

Recursive Weight Heuristic for Random k -Sat

Dimitrios Athanasiou, Marco Alvarez Fernandez

Delft University of Technology, Delft, The Netherlands

Abstract. Look-ahead SAT solvers are the most suitable to solve random k -SAT formulas. Based on the DPLL framework, they use branching heuristics. We generalise the application of the efficient recursive weight heuristic for k -SAT problems. The heuristic and an efficient implementation are presented in this paper. Parameter tuning is performed providing experimental results. Further experiments demonstrate the improvement of the performance of the heuristic extending *march*, the strongest complete solver on random k -SAT formulae as shown by the results of SAT 2009 competition.

1 Introduction

Applications of Satisfiability (SAT) solving range from hardware and software verification to logic synthesis, automatic test generation, artificial intelligence and concrete combinatorial problems[4]. There are two general approaches in SAT solving, the "conflict-driven solver", suitable for verification problems, and the "look-ahead" solver, suitable for difficult k -SAT problems[6]. Even though k -SAT problem has been exhaustively studied and powerful SAT solvers have been implemented that are able to solve huge SAT instances efficiently, random k -SAT formulas are still hard to solve.

In order to do so, look-ahead SAT solvers are based on the DPLL framework[1]. According to the DPLL framework, Look-ahead SAT solvers construct the search tree of a k -SAT instance by choosing on which variable to branch on and which value of that variable will be explored first. This process is accelerated significantly by using heuristics. There are two kinds of heuristics in the context of a SAT solver: decision heuristics, which decide which will be the next variable to branch on and direction heuristics, which decide the order with which the possible values of the branched variable will be explored.

In [7] a decision heuristic called recursive weight heuristic was introduced and applied on 3-SAT instances. In this paper, a generalization of that heuristic is presented making it applicable in any k -SAT instance. An efficient implementation of the heuristic is analysed. The heuristic was implemented and integrated in *march*¹, the strongest complete solver for random k -sat formulas as shown by the results of SAT 2009 competition². The heuristic has some parameters that highly affect its performance. An empirical exploration of those parameters is performed and

¹ See http://www.st.ewi.tudelft.nl/sat/march_dl.php

² See <http://www.satcompetition.org>

the optimal values are reported in order to facilitate the tuning of the parameters and the extensive use of the recursive weight heuristic. Finally, experiments are conducted in order to demonstrate the improvement of the application of the heuristic compared to the latest release of march, *march_hi*.

In section 2, the heuristic and the basic concepts behind it are explained. In section 3, the implementation is analysed. In section 4, the parameters of the heuristic are tuned according to empirical results. In section 5, the improvement of the performance is demonstrated with experimental tests. Section 6 summarizes our work and the derived conclusions.

2 Recursive Weight Heuristic

In [7], a model that generalized existing work on look-ahead branching heuristics was introduced. The model is referred to as the recursive weight heuristic and it was applied on 3-SAT. The logic behind the heuristic is that $h_i(\neg x)$ represents the tendency of a variable x to be falsified after i iterations. When $h_i(x)$ is calculated (the tendency of x to be assigned true), for each clause where x participates, the heuristic values of the negation of all the other literals in the clause are combined via multiplication. The more the tendency of all other literals to become falsified the more the tendency of x to be assigned true should be in order to satisfy the clause and consecutively, the entire formula. As far as clauses are concerned, the weight of a clause represents the tendency of that clause to become falsified. Thus, the weight of a clause $(x \vee y \vee z)$ is:

$$w(x \vee y \vee z) = h_i(\neg x) \cdot h_i(\neg y) \cdot h_i(\neg z) \quad (1)$$

We generalize that model for k -SAT. Let $VAR(F)$ refer to the set of variables in formula F and $n = |VAR(F)|$. First, the heuristic values h_0 are initialized to 1:

$$h_0(x) = h_0(\neg x) = 1 \quad (2)$$

Next, the heuristic is calculated recursively until the desired accuracy - number of iterations - is reached. At each step, the heuristic values $h_i(x)$ are scaled dividing with μ_i , which represents the average heuristic value of iteration i :

$$\mu_i = \frac{1}{2n} \sum_{x \in VAR(F)} (h_i(x) + h_i(\neg x)) \quad (3)$$

Finally, in each iteration, the heuristic values h_{i+1} are computed for each variable in F in the following way: for every clause all the other literals y of that clause get weight $h_i(\neg y)/\mu_i$. The product of the weights of the literals of each clause is calculated. Last, the derived products of each clause are summed.

$$h_{i+1}(x_j) = \sum_{(y_1 \vee y_2 \vee \dots \vee y_{k-1} \vee x_j)} \left(\frac{\prod_{w=0}^{k-1} h_i(\neg y_w)}{\mu_i^{k-1}} \right)$$

$$\begin{aligned}
& + \gamma \sum_{(y_1 \vee y_2 \vee \dots \vee y_{k-2} \vee x_j)} \left(\frac{\prod_{w=0}^{k-2} h_i(\neg y_w)}{\mu_i^{k-2}} \right) \\
& + \gamma^2 \sum_{(y_1 \vee y_2 \vee \dots \vee y_{k-3} \vee x_j)} \left(\frac{\prod_{w=0}^{k-3} h_i(\neg y_w)}{\mu_i^{k-3}} \right) \\
& + \dots \\
& + \gamma^{k-2} \sum_{y \vee x_j} \frac{h_i(\neg y)}{\mu_i}
\end{aligned}$$

The length of the clauses vary as the process advances and clauses are reduced. In the heuristic, clauses with different length have different importance. Clauses of smaller length pose bigger constraints and therefore it is preferable to branch on variables that have many occurrences in such clauses - this is the idea on which Mom's Heuristic was based [2]. In the recursive weight heuristic this fact is modelled with the use of γ , in the sense that the latter represents the heuristic value of falsified literals. For example, when a clause:

$$(x \vee y \vee w \vee z) \tag{4}$$

is reduced to the clause:

$$(x \vee y) \tag{5}$$

it implies that the variables w and z have been set to false and therefore the corresponding literals in the clause were falsified. This fact means that there are now less opportunities for this clause to be satisfied and thus, its literals should have a higher priority to be selected in the next branching step. The average value of $\frac{h_i(x)}{\mu_i}$ is 1 and therefore γ should be greater than 1 in order for falsified literals to have greater impact on the calculation of the heuristic than non-falsified literals. The optimal levels of the value of γ are experimentally explored in subsection 4.1.

3 Implementation

The algorithm that implements the recursive weight heuristic focuses on reducing the cost of the calculation. Stronger heuristics can simplify more a problem, yet the time spent for their calculation may become more than it would be spent for solving a less simplified problem. This is a trade-off that is always involved in constructing efficient heuristic algorithms.

A naive approach of the calculation of the heuristic value for each variable is discussed first. For each variable x iterate through each clause, iterate through the literals of each clause and calculate the heuristic value $h_i(x)$. This approach results in a complexity of $O(n^{k-1})$ because the variables are traversed repeatedly for each distinct clause length. The redundancy here is that the product of the heuristic values of the literals of each clause are calculated as many times as the number of the variables. Thus, eliminating this redundancy can lead to a much more efficient algorithm.

Instead of iterating through each variable, we iterate through each clause and calculate the product of all its literals. Then for each literal l we calculate its heuristic value by dividing the product with the heuristic value of the falsification of l . That way we iterate through the variables twice to achieve a complexity of $(O(n^2))$. In [7] the recursive weight heuristic is applied only on 3-SAT. Therefore the naive approach was followed since the complexity for $k = 3$ is also $(O(n^2))$. The generalised algorithm of the recursive weight heuristic for k -SAT is shown in Algorithm 1.

The algorithm begins with the initialization of the $h_0(x)$ and $h_0(\neg x)$ for all variables to 1. The *sum* of the heuristics is therefore $2n$ initially. The current recursion round is indicated by i which is initially 1. A loop follows with a repetition for every recursion round. The desired number of recursions is stored in variable *accuracy*. In each round i , the average μ_{i-1} is calculated by dividing the *sum* by $2n$. Next, in order to avoid redundant calculations, we calculate the constant part of the calculation for each clause of distinct length (binary, ternary, etc.). An array $\gamma[j]$ with $j = 2, 3, \dots, k$ and $\gamma[j]$ being the γ to the power of $k - j$ (the number of falsified literals in the clause) is initially calculated so that whenever the factor is needed for a clause of length j , the value can be retrieved directly. After retrieving the appropriate γ , it is divided by μ_i to the power of the length of the clause minus one. In order to reduce the number of calculations, this constant factor for each distinct clause length is stored in an array *ConstantFactor*[j], again so that j indicates the length of the clause. To continue, for every clause in the formula the product of the negation of its unassigned literals is calculated for a cost of calculations equal to the number of unassigned literals. The literals of the clauses are traversed a second time in order to calculate their heuristic values. In every occurrence of the literal x the heuristic is incremented by adding the product of the clause divided by the negation of the heuristic of x because it was included in the first place when the product was calculated. At this point, the heuristic values are filtered so that they do not exceed a certain upper/lower bound which is different for k -SAT problems with different k and has to be found empirically (see subsection 4.2). Finally, *sum* is updated with the summation of all the heuristic values of all variables of the current recursion so that it can be used for the calculation of μ_i in the next round.

4 Parameter Tuning

There are two parameters that have to be tuned in order to optimize the performance of the heuristic. The first is the γ and the other is the upper bounds used for the filtering of the heuristic values.

The tests that follow in this section have been conducted using March, a SAT Solver in which the recursive weight heuristic was implemented. The comparison of the performance is based on the count of nodes that were searched during the execution of 25 5-SAT unsatisfiable instances and 25 7-SAT unsatisfiable instances. Since the count of nodes is not affected by the hardware, it is not relevant to publish hardware details. Instances of the SAT problems were randomly generated using the ratios

Algorithm 1 CALCULATEWEIGHTS(formula \mathcal{F})

```
1: for  $x \in VAR(F)$  do
2:   for  $i = 1$  to accuracy do
3:      $h_i(x) := 0$ 
4:   end for
5: end for
6: for  $x \in VAR(F)$  do
7:    $h_0(x) := 1$ 
8:    $h_0(\neg x) := 1$ 
9: end for
10:  $sum := 2n$ 
11:  $i := 1$ 
12: for  $i = 1$  to accuracy do
13:    $\mu_{i-1} := sum/(2n)$ 
14:   for  $j=2, 3, \dots, k$  do
15:      $ConstantFactor[j] := \gamma^{k-j}/(\mu_i)^{j-1}$ 
16:   end for
17:   for  $C \in F$  do
18:      $product := ConstantFactor[length(C)]$ 
19:     for  $l \in C$  do
20:        $product := product * h_{i-1}(\neg l)$ 
21:     end for
22:     for  $l \in C$  do
23:        $h_i(l) := h_i(l) + product/h_{i-1}(\neg l)$ 
24:     end for
25:   end for
26:    $sum := 0$ 
27:   for  $x \in VAR(F)$  do
28:     if  $h_i(x) > upperBound$  then
29:        $h_i(x) := upperBound$ 
30:     end if
31:     if  $h_i(\neg x) > upperBound$  then
32:        $h_i(\neg x) := upperBound$ 
33:     end if
34:     if  $h_i(x) < lowerBound$  then
35:        $h_i(x) := lowerBound$ 
36:     end if
37:     if  $h_i(\neg x) < lowerBound$  then
38:        $h_i(\neg x) := lowerBound$ 
39:     end if
40:      $sum := sum + h_i(x) + h_i(\neg x)$ 
41:   end for
42: end for
```

of clauses to variables that produce the hardest problems (21.3 for 5-SAT and 89 for 7-SAT [3]). 5-SAT instances had 50 variables and 1065 clauses and 7-SAT instances had 40 variables and 3560 clauses. March was used in order to find whether tests were satisfiable or unsatisfiable and as it was mentioned only unsatisfiable instances were used because they are more stable than satisfiable problems. For the latter the factor of luck is important and can produce very different results for small changes to the parameters. Finally, the accuracy level that was used for the tests was 3.

4.1 Tuning of γ

The constant γ represents the heuristic value of a falsified literal. The results of the tests for 5-SAT instances can be seen in Figure 1. A radical improvement is observed as γ becomes bigger than 1. The improvement continues as γ increases until it reaches the value of 6. After that point the performance gets worse as γ gets higher and higher values. Further tests were run for the values of γ around 5. The results can be seen in Figure 2. The performance is optimized for the value of 5.1.

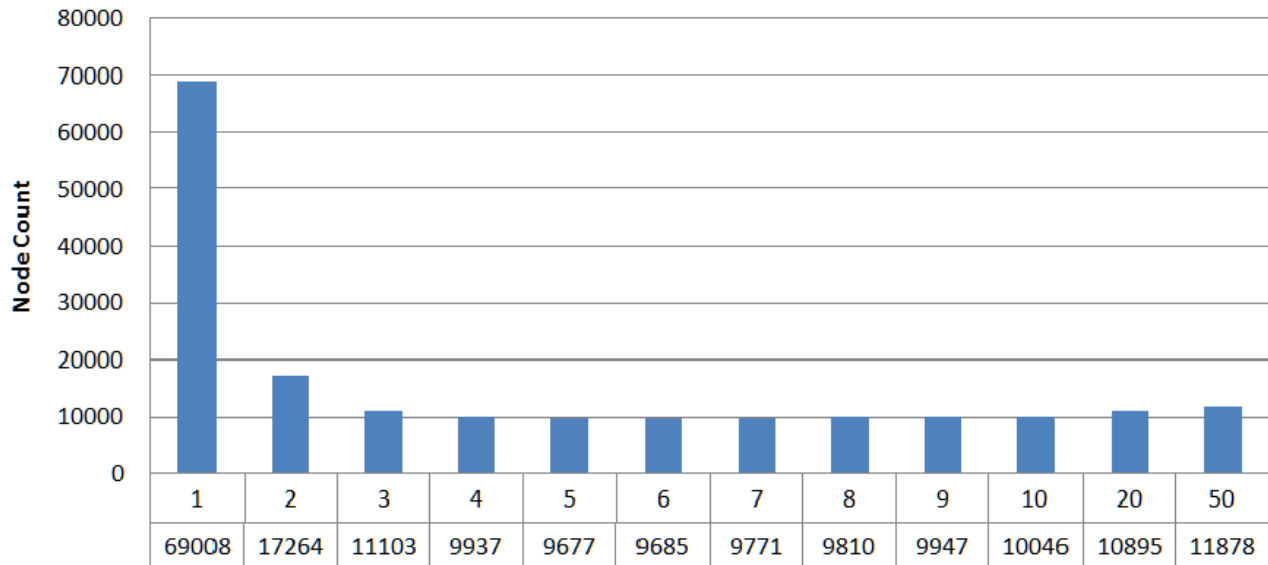


Fig. 1. Average number of nodes searched for 5-SAT instances against the value of γ

The results of the tests for 7-SAT instances are very similar to the ones that correspond to 5-SAT instances. The behaviour of the performance indicates that the optimal value of γ is around 5. Once again, further

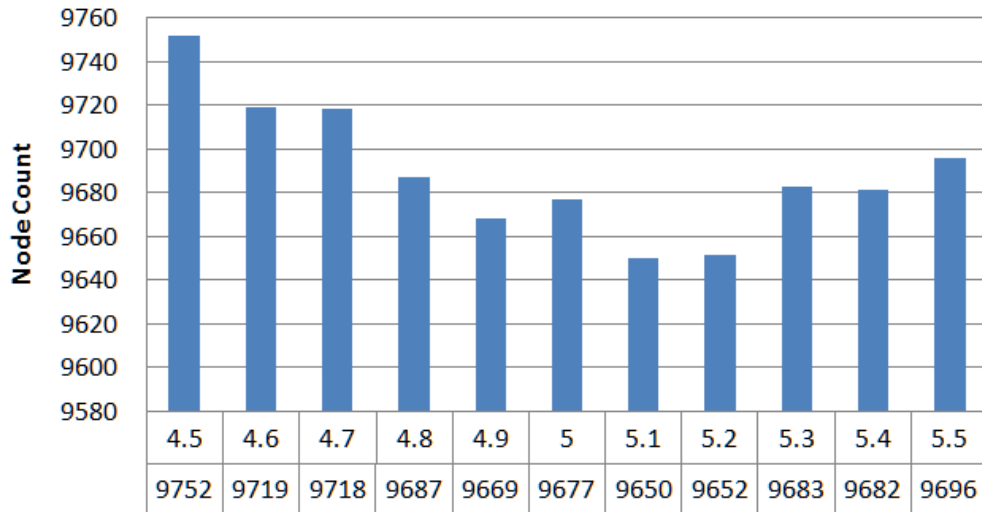


Fig. 2. Average number of nodes searched for 5-SAT instances against the value of γ

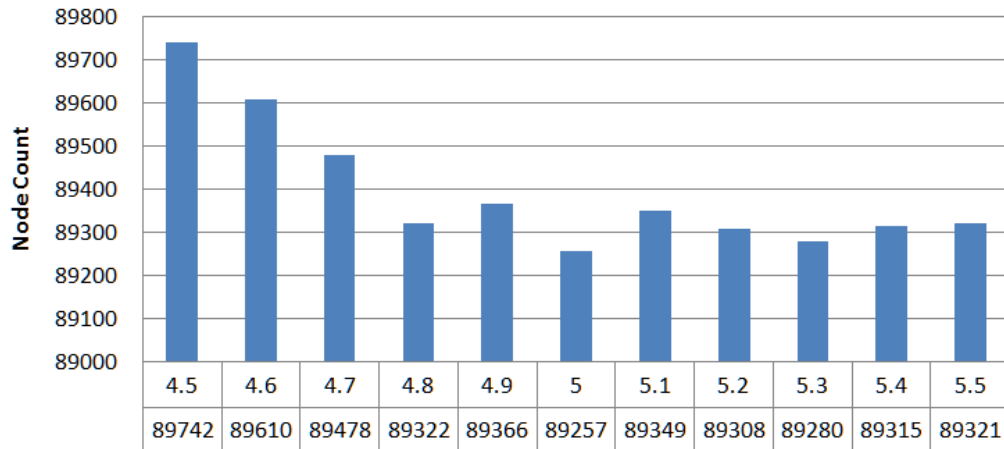


Fig. 3. Average number of nodes searched for 7-SAT problems against the value of γ

tests were run in order to specify the optimal value of γ . The results are shown in Figure 3. The performance is optimized for the value of $\gamma = 5$. It is important to mention at this point that the finding that the optimal value of γ is approximately 5 is in accordance with the experimental findings in the work of Oliver Kullmann[5] for a different weight heuristic. This coincidence implies the existence of a common behaviour of the weights between different weight heuristics.

4.2 Tuning of the upper/lower bounds

For each literal x in a clause its heuristic value in iteration $i + 1$ is $h_i(x)/\mu_i$. Taking under consideration that the γ represents the heuristic value of a falsified literal divided by μ_i , we expect that the heuristic value of each literal should be at most γ . Otherwise, a falsified literal would be less significant than a literal that is satisfied which is against the logic of the recursive weight heuristic. Thus, an upper bound should be used to normalize the values of the heuristic. In addition, it is necessary to apply a lower bound to prevent the heuristic value of a literal to become zero and subsequently convert the whole weight heuristic value into zero. The lower bound was experimentally set to 0.1 serving its purpose. On the other hand, the upper bound needs tuning in order to spot the optimal value for each case of k -SAT problems.

In the implementation the upper bound is applied on the heuristic value of each literal as soon as it is calculated and if it is exceeded, the heuristic value is set to the bound. Tests were run on the same set of 5 and 7 SAT instances in order to experimentally obtain the optimal values of the upper bound and to compare it with the expectations that were created by the aforementioned intuition. The results present the same behaviour for both 5-SAT and 7-SAT instances. Beginning from a low initial value the performance increases as the upper bound increases. At some point optimal performance seems to be approached and further increasing of the upper bound causes a decrease in the performance. Results for 7-SAT instances are shown in Figure 4. The optimal performance appears to be around the value 11000. Further exploration for the values around 11000 reach the conclusion that the value 10900 can be used as the optimal one as shown in 5.

Observing the results a relation can be derived between k and the upper bound values. The upper bound appears to be approximately $(5^k)/7$. A comparison between the actual optimal values for 4,5,6,7-SAT instances and the values proposed by the aforementioned relation is shown in Table 1. The values seem to converge and the difference between the performance of the proposed values and the actual ones is negligible. However, we see that in case of 4-SAT instances the variance is significant. In any case, the proposed relation can serve as a starting point for the search of the optimal value of the upper bound for any k -SAT instances.

In addition, in Table 2 a comparison between the actual values of the upper bound and γ is shown. The expected behaviour is confirmed. As k increases, the optimal upper bound increases relatively proportionally to the average μ_i value where $i = 3$ ($\bar{\mu}_3$) since the tests were run for an

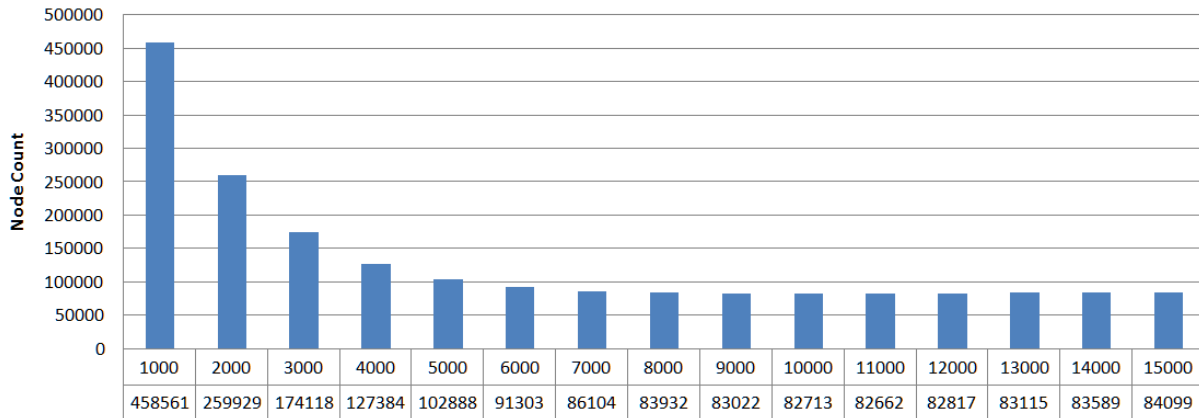


Fig. 4. Average number of nodes searched for 7-SAT problems against the value the upper bound

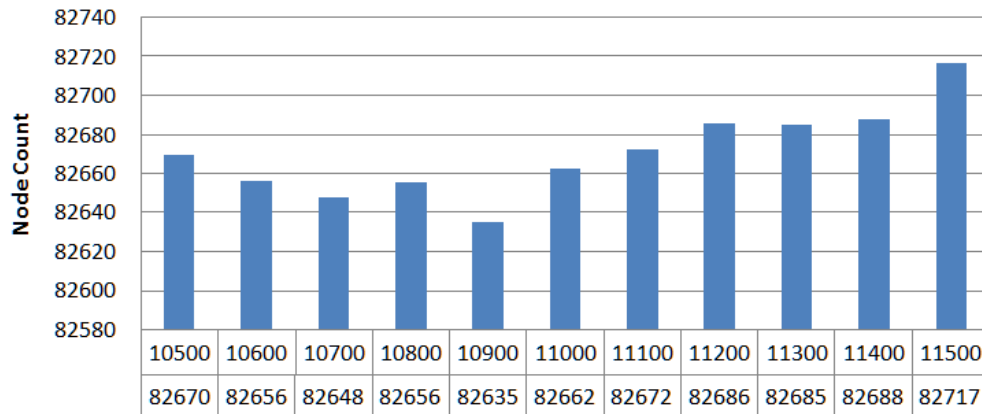


Fig. 5. Average number of nodes searched for 7-SAT problems against the value the upper bound

Table 1. Upper bound optimal values for 4,5,6,7-SAT instances in comparison with $(5^k)/7$

k	Upper bound	$(5^k)/7$
4	130	89
5	480	446
6	2150	2232
7	10900	11160

accuracy level of 3. Consequently, the heuristic values are maintained in a level not more than 50% of the value of γ .

Table 2. Upper bound optimal values for 4,5,6,7-SAT instances in comparison with γ

k	Upper bound	$\bar{\mu}_3$	(Upper bound)/ $\bar{\mu}_3$	γ	(Upper bound/ $\bar{\mu}_3$)/ γ
4	130	53	2.45	5	49%
5	480	236	2.03	5	41%
6	2150	1118	1.92	5	38%
7	10900	5800	1.88	5	38%

5 Experimental Results

After optimizing the parameters that affect the performance of the heuristic, experimental tests were run in order to assess the improvement of the generalised implementation of the recursive weight heuristic on k -SAT instances compared to *march_hi*, the latest version of the SAT solver that implements the recursive weight heuristic only for 3-SAT instances and for $k > 3$ it applies the heuristic with just one iteration (accuracy is 1). The dataset of the experiments consisted of 100 unsatisfiable 5-SAT instances with 65 variables and 1385 clauses and of 100 unsatisfiable 7-SAT instances with 40 variables and 3560 clauses. To measure the performance the size of the search tree (node count) and the execution time are reported. The experiments were executed on the hardware specifications shown in Table 3.

Table 3. Hardware specifications

CPU	Intel [®] Core [™] 2 Duo P7350 @ 2.0GHz
MEMORY	4GB DDR2 @ 800MHz
OS	Ubuntu 9.10 64-bit

In Figure 6 a comparison between the size of the search tree is shown between the implementation of the recursive weight heuristic for accuracy levels of 1,2,3 and *march_hi*. The reduction of the size of the search tree is significant as the accuracy level increases. In fact, for accuracy level 3, the size of the search tree is approximately half than it is for accuracy level 1 for both 5-SAT and 7-SAT instances. As far as time is concerned, it is observed in Figure 7 that the recursive weight heuristic performs better than *march_hi* for accuracy level 2. In spite of the radical reduction of the search tree size from accuracy level 2 to 3, the increase of the cost of the heuristic exceeds the benefit of the reduced size leading to a decrease in the performance. It is therefore concluded that accuracy level of 2 results in the best performance for 5-SAT and 7-SAT instances in contrast with 3-SAT instances, where accuracy level of 3 performs the best [7].

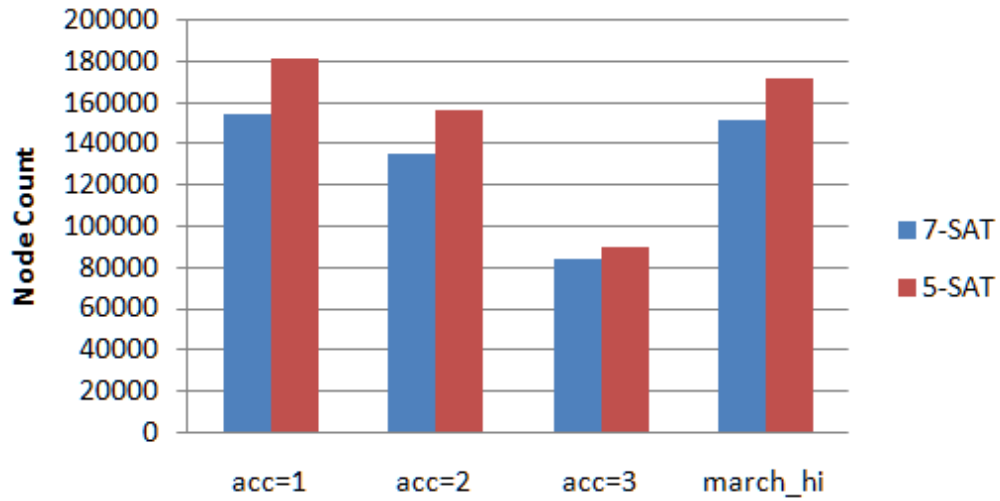


Fig. 6. Comparison of the average number of nodes searched between the generalized implementation of the weight recursive heuristic for accuracy of 1,2,3 and march_hi

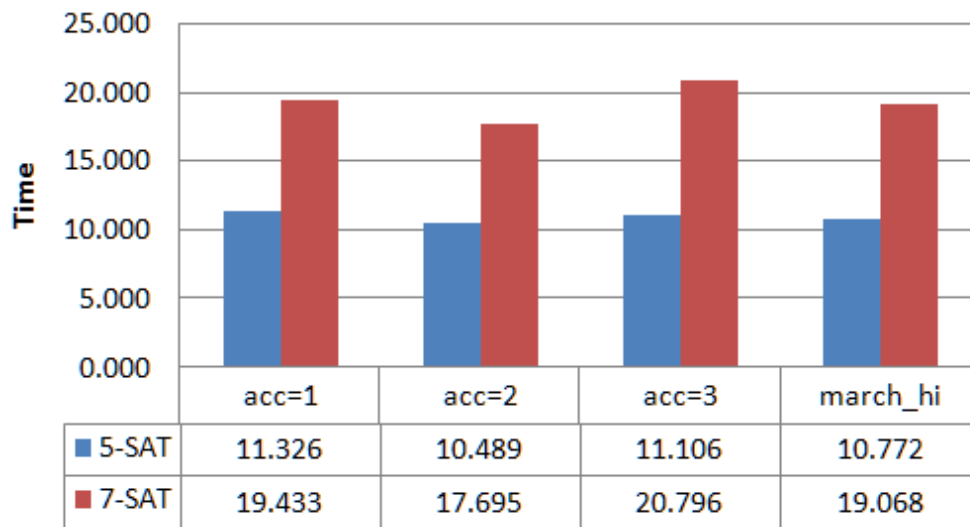


Fig. 7. Comparison of average execution time in seconds between the generalized implementation of the weight recursive heuristic for accuracy of 1,2,3 and march_hi

6 Conclusion

A generalization for the recursive weight heuristic, a branching heuristic for look-ahead SAT solvers, was performed. The ways to implement it efficiently were explored and the outcome is presented and analysed. In addition, the tuning of the parameters that affect the performance of the heuristic was researched and the results were shown together with a relation that can be used for further exploring k -SAT problems. Finally, the improvement of the performance of a SAT solver using the recursive weight heuristic was shown for 5-SAT and 7-SAT instances.

References

1. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
2. Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995.
3. Gabriel Istrate. Critical behavior in the satisfiability of random k -horn formulae.
4. Himanshu Jain and Edmund M. Clarke. Efficient sat solving for non-clausal formulas using dpll, graphs, and watched cuts. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 563–568, New York, NY, USA, 2009. ACM.
5. Oliver Kullmann. Investigating the behaviour of a sat solver on random formulas. Technical report, University of Wales Swansea, 2002.
6. Oliver Kullmann. Present and future of practical sat solving. pages 283–319, 2008.
7. Sid Mijnders, Boris de Wilde, and Marijn J.H. Heule. Symbiosis of search and heuristics for random 3-sat. In *Federated Logic Conference July 2010*, 2010.