# Monte-Carlo Style UCT Search for Boolean Satisfiability

Alessandro Previti[1], Raghuram Ramanujan[2], Marco Schaerf[1], and Bart Selman[2]

[1] Dipartimento di Informatica e Sistemistica Antonio Ruberti, Sapienza, Università di Roma, Roma, Italy
[2] Department of Computer Science, Cornell University, Ithaca, New York

**Abstract.** In this paper, we investigate the feasibility of applying algorithms based on the Uniform Confidence bounds applied to Trees [12] to the satisfiability of CNF formulas. We develop a new family of algorithms based on the idea of balancing exploitation (depth-first search) and exploration (breadth-first search), that can be combined with two different techniques to generate random playouts or with a heuristics-based evaluation function. We compare our algorithms with a DPLL-based algorithm and with WalkSAT, using the size of the tree and the number of flips as the performance measure. While our algorithms perform on par with DPLL on instances with little structure, they do quite well on structured instances where they can effectively reuse information gathered from one iteration on the next. We also discuss the pros and cons of our different algorithms and we conclude with a discussion of a number of avenues for future work.

## 1 Introduction

The Upper Confidence bounds applied to Trees (from now on UCT) algorithm, introduced by Kocsis and Szepesvári in [12], is an (increasingly popular and succesful) adaptation of the work on Upper Confidence Bounds (UCB) by Auer, Cesa-Bianchi and others [2,3,4] on the multi-armed bandit problem to tree search. It has been successfully used in many game playing programs, the most notable being MoGo which is one of the strongest computer Go players [10,16]. In this paper we perform a preliminary investigation into the application of UCT-style search algorithms to satisfiability testing of propositional formulas in Conjunctive Normal Form (CNF).

Rather than explore the search space in a depth-first fashion, in the style of DPLL [9,8], UCT repeatedly starts from the root node and incrementally builds a tree based on estimates of node utilities and node visit frequencies computed from previous iterations. In most implementations of UCT, the estimated utility of a new node is computed using Monte-Carlo methods, i.e., by generating

R. Pirrone and F. Sorbello (Eds.): AI*IA 2011, LNAI 6934, pp. 177–188, 2011.
© Springer-Verlag Berlin Heidelberg 2011

random completions of the search (termed “playouts”) and averaging their outcomes. This utility is revised each time the search revisits the node using the estimated values of the children. This technique is especially effective when no adequate heuristics is available to perform this value estimation task.

Here we present in detail a family of algorithms called UCTSAT that employ the UCT search control mechanism but use different mechanisms to estimate the utility of a node. In the first version, called $\text{UCTSAT}_h$, a heuristic is used to estimate the initial utility of a node, more precisely, the heuristic used is the fraction of the total set of clauses that are satisfied by the partial assignment associated with the node. While the results have been promising, especially when applied to structured instances, we also experiment with two more variants, called $\text{UCTSAT}_{cp}$ and $\text{UCTSAT}_{sbs}$, that use search strategies that are closer to the more traditional usage of UCT algorithms, that is using random tryouts in a MonteCarlo style. A very short description of the $\text{UCTSAT}_h$ algorithm has been described, but not presented, in [14].

While we do not expect UCTSAT to outperform the state of the art SAT solvers (especially with respect to CPU time), we believe that the development of an algorithm based on a radically different search technique is important for at least two reasons: (a) the hardness of SAT instances is related to the algorithm used [1,7], and hence UCTSAT, which uses different search strategies, can provide useful and new insights into the complexity of SAT instances; and (b) because such algorithms can be useful when included in a portfolio of algorithms (see, for example, [17]) where very different solution techniques can help expand the range of applicability of the portfolio.

The remainder of this paper is organized as follows. In Section 2 we briefly describe the UCT algorithm. Section 3 presents all three versions of the UCTSAT algorithm for satisfiability testing. In Section 4 we present preliminary experimental results from applying UCTSAT to a variety of benchmark problems, and compare it to our own DPLL implementation and to WalkSAT. Section 5 concludes with a discussion of our results, and outlines a few topics that deserve further investigation.

## 2 Upper Confidence Bounds Applied to Trees (UCT)

Monte-Carlo tree search algorithms such as UCT [12] have recently received a great deal of attention from the planning and game-playing community, in particular due to their success in the domain of Go [10,16]. UCT builds on the UCB1 algorithm for multi-armed bandits [2], which is used to guide the search tree construction process. Exploration of under-sampled actions is balanced against exploitation of known good actions to generate asymmetric trees that are deeper in more promising regions of the search space and shallower elsewhere.

Algorithm 1 describes the recursive procedure UCT uses to build the search tree. $T(s, a)$ is the domain transition function that returns the state $s'$ reached

from taking action $a$ in state $s$. The algorithm maintains two lookup tables — $n(s)$ tracks the number of times state $s$ has been visited and $Q(s)$ tracks the current estimated utility of the state $s$. The action selection operator $\pi(s)$ is repeatedly applied to descend down the tree until a previously unvisited (or terminal) node $k$ is reached. $k$ is added to the tree and an estimate of its utility is computed which is used to update $Q(s)$ and $n(s)$ for all nodes $s$ on the path from the root node to $k$, according to lines 11 and 12. Under this scheme, the size of the tree grows by one node on every iteration.

We describe $\pi(s)$ and the utility estimation step in greater detail below:

- **Action Selection:** Given a state $s$, the action selection operator $\pi(s)$ returns the action $a$ that maximizes an upper confidence bound on the utility of the resulting state $s' = T(s, a)$:

$$\pi(s) = \operatorname*{argmax}_{a} \left( Q(T(s,a)) + c \cdot \sqrt{\frac{\log n(s)}{n(T(s,a))}} \right)$$

  If $n(T(s, a)) = 0$ for an action $a$, then it is selected first, before any actions are re-sampled. Ties are broken randomly. The constant $c$ is tuned empirically and controls the extent to which exploration or exploitation is favored.
- **Value Estimation:** For terminal nodes, the true utility of the state is returned. For non-terminal nodes, an estimate of the true utility is returned. This estimate can be computed using a domain-independent approach, such as a random playout (which is also the traditional solution), or using a domain-specific heuristic. Notice that the function can return either a definite answer (SAT/UNSAT) or a reward $r$ for the node. To be more precise, both SAT and UNSAT are represented as integer constants in all of the algorithms.

A UCT search consists of repeatedly calling the function given in Algorithm 1 on the root node for as long as time allows. At that point, the action that leads to

**Algorithm 1.** The UCT Algorithm

1: *int* **Function** $UCTRecurse(s : \text{state})$
2: **if** $s$ is a terminal state **then**
3:     Add $s$ to the search tree if $n(s) = 0$
4:     $r \leftarrow$ true utility of $s$
5: **else if** $n(s) = 0$ **then**
6:     Add $s$ to the search tree
7:     $r \leftarrow$ estimated utility of $s$
8: **else**
9:     $r \leftarrow UCTRecurse(T(s, \pi(s)))$
10: **end if**
11: $n(s) \leftarrow n(s) + 1$
12: $Q(s) \leftarrow Q(s) + (r - Q(s))/n(s)$
13: **return** $r$

the state with the highest average utility is returned. Alternate schemes include returning the action with the most number of visits and returning the action with the highest lower confidence bound. In practice, there is little difference between these approaches.

## 3 UCTSAT

Typical UCT implementations estimate the utility of a node $n$ on the first visit by sampling the search space subsumed by $n$, via random or pseudo-random playouts. This idea is very appealing when no good heuristics are available for a domain. The pseudo-code for the recursive tree-building component of our procedure (which we call UCTSAT) is given by Algorithm 2.

**Algorithm 2.** The UCTSAT Algorithm

```
 1: int Function UCTSATRecurse(s : state)
 2: if n(s) = 0 then
 3:    Add s to the search tree
 4:    r ← estimate(s)
 5:    if r = SAT then
 6:       print "Formula is satisfiable"
 7:       exit
 8:    else if r = UNSAT then
 9:       Mark s as closed
10:    else
11:       var(s) ← chooseVariable()
12:    end if
13: else
14:    r ← UCTSATRecurse(T(s, π(s)))
15:    if r = UNSAT then
16:       if all the children of s are closed then
17:          Mark s as closed
18:          return UNSAT
19:       else
20:          r ← 0
21:       end if
22:    end if
23:    n(s) ← n(s) + 1
24:    Q(s) ← Q(s) + (r − Q(S))/n(s)
25: end if
26: return r
```

Analogously to UCT, a UCTSAT search comprises repeated invocations of Algorithm 1 on the root node. UCTSAT behaves like a cross between a backtracking (DPLL-style) and a randomized algorithm (for example, WalkSAT [15]). It is a complete procedure that explores the search space in a very different fashion

to that of DPLL. While DPLL only backtracks when it has finished completely evaluating a branch, UCTSAT repeatedly starts from the root node and only goes one level deeper on each iteration. As in UCT, the UCB1 formula is used to control the descent down the tree, where each step involves making a variable assignment and simplifying the original formula. In the flavor of local search methods, the most promising branch is typically chosen at each step, but occasional deviations to sub-optimal branches (that may still lead to solutions) also occur. The search terminates when either:

1. a satisfying assignment is found (line 5)
2. the formula is determined to be unsatisfiable (line 17, when $s$ is the root)
3. or the specified number of iterations is exceeded

We highlight the key differences between Algorithm 2 and Algorithm 1 below:

– **Variable Assignment Look-up Table**: In addition to the look-up tables $n(s)$ and $Q(s)$ employed by UCT, we use an additional look-up table $var(s)$ that stores the variable that will be assigned at state $s$. This table is updated when the node $s$ is first created (line 11). The function $chooseVariable()$ we used in line 11 of Algorithm 2 returns the variable with the highest number of occurrences in the simplified formula.
– **Action Selection**: In game-tree search, an action corresponds to a move in the game. Here, an "action" is the process of assigning a value to a variable. As in UCT, an upper confidence bound is used to choose among the possible assignments.
– **Handling Terminal Nodes**: In UCT, the information from terminal and non-terminal nodes is propagated up the tree in an identical fashion. UCTSAT, on the other hand, handles terminal nodes as a special case. When a satisfying assignment is found, the search promptly terminates. When a contradiction is encountered at a node $s$, it is marked as "closed" and $s$ is never revisited by the search. A negative signal (value of 0) is propagated up to penalize this branch of the tree. When all the children of a node $s$ have been closed (line 17), then $s$ is closed as well — this mechanism propagates information unsatisfiable assignments up the tree.
– **Value Estimation**: We will experiment with various estimation functions, from simple heuristics to different forms of playouts. A more detailed analysis of the comparison between UCT and UCTSAT on this issue will be presented after the definition of the UCTSAT variants.

In the first version of UCTSAT, called UCTSAT$_h$, we replace the playouts with a simple heuristic while retaining the multi-armed bandit approach of balancing exploitation (i.e., DPLL-style depth-first expansion of the search tree) and exploration (i.e., breadth-first expansion). In fact, line 4 is replaced by $r \leftarrow h(s)$, where $h(s)$ is the fraction of clauses that have been satisfied having reached it.

While UCTSAT$_h$ seems a very promising alternative to DPLL-like algorithms, at least on structured instances, here we also want to experiment with two different mechanism to generate playouts. The first such algorithm we developed,

called UCTSAT$_{cp}$ (UCTSAT with complete playouts), uses $n$ (complete) random playouts to estimate $r$. More precisely, we generate $n$ random playouts, each generating a complete assignment to all of the (unassigned) variables of $s$. For each playout, if it satisfies the formula we are done, otherwise we compute its heuristic value $h(s, s')$ and then compute the average over all the playouts. The pseudo-code for the function is:

**Algorithm 3.** Complete playout

```
1:  int Function estimate(s : state)
2:  value ← 0
3:  for i = 1 to n do
4:     for all p ∈ unassignedLiterals(s) do
5:        p ← choose a Random value
6:        s' = update(s, p)
7:     end for
8:     r ← h(s, s')
9:     if r = SAT then
10:       print  "Formula is satisfiable"
11:       exit
12:    end if
13:    value ← value + r
14: end for
15: return  value/n
```

To fully specify the behavior of UCTSAT$_{cp}$ we still need to clearly define the functions $h(s, s')$, used in line 8 of Algorithm 3. We experimented with various choices, however here we report the results obtained using

$$h(s, s') = \Sigma_{c \in clauses(s)} SatVars(c, s')/sizeof(c, s)$$

where, for each clause of the simplified formula associated to state (partial assignment) $s$, we compute the number of literals satisfied by $s'$ ($SatVars(c, s')$) and divide it by the size of the clause in $s$. This metric tries to capture the probability that a solution exists in the close proximity of the current (falsifying) assignment.

While UCTSAT$_{cp}$ performs quite well it can be improved if we allow for a step-by-step choice of the variables in the playouts. More precisely, we define UCTSAT$_{sbs}$ where we choose one (unassigned) variable at the time, assign to it a random variable and then check whether the formula is already falsified in order avoid generating useless complete assignments. Moreover, after each assignment we can perform unit propagation so that forced assignments are immediately performed and we do not choose random values for forced variables. The idea of generating either complete assignments or step-by-step ones is not novel and has already been used in the literature, see, for example, the work of

Lombardi et. al. [13]. These changes are incorporated into this new version of the function *estimate* (Algorithm 4), where $update(s, p)$ not only assigns a value to the variable $p$ but also performs unit-propagation.

**Algorithm 4.** Step by step playout

```
1: int Function estimate(s : state)
2: value ← 0
3: for i = 1 to n do
4:    while ¬contradictory(s) do
5:       p ← random variable in unassignedLiterals(s)
6:       p ← choose a Random value
7:       s' = s
8:       s = update(s,p)
9:       if satisfied(s) then
10:         return SAT
11:      end if
12:   end while
13:   r ← h(s, s')
14:   value ← value + r
15: end for
16: return value/n
```

We can now summarize the main differences between the 3 variants of UCT-SAT presented in the paper and compare them with the UCT basic algorithm:

- **Value Estimation**: In the style of UCT, both $\text{UCTSAT}_{cp}$ and $\text{UCTSAT}_{sbs}$ employ a number of random playouts to estimate the utility of a node. Each playout will either find a satisfying assignment or will return a heuristic assessment of the solution found $h(s, s')$ (lines 13 and 8). On the other hand, $\text{UCTSAT}_{h}$ uses a heuristics that computes, for each state $s$, the fraction of clauses satisfied in $s$ with respect to the total number of clauses. This heuristics makes this variant of UCTSAT more similar to the methodologies used in most variants of DPLL, where the maximization of the number of satisfied clauses is used as a heuristics to choose the branching literal.
- **Using playouts**: In UCT, each playout will report the value of the terminal position reached, these values usually are one of +1 (win), 0 (draw), -1(loss). In our case, if a playout reports 1 (SAT) we are done, so in general, all playouts will report 0 (UNSAT) and, thus, it makes little sense to average these values. To overcome this problem, we assign to each playout a heuristic value $h(s, s')$ that should amount to how far away the playout is from a solution. Since there is clearly no metrics that can exactly compute this value, we tried several such functions. The main difference between $\text{UCTSAT}_{cp}$ and $\text{UCTSAT}_{sbs}$ is the playouts-generation mechanism, since in $\text{UCTSAT}_{cp}$ the playouts are fully generated at the beginning, while in $\text{UCTSAT}_{sbs}$ the playouts are generated one literal at the time, thus avoiding to generate too many inconsistencies early in the creation process.

We can now briefly compare the properties of all three variants of UCTSAT with both DPLL-like and WALKSAT-like algorithms. An experimental comparison is discussed at length in Section 4.

The advantages and disadvantages of UCTSAT with respect to DPLL-like algorithms can be summarized as follows:

+ Once UCTSAT has visited all the children of a node, their estimated utilities and visit counts can help it make an informed decision about which assignment to focus on at a node.
+ UCTSAT can exit a dead-end branch without the need to completely explore the branch.
+ The above two properties mean that UCTSAT can, in most cases, create more compact trees.
– UCTSAT needs to keep in memory (almost) all of the visited tree, making it a memory-intensive procedure.
– UCTSAT needs to visit each node multiple times.

We summarize the pros and cons of UCTSAT with respect to local search methods such as WALKSAT below:

+ UCTSAT uses the information obtained from each previous iteration to guide the next one, while (standard) WalkSAT restarts each try from scratch. While more advanced versions of WalkSAT use adaptive strategies to guide the restart, we conjecture that UCT makes a more informed decision.
+ UCTSAT is a complete algorithm that can prove the unsatisfiability of a formula by closing all the nodes.
– UCTSAT needs to retain all of the visited tree and the associated utility estimates and visit counts, making it more memory-intensive than WalkSAT.

## 4 Experimental Analysis

This work is a preliminary assessment of the feasibility of applying UCT-style methods to solve SAT problems. As such, we have focused our efforts on understanding whether the various variants of UCTSAT are capable of solving SAT instances using smaller search trees than DPLL. To simplify the comparisons, we contrast our algorithms against an in-house, no-frills implementation of DPLL, and against WalkSAT (where the number of flips is used as the comparison benchmark). Our DPLL implementation uses the same heuristic for picking the next variable for assignment as UCTSAT, i.e., the variable with the maximum number of occurrences in the simplified formula. The choice of which branch to explore first is made non-deterministically.

Empirical tuning of the exploration bias constant $c$ revealed that on most instances, a value of $c$ very close to 0 yielded the best performance on average; we therefore fix $c$ to 0 in all our experiments. Our WalkSAT runs use the novelty heuristic with a maximum of $200,000$ flips allowed per try. To keep the comparison fair and run-times reasonable, DPLL and UCTSAT also time-out once

the size of the search tree exceeds 200,000 nodes. Since the three algorithms are all non-deterministic, we perform 100 independent runs of each algorithm per instance, and report the average tree size over all successful runs.

Our experiments use instances from the SATLIB repository [11]. We focus our attention only on satisfiable instances; for unsatisfiable instances, UCTSAT$_h$ and DPLL construct identical trees since they use the same variable choosing heuristic. Having fixed the variable ordering, proving unsatisfiability requires both algorithms to visit the same set of nodes. It is meaningless to measure WalkSAT's performance on unsatisfiable instances since it is an incomplete method. Our first experiment uses uniform random 3-SAT instances of various sizes from the phase-transition region. Figure 1 presents a plot of the size of the trees explored by the various algorithms, as a function of the number of variables in the formula. Each data point corresponds to the average number of visited nodes of the algorithm over 100 instances. Notice that we use a logarithmic scale in all of our plots.

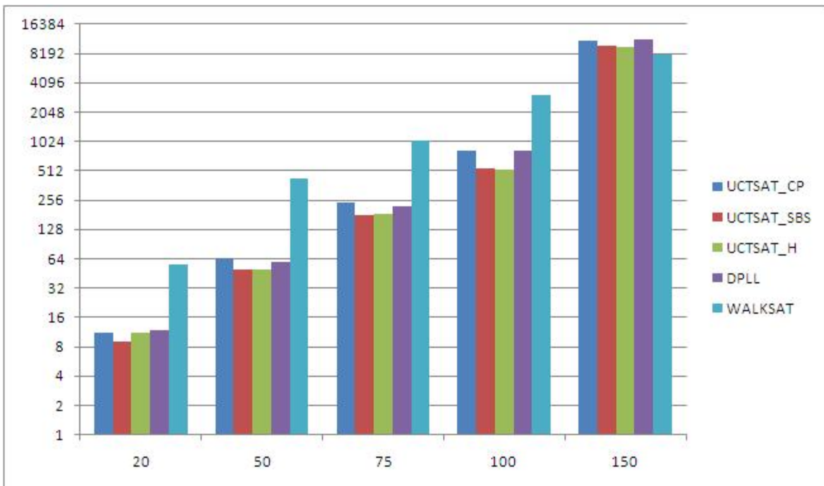


**Fig. 1.** Average tree sizes for uniform random 3-SAT instances

We observe that the rate of growth is similar for all the algorithms (with the exception of WalkSAT). Moreover, the size of the trees constructed by UCTSAT and DPLL is generally close, with UCTSAT$_{sbs}$ and UCTSAT$_h$ marginally outperforming DPLL. We believe that this similarity in tree sizes is due to the unstructured nature of these instances. UCTSAT works well when each exploration of the tree yields information that can be successfully used in subsequent ones. Little such information can be gained from unstructured instances and in such settings, UCTSAT only adds overhead to the DPLL machinery.

Our second experiment uses instances from the SAT encoding of graph coloring problems ("flat graph coloring"). These instances are randomly generated but have some underlying structure due to the encoding. These results are presented in Figure 2, and are qualitatively similar to those presented in Figure 1 — $\text{UCTSAT}_{sbs}$ slightly outperforms DPLL, while $\text{UCTSAT}_{cp}$ has worse performances. $\text{UCTSAT}_{h}$ is comparable with DPLL.

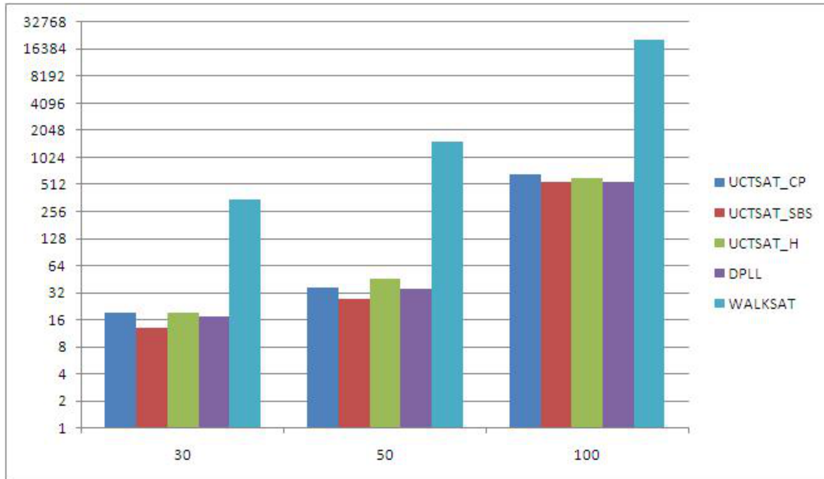


**Fig. 2.** Average tree sizes for flat graph coloring instances

Finally, we present some results on structured instances drawn from real-world problems, namely instances from circuit fault analysis (single-stuck-at-fault, or SSA).Figure 3 presents the average size of the search tree constructed by DPLL, UCTSAT and WALKSAT on 4 SSA instances. These results show that, when there is an underlying structure in the instances, both the $\text{UCTSAT}_{sbs}$ and $\text{UCTSAT}_{h}$ variants of UCTSAT can exploit it very effectively, by building a much smaller tree. It is not very clear why $\text{UCTSAT}_{cp}$ performs so poorly on these instances.

## 5 Discussion and Conclusions

In this paper, we have presented the UCTSAT family of algorithms based on UCT to solve CNF satisfiability problems. This family includes algorithms ($\text{UCTSAT}_{cp}$ and $\text{UCTSAT}_{sbs}$) using playouts to estimate the utility of nodes as well as an algorithm ($\text{UCTSAT}_{h}$) using a simple heuristic based on the fraction of satisfied clauses. Our initial experimental results show that UCTSAT does not perform well when instances have no underlying structure, but performs very well when it can successfully apply the information it gathers on one iteration on successive visits to the same node in the tree.

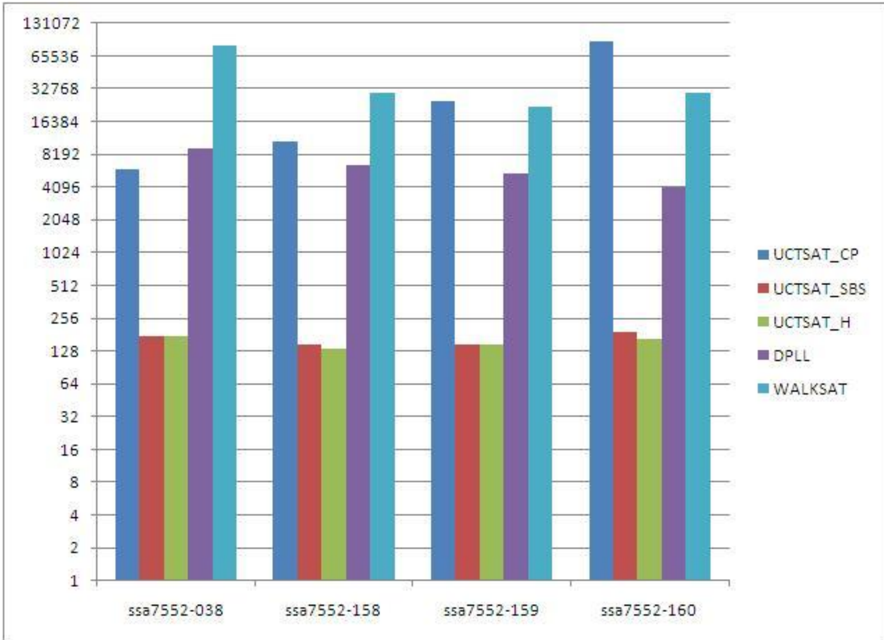


**Fig. 3.** Average tree sizes for SSA circuit fault analysis instances

UCT-based algorithms have already been successfully used in many applications, mostly in games such as GO [10,16]. All of these applications adopted a playout-based version of UCT, where the estimate is computed by generating random completions of the game. However, our experiments suggest that, at least in settings where a good heuristics to estimate the value of a node is available, using such a heuristics can be competitive (and even outperform) playouts-based algorithms.

There are many interesting avenues for future work. These include:

- Performing a systematic analysis of problem classes to gain insights into the classes of formulas that are better suited to UCTSAT.
- Experimenting with different heuristics to assess the quality of the playouts in both UCTSAT$_{cp}$ and UCTSAT$_{sbs}$.
- Gaining a better understanding of the advantages and disadvantages of using a heuristics to estimate the value of a node w. r. t. the use of playouts.
- Extending UCTSAT to solve Quantified Boolean Formulas (QBF), by extending the algorithms presented in [5,6]. We believe that UCT-style search can be effective in solving QBF instances that have a game-playing structure and is, therefore, closer to algorithms for Go and other games.

## References


1. Aguirre, A., Vardi, M.Y.: Random 3-SAT and BDDs: The plot thickens further. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 121–136. Springer, Heidelberg (2001)
2. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2), 235–256 (2002)
3. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: The nonstochastic multi-armed bandit problem. SIAM Journal on Computing 32(1), 48–77 (2003)
4. Auer, P., Ortner, R.: UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. Periodica Mathematica Hungarica 61(1), 55–65 (2010)
5. Cadoli, M., Giovanardi, A., Schaerf, M.: An algorithm to evaluate quantified boolean formulae. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998), pp. 263–267 (1998)
6. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An algorithm to evaluate quantified boolean formulae and its experimental evaluation. Journal of Automated Reasoning 28(2), 101–142 (2002)
7. Coarfa, C., Demopoulos, D.D., San Miguel Aguirre, A., Subramanian, D., Vardi, M.Y.: Random 3-SAT: The plot thickens. Constraints 8(3), 243–261 (2003)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
9. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
10. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th International Conference on Machine Learning, pp. 273–280. ACM, New York (2007)
11. Hoos, H.H., Stützle, T.: SAT20000: Highlights of Satisfiability Research in the year 2000, chapter SATLIB: An Online Resource for Research on SAT. In: Frontiers in Artificial Intelligence and Applications, pp. 283–292. Kluwer Academic, Dordrecht (2000), Web site available at: `http://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html`
12. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
13. Lombardi, M., Milano, M., Roli, A., Zanarini, A.: Deriving information from sampling and diving. In: Serra, R., Cucchiara, R. (eds.) AI*IA 2009. LNCS, vol. 5883, pp. 82–91. Springer, Heidelberg (2009)
14. Previti, A., Ramanujan, R., Schaerf, M., Selman, B.: Applying uct to boolean satisfiability. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 373–374. Springer, Heidelberg (2011)
15. Selman, B., Kautz, H.A., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Citeseer (1996)
16. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii, pp. 175–182 (2007)
17. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32(1), 565–606 (2008)