hence, optimality.

Algorithm 2 summarizes UCT in pseudo code. For each node $n$, UCT tracks the number of times $n$ has been visited, $v(n)$, and its current estimated utility $Q(n)$. The state of $n$ is denoted $s(n)$.

The tree policy is given by

$$\pi(n) = \text{argmax}_{k \in \textsc{Children}(n)} \left( Q(k) \pm \sqrt{\frac{\log v(n)}{v(k)}} \right),$$

where

$$\textsc{Children}(n) = \{k \mid a \in \textsc{Actions}(s(n)), s(k) = T(s,a)\},$$

i.e., the nodes of the states that are successors of $s(n)$. The constant $c$ is tuned empirically and controls the balance between exploitation (left-hand term) and exploration (right-hand term). The $\pm$ is a + when Max is on the move and − when it is Min to move.

**Algorithm 2** UCT algorithm

```
function UCTSearch(s)
    Create root node r with initial state s
    while time remains do
        UCTRecurse(r)
    end while
    return action leading to π(r)
end function

function UCTRecurse(n)
    if v(n) = 0 then
        if s(n) is terminal then
            Q := Utility(s(n))
        else
            Add Children(n) to the search tree
            Q := simulated utility of s(n)
        end if
    else
        Q := UCTRecurse(π(n))
    end if
    v(n) := v(n) + 1
    Q(n) := Q(n) + (Q − Q(n)) / v(n)
    return Q
end function
```

## 4 Adopting UCT to QBF

As detailed in, e.g., (Sipser 2006), the evaluation of a QBF can be viewed as a game between two players, an *evaluation game*. Consider a formula $\varphi$ in clausal form. Then an action corresponds to choosing a truth assignment of the variable of the outermost quantifier $Q_1$. If $Q_1 = \exists$ then we let Max choose the assignment, and Min chooses if $Q_1 = \forall$. Max tries to show that $\varphi$ is true, and Min that it is false. Thus, if both players play optimally, then $\varphi$ is true if Max wins, and false if Min wins.

Hence, QBF is a two-agent adversarial planning problem, so we can use UCT to play such an evaluation game. However, UCT cannot guarantee optimal play, unless the entire search tree is expanded, which is infeasible except for the simplest instances. Instead, we augment UCT to make it complete for QBF without exhausting the state space, by noting the following logical short cuts:

1. For $\exists$:
   (a) Let $\varphi = \exists x\, \psi$. If $\psi[x/v] = \mathit{true}$, then $\varphi = \mathit{true}$, where $v \in \{\mathit{true}, \mathit{false}\}$.
   (b) Let $\varphi = \exists x\, \psi$. If $\psi[x/\mathit{true}] = \mathit{false}$ and $\psi[x/\mathit{false}] = \mathit{false}$, then $\varphi = \mathit{false}$.
2. For $\forall$:
   (a) Let $\varphi = \forall x\, \psi$. If $\psi[x/v] = \mathit{false}$, then $\varphi = \mathit{false}$.
   (b) Let $\varphi = \forall x\, \psi$. If $\psi[x/\mathit{true}] = \mathit{true}$ and $\psi[x/\mathit{false}] = \mathit{true}$, then $\varphi = \mathit{true}$.

Using these observations, we arrive at a complete procedure for evaluating QBFs, as detailed in Algorithm 3. We shall refer to this algorithm as $\text{UCT}_{\text{QBF}}$.

The utility of a non-terminal state, i.e., a formula $\varphi$, is estimated by randomly assigning values to the remaining variables (or to as many as is necessary for determining the truthhood of $\varphi$). We perform a constant number $p$ of such simulations and return the average of the utilities as the estimation. If a simulation results in the formula being true, its utility is 1. Otherwise, if a simulation leads to $\varphi$ being false, then we let the utility of that simulation be

$$Q = -1 + \frac{\text{true clauses in } \varphi}{\text{total number of clauses in } \varphi}$$

in order to gauge how "close" the formula is to being true. The precise formulation is given in Algorithm 4

## 5 Results

We pursue a preliminary assessment of the feasibility of a Monte Carlo based approach to solve QBF instances. Thus, our foremost concern is not execution time or comparison against state-of-the-art QBF solvers. Instead, we look at the size of the search trees, i.e., the number of assignments needed to determine the truthhood of a formula. A similar approach was taken in (Previti et al. 2011) to evaluate the feasibility of UCT on SAT instances. In order to determine the size of the search trees accurately, we compare our implementation of $\text{UCT}_{\text{QBF}}$ against our own implementation of QDPLL. Furthermore, an in-house implementation allows us to employ the same formula simplifications (unit propagation and monotone literals) to both $\text{UCT}_{\text{QBF}}$ and QDPLL, ensuring a fair comparison.

Through experiments we have found that setting the exploration bias constant $c$ to 0 leads to the best performance on average, hence, $c$ was fixed at 0 in the results detailed in this section. For $\text{UCT}_{\text{QBF}}$, we used $p = 5$ simulations to estimate the utility of a non-terminal node. The benchmark has been carried out on a machine with four 3.5GHz Intel Core i5 processors and 8GB RAM space. The implementation does not support parallel computations, hence, each benchmark instance was only run on one processor.

We use QBF instances from the QBFLIB repository (Giunchiglia et al. 2005), which range over various verification problems and some miscellaneous instances. To ensure the correctness of the results, they were compared to those of DepQBF[2], which has competed in multiple

[2] `http://lonsing.github.io/depqbf/`

**Algorithm 3** $\text{UCT}_{\text{QBF}}$ algorithm

```
type Result = True | False | Utility of float

function UCTQBFSearch(φ)
    Create root node r with initial state given by φ
    Q := Utility 0.0
    while Q is instance of Utility do
        Q := UCTQBFRecurse(r)
    end while
    return Q
end function

function UCTQBFRecurse(n)
    if v(n) = 0 then
        Simplify(s(n))
        if all clauses in s(n) are true then
            Mark n as true
            return True
        else if some clause in s(n) is false then
            Mark n as false
            return False
        else
            Let Q1x1Q2x2...QmxmψbeQBFofs(n)
            Create node n_t of Q2x2...QmxmψbeQBFofs(n)[x1/true]
            Create node n_f of Q2x2...QmxmψbeQBFofs(n)[x1/false]
            Add n_t and n_f to the search tree
            r := SimulateUtility(s(n))
        end if
    else
        R := UCTRecurse(π(n))
        Let Q1x1Q2x2...QmxmψbeQBFofs(n)
        if Q1 = ∃ and R = True then
            Mark n as true
            return true
        else if Q1 = ∃ and both children are marked false
then
            Mark n as false
            return False
        else if Q1 = ∀ and s = False then
            Mark n as false
            return False
        else if Q1 = ∀ and both children are marked true
then
            Mark n as true
            return True
        else
            if R = True then
                Q := Utility +1.0
            else if s = False then
                Q := Utility −1.0
            else
                Q := R
            end if
        end if
    end if
    v(n) := v(n) + 1
    Q(n) := Q(n) + (Q − Q(n)) / v(n)
    return Q
end function
```

**Algorithm 4** Simulation in $\text{UCT}_{\text{QBF}}$.

```
function SimulateUtility(s)
    e := 0
    for all i = 1 to p do
        Let Q1x1Q2x2...QmxmψbeQBFofs
        ψ' := ψ
        for all j = 1 to n do
            Let v be a random boolean value
            ψ' := ψ'[x_j/v]
            if all clauses in ψ' are true then
                e := e + 1.0
                break
            else if some clause in ψ' is false then
                e := e − 1.0 + (true clauses in ψ') / (total number of clauses in ψ')
                break
            end if
        end for
    end for
    return e/p
end function
```

QBFEVAL events[3]. In total, we used 141 instances, which either $\text{UCT}_{\text{QBF}}$ or QDPLL could solve within 20 seconds. QDPLL could solve 139 of these instances, while $\text{UCT}_{\text{QBF}}$ could solve 106. QDPLL solved 35 instances, which $\text{UCT}_{\text{QBF}}$ could not solve, and two instances were only solved $\text{UCT}_{\text{QBF}}$. Of the 104 instances that both algorithms solved, QDPLL had a smaller search tree in 38 cases, while $\text{UCT}_{\text{QBF}}$ only had a smaller search tree in five cases. Their search tree was of the same size in the remaining 61 instances. Overall, QDPLL generated significantly smaller search trees as summarized in Figure 2.

## 6 Discussion

The immediate conclusion to draw from the results, is that UCT is not as efficient as a relatively simple implementation of QDPLL. It seems that there is not much to gain from a more breadth-first search as employed by UCT, confirmed by the observation that the best results are achieved by setting the exploration bias constant $c$ to 0.

Furthermore, it appears that the information gained on previous iterations to choose successful nodes is not increasing performance. One explanation for this, could stem from the fact that both sides, Min and Max, try to play optimally, even though such an effort is futile for Min if the formula in question is true, and vice versa for Max if the formula is false. Hence, by playing the best possible moves, one player is postponing the inevitable loss.

This futile effort leading to postponement of deciding the truthhood of a formula was also observed in another setting: Instead of always assigning a truth value to the variable of the outermost quantifier, one can group the variables of consecutive quantifiers, without changing the truthhood of the formula, e.g.,

$$\exists x \forall y \forall z\ \psi \equiv \exists x \forall z \forall y\ \psi.$$

[3] `http://www.qbflib.org/index_eval.php`

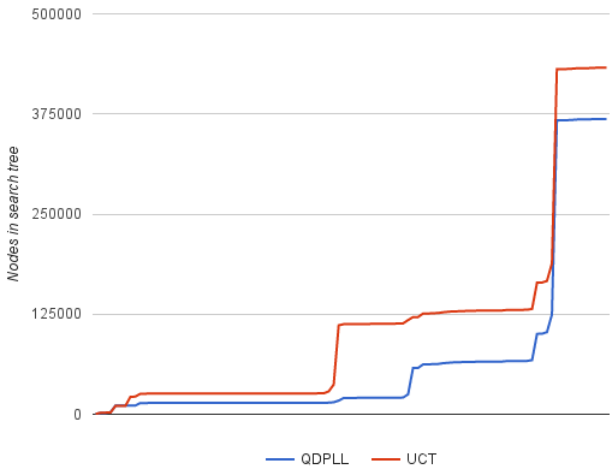


Figure 2: Graph of the accumulated size of the search trees on the 104 instances that both algorithms could solve.

Hence, in this case, after assigning a value to $x$, we are free to choose whether to first assign a value to $y$ or $z$. We experimented with choosing the variable that satisfied the most clauses if MAX is on the move, and the one satisfying the fewest clauses if MIN is on the move. Overall, this heuristic lead to bigger search trees being generated. Again, we conjecture that this is because the losing player is merely postponing the inevitable by choosing variables that delay the defeat.

## 7 Conclusion and future work

In summary, we conclude that our initial investigations has not shown Monte Carlo Tree Search to be a fruitful approach for the problem of Quantified Boolean Formulas. It appears that the adversarial nature of the UCT algorithm is not advantageous, since we need to determine the outcome from the initial state, as oppose to, e.g., Go, where the outcome of the game can not be determined until the very end of the game.

This finding seems to contradict the otherwise promising result of UCT applied to SAT (Previti et al. 2011). However, SAT and QBF differ in one fundamental aspect, namely that QBF is a two-player game and SAT can be viewed as a one-player game, since a SAT instance is simply a QBF where all the variables are existentially quantified. Thus, the adversarial aspect, which we conjecture is the drawback of MCTS, is not present in SAT.

More work needs to be done, before we rule out MCTS entirely. There were a few instances where $\text{UCT}_{\text{QBF}}$ outperformed QDPLL in our data set, and it would be interesting to investigate whether these form a family (or families) of formulas, where MCTS consistently outperforms QDPLL. Thus, MCTS can be still be used as part of a portfolio of algorithms, where different algorithms expands the applicability of the portfolio. See for example (Xu et al. 2008) for a discussion of such an approach in regard to SAT.

Finally, we performed preliminary work on formulas that were in non-clausal form, which seemed promising. However, we could not directly compare the performance on these instances with QDPLL, since the instances proved infeasible to convert to clausal form using a straightforward recursive translation. Still, it is promising that UCT is flexible enough to also handle formulas in non-clausal form, but a comparison with other algorithms is necesarry to deem it useful.

## References

Audemard, G., and Saïs, L. 2005. A symbolic search based approach for quantified boolean formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, 16–30. Springer.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Cadoli, M.; Schaerf, M.; Giovanardi, A.; and Giovanardi, M. 2002. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *Journal of Automated Reasoning, J. Autom. Reasoning, J Auto Reas, J Autom Reasoning, J. Automat. Reason* 28(2):101–142.

Chaslot, G. 2010. Monte-Carlo tree search. Master's thesis, Universiteit Maastricht.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Commun. ACM* 5(7):394–397.

Giunchiglia, E.; Narizzano, M.; Pulina, L.; and Tacchella, A. 2005. Quantified Boolean Formulas satisfiability library (QBFLIB). `www.qbflib.org`.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*. Springer. 282–293.

Papadimitriou, C. H. 1994. *Computational Complexity*. Addison-Wesley.

Previti, A.; Ramanujan, R.; Schaerf, M.; and Selman, B. 2011. *AI*IA 2011: Artificial Intelligence Around Man and Beyond: XIIth International Conference of the Italian Association for Artificial Intelligence, Palermo, Italy, September 15-17, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg. chapter Monte-Carlo Style UCT Search for Boolean Satisfiability, 177–188.

Sipser, M. 2006. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology, Boston.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research* 32:565–606.