Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas

Byungki Cha and Kazuo Iwama
Department of Computer Science and Communication Engineering
Kyushu University, Fukuoka 812, Japan

Abstract

New types of test-instance generators have been developed for generating random CNF (Conjunctive Normal Form) formulas with controlled attributes. In this paper, we use these generators to test the performance of local-search-based SAT algorithms. For this purpose, the generator which produces formulas having exactly one satisfying truth assignment is especially desirable. It is shown that (i) among several different strategies of local search, the weighting strategy is overwhelmingly faster than the others and that (ii) local search works significantly better for instances of larger clause/variable ratio, which allows us to come up with a new strategy for making local search even faster.

1 Introduction

It is needless to say that selecting good test instances is crucial when evaluating the performance of combinatorial algorithms empirically. As a common practice, we usually include into the benchmark set both random and natural instances because both types of test instances have their own merits (and demerits). Natural instances of course reflect the real world but they often lack the generality. By contrast, random instances can be obtained in relatively simple ways, by which we can know the average performance of algorithms and its growth ratio as the size of instances grows. However, critics always say that they are too artificial.

In the case of local search algorithms for satisfiability testing [Gu, 1992; Selman *et al*., 1992], the nature of local search particularly makes difficult to use random test instances. The reason is not unique: Local search is incomplete algorithms, so we cannot use unsatisfiable predicates for testing. It inherently works very well for formulas having many satisfying truth assignments (solutions). Henceforth we should select instances having very few solutions to test its critical performance, but it is essentially hard in the case of random instances. The only way of trying to do so is to select formulas from the so-called cross-over region (the clause/variable ratio ~ 4.2 ~ 4.3 in the case of 3SAT formulas, see e.g., [Cheeseman *et al*., 199l]). However, no one knows reasonable ways of proving that such instances actually have sufficiently few solutions. (By simple experiments for formulas of 20 variables, we found that formulas at the cross-over region have usually ten or more solutions if they are satisfiable.) Thus one can hardly deny the criticism that random instances are too easy and are inadequate for local search algorithms. Ironically, many papers claim the merit of local search mainly using random instances [Morris, 1993; Selman and Kautz, 1993a; 1993b; Selman *et al*., 1992].

New types of test-instance generators, called *AIM generators* [Asahiro *et al*., 1993], have been developed to answer such criticism about random SAT instances. They are also random generators but accept several parameter values to control attributes of the generated instances. In the present situation, the most important attribute is the number of solutions. AIM generators include the generator, called k ONESAT-GEN, which produces kSAT formulas having exactly one solution over a wide range of clause/variable ratio. Also, another AIM generator, called ibSAT-GEN, can generate satisfiable instances at the clause/variable ratio when the ordinary random generator can never generate satisfiable instances.

In this paper, we use these new types of random instances for testing the performance of several algorithms based on local search. Two major results are as follows:

(1) Among several different strategies of local search, the weighting strategy is overwhelming faster than the others.

(2) Our test instances made clear that local search works better for instances of larger clause/variable ratio even though the number of solutions is very few. To exploit this nature, we propose a new strategy, namely, to modify given instances so that they will become easier for local search.

In the next section, we survey existing strategies of local search. In Section 3, AIM generators are briefly described. In Sections 4 and 5, we present our experimental work and show the two major results mentioned above.

2 Local Search Strategies

A *literal* is a (logic) *variable* x or its negation $\overline{x}$. A *clause* is a sum of one or more literals. A (CNF) *formula* is a product of clauses. A specific assignment of *true* (or 1) and *false* (or 0) to all the variables is called a *cell*. (Note that the terminology in this paper is associated with the Karnaugh map being popular in switching theory.) For cells C_1 and C_2, the *Hamming distance* between C_1 and C_2 is the number of variables for which the assignment is different (true for C_1 and false for C_2 or vice versa). C_1 is a *neighbor* of C_2 if their Hamming distance is one. It is said that a clause A *covers* a cell C if the truth assignment denoted by C makes A *false*.

The number of overlaps of a cell C, denoted by $OL(C)$, is the number of the clauses that cover C. A cell C is called a *local minimum* if $OL(C') \geq OL(C)$ for all neighbors C' of C. A cell is called a *solution* if $OL(C) = 0$.

Basically, local search for CNF satisfiability tries to reach a solution by gradually moving to cells of a smaller number of overlaps hoping that it can eventually get to a cell of zero overlaps. Its fundamental structure is as follows:

Algorithm Local_Search
  $C :=$ a randomly selected initial cell.
  until C becomes a solution **do**
  if C is not a local minimum **then do** α **else do** β

A number of basic operations have been proposed to be included in α and/or β. Those are summarized as follows:

(1) *Moving Downward*. This is the most basic operation and is only applicable when the current cell is NOT a local minimum. It moves to one of the neighbors whose overlaps (less than the current one) are the least (using random selection for tie break).

(2) *Moving to Minimum*. It moves to one of the neighbors where overlaps are the least whether or not the current cell is a local minimum.

(3) *Random Walk* [Selman and Kautz, 1993b]. This is also a moving rule; it moves to one of the neighbors such that there is at least one clause which is unsatisfied currently but becomes satisfied at the new neighboring cell. (This may even cause increased overlaps.) Usually there are many such neighbors, so we need some selection method. The easiest one is random selection. One can see that the Moving-Downward rule meets the condition of this Random-Walk.

(4) *Restart*. Stop the current search and restart it from a new randomly selected cell.

(5) *Weighting*. Introduced by several different researchers independently [Selman and Kautz, 1993a; Morris, 1993]. The idea is, roughly speaking, that if one gets stuck at a hole, i.e., a local minimum, one can get out of it by "filling" the hole. More formally, a *weight*, $w(C) > 0$, is associated with each clause C. Its value is initially one for all clauses. Now the definition of the number of overlaps is changed into the sum of weights of the clauses that cover the cell. Weighting is the operation which adds one to $w(C)$ of all the clauses C which cover the current cell.

Existing local search algorithms can be described as combinations of these building blocks. For example,

(a) *WEIGHT*: The recent algorithm based on weighting [Selman and Kautz, 1993a; Morris, 1993] is composed of $\alpha =$ Moving Downward and $\beta =$ Weighting.

(b) *GSAT*: The well-known algorithm proposed by Selman et al. [Selman *et al.*, 1992]: $\alpha = \beta =$ Moving to Minimum + Restart at every fixed number (= MAXFLIP in [Selman *et al.*, 1992]) of rounds.

(c) *GSAT + Random Walk*: Proposed in [Selman and Kautz, 1993b] $\alpha = \beta =$ Moving to Minimum with probability $p(\approx 0.5)$ + Random Walk with probability $(1-p)$.

(d) *RESTART*: The most basic local search. $\alpha =$ Moving Downward and $\beta =$ Restart.

As will be shown in Section 4, WEIGHT is overwhelmingly stronger than the others. We will make several attempts to improve the WEIGHT algorithm furthermore by (i) combining it with other heuristics such as prohibiting revisits of recently visited cells, (ii) changing the way of weighting such as using random values and (iii) changing the input formulas into (possibly) easier ones. Those will be discussed in Sections 4 and 5.

3 Random Formulas

AIM project [Asahiro *et al.*, 1993] currently holds four different types of random generators, called (i) kSAT-GEN, (ii) $k\overline{\text{SAT}}$-GEN, (iii) kONESAT-GEN and (iv) kFEWSAT-GEN. In this paper, we used (i) and (iii) and all the instances are 3SAT formulas, although the generators can also generate more general formulas.

kSAT-GEN generates satisfiable kSAT formulas with a specified literal distribution even of the specified distribution means for right side of the cross-over region where the conventional random generator can never produce satisfiable formulas.

Generator kSAT-GEN:

Input: A literal distribution $N = (N(x_1), N(\overline{x_1}), \cdots, N(x_n), N(\overline{x_n}))$, where $N(x_i)$ (resp. $N(\overline{x_i})$) denotes the number of occurrences of literal x_i (resp. $\overline{x_i}$).

Output: A satisfiable k-formula whose literal distribution is N.

Step 1. Compute the number t of clauses by $t = \frac{1}{k}\sum_{v=x_1,\overline{x_1},\cdots,\overline{x_n}} N(v)$. Initially set $f_{now} = (w_1)(w_2)\cdots(w_t)$ where each w_i is an empty clause.

Step 2. Create a single cell at random and let $C_{ans} = v_1 + v_2 + v_3 + \cdots + v_i + \cdots + v_n$ ($v_i = x_i$ or $\overline{x_i}$) be the clause that covers only that cell. In the following we also use C_{ans} to denote the set $\{v_1, v_2, \cdots, v_n\}$ of literals.

Step 3. If $\sum_{v \in C_{ans}} N(\overline{v}) \geq t$, then go to Step 5; otherwise go to Step 4 ($\overline{v}$ denotes $\overline{x_i}$ if $v = x_i$, and x_i if $v = \overline{x_i}$). Each clause must have at least one literal $\overline{v}$ such that $v \in C_{ans}$ to make it not cover the

cell C_{ans}. Hence, if $\sum_{v \in C_{ans}} N(\overline{v}) < t$, we cannot prepare t such clauses.

Step 4. Select an integer i from $\{1, \cdots, n\}$ at random such that $N(\overline{v_i}) < N(v_i)$ and then change $C_{ans} = v_1 + \cdots + v_i + \cdots + v_n$ into $v_1 + \cdots + \overline{v_i} + \cdots + v_n$. Go to Step 3.

Step 5. For $j = 1, \cdots, t$, select an integer i from $\{1, \cdots, n\}$ at random and put $\overline{v_i}$ such that $v_i \in C_{ans}$ into w_j. If the number of $\overline{v_i}$'s in $w_1, \cdots, w_{j-1}$ is already full, i.e., is equal to the limit denoted by N, then try again the random selection of i.

Step 6. Now in $f_{now} = (w_1)(w_2)\cdots(w_t)$, each w_i contains a single literal. It is convenient to consider a pool P of remaining literals, namely, P contains l_i x_i's where $l_i = N(x_i)$−{the number of x_i's already used in Step 5} and similarly contains m_i $\overline{x_i}$'s. Just select a literal at random from P and put it into w_j which does not contain k literals yet. □

As mentioned before, k is always three within this paper. Furthermore, we adopt as flat literal distribution as possible, which is specified by the clause/variable ratio. Namely, when the number of variables is n and the ratio is r, we first compute the number of clauses t by $t = \lceil n \cdot r \rceil$. Then the whole number of literals is $3t$ and each literal should appear at least $\lfloor 3t/2n \rfloor$ times. $3t - 2n \cdot \lfloor 3t/2n \rfloor$ different literals, selected at random, should appear one more time. When the ratio is larger than the cross-over ratio (4.2 ~4.3), the normal, random generation can only generate mostly unsatisfiable formulas but *k*SAT-GEN always generates satisfiable ones.

When the ratio is smaller than the cross-over ratio, the normal, random generation mostly generates satisfiable formulas. However, that cannot control the number of solutions, which is usually too many to test the performance of SAT algorithms. (Many algorithms, especially local search, can find a solution very quickly.) *k*ONESAT-GEN is convenient in such an occasion. Namely, it always generates satisfiable formulas that have only one solution at any clause/variable ratio. The generation algorithm is completely different from *k*SAT-GEN; it is rather based on the generator of *unsatisfiable* formulas as follows (because of the space problem, we cannot state the algorithm in detail, see [Asahiro *et al.*, 1993].):

Generator $\overline{\text{SAT}}$-GEN (Basic idea):

Step 1. Select a variable x at random and let the formula $f_{now} = x\overline{x}$ initially.

Step 2. One of the following (2-1) through (2-4) is randomly chosen and executed:

(2-1) Select a clause A in f_{now} and select a variable x both at random. f_{now} is modified by *splitting* A into $(A + x)(A + \overline{x})$.

(2-2) Select a clause A in f_{now} and a literal x' in A at random. f_{now} is modified by *deleting* (simply removing) x' from A.

(2-3) Select, again randomly, a pair of clauses A and B in f_{now} such that A covers B (if any). We remove the (same or smaller) clause B from f_{now}

(2-4) Construct a random clause A and add A into f_{now}.

Step 3. Note that each of *(2-1)-(2-4)* keeps unsatisfiability of f_{now} and therefore f_{now} is always unsatisfiable. So, repeat step 2 as many times as we wish, for example, until all clauses include three literals and sufficient number of clauses are generated. □

kONESAT-GEN can be obtained by modifying SAT-GEN as follows: We start with a simple initial formula having only one solution instead of $x\overline{x}$: (i) Similarly as kSAT-GEN, kONESAT-GEN first creates a random cell C_{ans}. Then it constructs the initial formula such that it has exactly one solution which is induced by C_{ans}. Let, for example, $C_{ans} = x_1 + \overline{x_2} + x_3 + x_4 + x_5$ for variables $x_1, \cdots, x_5$. Then the initial formula is $f_{initial} = (\overline{x_1})(x_1 + x_2)(x_1 + \overline{x_2} + \overline{x_3})(x_1 + \overline{x_2} + x_3 + \overline{x_4})(x_1 + \overline{x_2} + x_3 + x_4 + \overline{x_5})$ (one can easily verify that f_{now} becomes true only if one sets $x_1 = x_3 = x_4 = x_5 = 0$ and $x_2 = 1$). (ii) Just as SAT-GEN, kONESAT-GEN repeats to apply the four rules appropriately until the parameter values of the formula fit the specified one. It should be noted that these rules never increase the number of solutions. However, rules (2-2) and (2-4) may decrease the number of solutions, i.e., may change the formulas into unsatisfiable ones. Therefore, we have to be careful when applying those rules. Actually several details are needed to make the formula 3SAT and to make it satisfy the literal distribution. See [Asahiro *et al.*, 1993] for those details.

Instance generators based on the same idea, i.e., generating instances with controlled attributes, especially with predetermined solutions, have been developed for other problems like the Hamiltonian Circuit problem [iwama and Miyano, 1995a] and the logic optimization problem [iwama and Hino, 1994]. All these generators generate instances from their solutions. So, if one can reverse this solution-to-instance process, then it might allow "cheating". We have developed how to guarantee the security of generators in this sense [iwama and Miyano, 1993; 1995b].

4 Strength of Clause Weighting

We tested several local search algorithms using the random formulas described in the previous section. Random formulas are denoted by $tn(r)$, where t is r (pure random generation), o (generated by 30NESAT-GEN) or y (generated by 3SAT-GEN), n is the number of variables, and r is the clause/variable ratio. For example, o100(2.0) means random formulas of 100 variables, of 2.0 ratio and generated by 30NESAT-GEN, i.e., having exactly one solution.

Table 1 shows the performances of several algorithms. Each entry shows the average number of cell moves (the upper portion) and the average CPU time in second over SPARC Station 10 (the lower portion). The average was

taken for 100 instances. If an algorithm does not stop within specified MAX cell moves for some instances, we also give the percentage of successful computation after the number of cell moves. The MAX cell moves is 1,000,000 for o200(2.0), 250,000 for ol00(all ratio) and 25,000 for the others. Since our implementation of algorithms was not so polished, the figures for CPU time should be used only for relative comparison.

Since the local search algorithms depended on luck in principle, we cannot avoid a large diversity of computation time. To observe this diversity we conducted three experiments: Fig. 1(a) shows the distribution of cell-move steps for 100 different ol 00(2.0) instances. The rightmost shaded bar represents the number of instances that need more the 8,000 moves, which are three between 8,000 and 10,000, six between 10,000 and 15,000 and the worst one took almost 10^5 moves. Fig. 1(b) shows the same distribution for a *single* instance where 100 searches from randomly selected initial cells are carried out. Fig. 1(c) is also for a single instance where 100 searches are carried out from *the same* initial cell. (Note that we still use randomization for tie break in Moving Downward.) It might be interesting that three distributions are fairly alike.

The first four columns of Table 1 show the performance of the four existing algorithm described in Section 2. One can conclude that:

(1) Most importantly, WEIGHT is much faster than the other three algorithms.

(2) [Selman and Kautz, 1993b] claims that GSAT + rwalk is faster than GSAT, but it is only for pure random instances of 4.3 ratio. When the ratio is larger, it is slower than GSAT for y instances.

(3) Not surprisingly, single-solution instances are much harder than other types of instances. No algorithms but WEIGHT can cope with 50-variable instances which is of surprisingly smaller size than the result obtained using pure random instances [Morris, 1993; Selman and Kautz, 1993a; 1993b; Selman *et a*/., 1992]. One should note that it is a merit that we can use smaller-sized instances to test algorithms since they make it easier to analyze the behavior of algorithms for the purpose of possible improvements.

(4) Generally speaking, local search works better for instances of larger clause/variable ratio. This will be more considered in the next section.

Since WEIGHT is overwhelmingly good in its basic form, it is natural to try to seek its improvement rather than to try to find completely new strategy. The fifth column of Table 1, W+mflip, is the combination of WEIGHT and Restart. The last column, W+rcell, is combination of WEIGHT with *Prohibiting Recent Cells,* i.e., when moving to a neighboring cell, the cells that have been visited in the last ib rounds are excluded from the candidate. Revisiting the same cell many times appears to be redundant, which is suppressed by this rule.

Unfortunately both attempts do not seem to be successful. We learned by experiment that WEIGHT occasionally moves forward and backward between two cells until the number of overlaps at both cells increases sufficiently large. This action appears to be important, which is prohibited if one imposes the Prohibiting-Recent-Cell rule. (Note that in the case of GSAT, this rule certainly shortens the computation time as much as 50%.)

We also experimented with different weighting mechanisms. The basic type is (i) to add a unit weight (1) to each clause covering the current cell. Other ways experimented are: (ii) add 2 to each clause, (iii) add a real value to each clause that is minimum enough to get rid of the current local minimum, (iv) add a random value between 0.5 and 1.5 to each clause and (v) add 1 to a *single* clause selected at random. Again we were not able to see any important difference in the performance. For example, (steps, CPU time, success percentage) of (v) for o200(2.0) is (8194, 3186, 52) against (6375, 2916, 62) of the basic method above mentioned.

5 Increasing Clause/Variable Ratio

As discussed in the previous section, WEIGHT is much better than the other local search algorithms but its further improvement seems to be hard. Then what about changing not the algorithm but the instances so as to become easier for the algorithm? This might be hopeful if we can exploit the fact that local search generally works better for formulas of large clause/variable ratio.

Before discussing the way of increasing the ratio, what is the reason for the above fact? Fig. 2 shows the transition curve of the number of overlaps (without counting the weight) during the course of traversing the cells. (The curve was first used by Selman et al. [Selman and Kautz, 1993b]. In that paper, the curve is only decreasing for unknown reasons. In the present case it is no longer decreasing but includes many ups and downs.) In Fig. 2, (a) is the curve for ol00(2.0) and (b) is for ol00(4.7). Note that the average overlaps at each cell for (a) is 25 since a single clause of three literals covers 1/8 out of the total cells and there are 200 clauses in an ol00(2.0) instance. This average number is 58.75 for (b). See Fig. 2(a). The number of overlaps becomes under 5 very quickly and never becomes more than 5 after that until it gets to the solution. This few overlaps should be very rare, since the average is 25.

Hence one would first speculate that the algorithm only searches a small number of cells by moving back and forth. That is not very true: We counted the number of new cells (cells that have never been visited before) at each period of the execution of WEIGHT for ol00(2.0) instances. In near-average cases which need roughly 4000 cell-moves, the ratio of new cells is 81% for 0-1000th moves, 51% for 1000th-2000th moves, 64% for 2000th-3000th moves and 45% for 3000th-4000th moves. Thus it seems that there exists "a narrow valley" which eventually reaches the solution. Figs. 3(a) and 3(b) show the transition of the Hamming distance between the current cell and the solution corresponding to Figs. 2(a) and 2(b), respectively. One can see that this valley is quite long.

In the case of high clause/variable ratio, as shown in Fig. 2(b), this "valley" is not so deep or the number of overlaps in the valley is as large as between 20 and 30. Therefore, there seems to be a nice "slope" down to the solution which begins from a fairly distant place from the solution. Thus the probability of running across this slope is much higher than Fig. 2(a) where the valley is quite flat.

In order to increase the number of overlaps, the simplest way is to add random clauses. However, one should be careful because adding clauses destroy many solutions. What we propose in this paper is to add resolvents: Suppose that clauses *A* and *B* include exactly one common variable *x* and x appears affirmative in *A* and negative in *B*. Then, the *resolvent of A* and *B* is the clause which includes the literals of *A* except *x* and those of *B* except *x*. It is well known that adding resolvents does not destroy any solution.

For experiment, we used *o* 100(2.0). The number of added resolvents (mostly clauses of four literals) is changed from 100 to 400. Then the number of added clauses v.s. the number of cell-move steps and CPU time of WEIGHT is as follows:

100	2792(95)/374
200	2584(85)/423
300	2377(90)/400
400	2069(95)/438

Thus there is a clear tendency that WEIGHT runs faster as the number of added resolvents increases. It would be reasonable to claim that the number of cell moves, 2069 when 400 resolvents are added, is significantly less than 2838 of Table 1, since it was very hard to improve the figure by, say, even 10%, by many other attempts described in Section 4 (most of them increased the number of steps). One can notice, however, that the CPU time increases because it takes more time to compute the number of overlaps at neighboring cells. We need more experiments to claim that the decreased number of cell moves will become more important than the increase of computation time necessary in each cell move.

Thus this approach is somehow hopeful. However, we should try to find better ways of adding clauses: See Fig. 4 which illustrates, just as Fig. 2, the transition curve of overlaps when 300 resolvents are added. Although a lot of clauses are added, the "depth" of the valley is still very deep or we can find very little improvement from Fig. 2. Actually we tried some attempts such as adding a clause *A* + *B* for two clauses *A* and *B* such that they overlap with each other and the number of overlaps there is much less than the average. However, no good results are obtained at this moment.

6 Concluding Remarks

Our requirement for good test instances is twofold: One is that they can make clear the performance difference of different algorithms. The other, which might be more helpful, is that they can give us some hint to improve the algorithms. The main purpose of this paper is to claim that our new type of random instances can play both roles (Section 4 for the first role and Section 5 for the second role). Of course, there remain a lot of possibilities for further work in both roles.

An important question is whether local search is really better than Davis-Putnum-type algorithms and for what kind of instances it is so. For example, CSAT developed by Dubois et al. [Dubios *et al,* 1993], runs in almost the same number of branches for ol00(2.0) as the number of cell-visits of *WEIGHT.* Answering this question would be one of the most urgent requirement in this field.

References

[Asahiro *et al,* 1993] Y. Asahiro, K. Iwama and E. Miyano, Random generation of test instances with controlled attributes, *2nd DIM ACS Challenge Workshop,* 1993. (Also, to appear in Cliques, Coloring and Satisfiability, *DIM ACS Series in Disc. Math, and Theory Corny. Sci,* 1995)

[Dubios *et al,* 1993] O. Dubois, P. Andre, Y. Boufkhad and J. Carlier, SAT versus UNSAT, *2nd DIM ACS Challenge Workshop,* 1993.

[Gu, 1992] J. Gu, Efficient local search for very large-scale satisfiability problem, *SIGART Bull.,* Vol.3, pp.8-12, Jan, 1992.

[Morris, 1993] P. Morris, The breakout method for escaping from local minima, *Proc. AAAI-93,* 1993.

[Selman and Kautz, 1993a] B. Selman and H.A. Kautz, An empirical study of greedy local search for satisfiability testing, *Proc. AAAI-93,* 1993.

[Selman and Kautz, 1993b] B. Selman and H.A. Kautz, Local search strategies for satisfiability testing, *2nd DIM ACS Challenge Workshop,* 1993.

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

[Cheeseman *et al,* 1991] P. Cheeseman, B. Kanefsky and W. Taylor, Where the Really Hard Problems Are, *Proc. IJCAI-91,* pp.163-169, 1991.

[Iwama and Miyano, 1995b] K. Iwama and E. Miyano, "Intractability of read-once resolution," *Proc. 10th IEEE Conference on Structure in Complexity Theory,* 1995.

[iwama and Miyano, 1995a] K. Iwama and E. Miyano, "Better approximations of non-Hamiltonian graphs," *manuscript,* 1995.

[iwama and Miyano, 1993] K. Iwama and E. Miyano, "Security of test-case generation with known answers," *Proc. AAAI Spring Symposium Series,* 1993.

[iwama and Hino, 1994] K. Iwama and K. Hino, "Random generation of test instances for logic optimizers", *Proc. 31th ACM/IEEE Design Automation Conference,* pp.430-434, San Diego, 1994.

	RESTART	GSAT	GSAT+rwalk	WEIGHT	W+mflip	W+rcell
r50(4.3)	2859(75) 46	754 8	791 5	121 1	96 1	128 3
y50(4.3)	3268(90) 55	1059 12	1612(96) 11	178 2	177 2	159 3
y50(5.0)	1888(96) 35	907 12	1418(98) 11	134 1	118 1	126 2
y50(6.0)	432 12	617 10	1384(95) 13	101 1	96 1	93 1
y50(7.0)	317 10	457 8	1524(98) 16	94 1	96 1	81 1
o50(2.0)	25000(0) 157	4826(30) 30	5521(15) 21	1014(99) 6	878 9	910(99) 22
o50(3.4)	5889(26) 84	1979(99) 19	2267(97) 13	454 4	336 4	282 5
o50(4.7)	1895(97) 33	799 10	1441(90) 10	153 1	143 1	104 1
r100(4.3)	250000(0) 18935	12388(99) 965	3889 155	789 103	575 64	747 149
y100(5.0)	25000(0) 2036	1885 188	4086(76) 210	371 37	408 45	397 50
y100(7.0)	25000(0) 2527	929 139	2763(84) 212	218 32	194 30	216 38
o100(2.0)	250000(0) 18974	250000(0) 18837	250000(0) 9794	3923 279	21979 1259	6767(99) 924
o100(4.7)	250000(0) 25281	1878 169	15106(93) 713	286 28	279 28	267 32
o200(2.0)	1000000(0) 347181	1000000(0) 335278	1000000(0) 302753	6375(73) 2916	40725(57) 18159	10590(52) 4733

Table 1: Performance of several local search algorithms

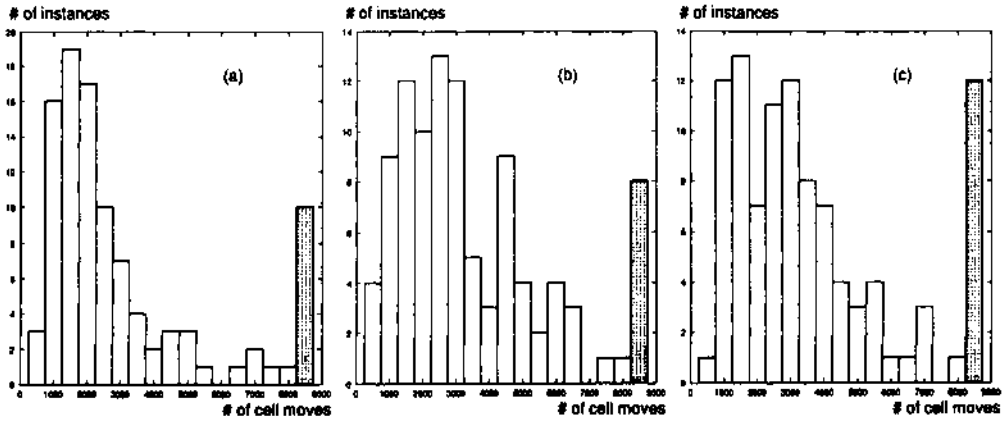
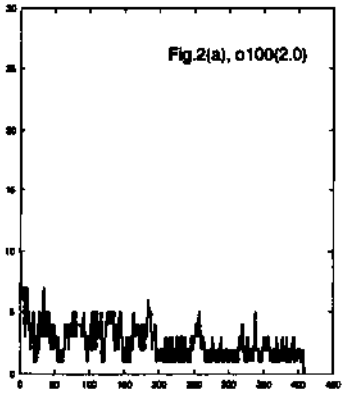


Figure 1: Diversity of cell-moves



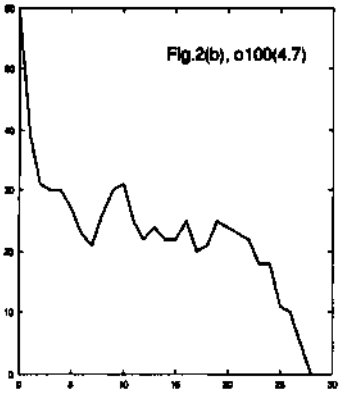
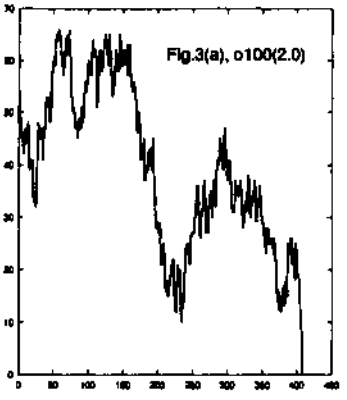


Figure 2: Transition of overlaps for o100(2.0) and o100(4.7)



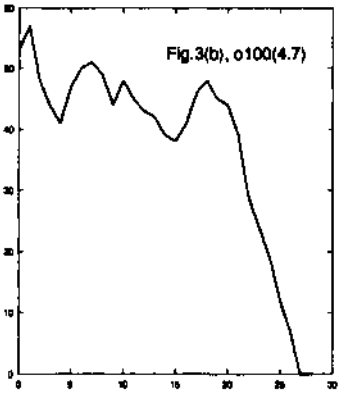


Figure 3: Transition of Hamming distance for o100(2.0) and o100(4.7)

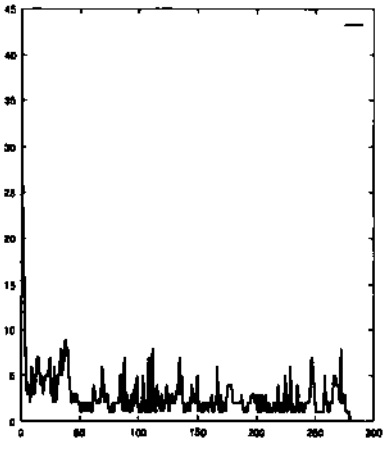


Figure 4: Transition of overlaps when 300 resolvents are added.