Local Search Algorithms for Partial MAXSAT

Byungki Cha*, Kazuo Iwama*, Yahiko Kambayashi and Shuichi Miyazaki***

*Department of Computer Science
Kyushu University
Fukuoka 812, Japan
{cha, iwama, shuichi}@csce.kyushu-u.ac.jp

**Department of Computer Science
Kyoto University
Kyoto 606, Japan
yahiko@kuis.Kyoto-u.ac.jp

Abstract

MAXSAT solutions, i.e., near-satisfying assignments for propositional formulas, are sometimes meaningless for real-world problems because such formulas include "mandatory clauses" that must be all satisfied for the solution to be reasonable. In this paper, we introduce Partial MAXSAT and investigate how to solve it using local search algorithms. An instance of Partial MAXSAT consists of two formulas f_A and f_B, and its solution must satisfy all clauses in f_A and as many clauses in f_B as possible. The basic idea of our algorithm is to give weight to f_A–clauses (the mandatory clauses) and then apply local search. We face two problems; (i) what amount of weight is appropriate and (ii) how to deal with the common action of local search algorithms, giving weight to clauses for their own purpose, which will hide the initial weight as the algorithms proceed.

Introduction

Local search has already become an important paradigm for solving propositional CNF satisfiability. Since it was shown to be surprisingly powerful (Gu 1992; Selman *et.al* 1992), local search has been intensively discussed by many AI-researchers, mainly focused on how to escape from local minimas or "plateaus." Strategies to that goal include Maxflips (Selman *et.al* 1992), Random Walk (Selman and Kautz 1993), Simulated Annealing (Selman and Kautz 1996; Spears 1996) and Weighting (Morris 1993; Selman and Kautz 1993; Cha and Iwama 1995; Cha and Iwama 1996; Frank 1996). By these efforts, random 3CNF formulas (at the hard region) are now considered to be "solvable" up to some 10^4 variables, which is a great progress compared to some 10^2 variables at the beginning of 90's.

Then it is an obvious movement to try to take advantage of this remarkable development for solving real-world problems (e.g., Kautz and Selman 1996), namely, by reducing them into CNF satisfiability. One important merit of using CNF satisfiability rather than conventional approaches (typically based on integer programming or first-order refutation) is that the reduction is quite systematic and often straightforward: Simply speaking, all we have to do is to generate clauses so that each of them will become false if something bad happens. Particularly when there are a lot of "irregular" constraints, we can really enjoy this merit. However, real-world formulas, i.e., formulas reduced from real-world problems, are sometimes much tougher than random formulas: First of all, their size becomes unexpectedly large because propositional variables can hold only true or false. (Intuitively speaking, in order to simulate one integer variable that can take k different values, we need k propositional variables or $\log k$ ones if binary coding is possible.) Secondly, even more serious is that real-world formulas are often unsatisfiable because of too many constraints. This is not surprising since one tends to put more constraints assuming that it implies better solutions.

Therefore, it does not appear to be realistic to stick to complete solutions or satisfying assignments for large-scale, real-world formulas. Instead, what is realistic is to use MAXSAT which gives us better solutions or truth assignments that unsatisfy a smaller number of clauses. Of course, local search does work for MAXSAT (Selman and Kautz 1996) and in fact its main target was MAXSAT itself (Hansen and Jaumand 1990) before Selman et.al claimed that it was also useful to search complete solutions (Selman *et.al* 1992). Unfortunately, the simple MAXSAT approach has a significant drawback. The third and probably the most important feature of real-world formulas is that they include "mandatory" clauses whose unsatisfaction makes solutions meaningless. (In (Freuder et.al 1995), Selman says "a near-satisfying assignment corresponds to a plan with a "magic" step, i.e., a physically infeasible operation.") The number of such mandatory clauses is usually not large, often a fraction of the whole formula.

Now it is obvious why we need *Partial MAXSAT* (*PMSAT* in short) introduced in (Miyazaki et.al 1996): An instance of PMSAT is composed of two CNF formulas f_A and f_B over the same variable set. It requires us to obtain a truth assignment or a solution that satisfies all the clauses in f_A and as many ones in f_B as possible. Namely, the goodness value of the solution is the number of satisfied clauses in f_B. Then how can we solve PMSAT using local search? One simple idea is to repeat mandatory clauses, i.e., clauses in f_A, and then to apply local search for obtaining MAXSAT solutions. Actually we do not have to repeat clauses but can give weight to each clause.

The objective of this paper is to investigate how this simple idea works. We mainly focus on two problems: (i) What amount of weight given to mandatory clauses is appropriate? (ii) Local search algorithms also give weight to clauses for their own purpose. This weighting will hide the initial weight given to the mandatory clauses as the algorithms proceed, which might become a significant obstacle against our goal, i.e., trying to satisfy all mandatory clauses.

For experiments, we used not only random 3CNF formulas but also a typical real–world formula which is to obtain a class–schedule table of universities. The data were taken from the real database of CS department, Kyushu University (Miyazaki et.al 1996), whose size is not so large but somehow realistic (60 students, 30 courses, 13 faculties, 3 class rooms and 10 time-slots). The formula includes 900 variables and some 300,000 clauses out of which some 2000 ones are mandatory. This is obviously a nice example for claiming the usefulness of Partial MAXSAT.

Why Simple MAXSAT Does Not Work

We first take a look at why simple MAXSAT does not work using the class-schedule example. The notation in this paper is as follows: A *literal* is a (logic) *variable* x or its negation $\overline{x}$. A *clause* is a *sum* (*disjunction*) of one or more literals. A (CNF) *formula* is a *product* (*conjunction*) of clauses. A *truth assignment* is a mapping from variables into {*true*, *false*} or {1, 0}. A formula f is said to be *satisfiable* if there is a truth assignment which makes all the clauses true; such an assignment is called a *satisfying* truth assignment.

An instance of the class-scheduling problem consists of the following information: (i) A set Σ_S of students, Σ_P of professors, Σ_R of classrooms, Σ_T of timeslots and Σ_C of courses. (ii) Which courses each student in Σ_S wishes to take, e.g., student s wants to take courses c_1, c_2 and c_3. (iii) Which courses each professor in Σ_P teaches, (iv) Which timeslots each professor cannot teach, (v) Which courses each classroom cannot be used for (because of its capacity). (vi) Which timeslots each classroom cannot be used for, and so on.

Now we generate a CNF formula from this information. Let A, B and C be the numbers of the total courses, timeslots and classrooms. Then we use variables $x_{i,j,k}(1 \leq i \leq A, 1 \leq j \leq B$, and $1 \leq k \leq C)$. Namely $x_{i,j,k} = 1$ means that course i is assigned to timeslot j and room k. Hence a particular truth assignment into those variables $x_{i,j,k}$ can be associated with a particular class schedule. Here is the translation algorithm:

Step 1. For each $i_1, i_2, j, k (i_1 \neq i_2)$, we generate the clause $(\overline{x_{i_1,j,k}} \vee \overline{x_{i_2,j,k}})$, which becomes false if different courses i_1 and i_2 are taught in the same room at the same time.

Step 2. Suppose for example that professor p_1 teaches courses c_2, c_4 and c_5. Then for each $j, k_1, k_2 (k_1 \neq k_2)$, we generate the clauses $(\overline{x_{2,j,k_1}} \vee \overline{x_{4,j,k_2}}) \wedge (\overline{x_{2,j,k_1}} \vee \overline{x_{5,j,k_2}}) \wedge (\overline{x_{4,j,k_1}} \vee \overline{x_{5,j,k_2}}) \wedge (\overline{x_{2,j,k_1}} \vee \overline{x_{2,j,k_2}}) \wedge (\overline{x_{4,j,k_1}} \vee \overline{x_{4,j,k_2}}) \wedge (\overline{x_{5,j,k_1}} \vee \overline{x_{5,j,k_2}})$. If two courses (including the same one) taught by the same professor p_1 are assigned to the same timeslot and different rooms, then at least one of those clauses becomes false. We generate such clauses for each of all the professors.

Step 3. For each i, we generate the clause $(x_{i,1,1} \vee x_{i,1,2} \vee \cdots \vee x_{i,B,C})$ which becomes false if course c_i does not appear in the class schedule.

Step 4. Suppose for example, student s_1 wants to take courses c_1, c_3, c_5 and c_8. Then for each j, k_1, k_2, we generate $(\overline{x_{1,j,k_1}} \vee \overline{x_{3,j,k_2}}) \wedge (\overline{x_{1,j,k_1}} \vee \overline{x_{5,j,k_2}}) \wedge (\overline{x_{1,j,k_1}} \vee \overline{x_{8,j,k_2}}) \wedge (\overline{x_{3,j,k_1}} \vee \overline{x_{5,j,k_2}}) \wedge (\overline{x_{3,j,k_1}} \vee \overline{x_{8,j,k_2}}) \wedge (\overline{x_{5,j,k_1}} \vee \overline{x_{8,j,k_2}})$. If two of those four courses are assigned to the same timeslot, then one of these six clauses becomes false. Construct such clauses for all the students.

Steps 5 – 7. More clauses are generated by a similar idea according to the other constraints (omitted).

To obtain a specific benchmark formula, we used the real data of the CS department, Kyushu University. It involves 30 courses, 10 timeslots, 3 rooms, 13 professors and 60 students, where each professor teaches two or three courses and has two or three inconvenient timeslots. Each student selects eight to ten courses. It should be noted that this formula, say f, is probably unsatisfiable because the request of students is so tight (there are ten timeslots and many students select ten courses).

Now what happens if we try to solve this f using simple MAXSAT? According to our experiment, it was not hard to obtain a solution that unsatisfies only 15 clauses out of the roughly 300,000 total ones. It might seem good but actually not: The result, namely the obtained class schedule, did not include eight courses out of 30. Also, there were five collisions of two courses in the same room at the same time and so on. Namely the number of the unsatisfied clauses is small but most of them are mandatory; if one of them is not satisfied then the solution includes a fatal defect.

Partial MAXSAT

Recall that an instance of PMSAT is composed of two CNF formulas f_A and f_B. We have to obtain a solution (an assignment) that satisfies all the clauses in f_A and as many ones in f_B as possible. Generally speaking, there is an implicit assumption such that f_A must be "easy," because it would be otherwise hard to obtain any solution at all regardless of its goodness. One example of this easiness is that f_A includes either only positive literals or only negative ones which we call *uni-polar*.

It seems that PMSAT has a large power of "simulating" other combinatorial optimization problems even under the easy-f_A assumption: For example, there is an approximation preserving reduction from MAX-Clique to PMSAT with uni-polar f_A. (Note that MAX-Clique is one of the hardest optimization problems that is believed to have no approximation algorithms of approximation ratio $n^{1-\epsilon}$ or better (Hastad 1996).) More formally we can prove that there is a polynomial-time algorithm that, given a graph G, outputs (f_A, f_B) which meets the following conditions: (i) f_A is uni-polar. (ii) There is a polynomial-time algorithm that computes a clique C of G from a solution Z of (f_A, f_B) such that the approximation ratio of C is k iff the approximation ratio of Z is k.

The reduction is pretty straightforward: For a graph G with n vertices v_1 through v_n, we use n variables x_1 through x_n. f_A is the product of clauses $(\overline{x_i} \vee \overline{x_j})$ such that there is no edge between v_i and v_j. Note that f_A contains only negated variables. f_B is set to be $(x_1) \wedge (x_2) \wedge \cdots \wedge (x_n)$. Its correctness is almost obvious: From a solution (an assignment) of (f_A, f_B), we can compute a clique easily, namely, by obtaining a set of vertices v_i such that x_i is set to be true. Since all the clauses in f_A are satisfied, no two vertices v_i and v_j being unconnected are not in the set. In other words, the set of vertices constitute a clique. Note that it is widely believed that there is no such approximation–preserving reduction form MAX-Clique to the normal MAXSAT.

A bit harder example is a reduction from MIN-Coloring, also known as a hard problem (Bellare et.al 1995). This time, we use $n^2 + n$ variables $x_{i,j}$ $(1 \le i \le n,\ 1 \le j \le n)$ and z_j $(1 \le j \le n)$. Setting $x_{i,j} = 1$ means that vertex v_i is given color j. f_A consists of the following two groups of clauses: (a) For each $1 \le i \le n$ and different j_1 and j_2, $1 \le j_1, j_2 \le n$, we generate $(\overline{x_{i,j_1}} \vee \overline{x_{i,j_2}})$. (b) For each $1 \le j \le n$ and different i_1 and i_2, $1 \le i_1, i_2 \le n$, such that there is an edge between v_{i_1} and v_{i_2}, we generate $(\overline{x_{i_1,j}} \vee \overline{x_{i_2,j}})$. A clause in (a) becomes false if a single vertex is given two or more different colors and a clause in (b) becomes false if two connected vertices are given the same color. f_B is written as

$$\begin{aligned} f_B &= f_{B_{1,1}} \wedge f_{B_{1,2}} \wedge \cdots \wedge f_{B_{1,n}} \wedge f_{B_{2,1}} \wedge f_{B_{2,2}} \\ &\quad \wedge \cdots \wedge f_{B_{2,n}} \wedge (z_1) \wedge (z_2) \wedge \cdots \wedge (z_n) \end{aligned}$$

where each $f_{B_{1,i}}$ consists of the single clause $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n})$ and $f_{B_{2,j}}$ consists of n clauses $(\overline{x_{i,j}} \vee \overline{z_j})$ for $1 \le i \le n$.

This reduction preserves approximation in the following way: First of all, one can see that if an assignment satisfies all the clauses in f_A, then (i) each vertex is given at most one color and (ii) no two connected vertices are given the same color. But it can happen that no color at all is assigned to some vertex. We next show that if we wish to satisfy as many clauses in f_B as possible, what we should do is to satisfy all the clauses in $f_{B_{1,1}}$ through $f_{B_{1,n}}$. Suppose that a clause in $f_{B_{1,i}}$ is not satisfied. That means the vertex v_i is not given any color. Then we can satisfy it by setting $x_{i,j} = 1$ for some j such that $x_{i',j} = 0$ for all i', i.e., by giving a color j, which is not currently used, to that vertex v_i. Clearly this change keeps all the clauses in f_A satisfied, and by this change, at most one clause $(\overline{x_{i,j}} \vee \overline{z_j})$ in $f_{B_{2,j}}$ changes from satisfied to unsatisfied. Therefore the number of satisfied clauses does not decrease at least. Similarly we should satisfy all the clauses $f_{B_{2,1}}$ through $f_{B_{2,n}}$ because if there is an unsatisfied clause in $f_{B_{2,j}}$, we can satisfy it by setting $z_j = 0$. Again, this change keeps all the clauses in f_A and $f_{B_{1,1}}$ through $f_{B_{1,n}}$ satisfied, and since only one clause (z_j) becomes false, the number of satisfied clauses does not decrease.

Thus, without loss of generality, we have to consider only assignments that satisfy f_A, $f_{B_{1,i}}$ for all i and $f_{B_{2,j}}$ for all j. Then if color j is actually used for some vertex, then the variable z_j must be set to 0 since at least one $(\overline{x_{i,j}} \vee \overline{z_j})$ becomes false otherwise. Thus the clause (z_j) under this assignment is false. Otherwise, if the color j is not used then all $\overline{x_{i,j}}$, $1 \le i \le n$, are true by themselves and hence we can make (z_j) true. As a result, the number of unsatisfied clauses in f_B is equal to the number of colors that are needed in the proper coloring. In other words, satisfying more clauses in f_B means better coloring (fewer colors).

Solving Partial MAXSAT using Local Search

Our basic idea of solving PMSAT is quite simple: We repeat each clause in f_A (or equivalently we can give initial weight to clauses in f_A but we will seldom use this expression to avoid possible confusion between this weighting and the other type of weighting carried out by local search algorithms). Then we simply apply local search algorithms to obtain a solution that (hopefully) minimizes the number of unsatisfied clauses. As local search algorithms, we tested two popular ones in this paper; one is based on the so-called weighting method and the other is based on GSAT+Random-Walk. In more detail, the former program (developed by the authors) adds +1 to the weight of each clause that is not satisfied at the current assignment whenever the current assignment is a local minima. The latter was developed by Liang et.al (Liang et.al 1996) that is claimed to be one of the fastest GSAT-type programs.

Initial Weighting Strategies

As an extreme case, suppose for example that a single clause in f_A is repeated $K + 1$ times where K is the number of all the clauses in f_B and that f_A is "easy" to be satisfied. Then to satisfy one more (original) clause in f_A pays even if it makes all the clauses in f_B false in the sense that the number of unsatisfied clauses decreases by at least one. It is very likely that the local-search algorithm first tries to satisfy all the clauses in f_A and after that it then tries to satisfy others (i.e., in f_B) as many as possible while keeping f_A all satisfied. That is exactly what we want.

Unfortunately this observation is too easy: After reaching some assignment that satisfies all f_A, the local-search algorithm will never visit any assignment that makes f_A false because such an assignment increases the number of unsatisfied clauses too large due to the repetition of f_A–clauses. That means the search space is quite restricted, which usually gives a bad effect to the performance of local-search algorithms. This turned out to be true by the following simple experiment: Suppose that both f_A and f_B are random formulas and the number of clauses in f_A is one ninth the number of clauses in f_B. Also suppose that the local search has already reduced the number of unsatisfied clauses well, say, to 20 (i.e., f_A is expected to include two such clauses on average). Then it often happens that all the clauses of f_A are satisfied by chance without using any repetition of f_A–clauses at all. However, if we assure the satisfaction of f_A by the (heavy) repetition, then it becomes easier to satisfy f_A-clauses but it becomes very hard to reduce the number of unsatisfied clauses in f_B, say, to less than 50.

Thus, we do need the repetition but its amount should be minimum. Then how can we compute an appropriate amount of the repetition? The idea is to make some kind of balance between f_A and f_B. To do so, we first suppose that the current assignment is not too bad, i.e., almost all f_A–clauses are already satisfied and many f_B–clauses are also satisfied. Under this assumption, we can imply several conditions on the current value of each variable. Using this information, we can then compute the average number, say, N, of f_B–clauses that are currently satisfied but will become unsatisfied when we change the assignment so that one new f_A–clause will become satisfied. This number N is a good suggestion of the number of repetitions of each f_A–clause. An example of this calculation for the class-schedule formula will be given in the next section.

Restart and Reset Strategies

As shown in (Cha and Iwama 1995; Cha and Iwama 1996; Frank 1996), the weighting method is very fast in terms of the number of cell–moves until the algorithm gets to a satisfying assignment. However, it takes more time to carry out a single cell–move. Actually, the GSAT-type program by Liang et.al, which we will call LWM hereafter, can make five to ten cell–moves while our weighting-type program carries out a single cell–move. (This difference of performance is also due to implementation at least in part.) Thus it is not an easy question whether should be preferred.

When we use weighting–type local search for PM-SAT, special care must be needed: Recall that each f_A–clause is repeated, for example, ten times. However, the algorithm can give weight to any clause, either in f_A or in f_B, if that clause is unsatisfied at a local minima. Note that giving +1 weight has exactly the same effect as repeating that clause one more time. Also it should be noted that the total amount of weight given by the algorithm is surprisingly large especially when the program is run for long time. Therefore it can well happen that the initial repetition of f_A–clauses will soon be overwhelmed by the vast amount of weight given by the algorithm.

We can observe this phenomenon in Fig. 1, which shows how the number of unsatisfied clauses in f_A (denoted by A-Clauses) and in f_B (denoted by B-Clauses) changes as the algorithm proceeds. The formula used is a random 3SAT formula of 400 variables, 4000 (total) clauses and 400 f_A–clauses. Each f_A–clause is repeated 100 times. (We also obtained very similar data for a formula of 800 variables, 8000 clauses and 800 f_A–clauses.) As one can see, the number of unsatisfied f_A–clauses drops immediately to zero thanks to the repetition, but when the number of steps (cell– moves) increases, it leaves from zero at some moment, say, T, and never comes back to zero. Namely, the effect of the initial repetition of f_A–clauses dies at the moment T and it is totally nonsense to continue the algorithm after that.

Then what should be done at that moment T? A simple answer is to stop the current search and restart the algorithm completely from the beginning (i.e., from a randomly selected initial assignment). We call this version RESTART. Another possibility is to reset only the weight given by the algorithm so far and to continue the search from the current assignment. This is called RESET. The next question is how to decide this moment T. Since the moment depends on the total weight given by the algorithm so far, one reasonable way is to decide it by the number of local minimas visited by the algorithm so far. (Recall that the algorithm gives weight whenever it gets to a local minima.) It is denoted by an integer parameter, Maxflips, (as Maxflips = 100) which means that the algorithm restarts (or resets) after it has visited local minimas Maxflips times.

Experiments

As mentioned in the preceding section, we tested three algorithms, RESTART, RESET and LWM. The value of Maxflips was set to 400 since that is not too small compared to the moment T discussed previously. (Note that if Maxflips is too small, we may never be able to reach good solutions at all. So, it is safer to make this value large but it may lose the efficiency.) Random formulas are denoted as r$a-b-c$ where a, b and c are integers that show the numbers of variables, the whole clauses and f_A–clauses, respectively. We also used the class-schedule formula for which the clauses are divided into f_A and f_B as follows: f_A includes all the clauses generated at Steps 1 and 2 (for the obvious reason). The clauses generated at Step 3 are also important. However, if we put them into f_A, then f_A becomes not uni–polar. So, we did not do so but repeated $(x_{i,1,1} \vee x_{i,1,2} \vee \cdots \vee x_{i,B,C})$ K_i times where K_i is the number of students who select course c_i, and put them into f_B. This does not lose sense because if c_i is missing, then it gives inconvenience to only the K_i students. All the other clauses are put into f_B.

Let us take a look at how to calculate the appropriate repetitions of f_A–clauses in the case of this class-schedule formula: Suppose that we are now close to a good solution, namely the current assignment looks like the one such that for each i only one of $x_{i,j,k}$, $1 \leq j \leq B$ and $1 \leq k \leq C$, is 1 and all the others 0. Furthermore suppose that there is an f_A-clause, say, $(\overline{x_{i,1,1}} \vee \overline{x_{j,1,1}})$ that is now 0, i.e., both $x_{i,1,1}$ and $x_{j,1,1}$ are 1. All f_A-clauses must be satisfied. So let us see what happens if we try to flip the variable $x_{i,1,1}$ from 1 to 0. Apparently this f_A-clause $(\overline{x_{i,1,1}} \vee \overline{x_{j,1,1}})$ is satisfied and fortunately it does not cause any bad effect to other f_A-clauses (i.e., no such clauses change from 1 to 0).

We next observe how this flip (of $x_{i,1,1}$ from 1 to 0) changes the satisfaction of f_B-clauses. As for the all-positive-literal clauses generated in Step 3, $(x_{i,1,1} \vee \cdots \vee x_{i,B,C})$ is expected to change from 1 to 0. That does not happen actually if some variable other than $x_{i,1,1}$ is also 1, but this probability is low according to

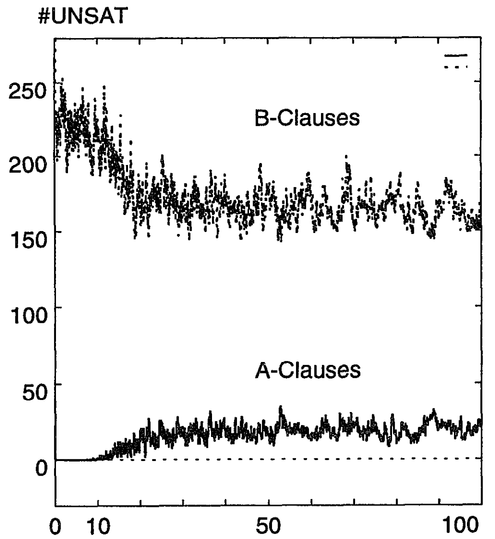

Fig. 1

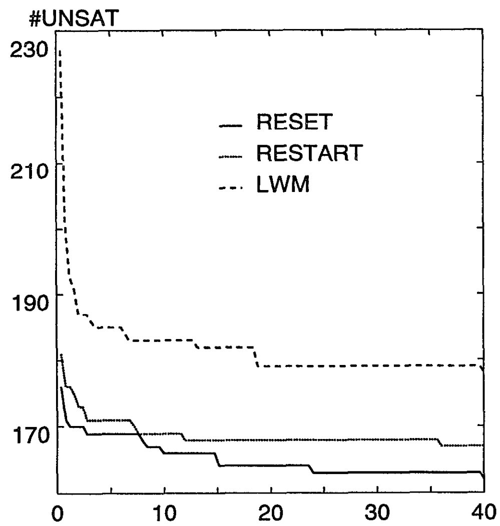

Fig. 2

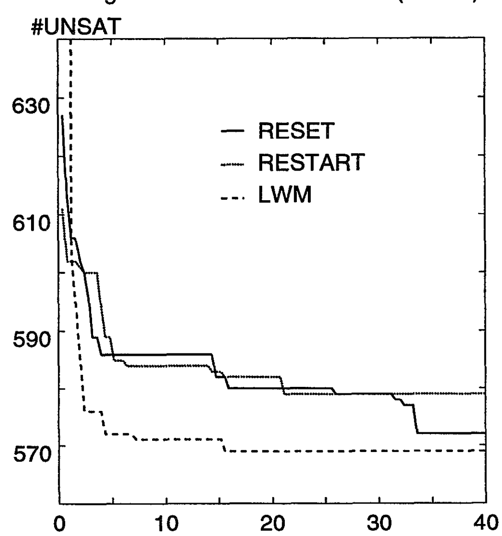

Fig. 3

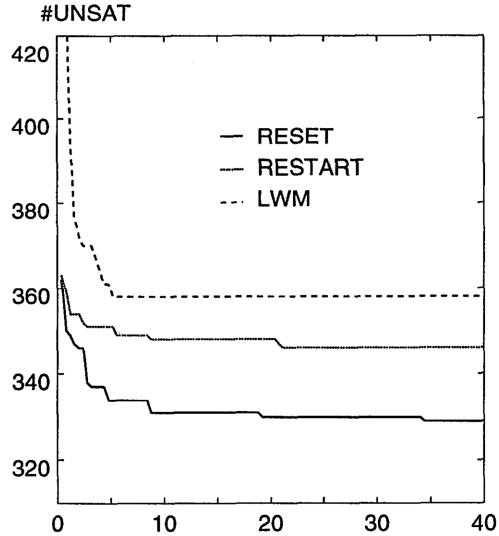

Fig. 4

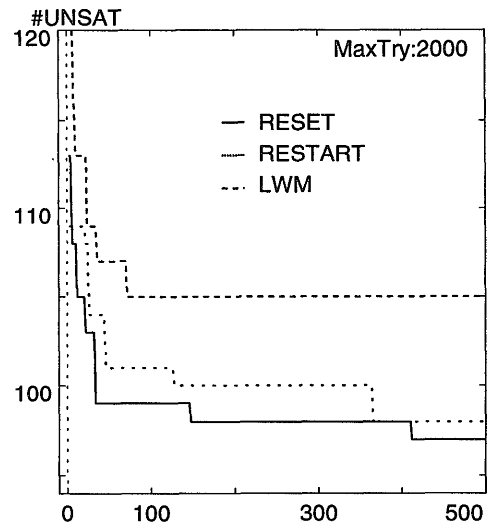

Fig. 5

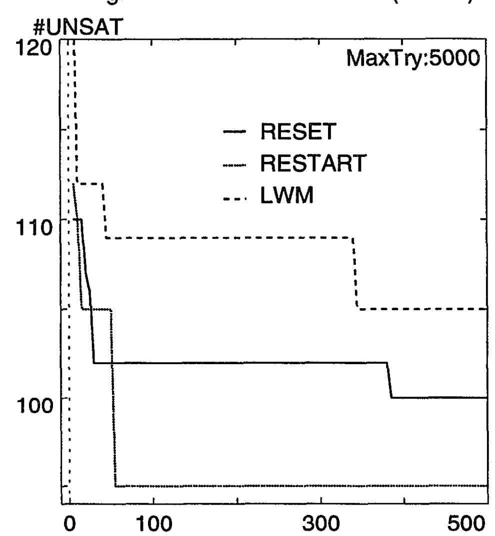

Fig. 6

the nearly-good-solution assumption mentioned above. Recall that this clause is repeated K_i times. As for the opposite clauses, i.e., the ones changing from 0 to 1, it is enough to only consider the clauses generated in Step 4 (others are negligible). Such clauses have the form of $(\overline{x_{i,1,1}} \vee \overline{x_{i',1,k}})$ and if this is 0, then it means some student among the K_i ones wishes to take both courses i and i' which collide at timeslot 1 in the current assignment. We can assume that the number of such students is only a fraction of K_i. It is not hard to see that $K_i = \frac{El}{A}$ on average where E and l are the number of students and the average number of courses selected by a student, respectively. (Recall that A is the number of courses.) Thus the number of satisfied clauses in f_B decreases by roughly $\frac{El}{A}$. As a result, flipping $x_{i,1,1}$ from 1 to 0 increases satisfied clauses by one in f_A but decreases by $\frac{El}{A}$ in f_B. This is the number N discussed previously, which is about 20 in our current example.

Figs. 2–4 show the performance of RESTART, RESET and LWM for random formulas, i.e., for r400-4000-400, r400-8000-800 and r800-8000-800, respectively. Each curve shows the number of unsatisfied f_B–clauses of the best solution (i.e., it satisfies all the f_A–clauses and most f_B–clauses) the algorithm has gotten by that number of steps (cell-moves). (This graph, in general, seems to be quite reasonable to show the performance of MAXSAT algorithms, which never appeared in the literature.) Each graph shows the average of results for four random formulas. The number of repetitions for each f_A–clause is 100 for all experiments. Generally speaking, RESET appears to be the best.

Figs. 5 and 6 show similar graphs for the class–schedule formula. The number of repetitions for f_A–clauses is 20 and Maxflips = 2000 (Fig. 5) and 5000 (Fig.6). The current best result is a solution that includes 93 unsatisfied (all f_B) clauses, which was obtained after some 5 million steps. We also tested the heavy weight (1000), but we were not able to get any solution that contains less than 115 unsatisfied clauses.

Remark. Very recently, Nonobe and Ibaraki (Nonobe and Ibaraki 1996) used our data for the class schedule and obtained a schedule table using a sophisticated CSP approach. Their result is slightly better than ours, i.e., the students have to abandon 86 courses (almost the same as 86 unsatisfied clauses in our case) in total, and they claim the result is optimal.

Concluding Remarks

Because of the time limit, we were not able to conduct several experiments including; (1) experiments to observe the effect of the number of repetitions of f_A–clauses, (2) experiments to investigate an optimal value for Maxflips, (3) experiments to test some different weighting strategies that are suitable to PMSAT and so on. It might be more important to investigate more basic requirements for solving PMSAT. For example, completely different approaches like backtracking may work better for PMSAT. Also there may be other methods to manage f_A–clauses than simply repeating them.

Acknowledgments

The first and fourth authors are Research Fellows of the Japan Society for the Promotion of Science and this research was supported by Scientific Research Grant, Ministry of Education, Japan, No. 0569 and 2273.

References

Bellare, M., Goldreich O. and Sudan, M. (1995). Free bits, PCPs and non-approximability - Towards tight results. *Proc. FOCS-95*, pp.422-431.

Cha, B. and Iwama, K. (1995). Performance test of local search algorithms using new types of random CNF formulas, *Proc. IJCAI-95*, pp.304-310.

Cha, B. and Iwama, K. (1996). Adding new clauses for faster local search, *Proc. AAAI-96*, pp.332-337.

Frank, J. (1996). Weighting for Godot: Learning heuristics for GSAT, *Proc. AAAI-96*, pp.338-343.

Freuder, E.C., Dechter, R., Ginsberg, M.L., Selman, B. and Tsang, E. (1995). Systematic versus stochastic constraint satisfaction, *Proc. IJCAI-95*, pp.2027-2032.

Gu, J. (1992). Efficient local search for very large-scale satisfiability problems, *Sigart Bulletin*, Vol.3, No.1, pp.8-12.

Hastad, J. (1996) *Proc. FOCS-96.*

Hansen, J. and Jaumard, B. (1990). Algorithms for the maximum satisfiability problem. *Computing*, 44, pp.279-303.

Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search, *Proc. AAAI-96*, pp.1194-1201.

Liang, D., Wu, Y. and Ma, S. (1996). Personal communication.

Miyazaki, S., Iwama, K. and Kambayashi, Y. (1996). Database queries as combinatorial optimization problems, *Proc. International Symposium on Cooperative Database Systems for Advanced Applications*, pp.448-454.

Morris, P. (1993). The breakout method for escaping from local minima, *Proc. AAAI-93*, pp.40-45.

Nonobe, K. and Ibaraki, T. (1996). Personal communication.

Selman, B. and Kautz, H.A. (1993). An empirical study of greedy local search for satisfiability testing, *Proc. AAAI-93*, pp.46-51.

Selman, B., Kautz, H. and Cohen, B. (1996). Local search strategies for satisfiability testing, *DIMACS Series in Discreet Mathematics and Theoretical Computer Science 26*, pp.521-531, 1996.

Selman, B., Levesque, H.J. and Mitchell, D.G. (1992). A new method for solving hard satisfiability problems, *Proc. AAAI-92*, pp.440-446.

Spears, W.M. (1996). Simulated annealing for hard satisfiability problems, *DIMACS Series in Discreet Mathematics and Theoretical Computer Science 26*, pp.533-557.