Inc*: An Incremental Approach for Improving Local Search Heuristics

Mohamed Bader-El-Den and Riccardo Poli

Department of Computing and Electronic Systems,
University of Essex, Wivenhoe Park, Colchester CO4 3SQ, United Kingdom
{mbbade,rpoli}@essex.ac.uk

Abstract. This paper presents Inc*, 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. Genetic programming is used to discover new strategies for the Inc* algorithm. We experimentally compare performance of local heuristics for SAT with and without the Inc* algorithm. Results show that Inc* consistently improves performance.

1 Introduction

Many NP-Complete problems, like scheduling, time tabling, satisfiability, graph colouring, etc., are routinely solved through the use of heuristics. A heuristic is effectively a rule of thumb or an educated guess that reduces the search required to find a solution. Heuristics make it possible to solve NP-Complete problems in practical situations which are beyond complete/exhaustive solvers. However, they provide no guarantee of success. So, finding ways to improve the performance of heuristics could have important and far-reaching ramifications.

We present Inc*, a general algorithm that can be used in conjunction with any local search heuristic to substantially improve the overall performance of the heuristic. Genetic programming is used to discover new strategies for the Inc* algorithm. We demonstrate the approach with Boolean satisfiability problems (SAT).

The paper is organised as follows. In Section 2 we introduce SAT problems and describe some of the best-known local-search heuristics used to solve them. We also review recent evolutionary systems developed for learning and evolving SAT heuristics. In Section 3, we introduce the Inc* algorithm along with the GP system used to evolve strategies for Inc*. A description of the experiments we performed with the GP Inc* framework is given in Section 4, Finally, we draw some conclusions in Section 5.

2 SAT problem

SAT is a classical combinatorial optimisation problem. It was the first problem to be proved to be NP-Complete [3]. Many heuristics have been proposed and successfully used for solving the SAT problem (e.g., [8, 18, 20, 19]). SAT has many different practical applications. Also, many other problems can be transformed into SAT problems.

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. So, if they fail, one cannot know whether that's because the formula is unsatisfiable or simply because the algorithm did 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 [4]. Incomplete algorithms are often based on local search heuristics (Section 2.1). 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.

2.1 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 [20]. 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. Algorithm 1 shows the general structure of a typical local-search heuristic for the SAT problem. The algorithm is normally repeatedly restarted for a certain number of times if it is not successful.

Some of the best-known heuristics of this type include:

GSAT [20] which, at each iteration, 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.

HSAT [8] In GSAT more than one variable may present the maximum gain. GSAT chooses among such variables randomly. HSAT, instead, it selects the variable with the maximum age, where the age of a variable is the number of flips since it is was last flipped. So, the most recently flipped variable has an age of zero.

WalkSat [19] 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.

Algorithm 1 General algorithm for SAT stochastic local search heuristics

```
1: L = initialise the list of variables randomly
2: for i = 0 to MaxFlips do
3:     if L satisfies formula F then
4:         return L
5:     end if
6:     select variable V from L using some selection heuristic
7:     flip V
8: end for
9: return no assignment satisfying F found
```

As one case see, 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 is typically the most important factor is 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. Mainly this is done to avoid (or escape from) local optima. This is done by selecting a random variable to flip from a designated set variables. There are different ways of choosing this set. For example, the set can include all the variables in the CNF formula, as in the GSAT, or just the variables in unsatisfied clauses, as in WalkSat.

2.2 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 [16]. 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 algorithms as in [10], although some researchers have also used incomplete or heuristic-based solvers to deal with incremental SAT problems [11]. The main problem with the latter is that heuristics give no grantee that a solution can be found. Their main advantage is speed.

In this paper we will introduce 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 will use Genetic Programming (GP) [13, 14] 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. We will explain this in more detail later.

2.3 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 [15], 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 [17]. A good overview of other algorithms of this type is provided in [9].

GP has evolved competitive SAT solvers. For example, Fukunaga evolved local search heuristics [6, 7]. 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 [12]. Furthermore, a general framework for evolving local-search 3-SAT heuristics, called GP-HH, has recently been proposed [1, 2]. 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 very competitive heuristics.

3 The Inc* Framework

3.1 Principles Behind Inc*

As 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, 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 are solved. The search is not restarted when a new instance is presented to the solver. In this way, it is hoped that the solver will effectively and progressively be 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 (of the same size as the previous one, but radically different from it) in the attempt to continue the progression towards the goal problem instance.

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 use one of the SAT heuristics to tests the satisfiability of this portion of the formula, which we will call the *clauses active list*. Depending on the result of the heuristic 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.

To illustrate the benefits of the main ideas behind Inc*, in Figure 1 we show the results of two simple experiments using GSAT. In both experiments, we added clauses to the active list one by one and we used GSAT after each addition to find an assignment that satisfies the clauses currently active. The graph on the left of the figure shows the case of a formula with 20 variables and 91 clauses. The graph reports the number of flips GSAT used to find a new satisfying assignment after the addition of each new clause and the number of variables in use in the active list. The plot on the right of the figure shows the number of flips required by GSAT to find assignments that satisfy the active list for a SAT instance with 50 variable and 218 clauses. The plots show that adding a new clause to the currently active clauses requires no flips or a very small number of flips most of the time. We even find that in both instances the full formula is actually satisfied by an assignment found before all clauses had been added.[1] This is really the reason why the use of a progression of problems may make a problem easier. On rare occasions, however, finding a new assignment after the addition of a clause may require hundreds or even thousands of flips. It is precisely to avoid these high peaks that Inc* backtracks.[2]

We have turned these ideas into a detailed algorithm (Algorithm 2). The algorithm starts by initialising all the variables in the formula F randomly and by activating 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 relatively a small number of flips to run with at the beginning. The number of allowed flips 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 to satisfy the active list after the addition of this clause. So, 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 AC is increased by adding new classes to it. Otherwise, the algorithm removes from AC a small set of clauses, giving preference to those with the lowest clause weight, and the number of allowed flips is increased, as previously mentioned.

Two key elements in the effectiveness of Inc* are the decisions taken at Steps 21 and 24 in Algorithm 2 as to how many clauses to add or remove from the active list after a success or failure, respectively. In this paper, 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.

[1] This is not entirely surprising, since it is well-known that in most hard SAT instances there are (sometimes numerous) redundant clauses. A redundant clause is a clause that has no effect on the SAT formula [21]. There are algorithms for finding and removing redundant clauses [5], but the process is complex and very time consuming, especially in large SAT instances.

[2] Recently, some researchers have attempted to detect possible hidden structural information inside real-world SAT instances (e.g., backbones, backdoors, equivalences and functional dependencies) in an effort to improve the efficiency of SAT solvers on hard instances. Inc* does not explicitly attempt to detect the hidden SAT structure. However, it effectively finds assignments for the variables that minimise the chance of violating the backbone of a SAT problem as early as possible in the construction of a solution. Furthermore, it does this quickly and without ever requiring complex operations, simply acting as a wrapper for standard SAT heuristics.

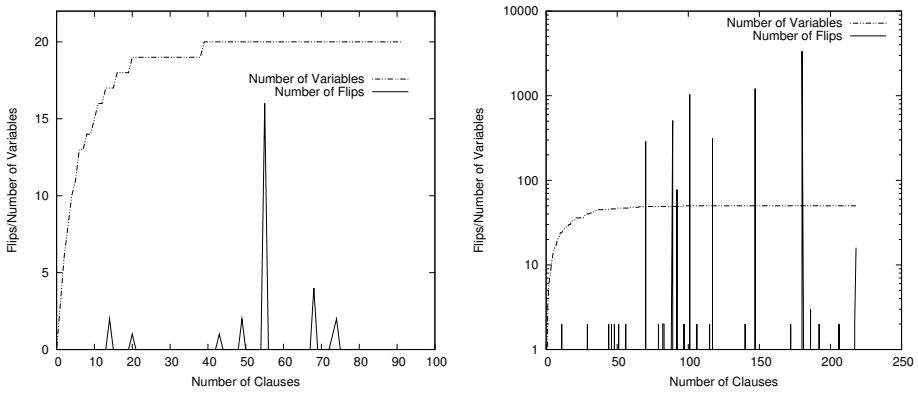

Fig. 1. Behaviour of Inc* with GSAT on two SAT problems (see text) when adding clauses one by one (and no backtracking).

3.2 Inc* Optimisation via GP

In this section we will describe the GP system used to evolve strategies for the Inc* algorithm. As we mentioned above, an evolved strategy (which takes the form of a computer program) needs to decide how many clauses the algorithm should add/remove to/from the active list after each success or failure at finding a valid variable assignment that satisfies the current active clauses of the full SAT formula.

We use a tree representation for programs. The function and terminal sets are shown Table 1. We constrain the representation 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 assumed to be reals. 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*.[3] 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 clauses should be removed from *AC*.

The other elements of the primitive set behave as follows: the function $add(d_1,d_2)$ 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 product of d_1 by d_2, $div(d_1,d_2)$ safely divides d_1 by d_2, and $neg(d_1)$ 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.

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

[3] 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*.

Algorithm 2 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 i = 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 = increment allowed flips
26:   end if
27: until Flips_Total < MaxFlips
28: return no assignment satisfying F found
```

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

Table 1. GP function and terminal sets.

Function Set	
$ifSuccess(d_1,d_2)$	: returns d_1 if the last attempt to solve the CNF formula was successful
$add(d_1,d_2)$	: returns the sum of d_1 and d_2 as doubles
$sub(d_1,d_2)$	: subtracts d_2 from d_1
$mul(d_1,d_2)$	: returns the multiplication of d_1 by d_2
$div(d_1,d_2)$	: safe 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
$sqrt(d_1)$	: returns the a safe square root of d_1
Terminal Set	
vNo	: total number of variables in the CNF SAT formula
cNo	: total number of clauses in the CNF SAT formula
used_cNo	: number the currently active variables
used_vNo	: number the currently active clauses
constX	: constant integer number form 0 to 9

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 have been expended to satisfy the initial active clauses, since no clauses are added or removed, no further flips would ever be necessary. 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 can avoid this and we can 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 extracting 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 the root of the tree), deleting the sub-tree rooted there, and then regenerating it randomly as in the initialisation phase.

4 Experimental Results

In our experiments we used a population of 1000 individuals, run for 51 generations. While strategies are evolved using 50 fitness cases, the generality of best of run indi-

viduals is then evaluated on an independent test set including more 500 SAT instances. This section will show a comparison between the performance of standard handcrafted heuristics (GSAT and WalkSat) and the same heuristics when combined with Inc* controlled by strategies evolved by GP. We have used the following parameters values for the Inc* algorithm.[4] We allow 100 flips to start with. Upon failure, the number of flips is incremented by 20%. We allow a maximum total number of flips of 100,000. The maximum number of tries is 1000 (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 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 by 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. The following is some evolved heuristics after they have been normalizef representing the diffrent groups:

- First group: $ifSuccess(abs(vno), neg(sroot(used\_vNo)))$
- First group: $ifSuccess(mul(used\_vno, div(used\_cno, 2)), neg(div(used\_vno, 9)))$
- Second group: $ifSuccess(cno, neg(sub(cno, div(used\_cno, 3))))$
- Third group: $ifSuccess(mul(vno, 2), abs(add(vno, sroot(used\_cno))))$

Table 2 shows a first set of experimental results. In particular, it shows the difference between the average performance of the GSAT and the average performance of GSAT combined with the best evolved Inc* strategies, which we will call IncGSAT. Both heuristics in this experiment are allowed a maximum of 100,000 total flips. The performance of the heuristics on an instance is the average of 10 different runs, to ensure the results are statistically meaningful. The AF column shows the average number of flips used by each heuristic in successful attempts only. As one case see IncGSAT has a better performance than GSAT in terms of success rate (as also shown in Figure 2) and average number of flips used to solve the test instances. So, IncGSAT managed to solve many more instance in less time. We believe this is due to the lack of random moves in the GSAT, which makes GSAT easy pray of local optima. IncGSAT improves GSAT by going forward and backward adding and removing clauses through the course of a run thereby avoiding the problem. The AT column shows the number of tries IncGSAT used. This corresponds to the number of times IncGSAT modified the search space to escape local optima and find a better path for satisfying the instance.

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

Table 2. Comparison between average performance of GSAT and GSAT with Inc* SR=success rate, AT = average tries, AF=average number of flips (out of a maximum of 100,000).

			GSAT		GSAT+Inc*		
Instance Test Set	variables	clauses	SR	AF	SR	AF	AT
uf20-(901-1000)	20	91	0.632	43853	1	1336.52	3.37
uf50-(901-1000)	50	218	0.368	75623	0.935	14949.3	12.29
uf75-(51-100)	75	325	0.348	78423.1	0.780	31614.6	18.47
uf100-(901-1000)	100	430	0.297	86599	0.723	39856	22.59

Table 3. Comparison between average performance of WalkSat and WalkSat with Inc* SR=success rate, AT = average tries, AF=average number of flips (out of a maximum of 100,000).

			WalkSat		WalkSat+Inc*		
Instance Test Set	variables	clauses	SR	AF	SR	AF	AT
uf20-(901-1000)	20	91	1	104.43	1	116.239	1.18
uf50-(901-1000)	50	218	1	673.17	1	696.174	4.95
uf75-(51-100)	75	325	1	1896.74	1	2000.59	8.07
uf100-(901-1000)	100	430	1	3747.32	1	3825.82	11.51
uf150-(51-100)	150	645	0.974	15021.3	0.987	14275	16.45
uf200-(51-100)	200	860	0.9	26639.2	0.936	28526.2	21.39
uf225-(51-100)	225	960	0.87	29868.5	0.91	31258.8	22.17
uf250-(51-100)	250	1065	0.816	38972.4	0.875	38304.2	24.09

Table 3 shows the results of another set of experiment using WalkSat and a combination of Sat* and WalkSat which we will call IncWalkSat. Also in this experiment, both heuristics were given a maximum of 100,000 total flips. Again, the performance of the heuristics on an instance is the average of 10 different runs. WalkSat is among the best performing local search heuristics for SAT.

We categorize the results in this table to 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 both heuristics have a perfect success rate of 100%. However, WalkSat used a slightly smaller number of flips, thereby running marginally faster than IncWalkSat on this group of problems. In the second group, which contains larger instances, however, IncWalkSat has a higher success rate than WalkSat, and the difference in the performance increases as the size of the instance increases. This means that Inc* can solve complex instances where local heuristics alone fail. This explains why when training the GP system on small instances some evolved strategies (the strategies in group three) always tried to go foreword by 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.

5 Conclusion

In this paper we provided a proof of concept, supported by results, of the ideas behind our Inc* algorithm, which tries to solve problems incrementally. Results on the

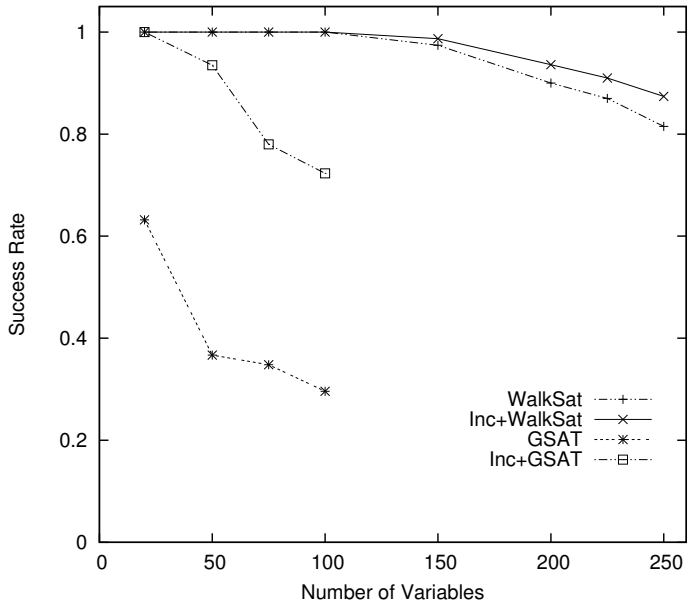

Fig. 2. Comparison between GSAT, WalkSAT, IncGSAT and IncWalk average success rate performance

SAT problem showed that combining local search heuristics with Inc* improved their performance especially on instance which standard local search alone failed to solve.

In future work, we will try to generalise the algorithm to other problem domains, including scheduling, time tabling, TSP, etc. Also we will test the algorithm on diffident types of SAT benchmarks (e.g., structured and handcrafted SAT problems). Also 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).

Acknowledgements

The authors acknowledge financial support from EPSRC (grants EP/C523377/1 and EP/C523385/1).

References

1. M. B. Bader-El-Den and R. Poli. A GP-based hyper-heuristic framework for evolving 3-SAT heuristics. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1749–1749, London, 7-11 July 2007. ACM Press.
2. 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.

3. 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.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
5. O. Fourdrinoy, E. Gregoire, B. Mazure, and L. Sais. Eliminating redundant clauses in sat instances. In *The Third International Conference on Integration of AI and OR Techniques 4th International Conference on Integration of AI and OR Techniques in Cons(CPAIOR'07)*, pages 71–83, Brussels, juin 2007.
6. A. Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 641–648, 2002.
7. 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.
8. I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proc. of AAAI-93*, pages 28–33, Washington, DC, 1993.
9. J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Evol. Comput.*, 10(1):35–50, 2002.
10. 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.
11. H. H. Hoos and K. O'Neill. Stochastic local search methods for dynamic SAT- an initial investigation. Technical Report TR-00-01, 1, 2000.
12. 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.
13. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
14. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
15. 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.
16. 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.
17. C. Rossi, E. Marchiori, and J. N. Kok. An adaptive evolutionary algorithm for the satisfiability problem. In *SAC (1)*, pages 463–469, 2000.
18. B. Selman and H. Kautz. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of theInternational Joint Conference on Artificial Intelligence(IJCAI-93)*, Chambry, France, 1993.
19. 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.
20. 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.
21. H. Zeng and S. A. McIlraith. The role of redundant clauses in solving satisfiability problems. In *Principles and Practice of Constraint Programming - CP 2005*, page 873, 2005.