Generating SAT Local-Search Heuristics using a GP Hyper-Heuristic Framework

Mohamed Bader-El-Den and Riccardo Poli

Department of Computing and Electronic Systems, University of Essex, UK

Abstract. We present GP-HH, a framework for evolving local-search 3-SAT heuristics based on GP. The aim is to obtain "disposable" heuristics which are evolved and used for a specific subset of instances of a problem. We test the heuristics evolved by GP-HH against well-known local-search heuristics on a variety of benchmark SAT problems. Results are very encouraging.

1 Introduction

Hyper-heuristics could simply be defined as "heuristics to choose other heuristics" [4]. 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 targeted 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 two main classes of hyper-heuristics. In a first class of hyper-heuristic systems, which we term *HH-Class 1*, the system is provided with a list of preexisting heuristics for solving a certain problem. Then the hyper-heuristic system tries to discover what is the best sequence of application for these heuristics for the purpose of finding a solution. Different techniques have been used to build hyper-heuristic systems of this class. Algorithms used to achieve this include, for example: tabu search [5], case-based reasoning [6], genetic algorithms [7], ant-colony systems [22], and even algorithms inspired to marriage in honey-bees [1].

The second approach used to build hyper-heuristic systems aims at evolving new heuristics by making use of the *components* of known heuristics. We term this class *HH-Class 2*. This is the approach we will adopt also in this paper. 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 hyper-heuristic system (as an in *HH-Class 1* 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 hyper-heuristic 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 in *HH-Class 2*,

both the grammar and the heuristics components are given to the hyper-heuristic systems. The system then uses a suitable evolutionary algorithm to evolve new heuristics. For example, in recent work [3] genetic programming [13, 14] was successfully used to evolve new heuristics in *HH-Class 3* for one-dimensional online bin packing problems. Very positive results with evolving offline bin-packing heuristics have recently been obtained in [16] where GP was used to evolve strategies to guide a fixed solver. In general, we can say that the *HH-Class 2* approach has more freedom to create new heuristics for a given problem than *HH-Class 1*. However, *HH-Class 1* is easier to implement since it does not require the decomposition of heuristics nor the use of a grammar.

The long term goal in our research is investigating the use of GP as a hyper-heuristic framework for evolving instance-dependent heuristics. That is, the aim is not to obtain general heuristics, but effectively "disposable" heuristics which are evolved and used for a specific instance of a problem. Here, we will make a first step in this direction, by exploring the evolution of heuristics which are specialised to solve *specific subsets of instances* of a problem. In particular we evolve heuristics specialised to solve SAT problems with a fixed number of variables. We do this with a grammar based strongly-typed GP hyper-heuristic system, which we call *GP-HH*.

2 SAT problem

The target in the satisfiability problem (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 [8]. Incomplete algorithms are often based on local search heuristics (see Section 2.1). 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.

2.1 Stochastic local-search heuristics

Stochastic local-search heuristics have been widely used since in the early 90s for solving the SAT problem following the successes of GSAT [21]. The main idea behind these

Algorithm 1 General algorithm for SAT stochastic local search heuristics

```
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
```

heuristics is to try to get an educated guess as to which variable will most likely, when flipped, give us a solution or to 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.

2.2 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 in the first direction – direct evolution – is FlipGA which was introduced by Marchiori and Rossi in [15]. 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 [17]. A good overview of other algorithms of this type is provided in [12].

The second direction, which we also adopt in this paper, is to use evolutionary techniques to automatically evolve local search heuristics. A successful example of this is the CLASS system developed by Fukunaga [9, 10]. 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 the genetic programming system where these rules are used instead of the standard GP operators (crossover and mutation). The results of 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 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 by Boyan and Moore in [2]. 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.

3 GP-HH for SAT

To construct a grammar suitable to guide GP-HH in the solution of SAT problems, we used a number of the well-know local-search heuristics. We decomposed these heuristics into their basic components. The heuristics considered are the following:

- GSAT: [21] which, at each iteration, flips the variable with the highest gain score, where the gain of the 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: [11] In GSAT more than one variable may present the maximum gain. GSAT chooses among such variables randomly. HSAT, instead, uses a more sophisticated strategy. Its selects the variable with the maximum age, where the age of the variable is the number of flips since it is was last flipped. So, the most recently flipped variable has an age of zero.
- GWSAT: [19] 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: [20] starts by selecting one of the unsatisfied clauses C. Then it flips randomly one of the variables that have a gain score of 0 (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.

We have designed a simple but flexible grammar, which gives GP-HH enough freedom to evolve really new heuristics. By analysing the previous heuristics, we classified the main components of these 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).

After trying different grammar representations we decided to design the grammar in such a way to produce nested functions to avoid using variables for passing data from a function 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 1.

In Group 2 we have two more components which are not directly taken from the list of heuristics above. The first, which we call `ALL_USC` (which stands for all unsatisfied clauses), returns a list of non-repeated variables found in all the unsatisfied clauses. We found that this component performed well especially on instances with a relatively small number of variables, as will be shown later. The second additional component, which we call `RAND_USC` (which stands for random unsatisfied clause), returns the variables in a randomly selected clause. The main difference between `RAND_USC` and `USC`, which

```
start →  FLIP v
v     →  RANDOM l
       | MAX_SCR l | MAX_SCR l, op
       | IFV prob, v, v
       | MIN_SCR l | MIN_SCR l, op
       | MAX_AGE l | MAX_AGE l, op
l     →  ALL | ALL_USC
       | RAND_USC | USC
       | IFL prob, l, l
       | SCR_Z l | SCR_Z l, op
op    →  TIE_RAND | TIE_AGE
       | TIE_SCR | NOT_ZERO_AGE
prob  →  20 | 40 | 50 |
         70 | 80 | 90
```

Fig. 1. The grammar used for evolving heuristics for SAT using GP-HH.

also returns a random unsatisfied clause, is that `USC` returns the same unsatisfied clause during the course of the execution of a heuristic, while `RAND_USC` randomly selects a different clause each time it is invoked.

The primitive `SCR_Z` selects a zero-damage variable (as in WalkSAT). We placed this component in group 2. It returns the input list if no variable with zero score gain is found. If, instead, a zero-damage variable is found, it returns a list which includes this variable only.

Most primitives which accept a list as input are provided in two versions: one with a single list argument, and one with a list and an object of type `op`. The non-terminal symbol `op` in the grammar specifies how to break ties between variables whenever multiple variables in a list satisfy a selection criterion. When `op` is not provided, a default tie-breaking strategy is used. For example, in `MAX_SCR` – the component that returns the variable with the highest score gain – if multiple variables have the same highest score, the first variable is returned by default. However, if the optional parameter `op` is provided and it is `TIE_AGE`, the tie will be broken by favouring the variable which has least recently been flipped. In some cases a specific option may have no meaning with a particular component. For example, `TIE_SCR` breaks ties by favouring the variable with highest score. Naturally, when used in conjunction with `MAX_SCR` this option has no effect.

We also included probabilistic branching components (`IFV` and `IFL`) in our heuristics. We classify branching components on the basis of their return type. For example, if the branch is between selecting a random variable from a list and selecting the variable with the highest gain score, we consider this probabilistic branching component as in Group 1 since it returns a variable. The parameter `prob` represents the probability of returning the first argument of an `IFV` or an `IFL` primitive.

The grammar in Figure 1 could describe any of the heuristics discussed above. For example, a statement describing the GWSAT heuristic with a noise parameter of 0.5 could be written as `FLIP IFV 50, MAX_SCR ALL, TIE_RAND, RANDOM USC`,

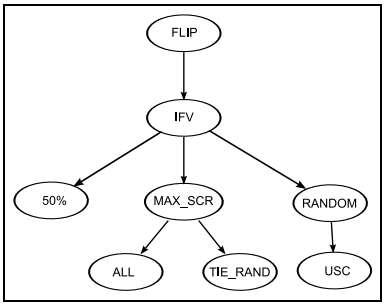

Fig. 2. GWSAT heuristic represented using the grammar adopted in GP-HH

where ALL returns all the variables in the CNF formula, TIE_RAND stands for "break ties randomly", MAX_SCR selects a variable with highest score and RANDOM selects a random variable from USC (unsatisfied clause). A tree representation of this individual is shown in Figure 2.

4 Experimental setup

We have implemented the full genetic programming hyper-heuristic framework for the SAT problem in C++ compiled with the gcc compiler. The system consists of two main parts: a grammar based GP engine and a SAT engine for handling SAT formulas.

The GP-HH system was applied to solve benchmark cases taken from the uniform random 3-SAT library SatLib.[1] All the problems in our benchmarks were satisfiable uniform random 3-SAT problems with 20, 50, 75 and 100 variables. To reiterate, the objective of the experiments was to evolve a separate heuristic that best performs on SAT instances of a given size, and not a general heuristics for 3-SAT problem. So, we are not trying to evolve heuristics that compete with general SAT solvers, although, as it will become clear later, unexpectedly we obtained solvers with a considerable degree of generality.

Normally a local-search heuristic starts by randomly initialising all the variables in the formula to either zero or one. We did this in testing. However, during evolution we started the evolved heuristics with all variables set to zero. This may have slightly reduced the total number of solved cases and may even have slightly increased the mean number of flips required by each heuristics in each run. We used this approach, however, because it reduces the randomness in the evolutionary process and makes it easier to compare results. Once again, this was done only during the evolution of heuristics, while in testing we initialised all the variables randomly, as customary.

The GP system initialises the population by using the grammar and selecting random primitives out of the functions and terminals that are consistent with the grammar. So, all initial heuristics are guaranteed to be syntactically valid SAT heuristic. The population is then manipulated by the following operators:

[1] A full set of benchmarks is available from http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

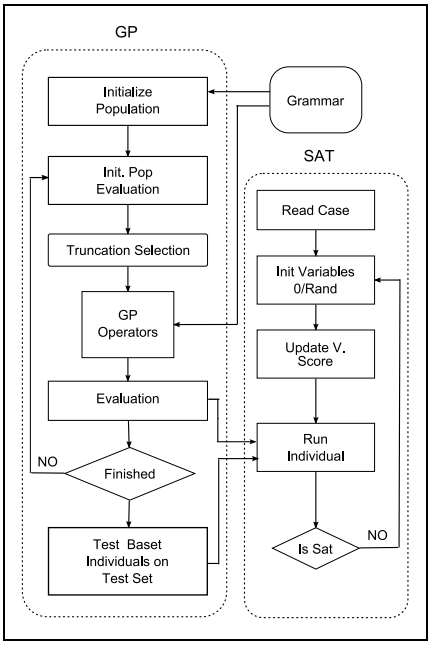

Fig. 3. Operations and interactions in the GP-HH framework.

- We use truncation selection, where only the best 40% of the population is allowed to reproduce.
- 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.
- Single point mutation is applied to 1% of the population. Again the grammar is used to ensure that all individuals are valid heuristics throughout the course of evolution.
- Individuals that have not been effected by any genetic operator are not evaluated again to reduce the computation cost of the evolution phase.

Figure 3 shows how the framework works and how the interaction between the two main engines, GP and SAT, operates.

As we mentioned before, we apply GP-HH to discover high-quality SAT solvers specialised for SAT instances of a particular size. So, we pass to the system sets of SAT instances all with the same number of variables. These form a training set of fitness cases on which individuals are tested. The fitness of each individual is based on three factors: a) how many cases have been solved out of the given fitness cases (SAT instances), b) the mean number of flips needed in the solved cases, and c) how many primitives (nodes) are present in an individual. Table 1 summarises the GP parameters used for each set of benchmarks.

Table 1. GP parameters

SAT set	Population size	Crossover rate	Mutation rate	Fitness cases	Max Flips
uf20	300	35%	1%	80	1000
uf50	250	40%	1%	100	4000
uf75	300	40%	1%	40	10000
uf100	250	40%	1%	100	12000

During the evaluation of the initial population, we use only a fraction of the training set. Also, the number of maximum flips allowed is smaller, than during the other generations. This is done to reduce the computation load involved with the evaluation of the initial population. Since this is randomly generated, a high percentage of individuals have very low performance. These time saving techniques help filter them out quickly.

Although the initial population in GP-HH is randomly generated and includes no handcrafted heuristics, individuals representing GSAT, HSAT and GWSAT were created in the initialisation in almost all experiments we did. This is because of their simple representation with our grammar. This gave evolved heuristic a chance to start competing with previously known good heuristics from the beginning. In some cases the standard heuristics dominated the early generations of runs. Nonetheless, GP was always able to eventually discover new and better heuristics, despite our using in all our training and testing sets hard SAT instances, where the clause-to-variable ratio is grater than or equal to 4.3. None of the instance used in testing and comparing the evolved heuristics have been used in the evolution phase.

5 Results

Evolving heuristics for SAT is hard, with each GP-HH run taking between a few hours to several days (for the biggest training sets) to complete. So, we cannot provide here a statistical analysis of GP-HH runs. All we can say is that most of our runs successfully evolved high-quality heuristics for the SAT instances in their training set. We feel that this deficiency is acceptable, since this is one of those cases where one is more interested in the *results* of a set of runs rather than the runs themselves, since the results of our runs are actual problem solvers. These we can study in great detail. So, in this section we present some of the results of the evolved heuristics for each instance set of the 3-SAT problem. We also compare the performance of these heuristics with that of well-known local-search SAT heuristics.

We start by showing a typical example of the heuristics evolved using the GP-HH framework. Figure 4 shows one of the best performing heuristics evolved for the 50-variables instance set (brackets were introduced to increase readability). As one can see evolved heuristics are significantly more complicated than the standard heuristics we started from (e.g., GWSAT). So, a manual analysis of how the component steps of an evolved heuristic contribute to its overall performance is very difficult.

However, it is possible to characterise the performance of SAT local search heuristics using certain numerical measures [18]. Depth and mobility are, perhaps, the two most important ones. *Depth* measures how many clauses remain unsatisfied during the

Table 2. Comparison of evolved and known SAT solvers on the uf100-953 instance set.

Solver	Mean depth	Mean mobility	Mean flips
GSAT()	2.13	5.7	99,006
WSAT(0.5)	5.65	15.7	9,421
Novelty(0.5)	4.76	18.9	4,122
GPHH100a	5.23	35,2	6,864
GPHH50a	8.17	42.9	11,154

execution of a heuristic. *Mobility* is a measure of how rapidly the heuristic moves in the search space. In general it is desirable to have algorithms with large mobility values which indicate that the heuristic is moving rapidly in the search space. Instead, it is better to have small values of depth, indicating that the average number of unsatisfied clauses is small during the course of execution of the heuristic.

Table 2 compares the depth and mobility of the GP-HH evolved heuristics against depth and mobility of reference human-designed heuristics. The comparison is done on the uf100-0953 SATLib benchmark, which consists of SAT instances with 100 variables and 430 clauses. The results for GSAT, HSAT and WSAT are taken form [18]. In this table we show two of the local search heuristics evolved using GP-HH, `GPHH100a`, which was evolved using 100-variable instances, and `GPHH50a`, which was evolved on 50-variable instances. In both cases SAT training instances were taken from the SATLib benchmark library. The results show that `GPHH100a` performs better than GSAT, HSAT and WSAT in terms of mobility and the average the number for flips used. However, GSAT and HSAT have lower (better) values of depth. This is because they use a very large number of flips, which cause these algorithms to have a smaller average number of unsatisfied clauses. `GPHH100a` did not outperform the Novelty heuristic, but the results are very close. We think this is a good result because `Novelty` is an extremely high performing heuristics and it wasn't one of the heuristics decomposed to construct our GP-HH grammar. So, we hope that by including components from `Novelty` in future research we may be able to further improve GP-HH. Table 2 also shows the performance of `GPHH50a`, that was trained on 50-variable instances. Despite this, it appears to perform rather well also on instances with 100 variable, showing some generalisation capability.

Some benchmark suites consisting of a number of SAT instances with between 30 and 100 variables were used in [12], where comparative results between a number of heuristics, some of which evolutionary, were presented. These suites have been used in a number of other studies. So, we chose the same suites to perform a wider range of tests on our evolved heuristics. In particular, we used Suite A, which encompasses instance with 30, 40, 50 and 100 variables, and Suite B, which includes instances with 50, 75 and 100 variables. More details can be found in [12].

In Tables 3 and 4 we provide comparative results of the GP-HH heuristics against other state-of-the-art evolutionary heuristics and human-designed heuristics on Suites A and B. The results of the GP-HH evolved heuristics are averages of 5 runs on the benchmark sets. In Tables 3 and 4 the results of the FlipGA and WSAT are taken from [12], while in Table 4 the results for `Novelty+` and `C2-D3` are taken from [10]. The num-

```
FLIP( IFV( 90, IFV( 40, MAX_SCR( ALL,
TIE_RAND), IFV( 70, RANDOM(USC)), IFV(
80, MAX_SCR( RAN_USC, NOT_ZERO_AGE), IFV(
20, MAX_SCR( ALL, TIE_RAND), MAX_SCR(
RAND_USC, TIE_AGE))))), IFV( 80, IFV(
50, MAX_SCR( ALL, TIE_AGE), MAX_SCR(
RAND_USC, TIE_RAND)), MAX_SCR( IFL( 70,
ALL_USC, USC) NOT_ZERO_AGE))))
```

Fig. 4. SAT heuristics evolved by GP-HH. Training set taken from the uf50 benchmark set.

ber of runs of these heuristics on the suites varied from 4 to 10. Note that we are testing evolved heuristics on *all* the instances in the suites. So, for example, heuristics evolved for 50 variable instances are also tested on the 75 and 100 variables instances. This gives us an indication of how general the heuristics are, though a thorough analysis of this issue is beyond the scope of this study.

In Table 3 and 4 two measures of the heuristics performance are shown: the success rate (SR) on the set and the average number of flips (AF) used by each heuristic. The results show that the heuristics evolved by GP-HH performed well compared to most local-search heuristics, outperforming some. The tables also show that the heuristics evolved by GP-HH outperformed FlipGA in terms of both the success rate and average number of flips.

From the results it can also be noticed that in some cases heuristics evolved for a instances with a larger number of variables have a considerable degree of generality, performing well also on problems with a smaller number variables.

6 Conclusion

In this paper we presented GP-HH, a framework for evolving "disposable" heuristics for the SAT problem, i.e., heuristics that are relatively fast to evolve and are specialised to solve specific sets of instances of the problem. We presented a comparison between GP-HH and other well-known evolutionary and local-search heuristics. The results show that the heuristics produced by GP-HH are competitive with these.

GP-HH produced heuristics that are on par with some of the best-known SAT solvers. We consider this a success. However, the heuristics evolved using CLASS2 are slightly better than the ones evolved by GP-HH. As mentioned in [10], these heuristics are slower than ours. This is because of the use of conditional branching as a GP primitive. As mentioned in Section 2.2 in most cases this requires to run two heuristics. We don't use this form of conditional branching (our branching instructions are probabilistic branches). So, GP-HH heuristics are faster than CLASS2 ones. Also the CLASS2 system used a large training sets and much longer evolutionary runs compared to GP-HH. It remains to be explored if by including more components in the grammar (e.g., those from `Novelty`), performing longer runs and feeding the GP-HH with a set of the well performing heuristics in the initial population, GP-HH could outperform CLASS2. This will be the target of our future research.

Table 3. Results for benchmark suite A. SR=success rate, AF=average number of flips (out of a maximum of 300,000). Results for FlipGA and WSAT are taken from [12].

	$n = 30$		$n = 50$		$n = 100$	
	SR	AF	SR	AF	SR	AF
FlipGA	1.0	25,490	1.0	127,300	0.87	116,653
WSAT	1.0	1,631	1.0	15,384	0.8	19,680
GPHH100a	1.0	1,864	1.0	12,872	0.92	54,257
GPHH50a	1.0	2,035	1.0	16,273	0.84	24,549
GPHH20a	1.0	1,457	0.95	18,384	0.66	32,781

Table 4. Results for benchmark suite B.

	$n = 50$		$n = 75$		$n = 100$	
	SR	AF	SR	AF	SR	AF
FlipGA	1.0	103,800	0.82	29,818	0.57	20,675
WSAT	0.95	16,603	0.84	33,722	0.6	23,853
Novelty+	N/A	N/A	0.966	17,018	0.716	34,849
C2-D3	N/A	N/A	0.972	19,646	0.786	40,085
GPHH100a	0.96	12,527	0.93	27,975	0.74	41,284
GPHH75a	1.0	18,936	0.95	26,571	0.59	29,495
GPHH50a	0.97	11,751	0.81	36,215	0.46	22,648

Furthermore, in future work we intend to test and evolve heuristics for a wider range of SAT problems. We also want to study the behaviour of GP-HH in more detail. In addition, we aim to further speed up evolution. Like most other GP systems, GP-HH populations include a large numbers of repeated individuals. So, a natural speed-up technique is to avoid the evaluation of repeated individuals. Additional savings could be obtained by avoiding the evaluation of repeated subtrees. We will also apply GP-HH to different combinatorial optimisation problems, e.g., job shop scheduling.

Acknowledgements

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

References

1. H. A. Abbass. 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, 27-30 2001. IEEE Press.
2. J. Boyan and A. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2000.
3. E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 860–869, Reykjavik, Iceland, 9-13 Sept. 2006. Springer-Verlag.

4. E. K. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: an emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 457–474. Kluwer Academic Publishers, 2003.
5. E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.
6. E. K. Burke, S. Petrovic, and R. Qu. Case-based heuristic selection for timetabling problems. *Journal of Scheduling*, 9(2):115–132, 2006.
7. P. Cowling, G. Kendall, and L. Han. An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1185–1190. IEEE Press, 2002.
8. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
9. A. Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 641–648, 2002.
10. A. Fukunaga. 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, 26-30 June 2004. Springer-Verlag.
11. 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.
12. J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Evol. Comput.*, 10(1):35–50, 2002.
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. R. Poli, J. Woodward, and E. Burke. A histogram-matching approach to the evolution of bin-packing strategies. In *Proceedings of the IEEE Congress on Evolutionary Computation*, Singapore, 2007. accepted.
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. D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.
19. 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.
20. 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.
21. 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, California, 1992. AAAI Press.
22. D. L. Silva, R. O'Brien, and E. Soubeiga. An ant algorithm hyperheuristic for the project presentation scheduling problem. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, pages 92–99, 2005.