Analysis and Extension of the Inc* on the Satisfiability Testing Problem

Mohamed Bader-El-Den and Riccardo Poli

Abstract— Inc* is a general algorithm that can be used in conjunction with any local search heuristic and that has the potential to substantially improve the overall performance of the heuristic. The general idea of the algorithm is the following. Rather than attempting to directly solve a difficult problem, the algorithm dynamically chooses a smaller instance of the problem, and then increases the size of the instance only after the previous simplified instances have been solved, until the full size of the problem is reached. Genetic programming is used to discover new strategies for Inc*. Preliminary experiments on the satisfiability problem (SAT) problem have shown that Inc* is a competitive approach.

In this paper we enhance Inc* and we experimentally test it on larger set of benchmarks, including big instances of SAT. Furthermore, we provide an analysis of the algorithm's behaviour.

I. INTRODUCTION

Inc*, which was first introduced in [1], is a general algorithm that can be used in conjunction with any local search heuristic to improve its performance. The general idea of the algorithm is the following. Rather than attempting to directly solve a difficult problem, let us first derive a sequence of progressively simpler and simpler instances of the problem. Then, let us give the solver these instances one by one starting from the simplest, and progressing in the sequence only after all previous simplified instances have been solved. Note that the search is not restarted when a new instance is presented to the solver. Thus, the solver is effectively and progressively biased towards areas of the search space where there is a higher chance of finding a solution to the original problem.

While this is the fundamental idea, the Inc* framework goes one step further and makes the choice of the simplified problems dynamic. The objective of this is to limit the chances of the algorithm getting stuck in local optima. Whenever the system detects that one of the simplified instances in the chain leading to the original problem is too difficult, it backtracks and creates a new simplified instance in the attempt to continue the progression towards the goal problem instance.

In this paper, we will study the use of the Inc* algorithm on the satisfiability problem (SAT). However, Inc* is expected to be useful in many combinatorial optimisation problems: especially the ones that could be easily decomposed into smaller instances, such as the graph colouring,

Mohamed Bahy Bader-El-Den and Riccardo Poli are with the Department of Computing and Electronic Systems, University of Essex, Wivenhoe Park, Colchester, UK (email: {mbbade, rpoli}@essex.ac.uk).

the graph labelling and travelling salesman problems. In these problems, a simpler instance of the problem could be either a graph with smaller number of vertices, graph with a smaller number of edges, or both.

The SAT is NP-complete [2] and a classical combinatorial optimisation problem. The target in SAT is to determine whether it is possible to set the variables of a given Boolean expression in such a way to make the expression true. The expression is said to be satisfiable if such an assignment exists. If the expression is satisfiable, we often want to know the assignment that satisfies it. The expression is typically represented in Conjunctive Normal Form (CNF), i.e., as a conjunction of clauses, where each clause is a disjunction of variables or negated variables.

There are many algorithms for solving SAT. Incomplete algorithms attempt to guess an assignment that satisfies a formula. If they fail, one cannot know whether that's because the formula is unsatisfiable or simply because the algorithm was not run for long enough. Complete algorithms, instead, effectively *prove* whether a formula is satisfiable or not. So, their response is conclusive. They are in most cases based on backtracking. That is, they select a variable, assign a value to it, simplify the formula based on this value, then recursively check if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable and the problem is solved. Otherwise, the same recursive check is done using the opposite truth value for the variable originally selected.

The best complete SAT solvers are instantiations of the Davis Putnam Logemann Loveland procedure [3]. Incomplete algorithms are often based on local search heuristics (see Section I-A). These algorithms can be extremely fast, but success cannot be guaranteed. On the contrary, complete algorithms guarantee success, but their computational load can be considerable, and, so, they can be unacceptably slow on large SAT instances.

SAT heuristics use one of two main strategies for choosing the next variable to flip. The first strategy is to make a greedy move. In other words, the SAT solver chooses to flip the variable which transforms the current solution state to a state which is closest to a solution. The gain of the variable (see Section I-A) is typically the most important factor in selecting such a move, although also the age of the variable is sometimes used to avoid looping. The second strategy is to perform a random walk. This is done to avoid (or escape from) local optima. This is achieved by selecting a random variable to flip from a designated set of variables. There are different ways of choosing this set. For example, the set can include all the variables in the CNF formula, as

in GSAT, or just the variables in unsatisfied clauses, as in WalkSat. We look at these algorithms in more detail in the next subsections.

A. Stochastic Local-search Heuristics for SAT

Stochastic local-search heuristics have been widely used since the early 90s for solving the SAT problem following the successes of GSAT [4]. The main idea behind these heuristics is to try to get an educated guess as to which variable will most likely, when flipped, give us a solution or will move us one step closer to a solution. Normally the heuristic starts by randomly initialising all the variables in a CNF formula. It then flips one variable at a time until either a solution is reached or the maximum number of flips allowed has been exceeded.

GSAT [4] works as follows. At each iteration, it flips the variable with the highest gain score, where the gain of a variable is the difference between the total number of satisfied clauses after flipping the variable and the current number of satisfied clauses. The gain is negative if flipping the variable reduces the total number of satisfied clauses.

WalkSat [5] starts by selecting one of the unsatisfied clauses C. Then it flips randomly one of the variables that will not "break" any of the currently satisfied clauses (leading to a "zero-damage" flip). If none of the variables in C has a "zero-damage" characteristic, it selects with probability p the variable with the maximum score gain, and with probability $(1 - p)$ a random variable in C.

B. Incremental SAT

In a standard SAT algorithm the input is a problem instance and the target is to state whether this instance could be satisfied or not, and what are the variable assignment that satisfies it. In some cases it is also important to know if the instance could be still satisfied if further (arbitrary) Boolean clauses were added to the current set. This is known in the literature as incremental or dynamic SAT [6]. In incremental SAT the solver normally starts with a certain number of clauses and determines whether this set can be satisfied or not. In case it is satisfied, the solver gives the user the opportunity of adding more clauses to the existing set. The solver then checks whether the solution is still valid. If not, it attempts to repair it.

Most incremental SAT solvers are based on exact SAT algorithms as in [7], although some researchers have also used incomplete or heuristic-based solvers to deal with the incremental SAT problem [8]. The main problem with this second kind of solvers is that heuristics give no guarantee that a solution can be found. Their main advantage is speed.

C. Evolutionary Algorithms and SAT Problem

There have been a number of proposals of using evolutionary algorithms for SAT. An example is FlipGA which was introduced by Marchiori and Rossi in [9]. There a genetic algorithm was used to generate offspring solutions to SAT using standard genetic operators. However, offspring were then improved by means of local search methods. The same authors later proposed ASAP, a variant of FlipGA [10]. A good overview of other algorithms of this type is provided in [11].

GP has evolved competitive SAT solvers. For example, Fukunaga evolved local search heuristics [12], [13]. Also, GP has been used to enhance the performance of exact algorithms for SAT by helping the algorithm decide which variables to start the backtracking process with or to evolve heuristics for initialising dynamic decisions [14]. Furthermore, a general framework for evolving local-search 3-SAT heuristics, called GP-HH, has recently been proposed [15]. The aim there is to obtain "disposable" heuristics which are evolved and used for a specific subset of instances of a problem. Results were promising, with GP-HH evolving competitive heuristics.

D. Structure and Contributions of this Papers

In [1], we introduced an approach that presents some similarity with incremental SAT, but where the objective is to solve SAT problems, not incremental SAT problems. In particular, we used Genetic Programming (GP) [16], [17], [18] to investigate the benefits of dynamically changing the number of active clauses during the course of solving SAT problems. So, the solver is given a CNF formula including *all* the clauses from the beginning, but we give the solver the ability to decide which clauses to start with and in which order to tackle them. This will be explained in more detail later on in the paper.

In principle Inc* [1] is a general algorithm that can be used in conjunction with any local search heuristic and any structured combinatorial optimisation problem, although, so far, we have only applied it to the SAT problem, where, in preliminary experiments, Inc* appeared to be a competitive approach. No explanation for why Inc* improved performance of traditional local-search SAT solvers was provided. Also, the experiments were limited to relatively small instances of SAT and small instance sets.

This paper makes three main contributions: we enhance Inc*, we experimentally test it on larger set of benchmarks, including big instances of SAT, and, finally, we provide an analysis of the algorithm's behaviour and an explanation for why it provides performance improvements.

The paper is organised as follows. In Section II, we describe the basic Inc* algorithm and the GP system used to evolve strategies for Inc*. Then, in Section III we introduce some novel enhancements to Inc*. A description of the new sets of experiments we performed with the GP Inc* framework is given in Section IV. Finally, we draw some conclusions in Section V.

II. THE INC* FRAMEWORK

A. Principles Behind Inc*

As we mentioned above, Inc* is a general algorithm that can be used in conjunction with any local search heuristic to improve its performance. The general idea of the algorithm is the following. Rather than attempting to directly solve a

difficult problem, we first derive a sequence of progressively simpler and simpler instances of the problem. Then, we give the solver these instances one by one starting from the simplest, and progressing in the sequence only after all previous simplified instances have been solved.

The Inc* framework is particularly applicable to the SAT problem, where one can easily and dynamically create the necessary set of simplified problems. Effectively the algorithm starts by selecting a subset of the clauses in the formula. It then uses one of the SAT heuristics to test the satisfiability of this portion of the formula, which we call the *clauses active list*. Depending on whether or not the heuristic is successful on this portion of the formula, the algorithm then increases or decreases the number of clauses in the active list. In some cases, adding a clause has no effect on the satisfiability of the active list with the current variable assignment, so no additional flips are necessary. In other cases, more work is needed to find a new valid assignment.

More specifically, as shown in Algorithm 1, Inc* starts by initialising all the variables in the formula F randomly. It then activates an initial set of clauses by adding them to active clause list AC. The algorithm then runs one of the SAT local search heuristics. The heuristic is given a relatively small maximum number of flips at the beginning. However, this number is incremented gradually if the SAT solver fails to satisfy the AC, until, of course, the total number of flips used exceeds a predefined maximum (*MaxFlips*). A weight is assigned to each clause, which indicates how many flips have been necessary in order to satisfy the active list after the addition of the clause. After each run of the SAT heuristic, clause weights are updated. If the heuristic found a variable assignment L that satisfies the current AC, then the size of the AC is increased by adding new classes to it. Otherwise, the algorithm removes a small set of clauses from AC, giving preference to those with the lowest clause weight, and the number of allowed flips is increased, as previously mentioned.

Two key elements in determining the effectiveness of Inc* are the decisions taken in Steps 21 and 24 of Algorithm 1 as to how many clauses to add or remove from the active list after a success or failure, respectively. In our work, we have used GP to find optimal strategies to make these decisions. In the next section, we describe the GP system used and the evolved strategies.

Random or scheduled restarting of a heuristic is known to improve the heuristic overall performance [19], [20]. Typically, the restarting is simply done by assigning randomly either 0 or 1 to all the variables in a CNF formula and then starting the heuristic again. It is important to note here that this is not what Inc* does. If the heuristic failed to solve the current active list, Inc* changes the number of clauses in the active list, but does not alter the value of the variables. So, it restarts from where it left.

B. Inc* Optimisation via GP

As we mentioned above, an evolved strategy (which takes the form of a computer program) needs to decide how

---

Algorithm 1 Inc* approach to solving SAT problems

1: L = random variable assignment
2: AC = small set of random clauses from the original problem
3: $Flips$ = number of allowed flips at each stage
4: $Flips\_Total = 0$ {This keeps track of the overall number of flips used}
5: $Flips\_Used = 0$ {This keeps track of the flips used to test the active list}
6: Inc_Flip_Rate = rate of increment in the number of flips after each fail
7: **repeat**
8:   **for** $Flips\_Used = 0$ to Flips **do**
9:     **if** L satisfies formula F **then**
10:       **return** L
11:     **end if**
12:     select variable V from AC using some selection heuristic
13:     flip V in L
14:   **end for**
15:   Flips_Total = Flips_Total + Flips_Used
16:   update clause weights
17:   **if** L satisfies AC **then**
18:     **if** AC contains all clauses in F **then**
19:       **return** L
20:     **end if**
21:     AC = add more clauses to the active list
22:   **else**
23:     sort AC
24:     AC = remove some clauses from the active list
25:     $Flips = Flips * Inc\_Flip\_Rate$
26:   **end if**
27: **until** $Flips\_Total < MaxFlips$
28: **return** no assignment satisfying F found

---

many clauses the algorithm should add/remove to/from the active list after each success or failure at finding a variable assignment that satisfies the current active clauses of the full SAT formula. The block titled "run a GP individual" in figure 1 shows exactly where the evolved strategy is placed in the Inc* algorithm.

We use a tree-based representation for programs. The function and terminal sets are shown Table I. We constrain the representation by requiring that the root node of each individual in the population be the binary function $ifSuccess(d_1, d_2)$, where d_1 and d_2 are of type real. This function returns the integer part of its first argument, $\lfloor d_1 \rfloor$, if the last run of the SAT heuristic was successful at satisfying the current AC. If this is the case, the value $\lfloor d_1 \rfloor$ is taken to represent how many clauses should be added to AC.[1] If, instead, the SAT heuristic failed to satisfy AC, then $ifSuccess$ returns the value $\lfloor d_2 \rfloor$, which is taken to represent how many

---

[1]Note that, to give complete freedom to evolution, negative return values are allowed. If $\lfloor d_1 \rfloor$ is negative clauses are removed, rather than added, from AC.
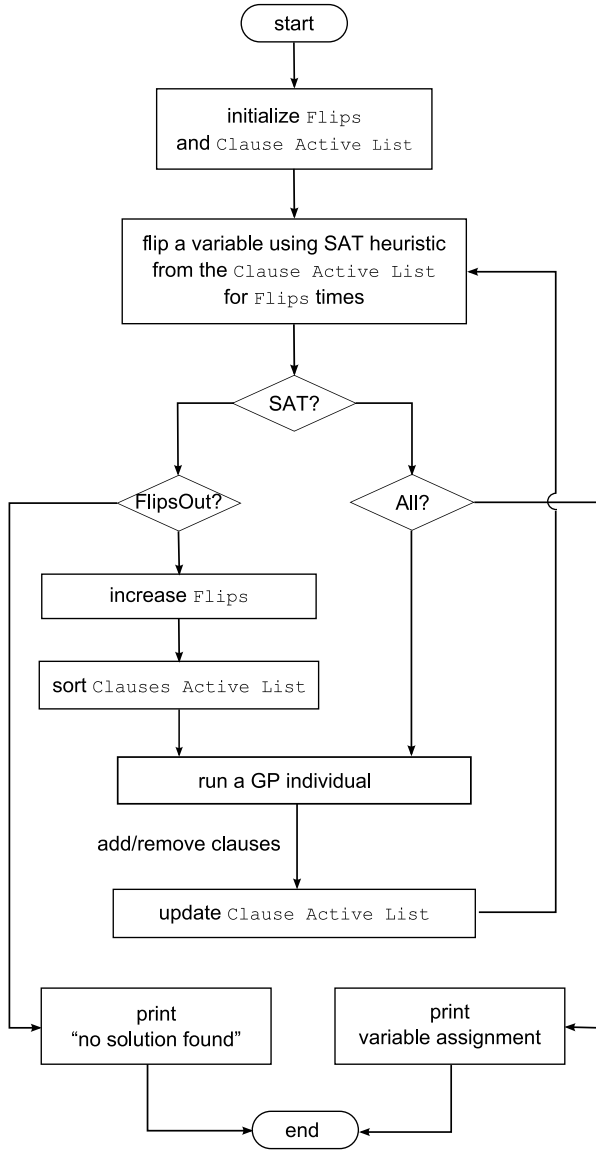
Fig. 1. A flowchart showing the main steps in the inc*/inc** algorithm, and where the GP individual (strategy) is placed in the algorithm.

clauses should be removed from *AC*.

The other elements of the primitive set behave as follows. The functions $add(d_1, d_2)$, $sub(d_1, d_2)$, $mul(d_1, d_2)$ and $div(d_1, d_2)$ are the standard arithmetic operations (note, division is protected to avoid division by 0 errors). The primitive $neg(d_1)$ inverts the sign of d_1. The terminals *vNo* and *cNo* return the total number of variables and the total number of clauses in the full SAT formula, respectively. The terminals *used_cNo* and *used_vNo*, instead, return the number of unique variables and the number of clauses currently loaded in the active list, respectively. Finally, *constX* represent random integers between 0 and 9.

In order to evolve general Inc* strategies, we used a training set including many SAT problems with different numbers of variables. The problems were taken from the widely used SatLib benchmark library. All problems were randomly generated satisfiable instances of 3-SAT. In total we used 50 instances: 10 with 100 variables, 15 with 150 variables and 25 with 250 variables. The fitness $f(s)$ of an evolved strategy *s* was measured by running the Inc* algorithm under the control of *s* on all the 50 fitness cases. More precisely

$$f(s) = \sum_i \left(inc_s(i) * \frac{v(i)}{10} \right) + \frac{1}{flips(s)}$$

where $v(i)$ is the number of variables in fitness case *i*, $inc_s(i)$ is a flag representing whether or not running the Inc* algorithm with strategy *s* on fitness case *i* led to success (i.e., $inc_s(i) = 1$ if fitness case *i* is satisfied and 0 otherwise), and $flips(s)$ is the number of flips used by strategy *s* averaged over all fitness cases. The factor $v(i)/10$ is used to emphasise the importance of fitness cases with a larger number of variables, while the term $1/flips(s)$ is added to give a slight advantage to strategies which use fewer flips (this is very small and typically plays a role only to break symmetries in the presence of individuals that solve the same fitness cases, but with different degrees of efficiency).

There is only one exception to this fitness calculation. In the system we keep a count of the number of attempts the SAT solver made at solving the *AC* list. If a maximum number of tries is reached, fitness is computed differently. Imagine, for example, what would happen if an evolved strategy added zero clauses after each successful attempt and removed zero clauses after each unsuccessful one. After a small number of flips had been expended to satisfy the initial active clauses, no further flips would ever be necessary (since no clauses are added or removed). So, the total number of flips used would never reach the maximum number of flips allowed, leading to an infinite loop. By using a maximum number of tries, we avoid this and we signal to the system that this individual (strategy) went into an infinite loop on the current fitness case. The system reacts by setting the fitness of this strategy to zero and stopping the evaluation of any remaining fitness cases.

The GP system initialises the population by randomly drawing nodes from the function and terminal sets. This is done uniformly at random using the GROW method, except that the selection of the function *ifSuccess* is forced for the root node and is not allowed elsewhere. After initialisation, the population is manipulated by the following operators:

- Roulette wheel selection (proportionate selection) is used. Reselection is permitted.
- The reproduction rate is 0.1. Individuals that have not been affected by any genetic operator are not evaluated again to reduce the computation cost.
- The crossover rate is 0.8. Offspring are created by generating a copy of a random subtree from the first parent and inserting it at a random point (excluding the root of the tree) in a copy of the second parent.
- Mutation is applied with a rate of 0.1. This is done by selecting a random node from the parent (including

Function Set		
$ifSuccess(d_1,d_2)$	:	returns d_1 if the last attempt to solve the formula was successful
$add(d_1,d_2)$	:	returns the sum of d_1 and d_2
$sub(d_1,d_2)$	:	subtracts d_2 from d_1
$mul(d_1,d_2)$	:	returns the multiplication of d_1 by d_2
$div(d_1,d_2)$	:	protected division of d_1 by d_2
$abs(d_1)$	:	returns the absolute value of d_1
$neg(d_1)$	:	multiplies d_1 by -1
$fneg(d_1)$	:	$abs(d_1)$ multiplied by -1, to force negative value
$sqrt(d_1)$	:	returns the a protected square root of d_1
Terminal Set		
vNo	:	total number of variables in the formula
cNo	:	total number of clauses in the formula
$used\_cNo$	:	number of currently active clauses
$used\_vNo$	:	number of currently active variables
$constX$	:	random integer number from 0 to 9

the root of the tree), deleting the sub-tree rooted there, and then regenerating it randomly as in the initialisation phase.

Figure 2 shows Some of the best evolved strategies, the strategies were manually edited and normalized for display purposes, the code for some of the best performing heuristics is so large and complex to show here.

III. INC**

In this section we introduce Inc**, a novel and enhanced version of the Inc* algorithm. We have modified two main behaviours on the Inc* algorithm.

The first modification is the weighting mechanism of the clauses. In the Inc* algorithm, only the weight of newly added clauses are updated, and the update is done only after the heuristic fails to satisfy the current active clauses, as described before. In inc**, the weight is modified as in Inc*, but, additionally, the weight of each unsatisfied clause is increased by one after each flip. Furthermore, the weight of all active clauses are multiplied by δ (with $0 < \delta < 1$) after each successful try in satisfying the active list, thereby decreasing the weight of this combination of clauses after they have been successfully satisfied.[2]

The second modification in Inc** regards the flip increment rate. In Inc*, the number of allowed flips is increased by a constant percentage. This sometimes causes Inc* to use more flips to solve some solvable instance than the standard heuristics without Inc* (this is explained the next section in more detail). In order to increase the number of flips more rationally, since we increase the number of allowed flips only

[2]The idea of increasing the weight of unsatisfied clauses is not entirely new. Indeed, we took inspiration from the Dynamic Local Search (DLS) heuristic [21]. However, weights in DLS are used in a completely different manner in DLS. In DLS, the target is to flip the variable which will minimise the weight of all unsatisfied clauses. This is more widely used in MAX-SAT than SAT. Updating and maintaining the weight in this category of algorithms is more sophisticated and computationally expensive than in Inc**, where the clause weight is only used to guide which clauses to add or remove from the active list.

on after each fail to satisfy the current active list, in Inc** we increase the number of allowed flips in proportion to the number of unsatisfied clauses in the active list. This simple modification allows the algorithm to change its behaviour depending on the size of the SAT instance being solved and the hardness of the current active list.

For example, consider the case of a large instance and assume we have many clauses in the active list but the current number of allowed flips is relatively small. In this case, if the number of unsatisfied clauses is large, Inc** grants the next try with this instance more flips than Inc* would. Conversely, if the number of unsatisfied clauses was small, Inc** gives a smaller number of maximum allowed flips to the next iteration, which makes sense since probably these clauses could be satisfied with fewer flips.

IV. EXPERIMENTAL RESULTS

The results are presented in two section. In Section IV-A we will show comparative results between the proposed algorithm combined with WalkSat, and WalkSat alone. In Section IV-B we will provide results on the behaviour of the algorithm on instances with different structural characteristics (namely, SAT backbone size).

A. Comparative Results

In these experiments we used a population of 1,000 individuals, run for 51 generations. While strategies are evolved using 50 fitness cases, the generality of best of run individuals is then evaluated on an independent test set including 500 SAT instances. In this section, will show a comparison between the performance of a standard handcrafted heuristic, WalkSat, and the same heuristics when combined with Inc* and Inc** controlled by strategies evolved by GP.

We have used the following parameters values for the Inc* algorithm:[3]

- We allow 100 flips to start with, and 2,000 for instances with more than 250 variables.
- Upon failure, the number of flips is incremented by 20%.
- We allow a maximum total number of flips of 100,000, and 400,000 for instances with more than 250 variables.
- The maximum number of tries is 1,000 (including successful and unsuccessful attempts).

The GP system has managed to evolve a number of successful strategies. Most of these can be categorised into three groups. In the first group, strategies start by activating a relatively small number of clauses w.r.t. the total, after which they then rapidly increase the number of active clauses. This was almost always the best performing group. In the second group, strategies start by activating a very large number clauses at the beginning, then they remove some clauses after each fail and try to go forward again until a solution for all clauses is found. Strategies in this category perform slightly worse than those in the first category. Strategies

[3]Many different combinations of parameter values have been tested, but this particular combination gave almost invariably the best results.

in the third group were generally outperformed by those in the other groups. Strategies in this group acted in an unexpected manner. Namely, these strategies kept moving forward, adding clauses after both successful and unsuccessful tries. In the testing phase, this kind of strategies performed well on instances with fewer than 100 variables in terms of number of flips used to solve the instance. However, they had a lower success rate than other strategies on larger instances. The reason of this will be explained after showing detailed results of the strategies.

Table II shows the results of a set of experiment using WalkSat and a combination of Inc* and Inc** with WalkSat. Instances with up to 250 variables were taken from SatLib. On these instances, heuristics where given a maximum of 100,000 total flips. Larger instances were taken from the benchmark set of the SAT 2007 completion and where given a maximum of 400,000 total flips. None of the test instances had been used in the GP training phase . The performance of the heuristics on an instance is the number of flips required to solve it averaged over 10 independent runs of a solver, to ensure the results are statistically meaningful. The AF column shows the average number of flips used by each heuristic in successful attempts only.

We categories the results in this table into two groups. The first group includes instances with no more than 100 variables. The second group includes instances with more than 100 variables. In the first group of problems all heuristics have a perfect success rate of 100%. While WalkSat used a slightly smaller number of flips than Inc* on this group of problems, Inc** was able to outperform both algorithms on instances with 20, 50 and 75 variables. In the second group of problems, which contains larger instances, however, Inc* and Inc** have a higher success rate than WalkSat, and the difference in the performance increases as the size of the instances increases, figure refg1 shows how the gap increases by the increase of the size of the instances. This means that Inc*/Inc** can solve complex instances where local heuristics alone fail. Again, Inc** tends to perform better than Inc*.

This explains why, when training the GP system on small instances, some evolved strategies (the strategies in group three) always tried to go foreword, adding more clauses after both successful and unsuccessful tries, as we mentioned above. Effectively, these strategies tried to imitate the standard heuristics behaviour, and, indeed, they were slightly faster on small instances. Table II also shows that Inc* and Inc** perform much better on more complex instances with larger number of variables, and that the gap in the performances increases too.

B. Behaviour on SAT Instances with Different Backbone Size

SAT problems may have multiple solutions. The backbone of a SAT problem consists of those variables for which logical values are the same in all possible solutions. Instances with large backbone sizes are known to be more difficult to solve for local search heuristics than instances with smaller backbones. Knowing which variables belong to the backbone

```
ifSuccess
    mul(used_vno, div(used_cno, 2))
else
    neg(div( used_vno, 9))


ifSuccess
    add(abs(used_vno), 5)
else
    mul(sub(used_vno - 2),
    div(used_cno, mul(3,-1)))


ifSuccess
    abd (abs(vno), div(used_cno,
    used_vno))
else
    neg(sroot( used_vNo))


ifSuccess
    add(abs(used_cno), 7)
else
    mul(div(used_cno, mul(3,-1),
    sub (used_vno - 4)))
```

Fig. 2. Some of the best evolved strategies for Inc*/Inc**. The strategies were manually edited for display purposes.

TABLE III
NUMBER OF FLIPS REQUIRED BY WALKSAT AND INC* (WITH WALKSAT) TO SOLVE SAT INSTANCES WITH DIFFERENT BACKBONE SIZES (SR=SUCCESS RATE, AF=AVERAGE NUMBER OF FLIPS).

backbone size	WalkSat		Inc*	
	SR	AF	SR	AF
10	1	752	1	670
30	1	1523	1	1541
50	1	2796	1	2191
70	1	3153	1	2340
90	1	6788	1	5680

of an instance would greatly help its solution. However, detecting the backbone variables in an instance is an expansive process.

To try and understand the reasons why the Inc* approach improves performance so much over an already highly effective solver (WalkSat), we have tested the performance of Inc* on different SAT instances all with the same number of variables (100 variables), but with different numbers of backbone variables. The instances are grouped into classes with 10, 30, 50, 70, 90 backbone variables out of the 100 total variables. Table III and Figure 3 show the difference in performance between Inc* and Walksat on these instance sets. Both heuristic were capable of finding solutions in all cases

TABLE II

COMPARISON BETWEEN AVERAGE PERFORMANCE OF WALKSAT AND WALKSAT WITH INC* AND INC** SR=SUCCESS RATE, AT = AVERAGE TRIES,
AF=AVERAGE NUMBER OF FLIPS

| name	#variables	#clauses	WalkSat		Inc*			Inc**		
			SR	AF	SR	AF	AT	SR	AF	AT
uf20	20	91	1	104.43	1	116.239	1.18	1	89.349	0.95
uf50	50	218	1	673.17	1	696.174	4.95	1	603.784	5.03922
uf75	75	325	1	1896.74	1	2000.59	8.07	1	1776.61	8.18039
uf100	100	430	1	3747.32	1	3825.82	11.51	1	3889.52	11.2706
uf150	150	645	0.97	15021.3	0.99	14275	16.45	1	6454.14	13.6706
uf200	200	860	0.9	26639.2	0.94	28526.2	21.39	1	24620.7	20.7922
uf225	225	960	0.87	29868.5	0.91	31258.8	22.16	0.99	29189.4	21.2627
uf250	250	1065	0.81	38972.4	0.87	38304.2	24.09	0.90	35021.5	23.2784
com360	360	1533	0.68	277062	0.78	221597	36.72	0.87	149616	31
com400	400	1704	0.66	172820	0.78	158485	30.66	0.79	161413	31
com450	450	1912	0.64	169113	0.72	140329	30.72	0.75	140489	30.36
com500	500	2130	0.38	271822	0.42	257987	36.52	0.44	263771	36.68
com550	550	2343	0.30	288379	0.42	271227	37.34	0.48	261428	37.28
com600	600	2556	0.44	257479	0.6	228347	35.74	0.6	251906	37.12
com650	650	2769	0.34	274112	0.44	268845	37.46	0.46	262329	41.07
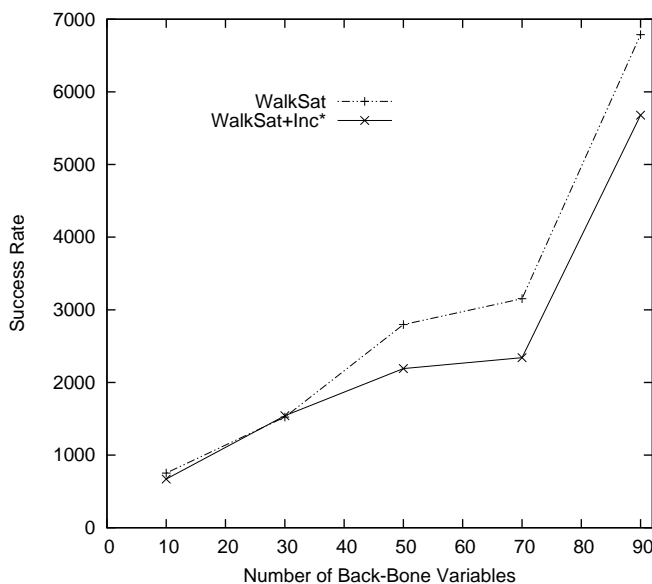
Fig. 3. Number of flips required by WalkSAT and Inc* (with Walksat) to solve SAT instances with different backbone sizes

(as shown in Table II their success rate is 100% on problems with 100 variables). However, Inc* was clearly faster than WalkSat on problems with more backbone variables.

Why is this? We conjecture that in Inc* the processes of progressively adding clauses to the clause list (based on their weights) and backtracking upon failure lead to satisfying (and implicitly identifying) early those clauses which have a higher-than-average frequency of backbone variables. Therefore, effectively, Inc*'s improved performance on difficult SAT instances is the result of its detecting and exploiting the structure of such instances.

V. CONCLUSION

In this paper we extended the recently proposed Inc* framework, and introduced Inc**, a SAT solver which significantly outperforms one of the best heuristic solvers for SAT, WalkSat. Results on the SAT problem showed that applying local search heuristics to progressively harder and harder versions of a problem with Inc* improves their performance both in terms of number of flips needed and, crucially, success rate. Thanks to relatively small modifications, Inc** further improves the performance of the Inc* algorithm.

In future work, we will try to generalise the algorithm to other problem domains, including scheduling, timetabling, TSP, etc. Also, we will test the algorithm on different types of SAT benchmarks (e.g., structured and handcrafted SAT problems). Furthermore, we would like to embed Inc* within a hyperheuristic framework where multiple agents perform the search in parallel. Each agent might, for example, use a different heuristic and would search for solutions to a part of the original problem (e.g., a subset of the clauses in a SAT formula).

REFERENCES

[1] M.B. Bader-El-Den and R. Poli. Inc*: An Incremental Approach for Improving Local Search Heuristics. In *Proceedings of 8th European Conference Evolutionary Computation in Combinatorial Optimization (EvoCop-08)* , page 194–205, 2008.

[2] S.A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[4] B. Selman, H.J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, CA, 1992. AAAI Press.

[5] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.
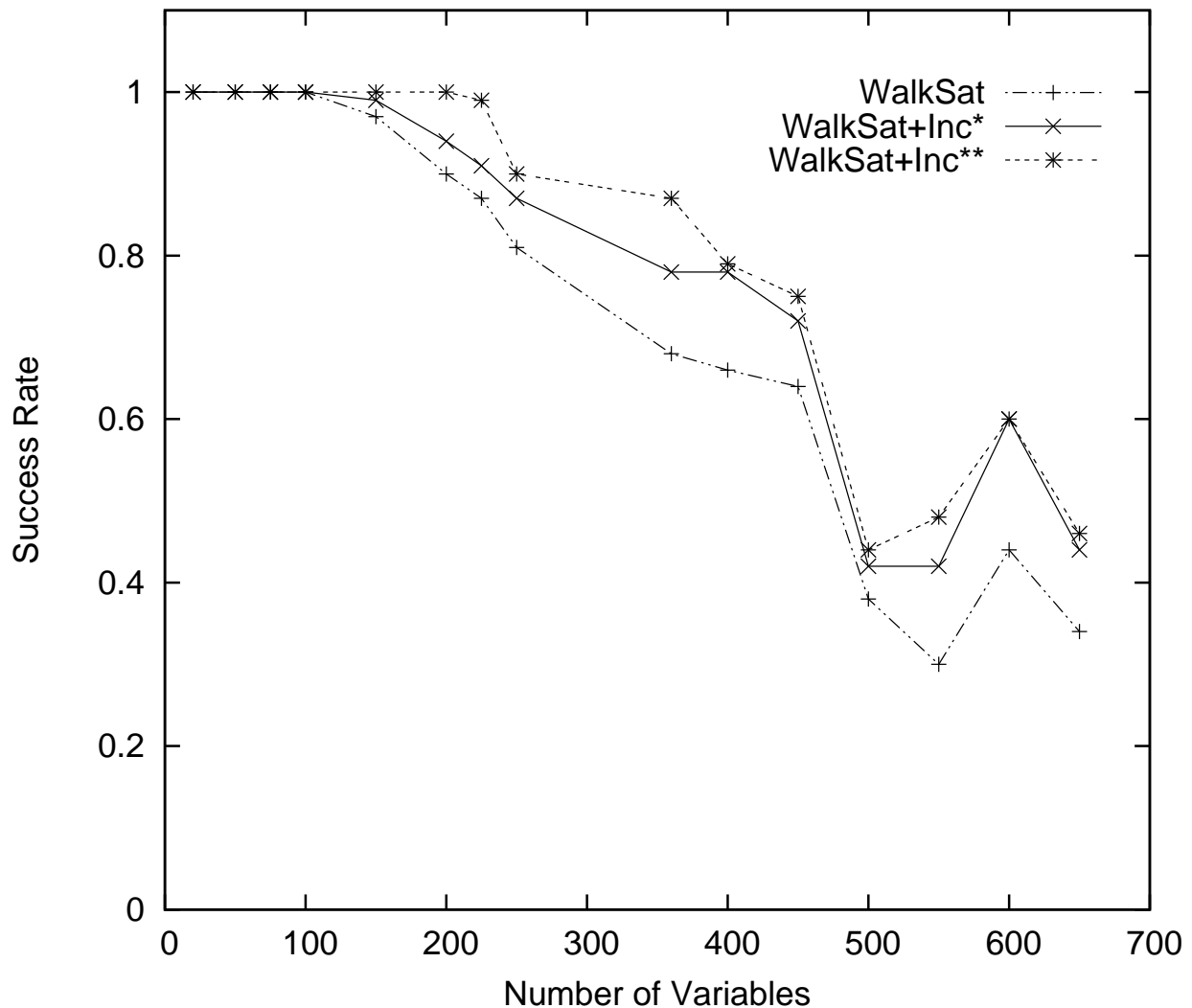
Fig. 4. Performance comparison between WalkStar, Inc*, and Inc**

[6] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[7] H. Han and F. Somenzi. Alembic: an efficient algorithm for cnf preprocessing. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 582–587, New York, NY, USA, 2007. ACM.

[8] H.H. Hoos and K. O'Neill. Stochastic local search methods for dynamic SAT- an initial investigation. Technical Report TR-00-01, 1, 2000.

[9] E. Marchiori and C. Rossi. A flipping genetic algorithm for hard 3-SAT problems. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[10] C. Rossi, E. Marchiori, and J. N. Kok. An adaptive evolutionary algorithm for the satisfiability problem. In *SAC (1)*, pages 463–469, 2000.

[11] J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Evol. Comput.*, 10(1):35–50, 2002.

[12] A. Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 641–648, 2002.

[13] A.S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *Genetic and Evolutionary Computation – GECCO-2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[14] R.H. Kibria and Y. Li. Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In P. Collet *et al.*, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 331–340, Budapest, Hungary, 10 - 12 Apr. 2006. Springer.

[15] M.B. Bader-El-Din and R. Poli. Generating SAT local-search heuristics using a GP hyper-heuristic framework. *Proceedings of the 8th International Conference on Artificial Evolution*, 36(1):141–152, 2007.

[16] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[17] W.B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[18] R.Poli, W.B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published by http://lulu.com and freely available at http://www.gp-field-guide.org.uk , 2008. (With contributions by J. R. Koza).

[19] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes and B. Selman Dynamic restart policies. In*Proceedings of Eighteenth national conference on Artificial intelligence*, page 674–681, Menlo Park, CA, 2002 AAAI Press.

[20] A.J. Parkes and J.P. Walser. Tuning Local Search for Satisfiability Testing. In*AAAI/IAAI, Vol. 1*, page 356-362, 1996.

[21] P. Morris. The Breakout Method for Escaping from Local Minima. In*AAAI*, page 40–45, 1993.