Chapter 11

EVOLVING EFFECTIVE INCREMENTAL SOLVERS FOR SAT WITH A HYPER-HEURISTIC FRAMEWORK BASED ON GENETIC PROGRAMMING

Mohamed Bader-El-Den[1] and Riccardo Poli[1]
[1]*Department of Computing and Electronic Systems, University of Essex.*

Abstract Hyper-heuristics could simply be defined as heuristics to choose other heuristics. In other words, they are methods for combining existing heuristics to generate new ones. In this paper, we use a grammar-based genetic programming hyper-heuristic framework. The framework is used for evolving effective incremental solvers for SAT. The evolved heuristics perform very well against well-known local search heuristics on a variety of benchmark SAT problems.

Keywords: hyper-heuristic, Inc, SAT, heuristics.

1. Introduction

Heuristic methods have contributed to the solution of many combinatorial optimisation problems such as bin packing, the travelling salesman problem (TSP), graph colouring, and the satisfiability problem (SAT). The performance of heuristics on a problem varies from instance to instance. Also, even on the same instance, randomised heuristics may be able to provide good solutions on one occasion, and bad on another. Hyper-heuristics (HHs) aim at providing a more robust approach raising the level of generality at which optimisation methods operate. They can be defined as "heuristics to choose heuristics" (Burke et al., 2003a). The main idea is to make use of different heuristic during the search for a solution.

SAT is one of the most studied combinatorial optimisation problems, and the first problem proved to be NP-Complete. In this paper we will use genetic programming (GP) in a HH framework to solve SAT problems. In particular, we

use GP to evolve local search heuristics to be used within the Inc* algorithm. Inc* (Bader-El-Den and Poli, 2008) 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 simplified instance of the problem, and tries to solve it. If the instance is solved, Inc* increases the size of the instance, and repeats this process until the full size of the problem is reached. 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. We look at all this in more detail below.

SAT 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. So, if they fail, one does not 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 (Davis et al., 1962). Incomplete algorithms are often based on local search heuristics (see next section). These algorithms can be extremely fast, but success cannot be guaranteed. On the contrary, complete algorithms guarantee success, but they computational load can be considerable, and, so, they cannot be used for large SAT instances.

Stochastic local-search heuristics. Stochastic local-search heuristics have been widely used since the early 90s for solving the SAT problem following the successes of GSAT (Selman et al., 1992). The main idea behind these heuristics

Algorithm 1

```
L = initialise the list of variables randomly
for i = 0 to MaxFlips do
  if L satisfies formula F then
    return L
  end if
  select variable V using some selection heuristic
  flip V in L
end for
return no assignement satisfying F found
```

is to try to get an educated guess as to which variable will most likely, when flipped, give us a solution or move us one step closer to a solution. Normally, the heuristic starts by randomly initialising all the variables in the 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.

The best heuristics of this type include:

- GSAT: (Selman et al., 1992) 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: (Gent and Walsh, 1993) In GSAT more than one variable may present the maximum gain. GSAT chooses among such variables randomly. HSAT, instead, uses a more sophisticated strategy. 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.

- GWSAT: (Selman and Kautz, 1993) with probability p selects a variable occurring in some unsatisfied clauses while with probability $(1-p)$ flips the variable with maximum gain as in GSAT.

- WalkSat: (Selman et al., 1994) 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, WalkSat

selects with probability p the variable with the maximum score gain, with probability $(1-p)$ a random variable in C.

- Novelty: (Hoos and Stützle, 2000) After selecting a random unsatisfied clause, Novelty flips the variable with the highest score unless it was the last variable flipped in the clause. If this is the case, with probability p the same variable is flipped, otherwise a random variable from the same clause is selected.

Evolutionary algorithms and SAT problem. Different evolutionary techniques have been applied to the SAT problem. There are two main research directions: direct evolution and evolution of heuristics.

An example of methods of the first type – direct evolution – is FlipGA which was introduced in (Marchiori and Rossi, 1999). There a genetic algorithm was used to generate offspring solutions to SAT using the standard genetic operators. However, offspring were then improved by means of local search methods. The same authors later proposed ASAP, a variant of FlipGA (Rossi et al., 2000). A good overview of other algorithms of this type is provided in (Gottlieb et al., 2002).

The second direction is to use evolutionary techniques to automatically evolve local search heuristics. A successful example of this is the CLASS system developed in (Fukunaga, 2002; Fukunaga, 2004). The process of evolving new heuristics in the CLASS system is based on five conditional branching cases (if-then-else rules) for combining heuristics. Effectively CLASS can be considered as a very special type of GP system where these rules are used instead of the standard GP operators (crossover and mutation). The evolved heuristics were competitive with a number of human-designed heuristics. However, the evolved heuristics were relatively slow. This is because the conditional branching operations used in CLASS evaluate two heuristics first and they then select the output of one to decide which variable to flip. Also, restricting evolution to use only conditional branching did not give the CLASS system enough freedom to evolve heuristics radically different from the human-designed heuristics (effectively, the evolved heuristic are made up by a number of nested heuristics). Another example of system that evolves SAT heuristics is the STAGE system introduced in (Boyan and Moore, 2000). STAGE tries to improve the local search performance by learning (online) a function that predicts the output of the heuristic based on some characteristics seen during the search.

Hyper-heuristics, GP and SAT

As we noted above, hyper-heuristics are "heuristics to choose other heuristics" (Burke et al., 2003a). A heuristic is considered as rule-of-thumb or "educated guess" that reduces the search required to find a solution. The difference

between metaheuristics and hyper-heuristics is that the former operate directly on the targeted problem search space with the goal of finding optimal or near optimal solutions. The latter, instead, operate on the heuristics search space (which consists of the heuristics used to solve the target problem). The goal then is finding or generating high-quality heuristics for a target problem, for a certain class of instances of a problem, or even for a particular instance.

There are different classes of hyper-heuristics. In one class of HH systems, the system is provided with a list of preexisting heuristics for solving a certain problem. Then the HH system tries to discover what is the best sequence of application for these heuristics for the purpose of finding a solution as shown in Figure 11-1. Different techniques have been used to build HH systems of this class. Algorithms used to achieve this include, for example: tabu search (Burke et al., 2003b), case-based reasoning (Burke et al., 2006b), genetic algorithms (Cowling et al., 2002), ant-colony systems (Silva et al., 2005), and even algorithms inspired to marriage in honey-bees (Abbass, 2001).

Another form of hyper-heuristic is one where the system produces (meta-)heuristics by specialising them from a generic template. The specialisation can take the form of one or more evolved components, which can modify the behaviour of the meta-heuristic or heuristic. This approach has given, for example, very positive results in (Poli et al., 2007) where the problem of evolving offline bin-packing heuristics was considered. There GP was used to evolve strategies to guide a fixed solver. This approach was also taken in (Bader-El-Den and Poli, 2008) where Inc* was originally proposed. This solver was applied to the SAT problem with good success. To further improve chances of success, a key element of Inc*, its strategy for adding and removing SAT clauses, was evolved using GP. (We will provide more information about this below, since the work presented in this paper relates closely to Inc*.)

A third approach used to build HH systems is to create (e.g., evolve) new heuristics by making use of the *components* of known heuristics. The process starts simply by selecting a suitable set of heuristics that are known to be useful in solving a certain problem. However, instead of directly feeding these heuristics to the HH system (as in the first type of HHs discussed above), the heuristics are first decomposed into their basic components. Different heuristics may share different basic components in their structure. However, during the decomposition process, information on how these components were connected with one another is lost. To avoid this problem, this information is captured by a grammar. So, in order to provide the HH systems with enough information on how to use components to create valid heuristics, one must first construct an appropriate grammar. Hence, in the hyper-heuristics of this third type, both the grammar and the heuristics components are given to the HH systems. The system then uses a suitable evolutionary algorithm to evolve new heuristics, Figure 11-2 shows an abstract model for the process. For example, in recent

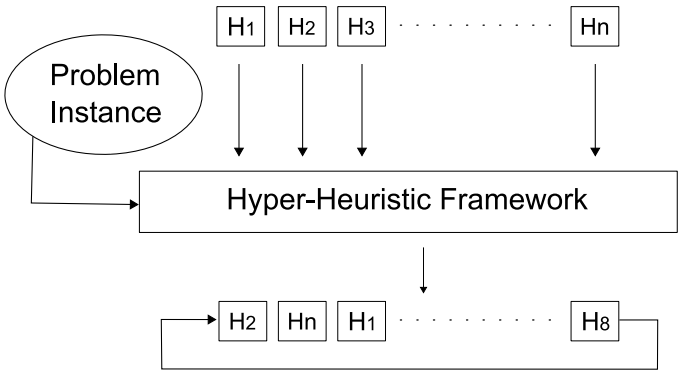

Figure 11-1. Shows an abstract structure of one of the HH types: In this type, HH tries to find the best sequence of heuristics to solve a certain problem.

work (Burke et al., 2006a) GP was successfully used to evolve new heuristics of for one-dimensional online bin packing problems. While in (Bader-El-Din and Poli, 2007) this approach was used in a system called GP-HH (for GP Hyper-Heuristic) to evolve heuristics for the SAT problem which are specialised to solve specific sets of instances of the problem. A comparison between GP-HH and other well-known evolutionary and local-search heuristics revealed that the heuristics produced by GP-HH are very competitive, being on par with some of the best-known SAT solvers.

Contributions of this Paper

The Inc* algorithm proposed in (Bader-El-Den and Poli, 2008) and briefly discussed above has been successful. There the choice of the simplified problems progressively leading to the original (hard) problem is dynamic so as to limit the chances of the algorithm getting stuck in local optima. Whenever the system finds a simplified instance too difficult, it backtracks and creates a new simplified instance. In the SAT context, the simplified problems are simply obtained by choosing subsets of the clauses in the original formula. The subset in current use is called 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. Naturally the choice of how many clauses to add or remove from the active list of Inc* after a success or failure is very important for the good performance of the algorithm. In (Bader-El-Den and Poli, 2008) genetic programming was used to discover good dynamic strategies to make such decisions optimally at run time. These strategies were constrained to act within the specific Inc* algorithm (which we report in Algorithm 2), which was human designed.

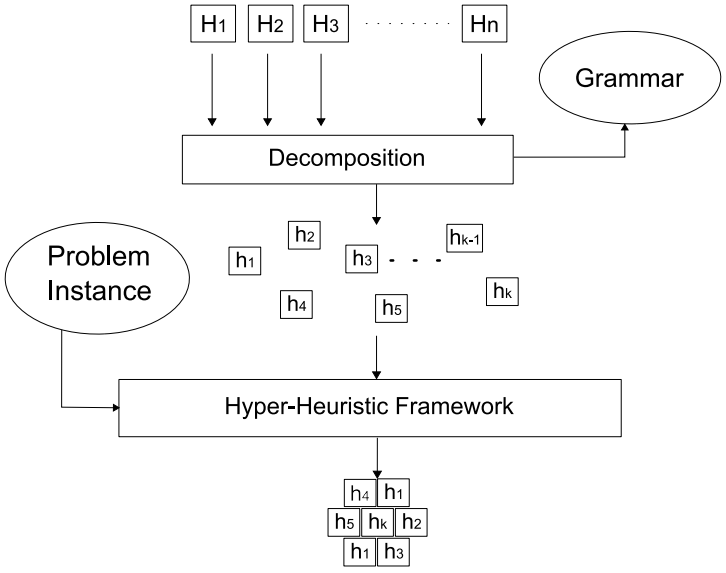

Figure 11-2. Shows an abstract structure of one of the HH types, In this type the input heuristics H are decomposed to their components h before evolving new heuristics from these components.

Encouraged by the success of Inc* and the GP-HH, in this work, we want to take Inc* one step further. We want to evolve local search SAT heuristics specifically designed to work well within Inc*, instead of just evolving the control element of Inc* which decides how many clauses to add or remove upon success or failure, which is what we did before. To do this, we make use of the grammar-based Hyper-Heuristic GP framework developed in (Bader-El-Din and Poli, 2007; Bader-El-Den and Poli, 2007). As one can easily see inspecting the local search heuristics for SAT presented above, all the heuristics share similar components, for example variable score, selection of a clause and conditional branching. The components of these heuristics plus additional primitives that are necessary to evolve more complex behaviours are used to generate the grammar for GP-HH, as described in the next section.

By giving GP-HH the freedom to design completely new Inc* searchers, we hoped to find algorithms for the solution of the SAT problem that are novel and even more powerful than Inc* or GP-HH alone.

2. GP hyper-heuristic for evolving Inc* SAT heuristics

As mentioned before, the most important step in designing a GP Hyper-Heuristic framework is constructing a grammar capable of describing heuristics for the targeted domain. The grammar shows the GP system how the elementary

Algorithm 2 Inc* approach to solving SAT problems.

$L =$ random variable assignment
$AC =$ small set of random clauses from the original problem
$Flips =$ number of allowed flips at each stage
$Flips\_Total = 0$ {This keeps track of the overall number of flips used}
$Flips\_Used = 0$ {This keeps track of the flips used to test the active list}
$Inc\_Flip\_Rate =$ rate of increment in the number of flips after each fail
repeat
  for $i = 0$ to Flips **do**
    if L satisfies formula F **then**
      return L
    end if
    select variable V from AC using some selection heuristic
    flip V in L
  end for
  Flips_Total = Flips_Total + Flips_Used
  update clause weights
  if L satisfies AC **then**
    if AC contains all clauses in F **then**
      return L
    end if
    AC = add more clauses to the active list
  else
    sort AC
    AC = remove some clauses from the active list
    $Flips$ = increment allowed flips
  end if
until $Flips\_Total < MaxFlips$
return no assignment satisfying F found

components obtained from the decomposition of previously-known heuristics could be connected to generate valid evolved heuristics. The grammar presented in this section is an extension of the one used in (Bader-El-Din and Poli, 2007; Bader-El-Den and Poli, 2007), which we modified so as to be more efficient and to better suit the Inc* framework. The grammar contains elements form the heuristic descried in the previous section.

The GP Hyper-Heuristic framework allows us to use components not used in handcrafted heuristics to see whether they can help the evolution process. This is important in our case because we are not just evolving local-search SAT heuristics but much more sophisticated heuristics for Inc*. For example, we

```
start  →   Flip v
v      →   Random l |
           MaxScr l, op |
           ScndMaxScr l, op |
           ZeroBreak l, v |
           Ifv condV, v, v |
l      →   ALL |
           UC |
           AllUC |
           Ifl condL, l, l |
condV  →   prob | size |
           NotMinAge |
           NotZeroAge |
           MinAge
condL  →   prob | size
prob   →   20 | 40 | 50 |
           70 | 90
size   →   Small | Midd |
           Large | LastAtmp
op     →   TieRandom | TieAge
```

Figure 11-3. The GP Hyper-Heuristic grammar used for evolving Inc* SAT heuristics.

added to the grammar new conditional branching conditions, based on the size of the current *clauses active list* as will be described in details later.

To construct our grammar, we classified the main components of SAT heuristics into two main groups. The first group of components, Group 1, returns *a variable* from an input list of variables (e.g., the selection of a random variable from the list or of the variable with highest gain score). The second group, Group 2, returns *a list of variables* from the CNF formula (e.g., the selection of a random unsatisfied clause which, effectively, returns a list of variables). The grammar is designed in such a way to produce functions with no side effects (i.e., we avoid using variables for passing data from a primitive to another). The aim was to reduce the constraints on the crossover and mutation operators, and to make the GP tree representing each individual simpler.

The grammar we used and its components are shown in Figure 11-3. The root of each individual (an expression in the language induced by the grammar) is the primitive `Flip`, which flips a variable `v`. The variable `v` is typically (but not always) selected from a list of variables, `l`, using appropriate primitives. There are three ways in which this can happen.

The first method is to select the variable randomly from `l`. This is done by the function `Random l`.

The second method is to choose the variable based on the score of the variables in the list. In other words, the choice depends on how many more clauses will be satisfied after flipping a variable. A positive sign of the score means that more clauses will be satisfied. A negative score is instead obtained if flipping the variable will cause fewer clauses to be satisfied. The selection based on score is done by either `MaxScr` or `ScndMaxScr`. These functions select the variable with the highest and second highest score from `l`, respectively. Both `MaxScr` and `ScndMaxScr` require a second argument, `op`, in addition to a list `l`. This argument specifies how to break ties if multiple variables have the same highest score. If `op` is `TieAge`, the tie will be broken by favouring the variable which has been flipped least recently, while `TieRandom` breaks ties randomly.

The third method for selecting a variable is through the primitive `ZeroBreak` which selects the variable that will not "unsatisfy" any of the currently satisfied clauses. If a such variable does not exist in the given list, the primitive returns its second argument `v`, which is selected by any of the previous methods.

The list of variables `l` can be selected from the CNF formula in three simple ways. The first two are directly taken from the handcrafted heuristics: `All` returns all the variables in the formula, while `UC` returns all the variables in one of randomly selected clause. The third method, `AllUC`, selects all the variables in all unsatisfied clauses. This primitive can be very useful (and, indeed, was often used by GP) in stages where the number of clauses in the *clauses active list* of Inc* is relatively small.

The grammar also includes conditional branching components (`IFV` and `IFL`). Branching components are classified on the basis of their return type. The condition `prob` means that the branching is probabilistic and depends on the value of `prob`. Also the branching can be done on other criteria like `size` of the *clauses active list*. While this is not in any of the handcrafted heuristics, we added it to our grammar to make GP able to evolve heuristics that adapt with the different stages in the Inc* algorithm, while it progressively adds and removes clauses from the active list. `LastAtmp` is *true* if the Inc* last attempt to satisfy the *clauses active list* was successful and *false* otherwise.

3. Experimental setup

In evolving Inc* SAT heuristics we used a training set including 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. While strategies are

evolved using 50 fitness cases, the generality of best of run individuals is then evaluated on an independent test set including many more SAT instances.

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).

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 (head) $Flip$ 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 using a specialised form of crossover. A random crossover point is selected in the first parent, then the grammar is used to select the crossover point from the second parent. It is randomly selected from all valid crossover points. If no point is available, the process is repeated again from the beginning until crossover is successful.
- 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.
- We used a population of 500 individuals.

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

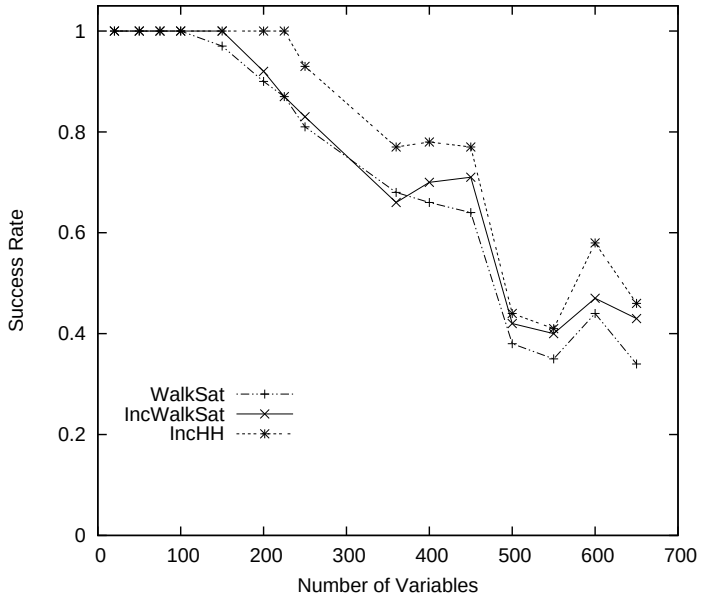

Figure 11-4. Average success rate performance of WalkSAT, IncWalkSat and IncHH.

- To start with we allow 2,000 flips for instances with more than 250 variables, while we use 100 flips for smaller instances.
- Upon failure, the number of flips is incremented by 20%.
- We allow a maximum total number of flips of 400,000 for instances with more than 250 variables, while we use 100,000 flips for smaller instances.
- The maximum number of tries is 1000 (including successful and unsuccessful attempts). This is mainly to prevent the system from getting into an infinite loop.
- We evolved Inc* SAT heuristics for one simple Inc* strategies which adds 15% clauses after each success and removes 10% after each failure.

4. Results

We start by showing a typical example of the search heuristics for Inc* evolved using the GP Hyper-Heuristics framework.

Figure 11-4 shows one of the best performing heuristics evolved for the Inc* strategy represented as a program tree. As one can see evolved heuristics are significantly more complicated than the standard heuristics we started from (e.g., GSat, WalkSat, Novelty). So, a manual analysis of how the component steps of an evolved heuristic contribute to its overall performance is difficult.

Although the initial population in GP is randomly generated and includes no handcrafted heuristics, we have noticed that most of the best evolved individuals contain patterns of instructions similar to those in hand crafted heuristics. This means that our grammar-based GP system has been able mix different heuristics patterns to create new heuristics for Inc*.

The Inc* strategy and parameter settings used during the evolution process are the same as the one used in the testing phase. Table 11-1 shows the results of a set of experiments comparing the performance of three algorithms: WalkSat alone, WalkSat with Inc* (IncWalk) and the heuristic evolved by GPHH with Inc* (IncHH). Instances with up to 250 variables were taken from SatLib (uf20 to uf250). Larger instances were taken from the benchmark set of the SAT 2007 competition. 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.

Table 11-1. Comparison between average performance of WalkSat and WalkSat with Inc* and Inc* with the evolved heuristic (IncHH) SR=success rate, BT = average back tracks , AF=average number of flips.

		WalkSat		IncWalk			IncHH		
name	clauses	SR	AF	SR	AF	BT	SR	AF	BT
uf20	91	1	104.43	1	136.32	1.13	1	98.54	0.96
uf50	218	1	673.17	1	702.52	4.25	1	723.52	3.13
uf75	325	1	1896.74	1	1970.59	8.15	1	1909.61	7.17
uf100	430	1	3747.32	1	3640.62	10.31	1	3769.42	9.07
uf150	645	0.97	15021.3	1	13526	15.44	1	6454.14	12.60
uf200	860	0.9	26639.2	0.92	27586.2	20.59	1	26340.8	19.09
uf225	960	0.87	29868.5	0.87	32258.8	21.27	1	34187.7	20.24
uf250	1065	0.81	38972.4	0.83	39303.5	25.15	0.93	39025.6	24.37
com360	1533	0.68	277062	0.66	225874	33.82	0.77	147617	31.87
com400	1704	0.66	172820	0.70	188435	31.64	0.78	191493	32.7
com450	1912	0.64	169113	0.71	190325	32.72	0.77	170459	30.36
com500	2130	0.38	271822	0.42	297683	35.72	0.45	265762	36.59
com550	2343	0.35	288379	0.40	278567	39.84	0.41	266429	38.23
com600	2556	0.44	257479	0.47	228347	36.64	0.58	297932	39.12
com650	2769	0.34	274112	0.43	285964	34.65	0.43	282352	44.30

We categorise the results in Table 11-1 into two groups. The first group includes relatively small instances with no more than 100 variables. The second group includes larger instances with more than 100 variables. In the first group of problems all heuristics have a perfect success rate of 100%. The IncHH performance is close to the performance to other heuristics regarding the number of flips used. However, IncHH was able to significantly outperform the

other heuristics solving more instances from the second group as shown also in Figure 11-4.

5. Conclusions

We have used the GP-HH framework for evolving customised SAT heuristics which are used within the Inc* algorithm. GP has been able to evolve high-performance heuristics for SAT problems.

In future work, we will try to generalise the framework 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). We also want to study the effect of grammar design on HH frameworks in more detail.

6. Acknowledgements

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

References

Abbass, H. A. (2001). MBO: Marriage in honey bees optimization - A haplometrosis polygynous swarming approach. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 207–214, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea. IEEE Press.

Bader-El-Den, Mohamed Bahy and Poli, Riccardo (2007). 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. ACM Press.

Bader-El-Den, Mohamed Bahy and Poli, Riccardo (2008). Inc*: An incremental approach to improving local search heuristics. In *EvoCOP 2008*. Springer. (to appear).

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

Boyan, J. and Moore, A. (2000). Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112.

Burke, E. K., Hyde, M. R., and Kendall, G. (2006a). Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 860–869, Reykjavik, Iceland. Springer-Verlag.

Burke, E. K., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003a). Hyper-heuristics: an emerging direction in modern search technol-

ogy. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 457–474. Kluwer Academic Publishers.

Burke, E. K., Kendall, G., and Soubeiga, E. (2003b). A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470.

Burke, E. K., Petrovic, S., and Qu, R. (2006b). Case-based heuristic selection for timetabling problems. *Journal of Scheduling*, 9(2):115–132.

Cowling, P., Kendall, G., and Han, L. (2002). An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In Fogel, David B., El-Sharkawi, Mohamed A., Yao, Xin, Greenwood, Garry, Iba, Hitoshi, Marrow, Paul, and Shackleton, Mark, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1185–1190. IEEE Press.

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

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

Fukunaga, A. (2004). Evolving local search heuristics for SAT using genetic programming. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103, pages 483–494, Seattle, WA, USA. Springer-Verlag.

Gent, I. P. and Walsh, T. (1993). Towards an understanding of hill-climbing procedures for SAT. In *Proc. of AAAI-93*, pages 28–33, Washington, DC.

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

Hoos, Holger H. and Stützle, Thomas (2000). Local search algorithms for SAT: An empirical evaluation. *J. Autom. Reason.*, 24(4):421–481.

Marchiori, E. and Rossi, C. (1999). A flipping genetic algorithm for hard 3-SAT problems. In Banzhaf, Wolfgang, Daida, Jason, Eiben, Agoston E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, Orlando, Florida, USA. Morgan Kaufmann.

Poli, Riccardo, Woodward, John, and Burke, Edmund (2007). A histogram-matching approach to the evolution of bin-packing strategies. In *Proceedings of the IEEE Congress on Evolutionary Computation*, Singapore. accepted.

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

Selman, B. and Kautz, H. (1993). Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of theInternational Joint Conference on Artificial Intelligence(IJCAI-93)*, ChambÃ©ry, France.

Selman, B., Kautz, H. A., and Cohen, B. (1994). Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle.

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

Silva, D. L., O'Brien, R., and Soubeiga, E. (2005). An ant algorithm hyper-heuristic for the project presentation scheduling problem. In Fogel, David B., El-Sharkawi, Mohamed A., Yao, Xin, Greenwood, Garry, Iba, Hitoshi, Marrow, Paul, and Shackleton, Mark, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, pages 92–99.