Weight Resets in Local Search for SAT

Abdelraouf Ishtaiwi, Ghassan Issa, Wael Hadi, and Nawaf Ali

***Abstract*—In this paper, we investigated the influence of resetting weights in what we refer to as safely satisfied sub areas within the search space. Our work is divided into two main tracks; track one is to search for sub areas within the search space where a group of connected clauses are all satisfied. In track two, a Weight Reset mechanism is designed and implemented within the Multi-Level Weight Distribution (mulLWD) algorithm, which produced a new algorithm known as mulLWD+WR.**

The impact of our new strategy, the Weight Reset mechanism, is illustrated via an extensive experimental range of evaluation on benchmarks obtained from the DIMACS and the SAT Competition 2017 problem sets. Our investigation and experimental evaluation shows that the Weight Reset mechanism, when compared to the state-of-the-art solving algorithms, can significantly improves the process of searching for solutions when solving hard Boolean satisfiability (SAT), Planning, scheduling, and many other hard combinatorial problems. Furthermore, the weight reset could be generalized to be employed by any Dynamic Local Search approach.

***Index Terms*—Satisfiability, optimization, dynamic local search.**

I. Introduction

Propositional satisfiability problems (SAT) has attracted the attention of many researchers in the last two decades due to its characteristics, such that, the SAT problem is the first known problem to be proven to be NP-complete [1], which implies that, finding a generic algorithm to solve every SAT problem is yet an open area of research. Nevertheless, Local Search (LS) heuristics, were able to solve large SAT problems that consists of hundreds of thousands of variables. Which is realistic enough to handle many real-world problems from computer science fields, specially, artificial intelligence.

In our study, we consider propositional satisfiability problems in the form of Conjunctive Normal Form (CNF). CNF formulas consist of conjunctions of clauses = $c1$, $c2$, $c3$, ...ci, where each clause is a set of disjunctions of literals, and each literal is a (propositional variable vj or its negations $-vj$),

$$F = \wedge c_i \vee c_i v_j \quad (1)$$

The task of solving a CNF formula is to assign a value $\in$ (0, 1) for each literal in F, so that F evaluates to true. In general, there are two main approaches to solving formulas in CNF, 1) Systematic search and 2) Stochastic local search (SLS) approaches. Applying systematic search techniques to solve randomly generated large SAT formulas is infeasible, as all the literals of the CNF formula must be visited in a consistent manner until a complete assignment that evaluate F to true is found or the search concludes that the problem has no true assignment. Thus, systematic search takes huge amount of time and the results of the search is obsolete when the time is constrained. Despite that, systematic search techniques have the advantage of guaranteeing to find a solution to the problem if one exists or conclude that the problem has no solution. On the other hand, Local search techniques have been proven to be able to handle problems of very large size (hundreds of thousands of variables and millions of clauses), much faster than systematic search [2] with the exempt that there is no guarantee to finding a solution if one exists. However, SLS solvers, when applied for solving problems in the industrial category of the SAT competition [3], performed poorly in comparison to the systematic solvers.

A. Dynamic Local Search Approach (DLS)

Stochastic local search (SLS) techniques follow a general scheme in which they start initially by randomly assigning Boolean values in (0, 1) to all literals in F. The initialization step produces a group of satisfied clauses where each satisfied clause (satCl) has at least one true literal, and a second group, unsatisfied clauses, where all the literals in each unsatisfied clause unSatCl are false. The cost function of the current assignment CF = $\sum$(unSatCl), is then improved by the search iteratively, by modifying the values of unSatCl literals, so that the $\sum$(satCl) is maximized (or the sum of the unSatCl is minimized), until either CF = zero, or the search has reached its predefined time limit.

The above scheme is very effective as long as the search can reduce the cost function CF, once the search is unable to reduce the cost function, it gets into trap known as local minimum. Therefore, an important challenge of SLS solvers is to escaping from local minima or avoiding getting into them. Many successful heuristics were produced to deal with local minima, for example restarting the search when stuck in local minima as in GSAT+Restart heuristic [4], or taking random moves as in GSAT+Walk heuristic [5], these two heuristics could successfully escape from local minima. However, they might lead the search away from reaching some optimal solutions. Another heuristic occurred in the year 1993 [6] known as Breakout heuristic, where weights are used to dynamically alter the search space so that cost increasing moves are permitted. The stat-of-the-art local search solvers, that use different variants of the weighting strategy of the Breakout heuristic, are known as Dynamic Local Search weighting solvers. There are many examples of

Manuscript received August 21, 2019; revised October 10, 2019.

The authors are with the Faculty of Information Technology, Petra University, Amman, Jordan, (e-mail: aishtaiwi@uop.edu.jo, gissa@uop.edu.jo, whadi@uop.edu.jo, n.ali@uop.edu.jo).

doi: 10.18178/ijmlc.2019.9.6.886

the state-of-the-art solvers that utilize weights, for example, mulLWD+ [7], DCCAlm [8], CSCCSat [9], BalancedZ [10], Score2 SAT [11], and previously DLM [12], SAPS [13], PAWS [14] and DDFW [15].

B. DLS Weighting Mechanisms

The DLS weighting solvers follow a general approach of handling weights during the search. That is, increasing weights (multiplicatively as in SAPS or additively as in PAWS) on all false clauses, and then periodically decreases the weights on the weighted clauses. The additive weighting DLS was first introduced in PAWS, which performed more efficiently when compared to SAPS [14]. Therefore, the weighting scheme used in PAWS has been used in many state-of-the-art solvers such as DCCAlm, CSCCSat and more recently Score2SAT algorithms. These solvers have shown an interesting performance and were superior to non-weighting LS techniques. Based on these facts a conclusion could be drawn that handling weights has an imperative role on the overall performance of all weighting DLS solvers. Hence, the decision of whether to increase the weights or decreasing them has become the factor that distinguishes the solvers performance from each other's. Therefore, the development of strategies to efficiently maintaining the weights has become a field of research for many researchers and specialists in the area of artificial intelligence.

In this study, we are interested in studying the weights behavior on some areas of the search space when a false clause (unSatCl) becomes satisfied (satCl), while its neighboring clauses, neighbor (unSatCl), are all satisfied. More specifically, we are interested in investigating a weight reset technique and its impact on the overall performance of the search process. Furthermore, we examine the co-relation between the clauses, the size of their neighborhoods on one side, and the weight reset mechanism on the other side. We will also show the effect of resetting the weight strategy when implemented in one of the weighting DLS solver, the mulLWD+.

C. Multi-level Weight Distribution (mulLWD+)

In the year 2005 [15] a new strategy known as Divide and Distribute Fixed Weight (DDFW) was introduced with two characteristics that uniquely distinguishes it from other solvers. One characteristic is that, the clauses at any given time of the search process are treated as two separate categories depending on whether the clauses are satisfied or unsatisfied. The Second and more important characteristic, is that, weights are increased and decreased in a combined weight distribution mechanism, such that the weights from the satisfied clauses are moved to the unsatisfied clauses. In the year 2017 [7], DDFW was extended into mulLWD+ which is still employ the categorization of the clauses, and weight distribution mechanisms with the difference that weights distribution is performed via moving weights to a false clause from clauses that are directly connected to the false clauses (first level neighbor), or from clauses that are in the neighborhood of the first level neighbors. In our current study, we implemented the weight reset mechanism within the mulLWD+ for two important factors: 1) the weight distribution is explicit and could be easily traced, and 2) mulLWD+ inherits an imperative factor from DDFW, where it is a domain independent algorithm that requires no parameter tuning.

II. A Satisfied Clause within a Satisfied Neighborhood

Maintaining the weights during the search, alter the search space so that the search does not get stuck and continue until an optimum solution is found, whether weights are incremented and periodically decremented as discussed in I-B, or as the case of weight distribution of DDFW and mulLWD+. However, we observed that some clauses can hold the weights while they are satisfied and all their neighbors are satisfied. For instance, consider the following:

Clause $unSatCl(c)$ where $\forall literals \in unSatCl(c) =$ false and $\exists$ a satisfied clause satCl(n) which is a neighbor of c, neighbor (unSatC l(c)), such that a literal(unSatC l(c)) $\in$ (unSatC l(c) $\wedge$ satC l(n)). And all the neighbors (unSatC l(c)) are satisfied.

We ran mulLWD+ on some problems obtained from the DIMACS benchmarks sets, and kept track of the changing status of the clauses from unSatCl to satCl, and examined the neighboring areas of each newly satisfied clause. Our initial observation indicates that newly satisfied clauses maybe connected with neighboring areas, where all the clauses are satisfied. Moreover, the experiment also indicates that this scenario can frequently occur during the run time of mulLWD+.

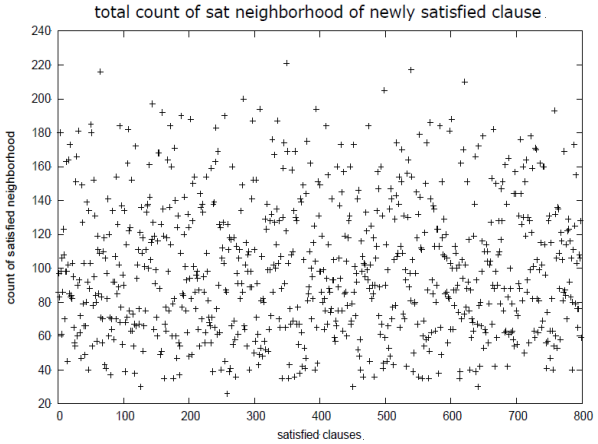

Fig. 1. Illustration of a clause and its neighboring clauses, the jnh210.

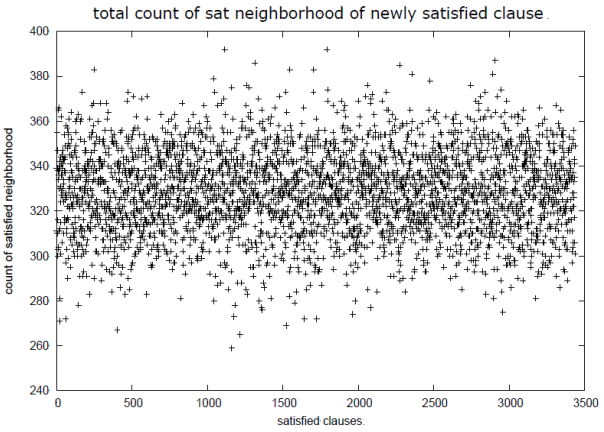

Fig. 2. Illustration of a clause and its neighboring clauses, the f800.

For instance, the above scenario is illustrated in figures (1,

2, 3), where we picked three problems of different sizes, the jnh210, the f800-hard, and the g125-17 from the DIMACS benchmarks problems. In each figure, we plotted the clauses distribution (x −axis), and the corresponding total number of fully satisfied neighborhoods (y − axis). We ran mulLWD+ on the three problems while keeping track of whether a clause and its neighboring clauses are all satisfied. The jnh210 is a small problem, with 100 variables and 800 clauses, that could be easily solved in comparison to f800, which is a more interesting problem that is randomly generated and a much harder problem with 800 variables and 3,440 clauses, and the last problem is the hardest and largest among the three problems which is generated from the graph theory area, known as graph coloring with 2125 variables and 66,272 clauses.

As we previously discussed, Fig. 1, Fig. 2, and Fig. 3, clearly indicate a high level of frequent occurrence of the situation where a clause and its all neighboring clauses are satisfied.

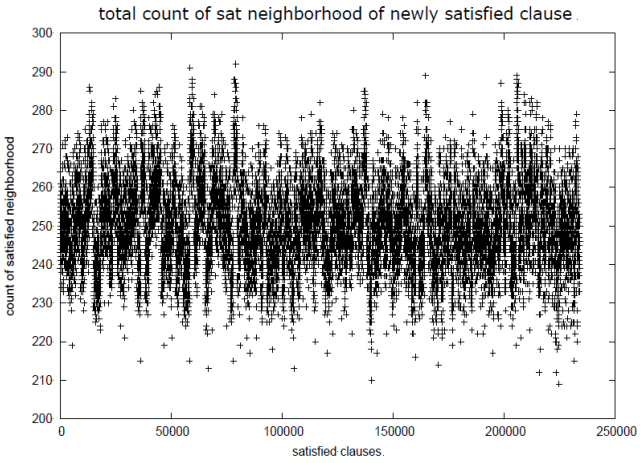

Fig. 3. Illustration of a clause and its neighboring clauses, the g125.

III. Weight Reset Strategy

Based on what has been discussed in section II, the following questions arise: Is it necessary for a clause to keep the weights after it has become satisfied, where all its neighboring clauses are satisfied? Also, does resetting the weight of a clause to its initial value has any effect, (whether negative or positive), on the overall performance of the search process? To answer these two questions, we performed a wide range of experiments, for instance, in results not mentioned here, we examined the process of resetting the weights of the clauses for the whole neighborhood to their initial value, in the moment of satisfying the last unsatisfied clause that belongs to it. The results indicate that resetting the weights in a completely satisfied neighborhood has a negative impact on the search process. It is our conjecture that, the cause for such impact is that clauses from other neighborhoods will be affected if they are connected to the rested neighborhood.

In another experiment, we rested the weights to their initial value, for a clause when the only remaining unsatisfied clause becomes satisfied, within a satisfied neighborhood. The initial results of our experiment were showing overall better and faster performance, as in some cases, the weight reset strategy helped the search reaching solutions in a faster CPU time. Algorithm 1 is the original mulLWD+ without the weight reset strategy, as we discussed in I-C, for further and more details about mulLWD+ please see [7]. The second algorithm, (Algorithm 2), shows the weight reset strategy pseudocode. Both algorithms are similar in terms of the beginning of the search process and the way the weights move between the clauses, such that weights move from the satisfied clauses to the unsatisfied clauses. The algorithms are also similar in the moving the weights from the first-level neighborhood when possible (the first neighborhood are all the clauses that share at least a same signed literals with any given clause); otherwise weights are moved from the second-level neighborhood (second level neighborhood clauses are any given clause that share at least a same signed literal with the first level neighboring clauses). The important difference between the two algorithms is that, the second algorithm adopted the process of resetting weights, as shown by the lines from 22 to 35 in Algorithm 2.

Algorithm 1 mulLWD+(F, Winit)

```
1: randomly instantiate each literal in F;
2: set the weight (wi) for each clause ci in F to Winit;
3: while solution is not found and not timeout do
4:     find and return a list L of literals causing the greatest reduction in
       weighted cost _w when flipped;
5:       if (Δ_w < 0) or (Δ_w = 0 and probability _ 15%) then
6:           randomly flip a literal in L;
7:       else
8:           for each false clause cf do
9:             search for a satisfied same sign neighboring
                 clause ck with maximum weight wk;
10:          if ck found then
11:              transfer a weight of one from ck to cf;
12:          else
13:            select the first satisfied same sign neighbor
                 of neighboring clause ck;
14:          end if
15:          if wk < Winit then
16:              randomly select a clause ck with weight
                 wk _Winit;
17:          end if
18:          if wk >= Winit then
19:              transfer a weight of one from ck to cf;
20:          end if
21:       end for
22:   end if
23: end while
```

Algorithm 2 mulLWD+(F, Winit)

```
1: randomly instantiate each literal in F;
2: set the weight (wi) for each clause ci in F to Winit;
3: while solution is not found and not timeout do
4:     find and return a list L of literals causing the greatest reduction in
       weighted cost _w when flipped;
5:     if (Δ_w < 0) or (Δ_w = 0 and probability _ 15%) then
6:         randomly flip a literal in L;
7:     else
8:         for each false clause cf do
9:             search for a satisfied same sign neighboring
               clause ck with maximum weight wk;
10:            if ck found then
11:                transfer a weight of one from ck to cf;
12:            else
13:                select the first satisfied same sign neighbour of
                   neighbouring clause ck;
14:            end if
15:            if wk < Winit then
16:                randomly select a clause ck with weight
                   wk_Winit;
17:            end if
```

```
18:         if wk >= Winit then
19:             transfer a weight of one from ck to cf;
20:         end if
21:     end for
22:     pick a satisfied clause satcl
23:     for all neighbors of satcl do
24:         search for an unsatisfied neighboring clause cf;
25:         if cf found then
26:             stop;
27:         else
28:             found = 1;
29:         end if
30:     end for
31:     if found = 1 then
32:         reset the weight of satcl to Winit;
33:         reset the makes and breaks of satcl;
34:     end if
35: end while
```

IV. Experimental Evaluation and Analysis

The general approach to this research is to examine the effect of weights in the search space. Therefore, we have performed a thorough assessment of the weight's behaviors during the research process. In order to carry out the assessment, we have selected a wide range of problems that were obtained from the SAT competition benchmarks[1] and the SATLIB DIMACS benchmarks[2]. The experimental work environment was as follows: Algorithms were applied to the selected problems, on the Linux operating systems, on a machine with a memory of 8 GB and an I5 CPU (2.5 GHz). Each algorithm performed 100 runs, on each problem, and each run was given 500 seconds. The computational mean was calculated when the solution was reached, as well as the percentage of solutions found for all the runs.

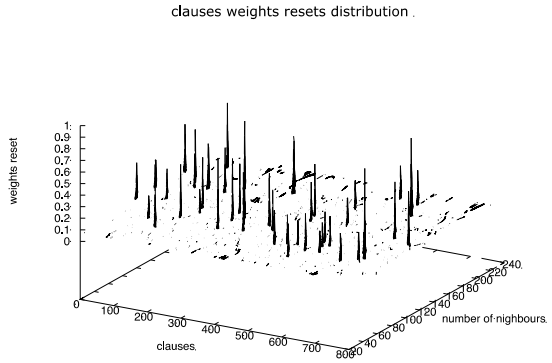

Fig. 4. Weight resets distribution, jnh.

We have included two tables of results. The first table, (Table I), shows the results of the algorithms performance on selected problems from DIMACS benchmarks set. The second table shows the results of the algorithms when applied to the SAT competition 2017 problem sets, (Table II). In both tables, we note that the algorithm that employs the weight reset strategy (mulLWD+WR) performed significantly better in all problem sets. For instance, in Table I, mulLWD+WR performed 10 times better than mulLWD+ in most problems. Moreover, mulLWD+WR best performance was on large and hard prob- lems. Where on small and easy problems, mulLWD+WR performed twice better. In Table II, mulLWD+WR performed significantly better on All the f lat problems and interestingly could solve the uniform-random-k5 and uniform-random-k7 where the mulLWD+ was timed out without reaching any solution.

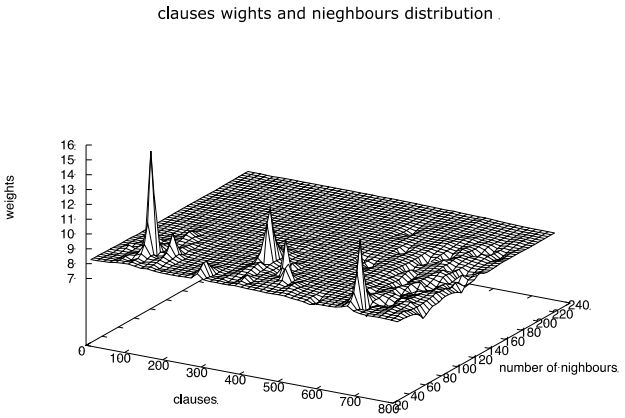

Fig. 5. Neighbors to clauses weight distribution, jnh.

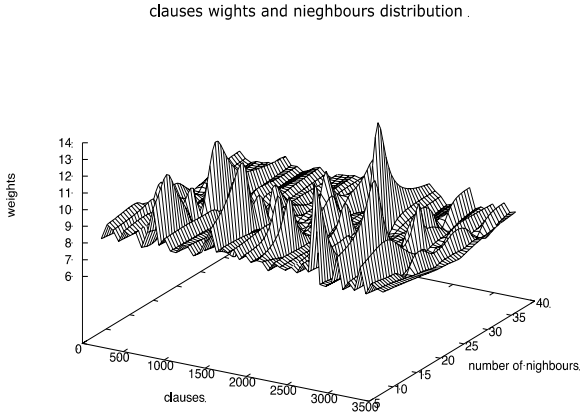

Fig. 6. Weight reset distribution, f800.

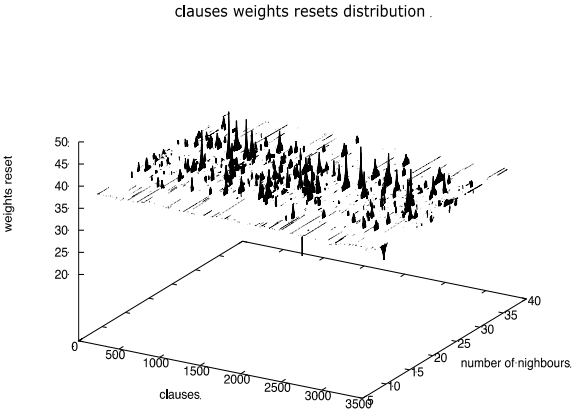

Fig 7. Neighbors to clauses weight distribution, f800.

For more illustration, we have included the Fig. 4 and Fig. 6 of the jnh and the f800 weight distribution respectively, that illustrate the relationship between the clauses, the number of neighboring clauses for each clause and how much weight. A clause gained during the search process until reaching a solution. We can clearly conclude that, clauses that have many clauses in their neighborhood were more weighted, and clauses which are linked with a fewer number of clauses were less weighted. This is logical since the smaller the number of clauses within a neighborhood, the harder the movement of weights among the clauses. Fig. 5 and Fig. 7 illustrate the

[1] (http"//www.satcompetition.org/)

[2] http"// http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html/)

relations between the number of clauses in the neighborhood of each clause, and the number of weight resets for each clause, for the jnh and f800 respectively.

TABLE I: COMPARISON RESULTS OF MULLWD AND MULLWD+ WEIGHT RESET PERFORMANCE, ON DIMACS BENCHMARKS

problems	MulLWD+		MulLWD+WR	
	time	%sol	time	%sol
bw_large.d	59.2	100	5.7	100
4blocks	180.18	100	47.3	100
aim-50-1_6-no-1	42.7	23	43.7	75
aim-200-6_0-yes1-4	0.06	100	0.03	100
ais12	26.9	100	7.57	100
bw_large.a	0.39	100	7	100
f1600-har	92.7	23	75.2	88
logistics.d	1.96	100	0.5	100
medium	0.05	100	0.02	100
uf400-hard	25.8	100	8.53	100
f800-har	119.1	99	48	100
bw_large.c	201.1	100	13.6	100
bw_large.a	0.6	100	0.08	100
huge	0.64	100	0.13	100
f800-har	76.3	98	46.03	100
4blocksb	270.4	100	41.6	100
BMS_k3_n100_m4_0	1.8	100	0.5	100
BMS_k3_n100_m4_499	28.5	100	5.71	100
e0ddr2-10-by-5-1	13.93	100	7.64	100
ewddr2-10-by-5-8	73.9	100	7.35	100
flat100-1	1.9	100	0.08	100
flat100-100	0.3	100	0.04	100
g125.18	8.96	100	0.13	100
g250.15	2.2	100	0.95	100
ii8a1	0.2	100	0.03	100
ii32e5	3.41	100	0.23	100
RTI_k3_n100_m4_499	0.3	100	0.05	100

TABLE II: COMPARISON RESULTS OF MULLWD AND MULLWD+ WEIGHT RESET PERFORMANCE, ON THE SAT COMPETITION BENCHMARKS

problems	MulLWD+		MulLWD+WR	
	time	%sol	time	%sol
fla-barthel-400-4	0.07	100	0.01	100
fla-barthel-420-2	0.07	100	0.03	100
fla-barthel-420-3	0.09	100	0.01	100
fla-barthel-420-4	0.03	100	0.01	100
fla-barthel-440-3	0.05	100	0.02	100
fla-barthel-520-2	0.04	100	0.01	100
fla-qhid-400-5	0.06	100	0.01	100
fla-qhid-420-2	0.03	100	0.006	100
fla-qhid-420-4	2.16	100	0.03	100
fla-qhid-480-2	0.06	100	0.006	100
fla-qhid-500-5	0.06	100	0.01	100
fla-qhid-540-4	0.06	100	0.006	100
unif-k5-r21.117-v200	125.5	100	4.32	100
unif-k7-r55.0-v50000	422.3	100	10.7	100
unif-k7-r56.0-v50000	n/a	n/a	12.6	100
unif-k7-r87.79-v90	n/a	n/a	11.6	100
unif-k7-r87.79-v98	n/a	n/a	108	100
unif-k7-r87.79-v102	n/a	n/a	100.7	100

With the f800 Fig. 7, the number of weight resets was performed much more than the jnh problem, in the since that it was distributed among most of the clauses in the problem. However, the number of weight resets for each clause was much less, which is also expected as f800 have more clauses and a harder problem to solve. This outstanding performance of mulLWD+WR, stressed that the weight-reset strategy has a positive impact on the process of research in general. If we look more deeply at the effect of weight reset strategy, we find that there are a number of possible scenarios that can result in such performance. One of the most important and possible scenarios is related to each individual clause and its neighborhood. That is, if a clause c was unsatisfied, such that all its literals evaluate to false, flipping the value of a literal (from zero to one or vice versa) in c will change its status to satisfied. Now, if all the clauses of the neighborhood of clause c are satisfied, after satisfying clause c, clause c can never be unsatisfied unless at least one of the clauses in the neighborhood of clause c becomes unsatisfied. Therefore, a clause weight can be safely rested to the initial weight, without affecting the overall search process. In other words, the weight reset smooth the weights safely and prevent a satisfied clause c from holding the weight (especially if it is heavily weighted), while clause c neighboring clauses are all satisfied.

V. CONCLUSION

In this research, two scenarios were experimentally investigated. The motivation of the first scenario was because of the observation during the study of weights and their dynamic changes during the search process. As a result of investigating the weights changes, we have come to a conclusion, which is supported by the experimental analysis that the weights in some areas of the search space may accumulate and hold the weights for long periods, without any positive impact on the overall performance of the search process. On the contrary, weights sometimes can be the cause of delaying the search process from reaching some optimal solutions to the problems under consideration.

Therefore, and based on what was observed in the first scenario, we have built a strategy for the optimal handling of weights on clauses which no longer need the weights, known as weight reset strategy. This strategy is concerned with clauses are satisfied. Our new strategy, the weight reset, has been proven via our experiments to be very effective when applied for solving the SAT problems.

ACKNOWLEDGMENT

The authors would like to acknowledge the financial support of the Scientific Research Committee at Petra University, and all the Information Technology faculty members who contributed to this work.

REFERENCES

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. the Third Annual ACM Symposium on Theory of Computing*, New York, USA, 1971, pp. 151–158.
[2] H. Hoos and T. Stulze, *Stochastic Local Search*, Cambridge, Massachusetts: Morgan Kaufmann, 2005.
[3] Sat competition. [Online]. Available: http://www.satcompetition.org
[4] B. Selman and H. Kautz, "Domain-independent extensions to gsat: Solving large structured satisfiability problems," in *Proc. 13th IJCAI*, 1993, pp. 290–295.
[5] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method" in *Proc. the Eighth National Conference on Artificial Intelligence*, 1990, vol. 1, pp. 17–24.

[6] P. Morris, "The Breakout method for escaping from local minima," in *Proc. 11th AAAI*, 1993, pp. 40–45.
[7] A. Ishtaiwi, M. Juarez, and G. Issa, "Multi level weight distribution in dynamic local search for sat," in *Proc. 3rd International Conference on Information Technology, Control and Computer Engineering ITCCE* 17, 2017, pp. 79–86.
[8] C. Luo, S. Cai, W. Wu, and K. Su, "Double configuration checking in stochastic local search for satisfiability," in *Proc. the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014, pp. 2703–2709.
[9] C. Luo, S. Cai, K. Su, and W. Wu, "Clause states based configuration checking in local search for satisfiability," *IEEE Trans. Cybernetics*, vol. 45, no. 5, pp. 1014–1027, 2015.
[10] A. Belov, H. Diepold *et al*. (2014). *Proceedings of Sat Competition 2014: Solver and Benchmark Descriptions*. [Online]. Available: http://dl.acm.org/citation.cfm?id=1865499.1865502
[11] T. Balyo, M. J. H. Heule, and M. Jarvisalo. (2017). Proceedings of Sat Competition 2017: Solver and Benchmark Descriptions. [Online]. Available: http://hdl.handle.net/10138/224324
[12] Z. Wu and B. Wah, "An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems," in *Proc. 17th AAAI*, 2000, pp. 310–315.
[13] F. Hutter, D. Tompkins, and H. Hoos, "Scaling and probabilistic smoothing: Efficient dynamic local search for SAT," in *Proc. 8th CP*, 2002, pp. 233–248.
[14] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira Jr., "Additive versus multiplicative clause weighting for SAT," in *Proc.19th AAAI*, 2004, pp. 191–196.
[15] A. Ishtaiwi, J. Thornton, A. Sattar, and D. N. Pham, "Neighbourhood clause weight redistribution in local search for SAT," in *Proc. 11th CP*, 2005, pp. 772–776.

Abdelraouf Ishtaiwi received the M.S degree in 2001 and Ph.D. degree in 2008 both in information and communication technology from Griffith University, Brisbane, Australia. Currently, he is an assistant professor at the University of Petra, Faculty of Information Technology in Amman, Jordan. His interests are in artificial intelligence, distributed systems and handling big data.

Ghassan F. Issa received his BS degree in electrical engineering from University of Toledo in 1983, and the BS in computer engineering from Trine University, Indiana in 1984. Also Prof. Ghassan Issa received his M.S and PhD in computer science from Old Dominion University, Norfolk Virginia in 1987 and 1992. Currently, he is a professor at University of Petra, the Faculty of Information Technology in Amman, Jordan. His interests are in artificial intelligence.

Wa'el Hadi is an associate professor in computer information systems at the Department of CIS, University of Petra. In research, his current interests include data warehousing, data mining, and knowledge management. Dr. Hadi received his PhD degree in computer information systems from the Arab Academy for Banking and Financial Sciences, Amman, Jordan. Dr. Hadi has published more than 50 articles in refereed journal as well as national and international conference proceedings.

Nawaf Ali graduated from J.B. Speed School of Engineering, University of Louisville, Kentucky, USA. He holds a Ph.D. in computer engineering and computer science. He worked in the Faculty of Information Technology, ISRA University and University of Petra, Jordan. Currently he is working in the Computer Engineering Department at American University of the Middle East, Kuwait. His research interest focuses on authorship identification, machine learning, data mining and recently in IoT and robotics applications.