Monte-Carlo Tree Search for the Maximum Satisfiability Problem

Jack Goffinet and Raghuram Ramanujan(✉)

Department of Mathematics and Computer Science, Davidson College, Davidson, NC 28035, USA
{jagoffinet,raramanujan}@davidson.edu

Abstract. Incomplete algorithms for the Maximum Satisfiability (MaxSAT) problem use a hill climbing approach in tandem with various mechanisms that prevent search stagnation. These solvers' conflicting goals of maintaining search mobility while discovering high quality solutions constitute an exploration-exploitation dilemma, a problem which has been tackled with great success in recent years using Monte-Carlo Tree Search (MCTS) methods. We apply MCTS to the domain of MaxSAT using various stochastic local search (SLS) algorithms for leaf node value estimation, thus offering a novel hybrid alternative to established complete and incomplete solution techniques. Our algorithm outperforms baseline SLS algorithms like Walksat and Novelty on most problem instances from the 2015 MaxSAT Evaluation. It also outdoes CCLS, a state-of-the-art incomplete MaxSAT solver, on a number of challenging industrial instances from the 2015 MaxSAT Evaluation.

1 Introduction

A canonical NP-complete problem, Boolean Satisfiability (SAT), has attracted a tremendous amount of attention from researchers in the A.I. and broader computer science community over the years. The Maximum Satisfiability problem (MaxSAT) is a generalization of SAT: in this setting, the goal is to find a truth assignment that satisfies the maximum number of clauses (i.e., constraints) in a given formula. Modern MaxSAT solvers generally fall into one of two categories. *Complete* solvers systematically explore the problem search space and can find provably optimal solutions to a given instance. Typically, they are implemented using branch-and-bound techniques, or make iterative calls to a SAT solver [19]. In contrast, *incomplete* solvers start with a random truth assignment and attempt to discover better assignments by interleaving greedy and random walk steps in the search space according to some policy. The WalkSAT algorithm [23] is the exemplar of this class of stochastic local search (SLS) methods. Their memorylessness means that these algorithms are incapable of proving that a solution is optimal. Incomplete SLS solvers are nonetheless popular as they are faster than their complete counterparts. Moreover, SLS algorithms are able to find approximate solutions to large problem instances that are beyond the capabilities of complete MaxSAT solvers.

M. Rueher (Ed.): CP 2016, LNCS 9892, pp. 251–267, 2016.
DOI: 10.1007/978-3-319-44953-1_17

SLS algorithms work by using various properties of the underlying formula as a guiding gradient. For example, WalkSAT uses the number of unsatisfied clauses, while modern solvers like CCLS [16] use more intricate strategies such as configuration checking [4]. However, this greedy approach, which is beneficial for refining the quality of a solution, also often leads the solver into "basins" in the search space that contain suboptimal solutions and from which escape is difficult [10]. Authoring an effective SLS algorithm thus requires careful management of this tension between preventing search stagnation and ensuring the reachability of high-quality solutions. We observe that this can be viewed as an example of the exploration-exploitation dilemma that is well-studied in the reinforcement learning community [27]. In recent years, Monte-Carlo Tree Search (MCTS) methods have proven to be an effective option for addressing this dilemma, particularly within the setting of sequential decision-making problems.

MCTS methods, such as the Upper Confidence bounds applied to Trees (UCT) algorithm [13], first came to the attention of the wider A.I. community due to their startling success in some challenging adversarial planning domains, most notably Go [8]. In the intervening years, they have been successfully used to advance the state-of-the-art in several other domains, including other games [2,3,5], probabilistic single-agent planning [12], and planning in partially observable settings [26]. In all these domains, a balance needs to be struck between verifying the value of known actions versus evaluating the potential of under-explored actions, when under computational constraints. Algorithms like UCT approach this tradeoff in a theoretically sound way, by modeling the search process as a sequence of plays of a system of heirarchically organized multi-armed bandits [13]. In this paper, we investigate the potential of UCT in the domain of MaxSAT. In particular, we present a novel hybrid algorithm named UCTMAXSAT that combines aspects of both complete and incomplete solvers. Our algorithm maintains a search tree that is methodically explored. An SLS algorithm is used to estimate the value of the leaf nodes in this tree and guides its future expansion. We find that our algorithm outperforms the underlying SLS algorithms on a wide variety of benchmark instances.

2 Related Work

Several attempts have been made in the past to combine aspects of the complete and incomplete paradigms in an attempt to produce strong SAT solvers. In 1998, Mazure et al. studied the use of an SLS algorithm to inform the variable selection strategy in a complete solver [17], while in 2002, Habet et al. studied the possibility of augmenting SLS algorithms with resolution mechanisms commonly used by complete solvers [9]. There have been fewer studies investigating hybrid approaches to MaxSAT. The most notable work in this area is that of Kroc et al. who proposed a framework where one runs both an SLS solver (WalkSAT) and a complete solver (MiniSAT [6]) in parallel on the same problem instance [14]. During the run, WalkSAT is forbidden from flipping any variable that has already been fixed by MiniSAT, but is otherwise unconstrained. MiniSAT, with its systematic approach, directs the search towards promising regions in the

search space, where WalkSAT is then tasked with quickly configuring the free variables. The authors demonstrate impressive results across a wide range of challenging MaxSAT instances with this technique [14]. The approach, however, fails on instances that are easily proven unsatisfiable, as in these cases, the MiniSAT solver terminates too quickly to be of much assistance to WalkSAT. Our algorithm, on the other hand, can still be applied to such problems and only relies on a single-processor programming model.

Similarly, there have also been several prior attempts to bring the strength of MCTS methods to bear on various combinatorial problems. One of the first such attempts was that of Previti et al. who described a UCT inspired algorithm for solving SAT instances [20]. Since then, others have investigated the feasibility of using MCTS methods for solving mixed integer programming [22] and constraint programming problems [15]. To our knowledge, this is the first work to study the potential of UCT in the MaxSAT setting.

3 The UCTMAXSAT Algorithm

We now describe UCTMAXSAT, a derivative of the original UCT algorithm with specific enhancements for solving unweighted MaxSAT problems. UCTMAXSAT constructs a search tree where each node corresponds to a variable in the supplied Boolean CNF formula φ. Every non-terminal node is treated like a two-armed bandit. Playing the "left" arm (i.e., exploring the subtree rooted at the left child of a node) corresponds to setting a variable to false, while playing the "right" arm corresponds to setting the variable to true. If we denote the set of variables in φ by V, then a sequence of $|V|$ such plays corresponds to a complete truth assignment to the variables. UCTMAXSAT incrementally grows a search tree by iterating over the following steps.

Tree Descent: At a node s, the algorithm chooses whether to go left or right by computing an upper-confidence bound on each child node's estimated value as shown [1]:

$$\text{UCB1}(s_i) = V(s_i) + c \cdot \sqrt{\frac{\ln n(s)}{n(s_i)}} \tag{1}$$

Here, $i \in \{0, 1\}$ refers to the arm index, s_i is the child node reached by playing arm i, $V(s_i)$ is the estimated value of node s_i (and is discussed in greater detail below), $n(s)$ denotes the number of prior visits to the node s, and c is the exploration bias parameter. Starting at the root node, UCTMAXSAT repeatedly selects the arm with the higher UCB1 score (breaking ties arbitrarily) to descend down the currently maintained search tree, until a previously unexpanded node (i.e., one for which $n(s) = 1$) or a terminal node (one corresponding to a complete assignment) is reached. With each variable assignment (arm play), the original formula is appropriately simplified — for convenience, we will use φ' to refer to this simplified formula in contrast to the original formula φ.

Variable Installation: If we have assigned all variables at the conclusion of the tree descent phase, then we compute the fraction of satisfied clauses in the formula and skip ahead to the "Value Propagation" phase. Otherwise, we are left with a partial truth assignment and must decide how to proceed with the current iteration of the search. In particular, we must decide what variable to branch on at the current node. We investigated a number of different heuristics for making this choice, that are summarized in Table 1. Intuitively, these heuristics may be understood as follows:

- $A(k)$: prefers variables that appear in many clauses in φ' — assigning such variables first simplifies the formula faster.
- $S(k)$: prefers variables that appear roughly the same number of times with each sign in φ' — such variables may be better candidates for systematic exploration.
- $M(k)$: prefers variables that both appear in many clauses, and in a "balanced" way, in φ'.

The setting of the parameter k can be used to weight clauses according to their size: $k = -1$ weights short clauses more heavily, $k = 1$ weights long clauses more heavily, and $k = 0$ weights all clauses equally. In addition, we also experimented with other heuristics from the SAT literature including the MOM, Bohm's, Jeroslow-Wang [24] and Max-Falsified [17] heuristics. The results we present in this paper are from using the $A(0)$ heuristic in UCTMAXSAT, owing to its simplicity and robust performance across a range of problem categories.

Tree Growth: Once a variable v has been installed at the current node s, we create the left and right child of this node, that we denote by s_l and s_r respectively. The node s_l is reached by setting v to false at s, while s_r corresponds to setting v to true at s. Both nodes are added to the search tree — thus, each iteration of UCTMAXSAT increases the size of the tree by two nodes.

Value Estimation: Once the tree has been augmented with s_l and s_r, we estimate the value of the decision to install v at s, i.e., the value of further exploring the search subspace rooted at the node s. We do this by performing two SLS runs, one starting at s_l (i.e., with v set to false) and one starting at s_r (with v set to true). We note the similarity between this estimation approach and the idea of *random playouts* that are commonly used

Table 1. The variable selection heuristics used. Here, $C(l)$ denotes the set of clauses in the simplified formula φ' in which the literal l appears, $|c|$ denotes the length of clause c, and $k \in \{-1, 0, 1\}$.

$A(k)$:	$\operatorname{argmax}_l \left[\sum_{c \in C(l)} \lvert c \rvert^k + \sum_{c \in C(\neg l)} \lvert c \rvert^k \right]$
$S(k)$:	$\operatorname{argmin}_l \left\lvert \sum_{c \in C(l)} \lvert c \rvert^k - \sum_{c \in C(\neg l)} \lvert c \rvert^k \right\rvert$
$M(k)$:	$\operatorname{argmax}_l \left[\left(\sum_{c \in C(l)} \lvert c \rvert^k \right) \left(\sum_{c \in C(\neg l)} \lvert c \rvert^k \right) \right]$

in UCT-based game-playing programs [7,8]. However, within our context, a random playout would correspond to simply assigning all free variables arbitrarily; instead, we start with an assignment to the free variables that we then attempt to iteratively improve using a local search. Critically, this initial assignment of the free variables is not random — instead, we assign these variables the same truth values as in the best global solution found by our algorithm so far. Then, during the ensuing SLS runs, we prohibit the algorithm from flipping any variables whose values have been already fixed higher up in the search tree. This is similar to the approach of Kroc et al. [14]. The SLS algorithm is thus constrained to explore the area of the search space that it has been corralled into by the tree descent process. For each SLS run, we keep track of the best solution seen (i.e., the highest fraction of clauses that were satisfied), which we denote by m_l and m_r. We then estimate the value of the node s to be $m = (m_l^2 + m_r^2)/2$. The intuition for why we use m_l^2 and m_r^2, rather than m_l and m_r directly, is the following: on most MaxSAT instances, it is fairly easy to satisfy a large percentage of clauses using any local search procedure, before the search process plateaus. Using a non-linear function of m_l/m_r allows us to magnify the difference between two high-quality candidate solutions. We also experimented with other functional forms — higher degree polynomials and exponentials — but using the square empirically proved to be the best choice for a wide range of problems.

Value Propagation: Once the SLS runs complete (after some fixed number of variable flips), the estimated value of the node s (i.e., m) is propagated up the tree. The value of each node s' that is on the path from the root node to s is updated using a simple averaging operator, as in the original UCT implementation [13]. The visit count is also incremented for each such s', as shown:

$$n(s') \leftarrow n(s') + 1$$
$$V(s') \leftarrow V(s') + \frac{(m - V(s'))}{n(s')}$$

In this way, feedback from the SLS runs influences future expansions of the search tree. Since the values m_l and m_r represent a guaranteed lower bound on the proportion of satisfied clauses, we also experimented with using a max backup operator, that sets the value of a node to the maximum of the incumbent and incoming values. This approach has been found to outperform the traditional averaging operator on problems where the heuristic value estimate is reliable [21,22]. In our setting, however, we found the averaging operator to be superior to the max. A node-closure procedure (similar to that described by Previti et al. [20]) is also built into the value propagation phase. If the node s is a terminal node (i.e., it assigns to the last remaining free variable in φ'), then s is marked "closed" as further visits to this node cannot improve our result. These "closed" flags are propagated upwards through the tree so that tree descents always end at non-closed nodes.

The steps outlined above are repeated until the computational budget allotted to the solver is exhausted. The best solution discovered at any point during the algorithm's run is then reported as the final result. A pseudo-code specification of the algorithm is given in Appendix A.

4 Results

We implemented three flavors of the UCTMAXSAT algorithm that only differ in the SLS algorithm that was employed for estimating the values of leaf nodes: UCTMAXSAT$_W$ (uses WalkSAT [23]), UCTMAXSAT$_N$ (uses Novelty [18]), and UCTMAXSAT$_C$ (uses CCLS [16]). The implementations of WalkSAT and Novelty were adapted from the UBCSAT suite [28], while UCTMAXSAT$_C$ was built on an implementation of CCLS provided by its authors. In the remainder of this section, we present our findings from running experiments on problem instances drawn from the 2015 MaxSAT Evaluation[1] and large random 3SAT formulas. All tests were conducted on an Intel Xeon 3.5 GHz quad-core processor running Linux Ubuntu 14.04, with 16 GB of memory.

4.1 The Exploration-Exploitation Trade-Off

We motivated this work by observing that balancing greed and mobility in local search constituted an exploration-exploitation dilemma that MCTS methods like UCT are well-equipped to handle. The parameter c in equation (1) controls this trade-off — smaller values of c correspond to a greater amount of greed and lead to UCTMAXSAT building highly asymmetric trees that are much deeper in more promising regions of the search space. Higher values of c encourage greater exploration and lead to "bushier" trees. The left panel of Fig. 1 highlights the impact of this parameter's setting on the performance of UCTMAXSAT$_W$. The plot shows the average quality of the solutions (over 50 independent runs of 300 s each) found by the algorithm for the `maxcut-140-630-0.7-1` instance, for various values of c. We note that there is a "sweet spot" for the setting of c that produces the best results, a fact we have confirmed for many different problem instances, as well as for UCTMAXSAT$_N$. This indicates that UCTMAXSAT is indeed effective at reconciling the greed-mobility tension in local search and does so in a sophisticated way — Fig. 1 demonstrates that neither pure-exploitation nor pure-exploration strategies are as successful as a blend of the two.

4.2 Allocating the Computational Budget

In this section, we address the following question: *How much of a fixed computational budget should we allocate to the tree-building and SLS components of* UCTMAXSAT? Using a fixed budget of 300 s of runtime, we investigated various allocation strategies, measured in flips per SLS run. For example, using 1000 flips per SLS run would permit us to run more iterations of UCTMAXSAT (therefore

[1] http://www.maxsat.udl.cat/15/, accessed on Feb. 21, 2016.

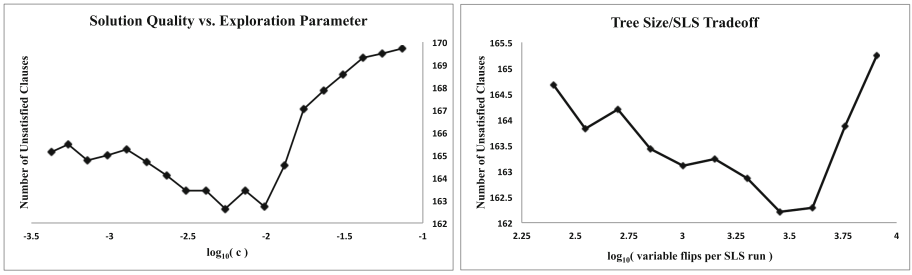

Fig. 1. Left: Effect of the exploration bias parameter c on the quality of the solutions found by UCTMAXSAT$_W$. Right: Comparison of different allocations of the computational budget to the tree-building and SLS components of UCTMAXSAT$_W$.

producing a larger search tree) than using 2000 flips per SLS run (which would require relatively more of the runtime budget to be allocated to the SLS runs). The right panel of Fig. 1 shows the effect of different allocations on the performance of UCTMAXSAT$_W$ on the `maxcut-140-630-0.7-1` problem instance. Each data point represents the average number of unsatisfied clauses found over 50 independent runs. At the left extreme of the plot, we are choosing very short SLS runs that permit us to build larger search trees; at the right extreme, we are performing long SLS runs and building smaller trees. The shape of the curve indicates an optimal allocation that is far removed from both extremes — at this point, there is a synergy between the tree-building and SLS components, and the combination of the two techniques outperforms either component by itself. We found this too to be a robust trend, that persisted independent of the problem instance used, the SLS algorithm or the setting of c. In all the experiments that follow, we use an allocation of 2000 flips per SLS run, which we empirically found to be the best general-purpose setting.

4.3 UCTMAXSAT Against Baseline SLS Algorithms

We now present a head-to-head comparison of two baseline SLS algorithms, WalkSAT and Novelty, to their MCTS-enhanced counterparts. Since the problem instances we use in our experiments vary widely in size, we report our results using the following *normalized difference in solution quality* metric:

$$\sigma(s_1, s_2) = \frac{s_1 - s_2}{\max\{s_1, s_2\}}$$

Here, s_1 and s_2 represent the quality of the best solutions (as measured by the number of unsatisfied clauses) found by the two algorithms being compared. We follow the convention of always using s_1 for the solution found by the SLS algorithm and s_2 for the solution found by UCTMAXSAT. Thus, positive values of σ indicate instances where UCTMAXSAT outperforms its SLS competition, while negative values of σ indicate instances where the situation is reversed. Dividing the difference in solution quality by the max term ensures that σ stays bounded within $[-1, +1]$. However, note that the measure is non-linear: for example, a

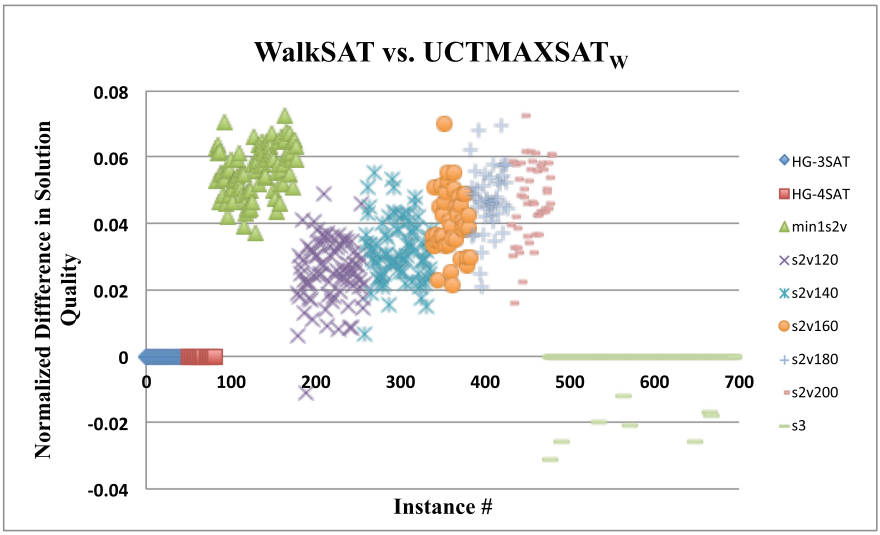

Fig. 2. A comparison of the solutions found by WalkSAT to those found by UCTMAXSAT$_W$ on random instances from the 2015 MaxSAT Evaluation.

score of $\sigma = 0.5$ indicates that UCTMAXSAT found a solution that violates only half as many clauses as the best solution found by its SLS counterpart. A score of $\sigma = 0.9$ indicates that the UCTMAXSAT solution is superior by an order-of-magnitude while scores close to 1.0 denote cases where UCTMAXSAT improved upon the SLS solution by several orders-of-magnitude.

Figures 2 and 3 present our results on the complete set of random unweighted MaxSAT problem instances from the 2015 MaxSAT Evaluation. Each problem instance corresponds to a data point, for which we report the σ measure (as defined earlier). The instances are color-coded according to the problem family from which they are drawn. The critical parameters for the algorithms (noise settings for WalkSAT and Novelty, c for UCTMAXSAT) were tuned on a per-instance basis and we compare the results obtained using the best parameter settings in each case. All results were collected based on a single run of the appropriate algorithm for 300 s — the same time control that is used in the MaxSAT Evaluation. As can be seen from these results, the MCTS tree improves the performance of both vanilla WalkSAT and Novelty on a broad range of instances. In particular, it improves on the best WalkSAT solution (i.e., $\sigma > 0$) on 393 instances and improves on the best Novelty solution on 364 instances (out of 695). There are 287 instances on which WalkSAT and UCTMAXSAT$_W$ tie (i.e., $\sigma = 0$); however, on 286 of these instances, WalkSAT found the optimal solution unaided, thus leaving no room for improvement. There were only 15 instances where UCTMAXSAT$_W$ was outperformed by WalkSAT (i.e., where WalkSAT found a solution with at least one fewer unsatisfied clause than UCTMAXSAT$_W$). Similarly, there was no room for improvement on 293 of the 309 instances where Novelty and UCTMAXSAT$_N$ tie; overall, Novelty outperformed UCTMAXSAT$_N$ on only 22 (out of 695) instances. This performance breakdown on the random instances

Table 2. A breakdown of the difference in performance between the specified SLS algorithm and its MCTS-enhanced counterpart on the random and crafted instances from the 2015 MaxSAT evaluation. The $\sigma > 0$ column presents the number of instances on which the MCTS algorithm outperformed SLS, the $\sigma < 0$ column indicates the number of instances on which the situation was reversed, and the $\sigma = 0$ column gives the number of instances for which both methods found solutions of equal quality.

	Random instances			Crafted instances		
	$\sigma > 0$	$\sigma = 0$	$\sigma < 0$	$\sigma > 0$	$\sigma = 0$	$\sigma < 0$
WalkSAT	393 (56.5 %)	287 (41.3 %)	15 (2.2 %)	336 (83.6 %)	63 (15.7 %)	3 (0.7 %)
Novelty	364 (52.4 %)	309 (44.5 %)	22 (3.1 %)	311 (77.4 %)	76 (18.9 %)	15 (3.7 %)

is summarized in the left section of Table 2. Overall, UCTMAXSAT$_W$ improves on WalkSAT by larger margins on average than UCTMAXSAT$_N$ improves on Novelty — this is unsurprising given that Novelty is a superior SLS algorithm and is thus more challenging to surpass.

Figures 4 and 5 present our results on the complete set of crafted unweighted MaxSAT instances from the 2015 MaxSAT evaluation. Overall, the trend is similar to what was observed on the random instances: once again, UCTMAXSAT outperforms WalkSAT and Novelty across a wide variety of benchmark domains, it tends to improve WalkSAT more so than Novelty, and in the overwhelming majority of cases when the algorithms tie, there is no room for improvement since both algorithms find the optimal solution. Interestingly, looking across the left and right partitions of Table 2, we see that the impact of MCTS on the underlying SLS algorithm is more pronounced on crafted instances than

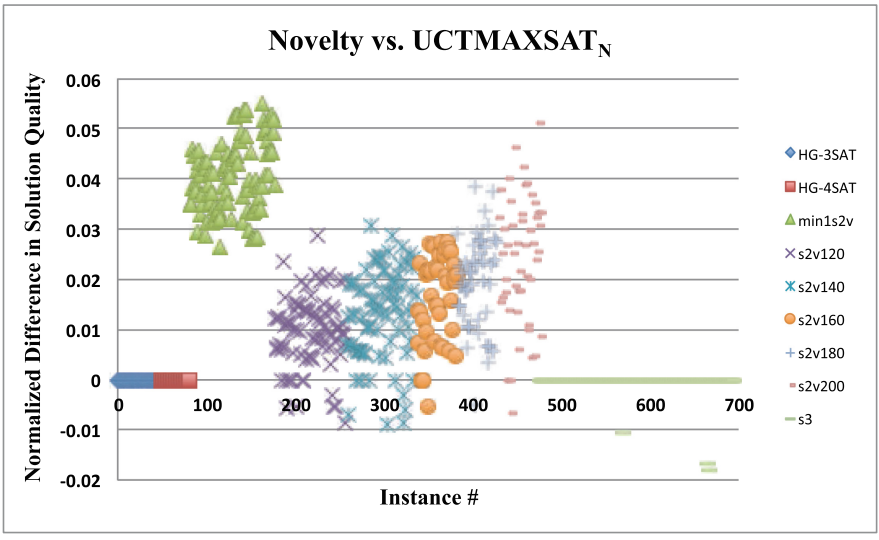

Fig. 3. A comparison of the solutions found by Novelty to those found by UCTMAXSAT$_N$ on random instances from the 2015 MaxSAT Evaluation.

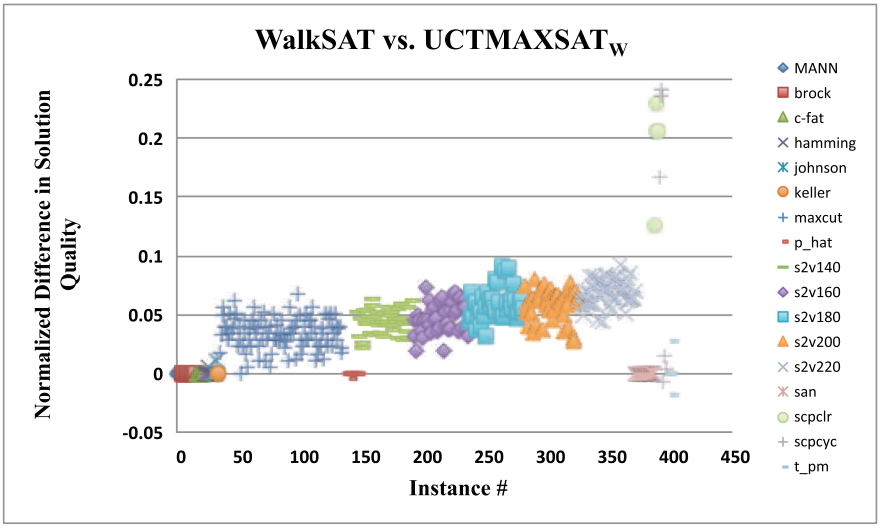

Fig. 4. A comparison of the solutions found by WalkSAT to those found by UCTMAXSAT$_W$ on crafted instances from the 2015 MaxSAT Evaluation.

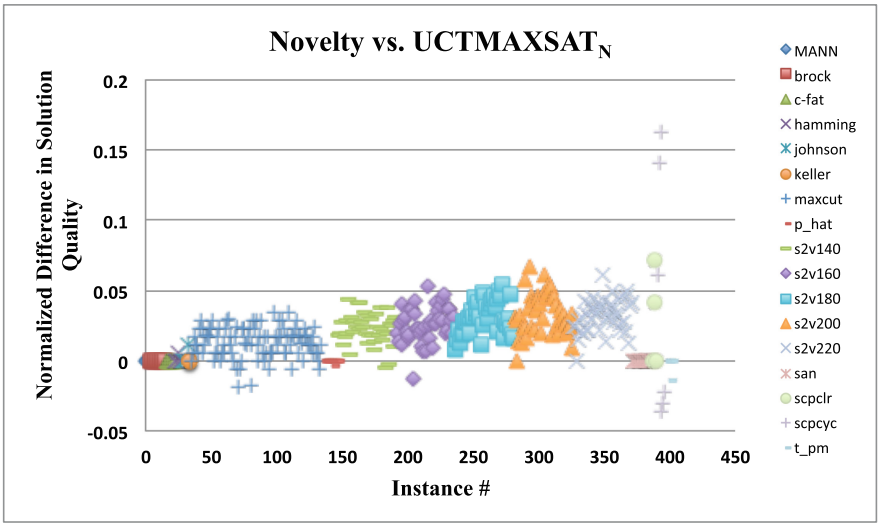

Fig. 5. A comparison of the solutions found by Novelty to those found by UCTMAXSAT$_N$ on crafted instances from the 2015 MaxSAT Evaluation.

on random instances. This is consistent with results that are well-known in the SAT community, namely that systematic approaches fare better in domains with more structure, whereas local search approaches perform better on random formulas [11].

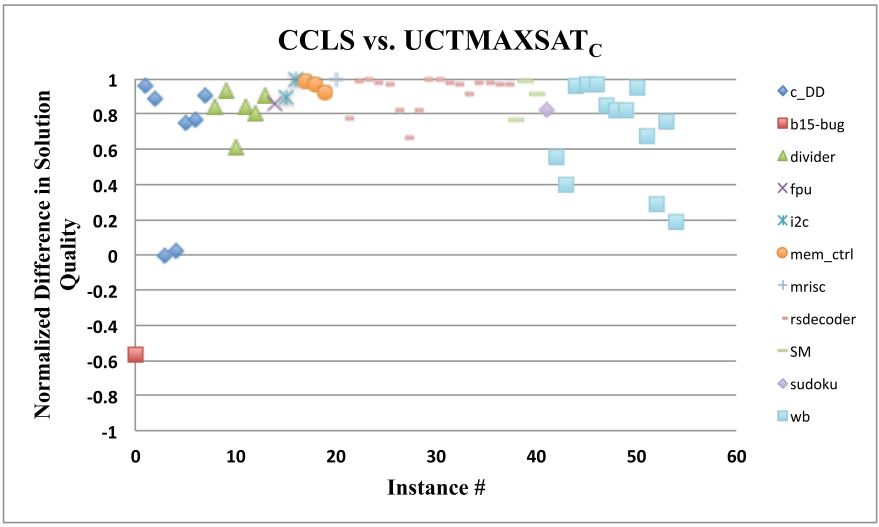

Fig. 6. A comparison of the solutions found by CCLS to those found by UCTMAXSAT$_C$ on industrial instances from the 2015 MaxSAT Evaluation.eps

4.4 Uctmaxsat$_C$ Against CCLS

In this section, we consider the benefits of augmenting CCLS [16], the top-performing incomplete MaxSAT solver in the Random and Crafted categories from the 2015 MaxSAT Evaluation[2], with a UCT tree search. We focused our initial attention on instances from the Industrial track of the MaxSAT Evaluation, having observed that CCLS fared poorly in this category. Using the same experimental set-up as in Sect. 4.3 (a single run of 300 s using the best parameter setting for each algorithm), we obtained the results shown in Fig. 6. As can be seen, UCTMAXSAT$_C$ handsomely outperforms CCLS on all but three instances. Indeed, the margin of improvement on many of the instances is by several orders of magnitude. A particularly stark example is the `mrisc_mem2wire` instance (the blue-grey '+' data point in the plot) where the best solution discovered by CCLS violates 397,032 clauses. In contrast, the best solution found by UCTMAXSAT$_C$ only violates 386 clauses. We saw similar margins of improvement on many of the other instances as well.

We also conducted experiments comparing the performance of UCTMAXSAT$_C$ to CCLS on random 3SAT formulas. Given that CCLS found the optimal solution for every single random instance that was used in the 2015 MaxSAT Evaluation, we chose to work with more challenging instances instead. We generated random 3SAT formulas with 700, 900, and 1100 variables, while maintaining the same clause-to-variable ratios as those used in the Evaluation. Figure 7 summarizes our results over a set of 105 formulas. CCLS outperforms UCTMAXSAT$_C$ on all

[2] http://www.maxsat.udl.cat/15/results-incomplete/index.html, retrieved on Feb. 21, 2016.

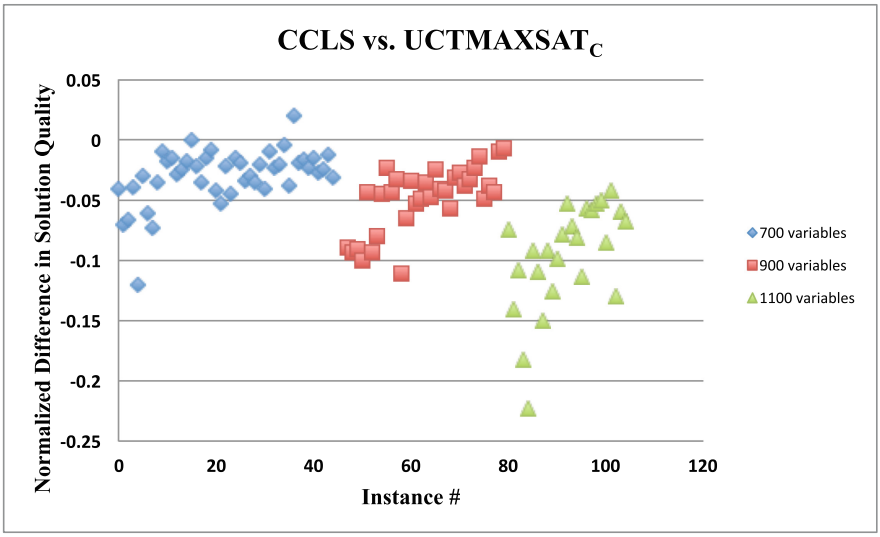

Fig. 7. A comparison of the solutions found by CCLS to those found by UCTMAXSAT$_C$ on large random 3SAT instances.

but two instances in this setting. It appears that the systematic tree search component really improves the performance of the underlying SLS solver on structured instances, but offers little benefit on random instances.

4.5 Summary

In summary, overlaying an MCTS search tree on top of existing SLS solvers has the effect of making them more "well-rounded" — the underlying solvers are improved in their traditional areas of weakness. For example, it is well-known that solvers like WalkSAT perform poorly on overconstrained random MaxSAT instances [29]. UCTMAXSAT$_W$ is able to boost the performance of native WalkSAT in this domain. Similarly, while CCLS is a top-performing solver on random MaxSAT instances, industrial problems are its Achilles Heel. In this case, UCTMAXSAT$_C$ substantially improves the performance of CCLS on industrial instances.

5 Future Work

This paper presented some exploratory work that demonstrated how techniques borrowed from the MCTS framework could improve incomplete solution methods for MaxSAT problems. We offer a few possible directions for future work.

– There is a large region of the algorithm parameter space that is ripe for exploration — for example, one could consider alternatives to the UCB1 bandit algorithm such as ϵ_n-GREEDY [1], or test other value back up operators (like a soft-max).

– In this work, we started with established SLS algorithms and attempted to "boost" their performance by combining them with tree search. However, many in the game-playing community have observed that certain random playout strategies work better in conjunction with UCT than others [8,25]. It would be interesting to investigate whether one could design novel SLS algorithms that specifically dovetail well with UCTMAXSAT (but are not necessarily effective as stand-alone solvers).
– Finally, it would be interesting to look into how one could extend the UCTMAXSAT algorithm to handle partial and weighted MaxSAT instances, and how the approach compares to the current state-of-the-art solvers.

6 Conclusions

In this paper, we presented a novel Monte-Carlo Tree Search based approach for solving MaxSAT problems. Our algorithm, UCTMAXSAT, elegantly combines aspects of systematic and local search. Our experiments with problem instances from the 2015 MaxSAT Evaluation show that UCTMAXSAT significantly improves the performance of baseline incomplete algorithms like WalkSAT and Novelty on almost every problem domain. Our algorithm also improves the performance of CCLS, a state-of-the-art incomplete solver, on industrial instances. In the future, we plan to expand on this work to extract better performance across a wider range of MaxSAT problem instances.

Acknowledgements. We are grateful to Shaowei Cai and his collaborators for sharing their implementation of the CCLS algorithm with us. We would also like to thank the anonymous reviewers for their useful comments and feedback.

A Appendix: The Uctmaxsat Algorithm

Algorithm 1. UCTMAXSAT main loop

```
1:
2: globals:
3:   φ: the input (non-empty) CNF formula
4:   ρ_best ← a random complete assignment          ▷ best solution found so far
5:
6: function UCTMAXSAT
7:     root ← CREATENODE(∅)                        ▷ pointer to root node of search tree
8:     repeat
9:         DESCEND(root, φ, ∅)
10:    until time limit is met
11:    return ρ_best
12: end function
13:
```

Algorithm 2. UCTMAXSAT helper routines

```
1:
2: function CreateNode(ρ)
3:     node ← new node
4:     node.visits ← 1
5:     ρ′ ← set of literals in ρ_best that do not contradict a literal in ρ
6:     Run SLS algorithm using ρ∪ρ′ as starting configuration, never flipping variables
       in ρ
7:     f ← fraction of satisfied clauses in best assignment found by SLS run
8:     node.value ← f^2
9:     Update ρ_best if the SLS run uncovered a new best solution
10:    return node
11: end function
12:
13: function Descend(node, φ′, ρ)        ▷ ρ is the current partial assignment (set of
    literals)
14:    if ρ is a complete assignment then
15:        Mark node as closed
16:        f ← fraction of clauses in φ that are satisfied by ρ
17:        return f^2
18:    else if node.visits = 1 then                    ▷ a previously unexpanded node
19:        node.variable ← A(0)                                    ▷ (see Table 1)
20:        node.left ← CreateNode(ρ ∪ {¬node.variable})
21:        node.right ← CreateNode(ρ ∪ {node.variable})
22:        r ← (node.left.value + node.right.value)/2
23:    else
24:        child, φ′, ρ ← SelectChild(node, φ′, ρ)
25:        r ← Descend(child, φ′, ρ)
26:    end if
27:    node.visits ← node.visits + 1
28:    node.value ← node.value + (r − node.value)/node.visits
29:    Mark node as closed if node.left and node.right are both closed
30:    return v
31: end function
32:
33: function SelectChild(node, φ′, ρ)
34:    if either node.left or node.right is closed then
35:        ch ← the still open child
36:    else
37:        ch ← child that maximizes Ucb1 score                    ▷ (see Eq. 1)
38:    end if
39:    if ch is the left child of node then
40:        φ′ ← φ′ simplified by assigning node.variable to False
41:        ρ ← ρ ∪ {¬node.variable}
42:    else
43:        φ′ ← φ′ simplified by assigning node.variable to True
44:        ρ ← ρ ∪ {node.variable}
45:    end if
46:    return (ch, φ′, ρ)
47: end function
48:
```

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Mach. Learn. **47**(2–3), 235–256 (2002). doi:10.1023/A:1013689704352
2. Balla, R.K., Fern, A.: UCT for tactical assault planning in real-time strategy games. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence, IJCAI 2009, pp. 40–45. Morgan Kaufmann Publishers Inc., San Francisco (2009). http://dl.acm.org/citation.cfm?id=1661445.1661453
3. Branavan, S.R.K., Silver, D., Barzilay, R.: Learning to win by reading manuals in a Monte-Carlo framework. J. Artif. Intell. Res. (JAIR) **43**, 661–704 (2012). doi:10.1613/jair.3484
4. Cai, S., Su, K.: Local search for boolean satisfiability with configuration checking and subscore. Artif. Intell. **204**, 75–98 (2013). doi:10.1016/j.artint.2013.09.001
5. Ciancarini, P., Favini, G.P.: Monte carlo tree search in Kriegspiel. Artif. Intell. **174**(11), 670–684 (2010). http://www.sciencedirect.com/science/article/pii/S0004370210000536
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the 23rd National Conference on Artificial Intelligence, vol. 1, AAAI 2008, pp. 259–264. AAAI Press (2008). http://dl.acm.org/citation.cfm?id=1619995.1620038
8. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th International Conference on Machine Learning, ICML 2007, NY, USA, pp. 273–280 (2007). http://doi.acm.org/10.1145/1273496.1273531
9. Habet, D., Li, C.-M., Devendeville, L., Vasquez, M.: A hybrid approach for SAT. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 172–184. Springer, Heidelberg (2002)
10. Hoos, H., Sttzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Publishers Inc., San Francisco (2004)
11. H. Hoos, H., Stützle, T.: Systematic vs. local search for SAT. In: Burgard, W., Christaller, T., Cremers, A.B. (eds.) KI 1999. LNCS (LNAI), vol. 1701, pp. 289–293. Springer, Heidelberg (1999)
12. Keller, T., Eyerich, P.: PROST: probabilistic planning based on UCT. In: McCluskey, L., Williams, B., Silva, J.R., Bonet, B. (eds.) Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, 25–29 June 2012. AAAI (2012). http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4715
13. 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)
14. Kroc, L., Sabharwal, A., Gomes, C.P., Selman, B.: Integrating systematic and local search paradigms: a new strategy for maxsat. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, 11–17 July 2009, pp. 544–551 (2009) http://ijcai.org/papers09/Papers/IJCAI09-097.pdf
15. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 464–480. Springer, Heidelberg (2013)

16. Luo, C., Cai, S., Wu, W., Jie, Z., Su, K.: CCLS: an efficient local search algorithm for weighted maximum satisfiability. IEEE Trans. Comput. **64**(7), 1830–1843 (2015)
17. Mazure, B., Saïs, L., Grégoire, E.: Boosting complete techniques thanks to local search methods. Ann. Math. Artif. Intell. **22**(3–4), 319–331 (1998). doi:10.1023/A:1018999721141
18. McAllester, D., Selman, B., Kautz, H.: Evidence for invariants in local search. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence, AAAI 1997/IAAI 1997, pp. 321–326. AAAI Press (1997). http://dl.acm.org/citation.cfm?id=1867406.1867456
19. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided maxsat solving: a survey and assessment. Constraints **18**(4), 478–534 (2013)
20. 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)
21. Ramanujan, R., Selman, B.: Trade-offs in sampling-based adversarial planning. In: Bacchus, F., Domshlak, C., Edelkamp, S., Helmert, M. (eds.) Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany, 11–16 June 2011. AAAI (2011). http://aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/view/2708
22. Sabharwal, A., Samulowitz, H., Reddy, C.: Guiding combinatorial optimization with UCT. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 356–361. Springer, Heidelberg (2012)
23. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Hayes-Roth, B., Korf, R.E. (eds.) Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, 31 July - 4 August 1994, vol. 1, pp. 337–343. AAAI Press/The MIT Press (1994). http://www.aaai.org/Library/AAAI/1994/aaai94-051.php
24. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) EPIA 1999. LNCS (LNAI), vol. 1695, pp. 62–74. Springer, Heidelberg (1999)
25. Silver, D., Tesauro, G.: Monte-carlo simulation balancing. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, NY, USA, pp. 945–952 (2009). http://doi.acm.org/10.1145/1553374.1553495
26. Silver, D., Veness, J.: Monte-Carlo planning in large POMDPS. In: Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A. (eds.) Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010, Proceedings of a Meeting Held 6–9 December 2010, Vancouver, British Columbia, Canada, pp. 2164–2172. Curran Associates, Inc. (2010). http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps
27. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning, 1st edn. MIT Press, Cambridge (1998)

28. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10–13 May 2004, Vancouver, BC, Canada, Online Proceedings (2004). http://www.satisfiability.org/SAT04/programme/23.pdf
29. Zhang, W., Rangan, A., Looks, M.: Backbone guided local search for maximum satisfiability. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI 2003, pp. 1179–1184. Morgan Kaufmann Publishers Inc., San Francisco (2003)