Engineering

Industrial & Management Engineering fields

Okayama University *Year* 2001

A minimal-state processing search algorithm for satisfiability problems

Nobuo Funabiki
Okayama University

Tokumi Yokohira
Okayama University

Toru Nakanishi
Okayama University

Shigeto Tajima
Osaka University

Teruo Higashino
Osaka University, Osaka

A Minimal-State Processing Search Algorithm for Satisfiability Problems

Nobuo Funabiki[1], Tokumi Yokohira[1], Toru Nakanishi[1], Shigeto Tajima[2], and Teruo Higashino[2]

[1]Department of Communication Network Engineering, Okayama University, Okayama 700-8530, Japan
{funabiki, yokohira, nakanisi}@cne.okayama-u.ac.jp

[2]Department of Informatics and Mathematical Science, Osaka University, Toyonaka 560-8531, Japan
{tajima, higashino}@ics.es.osaka-u.ac.jp

Abstract

The satisfiability problem (SAT) is a typical NP-complete problem where a wide range of applications has been studied. Given a set of variables U and a set of clauses C, the goal of SAT is to find a truth assignment to variables in U such that every clause in C is satisfied if it exits, or to derive the infeasibility otherwise. This paper presents an approximation algorithm called a minimal-state processing search algorithm for SAT (MIPS_SAT). MIPS_SAT repeatedly transits minimal states in terms of the cost function for searching a solution through a construction stage and a refinement stage. The first stage greedily generates an initial state composed of as many satisfied clauses as possible. The second stage iteratively seeks a solution while keeping state minimality. The performance of MIPS_SAT is verified through solving DIMACS benchmark instances.

Keywords

SAT, heuristic algorithm, optimization, DIMACS, MIPS_SAT.

1 Introduction

This paper presents a heuristic algorithm called a minimal-state processing search algorithm for satisfiability problem (MIPS_SAT). SAT is a typical NP-complete problem [1], and extensive studies on its complete or incomplete approximation algorithms have been reported. In a SAT instance, an m variable set U and an n distinct clauses C are given. A clause consists of literals connected by logical *or*. A literal is a variable or its negation. The goal of SAT is to find a truth assignment to variables such that the following Boolean formula is
made satisfiable if exits:

$$C_1 \wedge C_2 \wedge .... \wedge C_n \qquad (1)$$

or derives its unsatisfiability.

A wide range of practical applications has been reported for SAT and its related problems, in mathematical logic, inference, machine learning, constraint satisfaction, VLSI design automation, and computing theory. For example, in [2]-[4], the test pattern generations were reported using SAT formula. In [6], the maximum power dissipation estimation problem for CMOS circuits was transformed into a weighted max-satisfiability problem. In [7][8], SAT in database systems was studied.

A number of SAT algorithms have been developed. In [12], Gu et al. provided a general survey on them with their performance evaluations and practical applications. GRASP is one of best complete SAT algorithms which incorporates several search-pruning techniques with the non-chronological backtracking procedure. However, due to the computational complexity, even GRASP cannot solve large size SAT instances such as *f* and *g* in *DIMACS* [16]. An incomplete SAT algorithm can find only one random solution, although its multiple executions may produce different solutions. Besides, it usually cannot give an answer to an infeasible instance. The advantage of an incomplete algorithm is in the capability of finding a solution quickly even for large size instances. Within our knowledge, the discrete Lagrangian-based global-search method (DLM) [15] provides the best one. DLM formulates SAT as a discrete constrained optimization problem, where a Lagrangian function is defined to be minimized with the term for the constraint of satisfying every clause and the term for the objective function of minimizing the number of unsatisfied clauses. The performance of DLM is verified through solving DIMACS, where the search capability and the computation time are better than GSAT, GRASP, and other sophisticated algorithms. GSAT [17] is a randomized greedy local search algorithm, which performs well for solving hard random SAT instances. With help of three heuristic strategies [18], *clause weights*, *averaging in*, and *random walk*, GSAT can handle hard SAT instances with intricate underlying structures. *DIMACS* includes a variety of SAT instances from practical and theoretical worlds; circuit synthesis, circuit diagnosis, parity learning , artificially generated 3-SAT, randomly generated instances, large satisfiable 3-SAT, hard graph coloring, and towers-of-Hanoi encoding. Many papers have reported the simulation results for this benchmark suite.

The proposed MIPS_ SAT is an incomplete algorithm to formulate SAT as a discrete unconstrained optimization problem like by GSAT. MIPS_SAT repeatedly transits minimal states in terms of the cost function to solve a SAT instance through a construction stage and a refinement stage. The first stage greedily produces an initial truth assignment to variables. The second stage iteratively seeks a solution through transitions of minimal states with the hill-climbing capability. The performance is verified through solving DIMACS suite, where the solving capability and the computation time is summarized.

2 Framework of MIPS_SAT

In any state, MIPS_SAT assigns 1 (= True) or 0 (= False) to every variable in *U*. The unsatisfied clauses in *C* are kept in a clause list *CL* for further assignment changes. The state transition is designed to minimize the cost function *E*:

$$E = \sum_{i=1}^{n} w_i U_i(x) \quad (2)$$

where $U_i(x) = 0$ if the *i*th clause C_i is satisfied in a variable assignment state *x*, and $U_i(x) = 0$ otherwise, and w_i is the clause weight. As in the clause weight strategy [18], w_i is incremented when C_i is not satisfied by *x*. When $E = 0$ is achieved, the variable assignment state *x* represents a solution.

For efficient search, MIPS_SAT repeatedly visits minimal states in terms of the cost function *E*. A *minimal state* is defined as a state where any flip of a variable cannot reduce *E*. A *flip* is defined as a movement of changing an assigned value to its opposite one. This framework of minimal state transitions is based on a simple fact that a global minimum always exists among local minima. Thus, MIPS_SAT first generates an initial state greedily in the construction stage, and then, iteratively transits minimal states with hill-climbing and search-restarting functions in the refinement stage. All the *unit variables* in *unit clauses* are assigned the corresponding values and fixed in the construction stage beforehand as in the *David-Putnam-Loveland scheme* [21]. A *unit clause* is an unsatisfied clause that can be satisfied by exactly one unassigned variable, and a *unit variable* is an unassigned variable in a unit clause. An *unassigned variable* is a variable whose value is not assigned either 1 or 0.

3 Greedy Construction Stage

In the construction stage of MIPS_SAT, the unsatisfied clause list *CL* is initialized by *C*, and every unit clause is satisfied by assigning corresponding values to unit variables, which is repeated until no more unit clause exists in *CL*. Then, all unit clauses and variables are removed from *C* and *U* respectively, so that the succeeding procedures can avoid the unnecessary computation load. Let *n* and *m* be the number of clauses in *C* and the number of variables in *U* respectively. A greedy method for SAT sequentially produces an initial state of variable assignments to satisfy as many clauses as possible. An unassigned variable list *VL* is generated from *U*. To assign a value to one variable in *VL* sequentially, a *critical clause* in *CL* is detected for the corresponding value assignment. A *critical clause* is defined as a clause satisfiable by only one variable in the current state. Here, if two or more critical clauses exist, one variable in these clauses whose value assignment can satisfy the maximum number of clauses is selected. When no critical clause exists, a variable in *VL* whose value assignment can satisfy the maximum number of clauses in *CL* is selected. In either selection, the tiebreak is resolved randomly. Then, two lists *CL* and *VL* are updated. This sequential procedure is terminated when either list becomes null.

4 Iterative Improvement Stage

In the improvement stage of MIPS_SAT, the search process starts from the initial state by the construction stage, and repeatedly transits minimal states to minimize the cost function in (2). However, if only descent moves are allowed, the state may be trapped into a local minimum. Thus, three schemes for global convergence, *weight reset*, *variable shuffle*, and *variable reset*, are used together. The first scheme is designed to change the search direction, the second one provides a restating state without degrading the current cost, while the last one provides a restarting state far from the current one with degradation.

4.1 Descent Minimal State Transition

In a next minimal state, a variable of an unsatisfied clause in *CL* should be flipped to reduce *E*. At each iteration step, the change of *E* is checked for each variable flip for every clause in CL, which is started from a randomly selected clause to avoid a biased flip movement, and is stopped as soon as a variable flip of reducing *E* as in the hill-climbing [20]. Then, this variable is flipped, and the state of two lists and variable assignments is updated. This variable flip is repeated until a new minimal state is reached where *E* is not reduced by any variable flip. Then, to resume the state transition into a new state of satisfying harder clauses, the clause weight w_i for every unsatisfied clause in *CL* is repeatedly incremented by 1, until a new variable flip comes out.

As in [15[[20], hard SAT instances sometimes have large plateaus besides many local minima. In a plateau, some range of neighboring variable assignments gives the same value for *E*. To escape there, *sideway move* in [20] with a *tabu list* in [15] has been introduced, when hard instances are solved. Here, a variable flip for no change of *E* is also picked up. A tabu list is used together to avoid the cyclic state change, where any variable in the list is not selected for the sideway move. The list keeps the variables that have been flipped within a predefined number of flip movements *Tabu* since their last flips.

4.2 Weight Reset Scheme

Some clauses may have very large weights as the search proceeds. To escape from this state, every clause weight is initialized by the given initial clause weight *WB* regardless of its current value. Here, *WB* determines the degree to prevent the up-hill moves of increasing the number of unsatisfied clauses. For example, *WB* = 100 requires roughly 101 times of weight increments to accept an up-hill move. On the other hand, for *WB* = 1 requires only one time weight increment. Hence, when the divergence of search directions before and after this scheme's application should be restricted, *WB* should be assigned a large value. This scheme is actually applied when the minimum number of unsatisfied clauses is not updated within a predefined number of steps since its last application or the last update.

4.3 Variable Shuffle Scheme

When several trials of the weight reset scheme does not improve the solution quality, the variable for *CL* satisfying at most one critical clause is sequentially flipped. The number of critical clauses satisfied by each assigned variable is updated every time the state is changed. The flip procedure is repeated until no more such variables exist. The variable shuffle scheme can be regarded as continuous applications of sideway move and downhill moves with the unit clause weight.

4.3 Variable Reset Scheme

To further expand the search space, a restarting state with larger difference than the variable shuffle scheme is provided if the aboved-metioned schemes are insufficient. Actually, variables that are usually hard to be flipped, are flipped without any condition. Specifically, variables are flipped when their opposite value assignments satisfy many clauses with smaller clause weights compared to others. In addition, the number of flipped variables is increased when the solution quality has still not been improved after its several applications.

5 Simulations for DIMACS

For the performance evaluation of MIPS_SAT, the satisfiable DIMACS benchmark instances are solved. Like most existing SAT algorithms, the hardest instances "par32" and "hanoi5" cannot be solved.

5.1 Parameters in MIPS_SAT

In our tuning process of MIPS_SAT, it has first been fixed that the global convergence schemes are applied only for hard instances of "par16", "f", "g", and "hanoi4". They are applied when the minimum number of unsatisfied clauses has not been improved during *m* iterations, which assumes that every variable has been flipped once on average. For "par16", our simulations have found that *m/2* iterations can speed up the convergence. Actually, when the global convergence schemes are applied, the weight reset scheme is always applied first. Then, the shuffle scheme is applied when the best state has not been improved for five consecutive applications of the weight reset scheme. Finally, the variable reset scheme is applied when the best state has not improved for 50 consecutive applications of the shuffle scheme. For "hanoi4", the variable reset scheme is applied whenever the shuffle scheme is applied. Frequent applications of the variable reset scheme have been found to be very effective. This SAT instance has only one kind solution, and it has a lot of deep local minima where only one unsatisfied clause remains but is still far from the unique solution. Many restartings from a variety of states are necessary to escape from them and to reach the solution state. *WB* = 1 is normally used for SAT instances while *WB* = 10 is for "f600" and "f1000", *WB* = 50 for "f2000", *WB* = 300 for "g250.29", and *WB* = 100 for other "g" instances. We have found that *WB* should be assigned larger values in very large size instances. We have also found that sideway move should not be used for "par16". The tabu list length is 50 for "g" and "f", 100 for "hanoi4", and 0 for other instances.

5.2 Effects of Global Convergence Schemes

To see the effects of our schemes for global convergence, the following four cases are performed for hard SAT instances:
(1) case 1: weight reset,
(2) case 2: weight reset + variable shuffle,
(3) case 3: weight reset + variable reset, and
(4) case 4: weight reset + variable shuffle + variable reset. We note that MIPS_SAT cannot find any solution for hard SAT instances if the weight reset scheme is not adopted. Table 1 shows the number of successful runs among 10 runs for each case in "par16-2", "par16-5", "f2000", and "hanoi4". For the other instances, any case finds a solution in any run. This table indicates that each of three schemes individually helps the state of 2DOM to escape from a local minimum and converge to a solution. Particularly, they are very useful to solve "hanoi4".

In addition, the effect of the initial clause weight is examined for a large size SAT instance. Figure 1 depicts the change of the best number of unsatisfied clauses per 5,000 iteration steps to solve "g250.29", when *WB* is varied from 1 to 300. This figure indicates that the small value easily saturates the state improvement, where the weight reset scheme causes too much divergence in search directions.

5.3 Simulation Results for DIMACS

MIPS_SAT is applied to solve DIMACS SAT instances. A total of 10 runs are repeated with different random numbers for each instance. Table 2 shows the instance name, the number of successful runs among 10 runs, and the average, the minimum, and the maximum computation times (seconds) on Pentium-III 550 MHz only for hard SAT instances. For other SAT instances, MIPS_SAT can always find a solution in any run within 1.3 seconds on average. Table 2 indicates that MIPS_SAT finds a solution in any run even for hard SAT instances except for "hanoi4". For "hanoi4", the successful run rate is 70%. Most of existing algorithms cannot always solve these SAT instances. The extensive search capability of MIPS_SAT is very significant as an incomplete approximation algorithm, because an incomplete algorithm cannot guarantee either to find a solution even if it exists or to decide its infeasibility. A large number of repeated runs by
fast efficient algorithms may be necessary to improve the accuracy of the results obtained by incomplete algorithms. We conclude that MIPS_SAT contributes to achieving the goal of incomplete SAT algorithms by its extensive search capability and efficiency for a variety of instances in the NP-complete satisfiability problem.

6 Conclusion

This paper has presented MIPS_SAT, a minimal-state processing search algorithm for the NP-complete satisfiability problem. In MIPS_SAT, a construction stage first produces an initial search state by a simple greedy method, after all unit clauses are extracted and their corresponding unit variables are assigned values to satisfy them. Then, a refinement stage iteratively seeks a solution state while keeping minimality. This stage consists of the descent minimal state transition with three schemes for global convergence of the weight reset, the variable shuffle, and the variable reset. The performance of MIPS_SAT is verified through solving satisfiable DIMACS benchmark instances. The simulation results confirm the extensive search capability and efficiency of our MIP_SAT. The performance investigation for solving other instances than DIMACS benchmarks and the further improvement for solving hardest instances of "par32" and "hanoi5" will be in our future works.

References

1. M. R. Garey and D. S. Johnson, "Computers and intractability: a guide to the theory of NP-completeness," Freeman, 1979.
2. M. H. Schulz and E. Auth, "Improved deterministic test pattern generation with applications to redundancy identification," IEEE Trans. Computer-Aided Design., vol. 8, no. 7, pp. 811-816, July 1989.
3. T. Larrabee, "Test pattern generation using Boolean satisfiability," IEEE Trans. Computer-Aided Design., vol. 11, no. 1, pp. 4-15, Jan. 1992.
4. S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A transitive closure algorithm for test generation," IEEE Trans. Computer-Aided Design., vol. 12, no. 7, pp. 1015-1028, July 1993.

5. J. P. M. Silva and K. A. Sakallah , "Dynamic search-space pruning techniques in path sensitization," in Proc. 31st ACM/IEEE Design Automation Conf., pp. 705-711, 1994.
6. S. Devadas, K. Keutzer, and J. White, "Estimation of power dissipation in CMOS combinational circuits using Boolean function manipulation," IEEE Trans. Computer-Aided Design., vol. 11, no. 3, pp. 373-383, March 1992.
7. S. Guo, W. Sun, and M. A. Weiss, "Solving satisfiability and implication problems in database systems," ACM Trans. Database Systems, vol. 21, no. 2, pp. 270-293, June 1996.
8. S. Guo, W. Sun, and M. A. Weiss, "On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems," IEEE Trans. Knowl. Data Eng., vol. 8, no. 4, pp. 604-616, Aug. 1996.
9. J. de Kleer, "An assumption-based TMS," Artificial Intelligence, vol. 28, pp. 127-162, 1986.
10. M. Davis and H. Putnam, "A computing procedure for quantification theory," J. ACM, pp. 201-215, 1960.
11. M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," Commun. ACM, vol. 5, pp. 394-397, 1962.
12. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the satisfiability (SAT) problem: a survey," DIMACS Ser. Discrete Math.}, pp. 19-152, 1997.
13. R. Bayardo Jr. and R. C. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," Proc. 14th Int'l Conf. Artificial Intelligence, pp. 203-208, 1997.
14. J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," IEEE Trans. Computers., vol. 48, no. 5, pp. 506-521, May 1999.
15. Y. Shang and B. W. Wah, "A discrete Lagrangian-based global-search method for solving satisfiability problems," J. Global Optimization, vol. 12, pp. 61-99, 1998.
16. DIMACS SAT benchmarks, ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/.
17. B. Selman, H. J. Levesque, and D. G. Mitchell, "A new method for solving hard satisfiability problems," Proc. AAAI-92, pp. 440-446, 1992.
18. B. Selman and H. Kautz, "Domain-independent extensions to GSAT: solving large structured satisfiability problems," Proc. 13th Int'l Joint Conf. Artificial Intelligence, pp. 290-295, 1993.
19. B. Selman, H. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," DIMACS Ser. Discrete Math., vol. 26, pp. 521-531, 1996.
20. S. Hampson and D. Kibler, "Large plateaus and plateau in Boolean satisfiability problems: when to give up searching and start again," DIMACS Ser. Discrete Math., vol. 26, pp. 437-455, 1996.
21. B. Jaumard, M. Stan, and J. Desrosiers, "Tabu search and a quadratic relaxation for the satisfiability problem," DIMACS Ser. Discrete Math., vol. 26, pp. 457-477, 1996.
22. W. M. Spears, "Simulated annealing for hard satisfiability problems," DIMACS Ser. Discrete Math., vol. 26, pp. 533-557, 1996.
23. N. Funabiki and J. Kitamichi, "A two-stage discrete optimization method for largest common subgraph problems," IEICE Trans. Inform. Systems, vol. E82-D, no. 8, pp. 1145-1153, 1999.
24. N. Funabiki and T. Higashino, " A minimal-state processing search algorithm for graph coloring problems," IEICE Trans. Fundamentals, vol. E83-A, no.7, pp.1420-1430, July 2000.

Table 1 Effects of global convergence schemes

instance	case 1	case 2	case 3	case 4
par16-2	10/10	10/10	8/10	10/10
par16-5	9/10	10/10	10/10	10/10
f2000	8/10	9/10	7/10	10/10
hanoi4	2/10	3/10	2/10	7/10

Table 2 Simulation results for hard instances.

instance	suc.	CPU time (sec)		
	runs	ave.	min.	max.
par16-1	10/10	46.64	0.968	123.2
par16-2	10/10	143.9	24.44	308.4
par16-3	10/10	66.59	0.257	197.5
par16-4	10/10	77.72	10.12	209.6
par16-5	10/10	144.52	41.90	343.4
par16-1-c	10/10	9.696	0.859	22.23
par16-2-c	10/10	25.62	6.617	77.15
par16-3-c	10/10	21.73	0.328	75.10
par16-4-c	10/10	20.36	4.230	58.13
par16-5-c	10/10	24.92	4.171	48.84
f600	10/10	0.571	0.312	1.273
f1000	10/10	7.939	0.945	15.41
f2000	10/10	66.90	4.937	166.5
g125.17	10/10	120.2	15.67	436.8
g125.18	10/10	12.84	12.20	13.85
g250.15	10/10	73.23	72.42	73.68
g250.29	10/10	548.3	342.4	1226.7
hanoi4	7/10	3609.1	643.2	9076.4