Multiagent SAT (MASSAT): Autonomous Pattern Search in Constrained Domains

Xiaolong Jin and Jiming Liu

Department of Computer Science,
Hong Kong Baptist University
Kowloon Tong, Hong Kong
{jxl, jiming}@comp.hkbu.edu.hk

Abstract. In this paper, we present an autonomous pattern search approach to solving Satisfiability Problems (SATs). Our approach is essentially a multiagent system. To solve a SAT problem, we first divide variables into groups, and represent each variable group with an agent. Then, we randomly place each agent onto a position in the correspoding local space which is composed of the domains of the variables that are represented by this agent. Thereafter, all agents will autonomously make search decisions guided by some reactive rules in their local spaces until a special pattern (i.e., solution) is found or a time step threshold is reached. Experimental results on some benchmark SAT test-sets have shown that by employing the MASSAT approach, we can obtain performances comparable to those of other popular algorithms.

Keywords: Autonomous Pattern Search, Satisfiability Problem (SAT), Multiagent System, MASSAT

1 Introduction

Many problems, both practical and theoretical, can be viewed as *Constraint Satisfaction Problems* (CSPs) where some constraints among a group of variables are given. The goal is to find the values for all variables that will satisfy the given constraints. Actually, to solve a constraint satisfaction problem is to search one or some special patterns (i.e., solutions) of the variable value combinations in the solution space which is specified by the given problem, and the patterns should satisfy all constraints. In this paper, we will describe a novel autonomous pattern search approach to solving *Satisfiability Problems*, a special class of Constraint Satisfaction Problems.

1.1 Satisfiability Problems

A *satisfiability problem* is a problem of determining whether or not there exists a solution for a propositional formula. The following is the formal definition of *satisfiability problem*:

Definition 1. A *satisfiability problem* (SAT), P, consists of:

1. A finite set of propositional variables, $\mathbf{X} = \{X_1, X_2,..., X_n\}$.
2. A domain set, $\mathbf{D} = \{D_1, D_2,..., D_n\}$, for all $i \in [1,n]$, $X_i \in D_i$ and $D_i = \{True, False\}$.
3. A clause set, $\mathbf{CL} = \{Cl(R_1), Cl(R_2), ..., Cl(R_m)\}$, where each R_i is an subset of X, and each clause $Cl(R_i)$ is a conjunction or disjunction of the literals corresponding to the variables in set R_i.

It is well known that any SAT problem P can be transformed into a Conjunctive Normal Form (CNF), i.e., $P = \wedge_{i=1}^{m} Cl_i$, where each clause Cl_i is a disjunction of variables or their negations. If l_i denotes the length of clause Cl_i, and $\forall i \in [1,m], l_i \leq k$, then we call this propositional formula a k-SAT problem. It has been proven that 2-SAT is solvable in linear time, and $\forall k \geq 3$, k-SAT is NP-Complete. So far, the SAT community has primarily focused on 3-SAT problems, and has found that in 3-SAT problems there is a phase transition phenomenon around $m/n = 4.3$. The really hard 3-SAT problems exist in this region. In this paper, we will consider only those really hard problems.

Generally speaking, the methods for solving SAT problems fall into two categories: *complete* methods and *incomplete* methods. Complete methods systematically examine an entire search space. For a satisfiable formula, they can guarantee to find a solution. At the same time, for an unsatisfiable formula, they can prove its unsatisfiability. On the contrary, incomplete methods cannot guarantee to check the entire search space. Therefore, they cannot guarantee to find a solution for a formula even though it is satisfiable.

Complete Methods Complete methods assign values to partial variables and then check if there exist some unsatisfied clauses. If this is the case, they will backtrack to one of the previous variables to assign it with another value, and then repeat this process. Or else, they will select a new variable to branch until a solution is found or the unsatisfiability of the problem is proven. The following are some typical complete methods: REL_SAT [1],TABLEAU [2], SATZ [7], and GRASP [10].

Incomplete Methods Many incomplete methods for solving SAT problems are based on *local search*. Local search is a recent method, which was proposed in [4][15]. A local search procedure starts with a complete and randomly initialized assignment, then checks if it satisfies all clauses. If not, it will randomly or heuristically select a variable to flip (i.e., change its value). It repeats this process until a solution is found or a certain threshold (e.g., a maximum number of steps) is reached. There are two main streams in local search studies, namely, GSAT [15] and WalkSAT [12][14]. They both have many variations, for example, GWSAT [14], GSAT/Tabu [11] of GSAT, and Novelty, R-Novelty [12], Novelty+, and R-Novelty+ [6] of WalkSAT.

Recently, two more effective local search methods have been proposed, they are: SDF [13] and UnitWalk [5]. Generally speaking, SDF makes a greedy descent on its objective function similar to other local search methods. But a special feature of SDF is that it not only counts the number of unsatisfied clauses, but also considers how strongly a clause is satisfied. In addition, SDF makes the use of a weighting technique to continue its search process, if a local optimum is encountered. Another latest method is UnitWalk, a special form of local search. In the UnitWalk method, Hirsch and Kojevnikov employ *unit clause elimination* to guide the local search process. At each step, the UnitWalk algorithm checks if there are unit clauses in the current formula. If so, unit clauses will be eliminated first. Or else, the UnitWalk algorithm selects a variable and substitutes its value. As compared to other local search Methods, both SDF and UnitWalk exhibit good performance.

We will see that our new approach belongs to the category of incomplete methods. We will also compare our approach with other incomplete methods mentioned above.

1.2 Multiagent Systems

A *Multiagent System* is a cooperative system in which several agents interact and work together in order to achieve certain goals. Agents in such a system may be homogeneous or heterogeneous, and may have common or distinct goals. Interested readers are refered to [8] for more details. Our new approach is essentially to build a multiagent system that solves SAT problems.

Here, we want to highlight a special example of multiagent system, *Swarm*. Swarm is a software platform for simulating distributed multiagent systems. It has three main components: *agents*, *environment* where agents live, and *actions schedule*. In the Swarm system, swarm is the basic unit, which is a collection of agents living in a certain environment and executing a schedule of actions. As a Swarm-based computing paradigm, Liu et. al. in [9] proposed an approach called *ERA* to solving *constraint satisfaction problems*. In the ERA approach, each agent represents a variable. All agents select values in a common environment that corresponds to the domains of all variables. In order to solve a given CSP problem, the agents will make a series of search decisions, governed by their own reactive rules, until a solution state is reached. Using this approach, they can achieve promising results on some benchmark CSPs, e.g., *n-Queen Problems* and *Coloring Problems*.

2 The MASSAT approach

In the section, we will describe the MASSAT approach in detail. The basic idea of our approach is to create a multiagent system where a group of agents are placed in an environment that is composed of the domains of all vairables. Combining all agents' positions, we can obtain an assignment to the given SAT problem. In order to solve the problem, all agents will autonomously make search decision guided by their reactive rules in their environment until all of them find an appropriate position.

For the ease of description, first of all, we assume that we can divide n variables into u groups. Without lose of generality, we further assume that $G_i = \{X_{i1}, X_{i2}, ..., X_{ik}\}$ is the i^{th} variable group, and the domain set corresponding to G_i is $\{D_{i1}, D_{i2}, ...D_{ik}\}$.

2.1 An Environment

In the MASSAT approach, the environment consists of local spaces. One local space belongs only to a unique agent. Other agents cannot enter it. Each variable group corresponds to a local space.

Definition 2. A *local space,* s_i, corresponding to variable group G_i is a row vector $< p_1, p_2, ..., p_{|D_{i1} \times D_{i2} \times ... \times D_{ik}|} >$, where $\forall j \in [1, |D_{i1} \times D_{i2} \times ... \times D_{ik}|]$, p_j is a cell called a *position.*

In local space s_i, there are $|D_{i1} \times D_{i2} \times ... \times D_{ik}|$ positions. p^i_j can index the j^{th} position in the i^{th} local space s_i.

Definition 3. An *environment,* E, is a column vector composed of all local spaces, i.e.,$E = < s_1, s_2, ..., s_u >^T = \begin{pmatrix} s_1 \\ s_2 \\ ... \\ s_u \end{pmatrix}$.

Generally speaking, environment E is a regular (if all local spaces s_i have the same number of positions) or irregular (if not all local spaces s_i have the same number of positions) grid board. In environment E, we use $e(j, i)$ to index the j^{th} position in local space s_i. It means $e(j, i) \equiv p^i_j$. For convenience, we also call s_i the i^{th} row, i.e., row_i of E. The positions in environment E are not only used to serve for an agent to stay, but also used for information: *domain values.*

Definition 4. The *domain value* recorded in position $e(j, i)$, i.e., $e(j, i).value$ is the j^{th} element in Cartesian product $D_{i1} \times D_{i2} \times ... \times D_{ik}$.

If agent a_i stays at $e(j, i)$, and $e(j, i).value =< x_{i1}, x_{i2}, ..., x_{ik} >$ (where $\forall x_{iq}, q \in [1, k], x_{iq} \in D_{iq}$), it means $X_{i1} = x_{i1}$, $X_{i2} = x_{i2}$,..., and $X_{ik} = x_{ik}$.

2.2 An Agent

Definition 5. An *agent, a,* is an entity with a special tuple, i.e., $a = (Var, Act, Eval)$, where Var is a group of variables that agent a represents; Act is a set of actions that agent a can take; $Eval$ is a function that is used by agent a to evaluate the positions in its local space.

Definition 6. The *evaluation value* related to a position $e(j, i)$, i.e., $e(j, i).evaluation$ is a number that reflects the goodness if agent a_i stays at position $e(j, i)$.

The *evaluation value* of a position evaluated by an agent dynamically changes with the positions of other agents, namely, for a position in s_i, if the positions of other agents are changed, the evaluation value of this position will possibly be changed.

Now, let us introduce some important details on agents' actions.

Moving strategies We use function Ψ to define an agent's movement.

Definition 7. The movement function for agent a_i:

$$\Psi : [1, |D_{i1} \times D_{i2} \times ... \times D_{ik}|] \to [1, |D_{i1} \times D_{i2} \times ... \times D_{ik}|]$$

$\Psi(x)$ gives the new position of agent a_i, after it moves from position $e(x, i)$. So, the new position can be represented as $e(\Psi(x), i)$.

In our approach, we design three basic move strategies: *best-move*, *better-move*, and *random-move*. At each time step, agent can probabilitically choose one to perform.

- *Best-move*

  Definition 8. *best-move*: An agent moves to a best-position with a probability of *best-p*.

  Definition 9. A position $e(j, i)$ in local space s_i is called a *best-position*, if $\forall x \in [1, |D_{i1} \times D_{i2} \times ... \times D_{ik}|], e(j, i).evaluation \succ e(x, i).evaluation$.

  where $e(j, i).evaluation \succ e(x, i).evaluation$ means $e(j, i).evaluation$ is better than or at least equal to $e(x, i).evaluation$. We will see that the specific meaning of '$\succ$' is dependent on that of 'evaluation value'.
  If there exists more than one best-position, we let the agent randomly choose one. Assume there are b_n best-positions, the best-move strategy can be expressed as: $\Psi_{best}(j) = AbsPos(Random(b_n))$, where function $Random(k)$ can uniformly generate a random number between 1 and k. $AbsPos(k)$ is a function that returns get the k^{th} best-position in local space s_i. In the best-move function, the number of computational operations to find the best position for each i is $|D_{i1} \times D_{i2} \times ... \times D_{ik}|$. This strategy is instinctive to all agents.
- *Better-move*

  Definition 10. *better-move*: An agent moves to a better-position with a probability of *better-p*.

  Definition 11. A position $e(j, i)$ in local space s_i is called a *better-position*, if $e(x, i)$ is agent a_i's current position, and $e(j, i).evaluation \succ e(x, i).evaluation$.

  To perform a better-move, an agent will randomly select a position and then compare its evaluation value with that of its current position to decide whether or not to move to this position. This movement can be defined by Ψ_{better}:

  $$\Psi_{better}(j) = \begin{cases} j & if \quad e(r, i).evaluation \succ e(j, i).evaluation \\ r & if \quad e(r, i).evaluation \prec e(j, i).evaluation \end{cases}$$

  where $r = Random(|D_{i1} \times D_{i2} \times ... \times D_{ik}|)$. Although a better-move may not be the best choice for an agent, the computational cost required for this strategy is less than that of best-move. Only two operations are involved for deciding this movement, i.e., producing a random number and performing a comparison.

– *Random-move*

Definition 12. *random-move*: An agent randomly moves to a position in its local space with a probability of *random-p*.

Random-p will be relatively smaller than the probabilities of selecting best-move and better-move strategies. It is somewhat like a random-walk in local search. For the same reason as in local search, the random-move strategy is necessary, because without randomized movements the system will get stuck in some local optima, that is, all the agents are at best-positions, but it is not a solution state. In the state of local optima, no agent will move to a new position if using the strategies of best-move and better-move alone. Thus, the agents will lose their chances for finding a solution if without any techniques to avoid getting stuck in local optima.
Random-move can be defined as Ψ_{random}:

$$\Psi_{random}(j) = Random(|D_{i1} \times D_{i2} \times ... \times D_{ik}|)$$

Evaluation After an agent selects a best-move or a better-move, it will evaluate all or some positions in its local space before it actually move. We define the evaluation function Φ of the agent as follows:

$$\Phi : \ [1, |D_{i1} \times D_{i2} \times ... \times D_{ik}|] \to R$$

where R is the set of real numbers.

In local search, the most important issue is how to select the next variable to flip. To do so, an appropriate function is needed to evaluate the current state. Similarly, in the MASSAT approach, how to select the next position in solution space to move is also important. A good evaluation function Φ is crucial, which can make a pattern search process clever and more efficient.

A commonly used evaluation function in the SAT-related local search methos is based on the number of satisfied clauses. The number of unsatisfied clauses can also be defined as an evaluation function. In the former case, if $e(x, i).evaluation \succ e(y, i).evaluation$, it means that if agent a_i stays at position $e(x, i)$, more clauses will be satisfied than if it stays at position $e(y, i)$. In the latter case, $e(x, i).evaluation \succ e(y, i).evaluation$ means that if agent a_i stays at $e(x, i)$, fewer clauses will be unsatisfied than at $e(y, i)$.

In this paper, we will use two different evaluation functions as two different cases, and compare their results in performance. The first one is the number of satisfied clauses, that is, $\Phi = \sum_{i=1}^{m} T(Cl_i)$, where $T()$ is a function that returns the truth value of a clause, m is the number of clauses. We have found that, by employing this evaluation function, the approach is easy to get stuck in local optima, and cannot efficiently escape from them. So, as a common local optimum escaping technique, we introduce weighting into our MASSAT approach. General speaking, a weighting technique adds a weight to each clause. The higher a clause's weight, the more chance it will be satisfied. Intuitively, given an assignment, an unsatisfied clause is hard to satisfy. Hence, if we increase its weight, it will get more chance to be satisfied. Adding a weight to each clause, the evaluation function will be changed to $\Phi = \sum_{i=1}^{m} W_i \cdot T(Cl_i)$, where $T()$ and m are the same as in the previous formula, and W_i is the weight of clause Cl_i which is updated according to the following function:

$$W_i^{t'+1} = \begin{cases} W_i^{t'} & if \quad T(W_i^{t'}) = True \\ W_i^{t'} + \delta & if \quad T(W_i^{t'}) = False \end{cases}$$

where t' records the weights updating steps. δ expresses certain learning from the past failure assignments. So we call it *learning rate*. The higher δ is, the faster the learning will be. Initially, we set the weights of all clauses to 1, i.e., $W_i^0 = 1$. Generally speaking, in our MASSAT approach, we can update the weights of the clauses with two different strategies, i.e., to update weights after a time step is over, and to update weights after a local optimum occurs. The first one is easy to implement, but we found it is inefficient to improve the performance. Different from the first one that updates the weights of the clauses unconditionally, the second one only updates the weights of the clauses when a local optimum occurs. In the MASSAT approach, after a time step, if there is no agent that moves to a new position, obviously, it indicates the algorithm has gotten stuck in a local optimum. The second strategy updates the weights of the clauses in this situation. We can readily see that in both evaluation functions, all agents will instinctively try to move to a position with a higher evaluation value.

To differentiate the above two MASSAT mechanisms, we refer to the one that adopts formula $\Phi = \sum_{i=1}^{m} T(Cl_i)$ as evaluation function as MASSAT1, and the one that employs formula $\Phi = \sum_{i=1}^{m} W_i \cdot T(Cl_i)$ as MASSAT2.

2.3 The System Schedule

At each time step, all agents are given a chance to take a certain movement. If an agent selects a best-move or a better-move, it must evaluate all or some positions in its local space, and then decides to move to a certain new position or stay. Because in the MASSAT approach, the configuration of the distributed agents represents an assignment to the current SAT problem, an agent must consider the positions of all other agents if it evaluates the positions in its local space. But, because all agents have a chance to move, the question becomes which positions of other agents should be considered in this situation. If the new positions of other agents are considered, the system can only be implemented in a sequential way. Therefore, to make all agents move asynchronously, we employ a common blackboard to record the positions of all agents before each time step. At a time step, an agent gets the positions of other agents from the blackboard, no matter the corresponding agents have moved to a new position or not. After a time step, the system will update the new positions of all agents in the blackboard.

3 Experimental Results

In the preceding section, we have provided a detail description of the MASSAT approach. In this section, we will present some experimental results on some benchmark SAT problems.

To evaluate the MASSAT approach in solving SAT problems, we have conducted experiments on two commonly used benchmark problem packages from SATLIB [16]: uf100 of *Uniform Random-3-SAT* and flat50 of *Flat Graph Coloring*. All instances in uf100 and flat50 are 3-SAT problems, and the ratio m/n is 4.3. In an instance of Uniform Random-3-SAT problems, each clause has exactly 3 literals which are randomly selected from n variables and randomly decide whether add them or their negations to this clause. In an instance of flat100, clauses may have different number of literals, but at most 3.

In [5], the authors presented their experimental results on benchmark SAT problem packages, include the above two packages. To compare with those of other algorithms, we conducted the same experiments on each package using the MASSAT approach. In Table 1, we list the mean and median numbers of movements that our MASSAT approach takes to get a solution. We also list some related data extracted from [5], i.e., the mean and median numbers of flips other algorithms take to get solutions of the same problem packages.

Table 1. Results on uf100 and flat50. Both uf100 and flat50 have 1000 instances. The parameter setting is: Variable group size 5; Ratio of probability between best-move and random-move 80:1; Learning rate 1.

	uf100		**flat50**	
Algorithm	**Mean Flips/Moves**	**Median Flips/Moves**	**Mean Flips/Moves**	**Median Flips/Moves**
GWSAT	6,532	3,398	7,023	4,846
WalkSAT	3,652	1,981	3,896	3,078
WalkSAT/Tabu	2,485	1,041	61,393	1,265
Novelty	15,699	962	24,421	934
R-Novelty	1,536	581	7,109	739
SDF	876	515	773	537
UnitWalk	2,547	1,862	261	200
MASSAT1	2,950	2,563	3,548	3,111
MASSAT2	1,462	1,198	982	780

From the results, we can see that MASSAT2 is much better than MASSAT1. In addition, although the performance of MASSAT2 is not as good as SDF and UnitWalk. But it outperforms the other algorithms.

Here, one important issue we should clarify is the rationale behind the comparison between the movements in the MASSAT algorithm and the flips in other algorithms. In general local search algorithms, such as those listed in Table 1, the number of *flips* is a commonly used index to evaluate the performance. In the MASSAT approach, because each agent represents a group of variables, a movement of an agent will cause, in an extreme case, n/u variables change their values. If we just consider the value changes occur on the variables, the comparison in Table 1 is unfair to other algorithms. But, in essence, what those algorithms should count is how many time clicks they take to get a solution rather than values changed. Here, 'time click' means the minimum time unit. In this sense, time click is the same as time step in the MASSAT approach. In those local search algorithms, because of their sequential nature, the number of time clicks is equal to the number of flips. So, the correct way to compare the MASSAT approach and other algorithms is to compare their time clicks. But, because the MASSAT algorithm as implemented in this paper is a sequential simulation of the actual MASSAT algorithm, we recorded the movements of all agents and compared them with those of other algorithms in Table 1. Although one movement may possibly cause multiple flips, all these flips happen *simultaneously*. So, we think comparing movements with flips in this way is fair.

4 Conclusion and Future Work

In this paper, we have described an autonomous pattern (i.e., solution) search approach to solving SAT Problems, namely, MASSAT. In solving a SAT problem with this approach, we first divide variables into groups, and then represent each group with an agent. The environment for the whole multiagent system contains all possible domain values for the problem. All agents can move within their own local spaces, which represent their domain, to search a special pattern. We have introduced three elementary moving strategies: *random-move*, *best-move* and *better-move*. The movement of an agent will affect the evaluation values of positions in the environment. After being randomly initialized, the MASSAT-based system will keep on dispatching agents, according to a certain order, to choose their movements until a solution is found or a time step threshold is reached.

We also presented experimental results on two commonly used benchmark SAT problem packages: uf100 and flat50. Our experimental results have shown that by employing the MASSAT approach, we can obtain comparable results to some other popular algorithms. But, the results presented in the paper is based on a non-optimal implementation of the MASSAT approach in solving SAT problems. In addition, we use a sequential way to simulate the MASSAT approach. This limits its parallel nature.

From our experiments, we have noticed that, there are some further aspects we should study. One appears to be very promising in improving our MASSAT approach is how to divide variables into groups. In the experiments provided in this paper, we equally partition all variables. We have found that different sizes of variable groups may lead to different results. It indicates that the performance of the MASSAT approach is relative to variable group. Intuitively, if we could partition the variables with some identical or similar 'properties' into a group, in this case, all variables in the same group can 'synchronizely' move, then the performance of the MASSAT approach will possibly be better.

References

1. B. J. Bayardo Jr. and R. C. Schrag, Using CSP Look-back Techniques to Solve Real-world SAT Instances, in *Proceedings of the 14th National Conference on Artificial Intelligence*, pp. 203-208, 1997.
2. J. M. Crawford and L. D. Auton, Experimental Results on the Crossover Point in Random 3SAT, *Artificial Intelligence*, Vol. 81, No. 1-2, pp. 31-57, 1996.
3. S.A. Cook, The Complexity of theorem proving procedures, in *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation*, pp. 151-158, 1971.
4. J. Gu, Efficient local search for very large-scale satisfiability problem, *SIGART Bulletin*, vol. 3, pp. 8-12, 1992.
5. E. A. Hirsch and A. Kojevnikov, UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, in *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, pp. 35-42, 2002.
6. H. H. Hoos, On the run-time behavior of stochastic local search algorithms for SAT, in *Proceedings of the 16th National Conference on Artificial Intelligence*, AAAI'99, pp. 661-666, 1999.
7. C. M. Li and Anbulagan, Look-ahead Versus Look-back for Satisfiability Problems, in *Proceedings of CP'97*, pp. 341-345, 1997.
8. J. Liu, *Autonomous Agents And Multi-Agent Systems: Explorations in Learning, Self-Organization and Adaptive Computation*, World Scientific, 2001.
9. J. Liu, H. Jing and Y. Y. Tang, Multi-agent oriented constraint satisfaction *Artificial Intelligence*, vol. 136, pp. 101-144, 2002.
10. J. P. Marques-Silva and K. A. Sakallah, GRASP - A New Search Algorithm for Satisfiability, in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 1996.
11. B. Mazure, L. Sais, and É. Grégoire, Tabu Search for SAT, in *Proceedings of AAAI'97*, pp. 281-285, 1997.
12. D. McAllester, B. Selman, and H. Levesque, Evidence for Invariants in Local Search, in *Proceedings of AAAI'97*, pp. 321-326, 1997.
13. D. Schuurmans and F. Southey, Local search characteristics of incomplete SAT procedures, *Artificial Intelligence*, vol. 132, no. 2, pp. 121-150, 2001.
14. B. Selman, H. A. Kautz, and B. Cohen, Noise Strategies for Improving Local Search, in *Proceedings of AAAI'94*, pp. 337-343, 1994.
15. B. Selman, H. Levesque, and D. Mitchell, A New Method for Solving Hard Satisfiability Problems, in *Proceedings of AAAI'92*, pp. 440-446, 1992.
16. http://www.intellektik.informatick.tu-darmstadt.de/SATLIB/.