Evolving Local Search Heuristics for SAT Using Genetic Programming

Alex S. Fukunaga

Computer Science Department, University of California, Los Angeles, CA 90024, USA
fukunaga@cs.ucla.edu,

Abstract. Satisfiability testing (SAT) is a very active area of research today, with numerous real-world applications. We describe CLASS2.0, a genetic programming system for semi-automatically designing SAT local search heuristics. An empirical comparison shows that that the heuristics generated by our GP system outperform the state of the art human-designed local search algorithms, as well as previously proposed evolutionary approaches, with respect to both runtime as well as search efficiency (number of variable flips to solve a problem).

1 Introduction

Satisfiability testing (SAT) is a classical NP-complete, decision problem. Let V be a set of boolean variables. Given a *well-formed formula* F consisting of positive and negative *literals* of the variables, logical connectives $\vee$, $\wedge$, For example, the formula $(a \vee b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee c)$ is *satisfiable*, because if $a = true, b = false, c = true$, then the formula evaluates to true. On the other hand, the formula $(a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee \neg c)$ is *unsatisfiable* because there is no assignment of boolean values to a,b,c such that the formula evaluates to true. SAT has recently become a very active field of research because many interesting real world problems (e.g., processor verification, scheduling) can be reformulated as SAT instances, spurring the development of fast SAT solvers. Local search procedures for satisfiability testing (SAT) have been widely studied since the introduction of GSAT [13], and it has been shown that for many problem classes, incomplete local search procedures can quickly find solutions (satisfying assignments) to satisfiable CNF formula.

The most widely used SAT local search algorithms can be succinctly described as the template of Figure 1, where the "key detail" is the choice of *variable selection heuristic* in the inner loop. Much research in the past decade has focused on designing a better variable selection heuristic, and as a result, local search heuristics have improved dramatically since the original GSAT algorithm. Some of the most successful local search algorithms include: GSAT with Random Walk [11], Walksat [12], Novelty/R-Novelty [7], and Novelty+/R-Novelty+ [5].

Evolutionary computation has also been applied to SAT. A thorough review of previous applications of evolutionary algorithms to SAT can be found in [3].

The **CLASS** (**C**omposite **L**earned **A**lgorithms for **S**AT **S**earch) system, proposed in [1] was developed in order to automatically discover variable selection

```
T:= randomly generated truth assignment
For j:= 1 to cutoff
 If T satisfies formula then return T
 V:= Choose a variable using some
  variable selection heuristic
 T':=T with value of V reversed
Return failure (no satisfying
  assignment found)
```

Fig. 1. SAT local search algorithm template

heuristics for SAT local search. It was noted in [1] that many of the best-known SAT heuristics were expressible as decision-tree like combinations of a set of primitives, and thus, it should be possible for a machine learning system to automatically discover new, efficient variable selection heuristics by exploring the space of combinations of these primitives.

Whereas previous applications of evolutionary computation to SAT has focused on the design of evolutionary algorithms that searched the space of candidate solutions to the SAT problem directly, CLASS applies EC at the meta-level by generating heuristics which are embedded in a standard local search framework. It was shown that CLASS was capable of generating heuristics that were competitive with the best known SAT local search heuristics, with respect to the number of variable flips required to solve benchmark problems.

A major shortcoming of the original CLASS system was that although the automatically generated heuristics reduced the number of flips to find satisfying assignments, the actual runtime required to solve the problems was much slower (up to an order of magnitude) than the best available implementations of the standard algorithms. This was largely due to the increased complexity of the variable selection heuristic. It seemed that CLASS was making an unfavorable tradeoff between search efficiency, and the time spent on each decision. Although search efficiency is a key criterion for evaluating a SAT algorithm, in the end, what matter for practical applications (e.g., SAT solvers embedded in circuit verification tools) is runtime. Thus, from [1] it was not possible to conclude that CLASS had generated heuristics that were *truly better* than than state of the art algorithms. Furthermore, CLH-1, the best heuristic generated by CLASS, was generated by using a population pre-seeded with hand-coded heuristics, and used very sophisticated primitives which by themselves were as complex as the Novelty and R-Novelty heuristics. Although [1] reported two other heuristics (CH-1 and CH-2) which did not rely on this sophisticated "seed library" feature, it remained an open question whether a learning system using only simpler primitives could outperform CLH-1.

This paper describes CLASS2.0, which addresses the open issues described above which could not be resolved with the first version, and provides empirical evidence that genetic programming can be used to generate heuristics that outperform the state of the art SAT local search heuristics with respect to both runtime and the number of steps executed. In section 2, we give an overview of CLASS2.0, highlighting the improvements made compared to the first version. We then present new empirical results, comparing the heuristics learned by CLASS2.0 to state of the art local search algorithms, previously studied evolutionary approaches to SAT, and heuristics generated by CLASS1.0.

2 The CLASS2.0 System

2.1 Key Concepts and Definitions

We introduce some terminology to facilitate the discussion of the common structural elements of Walksat-family SAT variable selection heuristics throughout this paper.

Definition 1 (Positive/Negative/Net Gain) *Given a candidate variable assignment T for a CNF formula F, let B_0 be the total number of clauses that are currently unsatisfied in F. Let T' be the state of F if variable V is flipped.*

Let B_1 be the total number of clauses which would be unsatisfied in T'. The net gain *of V is $B_1 - B_0$. The* negative gain *of V is the number of clauses which are currently satisfied in T, but will become unsatisfied in T' if V is flipped. The positive gain of V is the number of clauses which are currently unsatisfied in T, but will become satisfied in T' if V is flipped.*

Definition 2 (Variable Age) *The* age *of a variable is the number of flips since it was last flipped.*

The best-known SAT variable selection heuristics can be succinctly described in terms of the above definitions. For example:
GSAT [13]: Select variable with highest net gain.
Walksat [12]: Pick random broken clause BC from F. If any variable in BC has a negative gain of 0, then randomly select one of these to flip. Otherwise, with probability p, select a random variable from BC to flip, and with probability $(1 - p)$, select the variable in BC with minimal negative gain (breaking ties randomly).
Novelty [7]: Pick random unsatisfied clause BC. Select the variable v in BC with maximal net gain, unless v has the minimal age in BC. In the latter case, select v with probability $(1-p)$; otherwise, flip v_2 with second highest net gain.

2.2 A Genetic Programming Approach to Learning Heuristic Functions

The CLASS system was motivated by the following key observations:

First, the well-known, SAT local search heuristics could all be expressed as a combination of a relatively small number of primitives. These primitives included: (1) scoring of variables with respect to a gain metric, (2) ranking of variables and greediness. (3)variable age, and (4) conditional branching Heuristics such as GSAT, Walksat, Novelty, Novelty+, could be implemented as relatively simple functions built by composing the various primitives discussed above. Even R-Novelty[7], which is one of the most complex heuristics proposed to date, can be represented as a 3-level decision diagram [5].

Second, historically, new heuristics were often generated by "structural" innovations which recombined existing primitives in new ways. For example, GWSAT and Novelty+ added random walk to GSAT and Novelty after observing behavioral deficiencies of the predecessors [11, 5].

Third, it appeared relatively difficult for even expert researchers to design successful heuristics, and the design of new heuristics required However, some major structural innovations involve considerable exploratory empirical effort. For example, [7] notes that over 50 variants of Walksat were evaluated in their study (which introduced Novelty and R-Novelty).

From these observations, [1] claimed that the domain of *SAT heuristic design* seemed to be a good candidate for the application of a genetic programming approach. CLASS represents variable selection heuristics in a Lisp-like s-expression language. In each iteration of the innermost loop of the local search (Figure 1), the s-expression is evaluated in place of a hand-coded variable selection heuristic.

To illustrate the primitives built into CLASS, Figure 2 shows some standard heuristics represented as CLASS s-expressions. Due to space restrictions, we can not define the language here, but a more detailed description of the CLASS language primitives can be found in [1].

```
Walksat:
(IfVarCond == NegativeGain 0
   VarBestNegativeGainBC0
   (If (rand 0.5)
        VarBestNegativeGainBC0
   RandomVarBC0))
```

```
Novelty:
(IfNotMinAge BC0 VarBestNetGainBC0
    (If (rand 0.5)
        VarBestNetGainBC0
        VarSecondBestNetGainBC0))
```

Fig. 2. Walksat and Novelty represented in the CLASS language.

CLASS uses a population-based search algorithm similar to strongly-typed genetic programming [9] to search for good variable selection heuristics (Figure 3). This algorithm is conceptually similar to the standard genetic programming algorithm [6]. The `Initialize` function creates a population of randomly generated s-expressions. The expressions are generated using a context-free grammar as a constraint, so that each s-expression is guaranteed to be a syntactically valid heuristic that returns a variable index when evaluated. The `Select` function picks two s-expressions from the population, where the probability of selecting a particular s-expression is a function of its rank in the population according to its objective function score (the higher an expression is ranked, the more likely it is to be selected). The `Compose` operator (detailed below) is applied to the parents to generate a set of children, which are then inserted into the population. Each child replaces a (weighted) randomly selected member of the population, so that the population remains constant in size (the lower the ranking, the more likely it is that an s-expression will be replaced by a child). The best heuristic found during the course of the search is returned.

The most significant difference between the CLASS learning algorithm and standard strongly typed genetic programming is how children are created. Instead of applying the standard crossover and mutation operators, CLASS uses a *composition* operator, which is intended to generate children whose behavior is a blend of the parents' behavior. Given two s-expressions `sexpr1` and sexpr2, composition combines them by creating a new conditional s-expression where the "if" and "else" branches are sexpr1 and sexpr2, i.e., `(conditional-sexpr sexpr1 sexpr2)`. The actual composition operator results in 10 children being created at each mating, where each child is the result of a different conditional s-expression combining the two parents. Each child is evaluated, and then is inserted into the population. CLASS uses composition rather than the standard genetic operators because (1) composition was a good model of how human researchers derived some of the most successful SAT local search heuristics, and (2) if at least one of the parent heuristics has the PAC-property [4], a subset of the children derived via composition are guaranteed to be PAC[1].

We have also explored the use of a more "standard" strongly-typed GP using standard crossover, selection, and mutation operators to CL. However, we have not yet been able to discover a parametrization of the standard GP to be competitive with our current algorithm.

The composition operator used by CLASS naturally leads to code "bloat", since each child will be, on average, twice as complex as its parents. Uncontrolled bloat would be a major problem in CLASS because fast evaluation of the s-expression heuristics is critical to achieving our goal of fast local search runtimes. We implemented a depth bound so that any interior nodes in the s-expression tree that exceed the given depth bound in the run are transformed to a leaf node (with a compatible type).

```
Initialize(population,populationsize)
For I = 1 to MaxIterations
Select parent1 and parent2 from population
 Children = Compose(parent1,parent2)
 Evaluate(Children)
 Insert(Children,Population)
```

Fig. 3. CLASS Meta-Search Algorithm

2.3 Fitness Function

A candidate s-expression is executed on a set of training instances from the distribution of hard, randomly generated 3-SAT problems from the SAT/UNSAT phase transition region [8].[1] First, the heuristic is executed on 200, 50 variable, 215-clause random 3-SAT instances, with a cutoff of 5000 flips. If more than 130 of these local search descents was successful, then the heuristic is run on 400,

[1] `mkcnf` (ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances/Cnfgen.tar.Z) was used to generate training instances, using the flag to force satisfiable.

100-variable, 430-clause random 3-SAT instances with a 20000 flip cutoff. The score of an individual is: (# of 50-var successes) + (5 * (# of 100-var successes)) + $(1/MeanFlipsInSuccessfulRuns)$ The 50-variable problems serve to quickly filter out very poor individuals and not waste time evaluating them further with more expensive training instances, while the larger 100-variable problems enable us to gain finer distinctions between the better individuals.

2.4 CLASS2.0

From the above description, it should be clear that searching for good heuristics using CLASS is *extremely* time-consuming. Each call to the evaluation function requires hundreds of descent of a local search algorithm, and during each descent, thousands or variable flips are executed, and each flip requires the evaluation of the candidate heuristic, i.e., each call to the fitness function results in hundreds of thousands of evaluations of the candidate s-expression. Each run of the CLASS genetic programming system (5000 fitness evaluations) took several days on a Pentium-3 based workstation, making experimentation with the system very cumbersome.

Although the original version, **CLASS1.0**, was able to generate some promising results, as reported in [1], it became difficult to make further progress without making the system substantially faster. In addition, the most important outstanding question was whether the heuristics discovered by CLASS were actually better than the state of the art, human-designed heuristics after the additional complexity of the heuristic was considered. Throwing additional computing power at the problem (e.g., using additional, faster CPUs in a parallel GP system) would make it easier to perform experiments, but it would not settle the issue of whether the resulting heuristics were actually faster than the standard heuristics when executed on the same CPU. Furthermore, the use of the *library* feature in CLASS1.0 (i.e., the CLASS-L extensions), which seeded the initial population with hand-coded heuristics, made it difficult to assess how powerful the representation was.

Therefore, after carefully studying the various performance bottlenecks in CLASS1.0, we developed a new system, **CLASS2.0**, which sought to address the crucial performance issues. Profiling CLASS showed that there were two functions that consumed the vast majority of computational resources. Not surprisingly, the execution of the variable selection heuristic code was the biggest bottleneck, especially when the s-expressions representing the heuristic were large (since the complexity of evaluating the s-expression grows exponentially with the depth of the expression tree). Second, when the variable selection heuristics were small (roughly depth 3 or shallower), then the function for flipping a variable and incrementally updating the gain values (defined in section 2.1) which were used by the variable selection heuristic was a major bottleneck. We addressed both of these issues. Also, all of the the CLASS-L extension features were eliminated from CLASS2.0.

2.5 Faster s-expression evaluation

CLASS1.0 was an ad-hoc, multilingual implementation, where the genetic programming component was implemented in `tcl`, and the local search engine was implemented in C++. The GP component would generate a candidate s-expression (as described above), then this s-expression would be translated to an equivalent C++ expression, which would then be compiled using the C++ compiler (gcc) and linked to the local search engine.

The fundamental problem with this architecture was that although the GP component must emit code that is executed by the local search component, it is extremely difficult and cumbersome, to correctly implement code in the GP component with enough introspective capability into the s-expression local search component such that the s-expressions could be optimized for fast execution. The current implementation, CLASS2.0, is implemented entirely in Common Lisp, allowing seamless integration between the GP and local search components. Although Lisp is often cited as being an interpreted language that is significantly slower than C/C++, modern ANSI Common Lisp implementations, using optional type declarations, are usually able to generate native machine code comparable (i.e., not much worse than a factor of 2) to code generated by C++ compilers. All of the results in this paper were obtained using CMUCL version 18e (http://www.cons.org/cmucl/)

Several improvements were implemented in order to enable faster expression of the evolved heuristic s-expressions. First, when a child is generated, CLASS2.0 probabilistically performs an *approximate simplification* of the s-expression. For example, consider the expression on the left below. The function `((rand x)` returns a random variable between 0 and x. Note that the probability of s-expr1 being evaluated is only 0.01. Thus, we can eliminate that subtree and "approximately" simplify the expression to the expression on the right.

```
(if (rand 0.1)                (if (rand 0.1)
    (if (rand 0.1)  -->           s-expr2
        s-expr1                   s-expr3)
        s-expr2)
    s-expr3)
```

This results in behavior that approximates the behavior of the original s-expression (i.e., 99% of the time, the modified expression returns the same value as the original expression), but executes faster due to the simpler structure. There is a tradeoff between the evaluation speedup gained by this technique versus possible loss in search efficiency due to the approximation.

Another improvement to s-expression evaluation was in the efficient computation and caching of subexpression values. For example, if an s-expression requires the evaluation of both "the variable in broken clause `BC1` with highest net gain", and "the variable in `BC1` with second highest net gain", then both of these can be computed by a single pass through the variables in BC1, rather than one pass for each query.

Also, if the same value is required multiple times during the evaluation of an s-expression, then it is worth caching the result when first computed. However, if it can't be guaranteed that an expression is required more than once (e.g., when it appears in a conditional subexpression), then it may in fact hurt performance to pay the overhead of caching the first instantiation. Before evaluating an evolved s-expression, CLASS2.0 applies a number of rules to transform the original s-expression into a semantically equivalent expression where some of the functions and terminals are the caching equivalents of the original functions and terminals (if the transformation module determines it worthwhile to do so).

2.6 Faster local search engine

The local search engine of CLASS1.0, while reasonably fast, did not have the fastest possible implementation of variable flipping. CLASS2.0 implements the most efficient local search engine found so far in the literature, with some enhancements that yielded significant speedups. A detailed description of these, as well as an extensive survey of efficient data structures and algorithms to support local search can be found in [2].

The standard implementation of SAT local search is the C implementation by Henry Kautz and Bart Selman (Walksat version 43).[2] This is an efficient, actively maintained implementation. We compiled Walksat-v43 using gcc and -O3 optimization. To evaluate the performance of the CLASS2.0 local search engine, we compared it against Walksat-v43, as well as CLASS1.0, as follows:

We hand-coded the standard Walksat-SKC heuristic [7] as an s-expression in the CLASS language (See Figure 2), generating CLASS1.0 and CLASS2.0 implementation of Walksat (Walksat-CLASS1.0, and Walksat-CLASS2.0). Each of the three solvers (Walksat-v43, Walksat-CLASS1.0, Walksat-CLASS2.0) was executed on the `uf250` set of 100, 250 variable, 1065 clause instances[3]. Each instance was run 10 times, with max-flips set to 10 million flips per instance. All experiments in this paper were performed on a 1.3GHz AMD Athlon CPU running Linux.

The results were as follows: All three solvers were 100% successful (i.e., all tries found satisfying assignments within the max-flips limit). Walksat-v43 solved all instances with an average of 73338 flips, executing 448962 flips per second. Walksat-CLASS2.0 solved all instances with an average of 68933 flips, executing 403781 flips per second. Walksat-CLASS1.0 solved all instances in 69014 flips, executing 278060 flips per second. As expected, the search effort (# of flips to solution) is almost identical for all three engines. The CLASS2.0 engine (implemented in Common Lisp) obtains 90% of the flip rate of the Walksat-v43 engine (implemented in C). CLASS2.0 is substantially faster than the CLASS1.0 engine (implemented in C++) due to significantly improved data structures. In theory, the CLASS2.0 engine should be somewhat faster if the entire system were rewritten in C, but we were satisfied that the underlying infrastructure was

[2] available at http://www.cs.washington.edu/homes/kautz/walksat
[3] available at http://satlib.org

fast enough so that the evolved heuristics, when embedded into the CLASS2.0 framework, had a chance of outperforming Walksat-v43.

3 Empirical evaluation of evolved heuristics

We compare the performance of a CLASS2.0-generated heuristic to the performance of the state of the art local search and evolutionary algorithms.

First, we extend the comparative study of SAT algorithms performed by Gottlieb, Marchiori, and Rossi [3] to include heuristics generated by CLASS. We used identical benchmark instances and computational resources, in order to allow a direct comparison with previously evaluated algorithms. The set `Suite B` consists of 50 75-variable problems and 50 100 variable problems. The set `Suite C` consists of 100 80-variable problems and 100 100-variable instances. All of the instances[4] are hard, satisfiable random 3-SAT instances from the phase-transition region, with a clause-to-variables ratio of 4.3. [5]

As with previous studies, we measured: (1) the *success rate*, which is the percentage of successful runs (runs where a solution is found), using a computation limit of 300,000 variable flips, and (2) the *average # of flips to solution (AFS)*, the mean number of flips to find a solution in a run, when a solution was found (i.e., flips in unsuccessful runs are not counted).

Algorithm	Suite B, n=75			SuiteB, n=100			Suite C, n=80			Suite C n=100		
	SR	AFS	time	SR	AFS	time	SR	AFS	time	SR	AFS	time
RFEA2+	96	39396	n/a	81	80282	n/a	95	49312	n/a	79	74459	n/a
ASAP	87	39659	n/a	59	43601	n/a	72	45492	n/a	61	34548	n/a
Walksat-v43	88.8	50706	0.21	62.6	44335	0.47	78.4	54178	0.34	67.0	40080	0.43
Novelty+	96.6	17108	0.06	71.6	34849	0.29	89.7	33518	0.16	75.7	31331	0.25
C2-D3	97.2	19646	0.11	78.6	40085	0.35	90.8	25744	0.19	80.9	37815	0.42
C2-D6	98.2	15199	0.16	82.0	38883	0.69	95.1	27527	0.32	83.8	36762	0.63

Table 1. Results with cutoff=300,000 flips: Success Rate (SR) (%), Average Flips to Solution (AFS), and Runtime (10 runs/instance for Walksat,Novelty+, C2-D3, and C2-D6

The data for RFEA2+ and ASAPare copied from [3]. The results for Suite B are based on 50 runs per instance for RFEA+, and 10 runs per instance for ASAP. For Suite C, the data for RFEA2+ is for 4 runs, and ASAP is for 5 runs.

The results for Walksat-v43 were obtained by running Kautz and Selman's Walksat version 43 code (see Section 2.6) using the `-best` heuristic flag and a noise setting of 50. This is the same configuration as the **WSAT** runs reported in [3]; we reran the algorithm so that we could obtain runtimes for comparison.

[4] Downloaded from http://www.in.tu-clausthal.de/ gottlieb/benchmarks/3sat

[5] Due to space restrictions, we only show results for the *largest* problem instances in Suites B and C of [3]. Experiments were performed with all 662 instances from Suites A, B, and C. the results were qualitatively similar for Suite A (omitted here because there were only 12 instances), and the smaller instances in Suites B and C excluded from Figure 1.

We also tried to find the best possible configuration of the Walksat-v43 solver (which implements not only Walksat but all of the `novelty` variants) on this benchmark set, by trying the `-best`, `-novelty`, `-rnovelty`, `-novelty+`, and `+rnovelty+` heuristics for a noise ranging from 10 to 90 (tested at 5% increments), and report the results for the best configuration on this benchmark set, which was Novelty+ (noise=75).

The heuristics evolved by CLASS2.0 included in this study are **C2-D3** and **C2-D6**. These are the results of running CLASS2.0 for 5500 individual evaluations. C2-D3 was evolved with a depth limit on the s-expression of 3, while C2-D6 was evolved with a depth limit of 6. We emphasize that *the training instances used during evolution are different from the benchmark instances*, although they are from the same problem class (generated using the mkcnf utility with the same control parameters).

The results in Table 1 show that the evolved heuristics C2-D3, and C2-D6 are quite competitive with the previous evolutionary algorithms as well as the state of the art local search algorithms. C2-D6 has a higher success rate than C2-D3, although it is paying a price in runtime due to the greater complexity of its s-expression.

Note that we simply picked C2-D3 and C2-D6 from the first GP runs we performed using a depth 3 limit and depth 6 limit. Heuristics with similar scores can be routinely evolved (see Section 4)

While the above study followed the standard methodology and allowed a direct comparison with the results of [3], the relatively low cutoff limit and the use of the success rate + AFS metric makes it difficult to see exactly how much faster one algorithm is than another. To provide another perspective, we repeated the experiments using a much larger resource threshold of 100,000,000 flips, and 1 run per instance.

The results are as shown in Table 2. For the hardest problems, C2-D3 and C2-D6 clearly outperformed Walksat-v43 and Novelty+.

Algorithm	Suite B, n=75			SuiteB, n=100			Suite C, n=80			Suite C n=100		
	SR	AFS	time	SR	AFS	time	SR	AFS	time	SR	AFS	time
Walksat-v43	100	244856	0.13	96	3134042	15.8	100	1790021	3.79	98	3633661	12.21
Novelty+	100	31027	0.07	100	2218112	5.42	100	167908	0.41	100	1228156	3.07
C2-D3	100	85555	0.34	100	267770	1.04	100	62194	0.24	100	233481	0.90
C2-D6	100	33656	0.28	100	404983	3.38	100	47506	0.40	100	268342	2.21

Table 2. Results with cutoff=100,000,000 flips: Success Rate (SR) (%), Average Flips to Solution (AFS), and Runtime

4 Empirical evaluation of the evolutionary algorithm

Is the GP-like population-based algorithm used in CLASS (Figure 3) a reasonably efficient meta-level search algorithm? We compared our algorithm to a simple, random generate-and-test (G&T) algorithm. G&T simply samples the space of s-expressions by repeatedly calling the random s-expression generated used during the initialization of the population in Figure 3. We ran each algorithm 10 times. Each run involved 5500 fitness function evaluations. For the GP, we generated and evaluated a population of 1000 and ran until 4500 children had been generated and evaluated. For G&T, 5500 random individuals were evaluated. Both algorithms were run with an s-expression depth limit of 6. We then compared the fitness function scores (Section 2.3) of the best individual found by the two algorithms. The mean value of the *best* individual score over 10 runs for the GP was 1377, and was 1213 for G&T. Also, the mean fitness of the final population in the GP was 1306, while the mean fitness of all individuals generated by G&T was 411. These results show that the GP was searching the space more effectively than G&T, and that the GP population was focusing on high-quality individuals (as opposed to getting lucky and generating a single better individual than G&T).

5 Conclusions

This paper summarized extensions to the work started in [1]. We showed that CLASS2.0, a genetic programming system for learning variable selection heuristics for SAT local search could generate heuristics that were truly competitive (both in terms of runtime and search efficiency) with state of the art local search algorithms, as well as previous evolutionary approaches.

Previously, a number of algorithms have been proposed which outperform Walksat and Novelty+ with respect to the number of flips required to solve satisfiable instances, including those reported in [1], as well as algorithms that used more sophisticated objective functions and/or clause weighting (e.g., SDF [10] and previous evolutionary algorithms [3]). Such previous studies have been more concerned with search efficiency than with the runtimes of the algorithms. However, search efficiency does not necessarily imply a superior algorithm, because the per-flip overhead of an efficient search algorithm , incurred due to a more complex objective function (e.g., clause weighting) or variable selection heuristic may be inherently too high for it to outperform the standard algorithms in practice. With the recent interest in real-world applications that have been formulated as SAT instances, SAT is no longer merely an academic benchmark problem. In the end, all that matters is speed, and a SAT solver must actually *run faster* than the standard algorithms in order to be truly useful to practitioners. From this perspective, CLASS1.0 was *far* from the goal of generating human-competitive results. A substantial engineering effort to implement CLASS2.0 was required in order to demonstrate that faster SAT solvers could indeed be generated using this approach.

Thus, the main contribution of this paper is a demonstration that genetic programming can be used to attack an important problem that has been intensively studied by human researchers for over a decade, and can routinely generate algorithms that objectively outperform the state of the art, hand-developed heuristics for a class of difficult instances. The performance of the evolved heuristics generalize fairly well to instances from other classes of SAT instances (not presented here due to space constraints, but generalization results for `C2-D3` and `C2-D6` are similar to those obtained for the heuristics evolved by CLASS-1.0[1]. Further improvements to the learning process to maximize the generalization is the major area of future work.

In addition, we showed that this could be accomplished without seeding the population with a "library" of hand-coded solutions or excessively complex primitives, as was done in [1]. We also showed that the CLASS genetic programming algorithm significantly outperformed random search in the space of CLASS-language s-expressions. These results confirm that the CLASS population-based algorithm is indeed a reasonable meta-learning algorithm, i.e., that the power of the system is not just in the representation but in the combination of the representation and the search algorithm.

Note: Source code for CLASS, including the C2-D3 and C2-D6 expressions, is available at the author's home page (search Google for current location).

References

1. A. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In *Proc. AAAI*, pages 641–648, 2002.
2. A. Fukunaga. Efficient implementations of sat local search. to appear in Proceedings of SAT-2004, May 2004.
3. J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Evolutionary Computation*, 10(1):35–50, 2002.
4. H. Hoos. *Stochastic local search - methods, models, applications.* PhD thesis, TU Darmstadt, 1998.
5. H. Hoos and T. Stutzle. Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24:421–481, 2000.
6. J. Koza. *Genetic Programming: On the Programming of Computers By the Means of Natural Selection.* MIT Press, 1992.
7. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. AAAI*, pages 459–465, 1997.
8. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proc. AAAI*, pages 459–65, 1992.
9. D. Montana. Strongly typed genetic programming. Technical report, Bolt, Beranek and Neuman (BBN), 1993.
10. D. Schuurmans and F. Southey. Local search characteristics of incomplete sat procedures. *Artificial Intelligence*, 132:121–150, 2001.
11. B. Selman and H. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *Proc. Intl. Joint Conf. Artificial Intelligence (IJCAI)*, 1993.
12. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI*, 1994.
13. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AAAI*, pages 440–446, 1992.