Automated Discovery of Local Search Heuristics for Satisfiability Testing

Alex S. Fukunaga fukunaga@is.titech.ac.jp
Tokyo Institute of Technology, Meguro, Tokyo, 152-8550, Japan

Abstract

The development of successful metaheuristic algorithms such as local search for a difficult problems such as satisfiability testing (SAT) is a challenging task. We investigate an evolutionary approach to automating the discovery of new local search heuristics. for SAT. We show that several well-known SAT local search algorithms such as Walksat and Novelty are composite heuristics that are derived from novel combinations of a set of building blocks. Based on this observation, we developed CLASS, a genetic programming system that uses a simple composition operator to automatically discover SAT local search heuristics. New heuristics discovered by CLASS are shown to be competitive with the best Walksat variants, including Novelty+. Evolutionary algorithms have previously been applied to directly evolve a solution for a particular SAT instance. We show that the heuristics discovered by CLASS are also competitive with these previous, direct evolutionary approaches for SAT. We also analyze the local search behavior of the learned heuristics using the depth, mobility, and coverage metrics proposed by Schuurmans and Southey.

Keywords

Genetic programming, satisfiability, constraint satisfaction, SAT, hyper-heuristic, hybrid genetic-local search

1 Introduction

Metaheuristics such as evolutionary algorithms and local search are powerful, generic tools for solving difficult, combinatorial optimization problems and constraint satisfaction problems. However, for difficult problems such as the TSP and SAT, which have already been the subject of years of intense study by researchers, developing a new metaheuristic implementation that performs well enough to advance the current state of the art for that problem is a very difficult task.

In this paper, we investigate the use of evolutionary computation to automatically generate new, metaheuristic strategies for the *Boolean satisfiability testing problem (SAT)*. Satisfiability testing 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 $\vee$, $\wedge$, the satisfiability problem is to determine whether there exists an assignment of true/false values to the variables in V such that F is true. 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 $(a \vee b) \wedge (\neg a) \wedge (\neg b)$ is *unsatisfiable* because there is no assignment of Boolean values to a, b such that the formula evaluates to true. SAT is currently a very active field of research because many interesting and important applications such as processor verification (Velev and Bryant, 2003) and AI planning (Kautz, 2006) can be reformulated as SAT instances. Local search procedures for SAT have been widely studied since the introduction of the first successful SAT local search algorithms in the early 1990's (Selman, Levesque, and Mitchell, 1992), and it has been shown that for many problem classes, incomplete local search procedures can quickly find solutions (satisfying assignments) to satisfiable CNF formula. Many new local search heuristics have been proposed, and local search heuristics have improved dramatically since the original GSAT algorithm. Significant improvements have included GSAT with Random Walk (Selman and Kautz, 1993), Walksat (Selman, Kautz, and Cohen, 1994), Novelty/R-Novelty (McAllester, Selman, and Kautz, 1997), and Novelty+/R-Novelty+ (Hoos, 1999).

This paper describes a genetic programming system which automatically discovers new SAT local search heuristics. We begin in Section 2 with an overview of SAT local search. We review the well-known, SAT local heuristics from the literature and identify the common building blocks of those heuristics. We then analyze the historical process by which these heuristics were discovered by researchers and show that recent advances were the result of combining existing building blocks. In Section 3, we describe CLASS, a genetic programming system that searches for good SAT variable selection heuristics. We empirically evaluate the performance of new heuristics evolved by CLASS. In Section 4, we show that CLASS can successfully discover variable selection heuristics which are competitive with standard local search algorithms, as well as previous evolutionary approaches to SAT. We analyze the local search behavior of the heuristics generated by CLASS, using a set of depth, mobility, and coverage metrics proposed by Schuurmans and Southey (2001). We also analyze the performance of the CLASS GP algorithm and show that it is successfully searching the space of candidate heuristics (Section 6). Section 7 compares our work to related work on automated composition and improvement of problem-solving heuristics, such as other systems from the AI literature on synthesizing problem solving heuristics, e.g., (Minton, 1996), as well as related work in the evolutionary computing literature such as hyper-heuristics (Burke, Kendall, Newall, Hart, Ross, and Schulenburg, 2003). We conclude in Sections 8-9 with a discussion of our results and directions for future work.

2 Local Search Algorithms for SAT

A generic SAT local search algorithms is shown in Figure 1. After generating an initial, random assignment of true/false values to the variables, one variable is "flipped" at every iteration until a solution is found or the algorithm runs out of time. The key design decision that distinguishes each of the different local search strategies is the design and implementation of the *variable selection heuristic* in the inner loop (Figure 1, Line 4), which is the procedure that decides the next variable to flip.

Many of the standard SAT local search procedures in the literature can be characterized as instances of the template of Figure 1 with a particular variable selection heuristic. We now introduce some terminology to facilitate the discussion of the common elements of GSAT/Walksat-family SAT variable selection heuristics throughout this paper.

```
1. A:= randomly generated truth assignment
2. For j:= 1 to cutoff
3.    If A satisfies formula then return A
4.       V:= Choose a variable using a **variable selection heuristic**
5.       A:=A with value of V flipped
6. Return FAILURE (no satisfying assignment found)
```

Figure 1: A generic SAT local search algorithm.

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* broken *(unsatisfied) in F. Let T' be the state of F if variable V is flipped. Let B_1 be the total number of clauses that are broken in T'. The* net gain *of V is $B_1 - B_0$. The* negative gain *of V is the number of clauses that are satisfied in T, but broken in T'. The* positive gain *of V is the number of clauses that are broken in T, but satisfied in T'.*

Definition 2 (Variable Age) *The* age *of a variable is the number of variable flips since it was last flipped. E.g., if we flip variable v_1, and then on the next iteration flip v_2, then at that point in time, v_1 has an age of 2, and v_2 has an age of 1.*

The following are some well-known heuristics that have been proposed in the SAT literature:

GSAT (Selman et al., 1992): Select variable from the formula with highest net gain. Break ties randomly.

HSAT (Gent and Walsh, 1993): Same as GSAT, break ties in favor of maximum age variable.

GWSAT(p) (Selman and Kautz, 1993): (Also known as "GSAT with Random Walk") With probability p, randomly select a variable in a randomly selected broken clause BC; otherwise same as GSAT.

Walksat(p) (Selman et al., 1994): 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(p) (McAllester et al., 1997): Pick random broken 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.

Novelty+(p,p_w) (Hoos, 1999): Same as Novelty, but after BC is selected, with probability p_w, select random variable in BC; otherwise continue with Novelty.

R-Novelty(p) (McAllester et al., 1997): Behaves the same as Novelty, except when the maximal net gain variable has the minimal age. In that case, let $best$ denote the variable with highest net gain, $secondbest$ the variable with second highest net gain, and $n = netgain(best) - netgain(secondbest)$. Let p be a randomly generated number between 0 and 1. There are four cases: 1) When $p < 0.5$ and $n > 1$, return $best$. 2) When $p < 0.5$ and $n = 1$, then with probability $2p$ return $secondbest$, otherwise return $best$. 3) When

$p \geq 0.5$ and $n = 1$, return $secondbest$. 4) When $p \geq 0.5$ and $n > 1$, then with probability $2(p - 0.5)$ return $secondbest$, otherwise return $best$. Every 100 flips, a random variable is selected from BC (this is intended to help escape loops)

2.1 Building Blocks for Variable Selection Heuristics

Based on previous descriptions, it is clear that these heuristics share some significant features. In fact, we can view all of the above heuristics as *composite heuristics* which are built from a set of *building blocks* for variable selection heuristics. We identify the following building blocks, from which all of the above heuristics can be built.

- *Scoring of variables with respect to a gain metric:* Variables are scored with respect to net gain or negative gain. Walksat uses negative gain, while GSAT and the Novelty variants use net gain.
- *Selecting a variable from some subset of variables:* GSAT allows the selection of any variable in the formula. Walksat and Novelty variants restrict the variable selection to a single, randomly selected broken clause. Note that flipping any variable in a broken clause will result in that clause being satisfied, although other clauses may be broken as a side effect.
- *Ranking of variables and greediness:* The variables in the subset are ranked with respect to the scoring metric. Of particular significance is the best (greedy) variable. Novelty also considers the second best variable.
- *Variable age:* The number of flips since a variable was last flipped provides a memory which is useful for avoiding cycles and forcing exploration of the search space. Age is used by the Novelty variants, as well as HSAT. A variant of Walksat with a tabu list was evaluated in (McAllester et al., 1997).
- *Branching:* In most of the heuristics, some simple Boolean condition (either a random coin toss or a function of one or more of the primitives listed above) is evaluated as the basis for a branch in the decision process.

2.2 A historical analysis of the process of SAT local search improvements

Having identified the building blocks of some historically significant SAT local search heuristics (specifically focusing on the line of local search algorithms exemplified by GSAT and Walksat), we now analyze the developments of these heuristics in a historical context, and identify the contribution of each heuristic, so that we have a *model for the process by which some key SAT heuristics were discovered*.

GSAT (Selman et al., 1992) was the first widely known SAT local search heuristic that was shown to be successful on many instances that were believed to be very difficult at the time. GSAT introduced the notion of scoring variables based on a net gain metric. This generated much excitement in the AI community, resulting in the appearance of numerous papers on SAT local search that tried to improve upon GSAT. An obvious weakness of GSAT is that because it is a greedy algorithm, GSAT tends to get stuck in local optima, so the original GSAT algorithm needed to be frequently restarted from scratch with a random assignment in order to get out of local optima. GWSAT (Selman and Kautz, 1993) addressed this problem and introduced several ideas: (1) a conditional branch based on a random variable (the walk probability *p*), and (2) selecting a variable from a randomly selected broken clause (on the walk moves). At around the same time, the HSAT heuristic (Gent and Walsh, 1993) introduced the usage of variable age. Gent and Walsh also introduced the notion of picking a variable other than the best variable according to the scoring metric (Gent and Walsh, 1993).

A major improvement in performance resulted from the introduction of Walksat (Selman et al., 1994). One significant difference from previous GSAT variants was the use of negative gain instead of net gain. Walksat also introduced the notion of a forced, conditional flip as a building block – if there is any variable in the selected clause with a 0 negative gain, then variable is flipped. The focus on a single randomly selected broken clause was new, but as noted above, the use of a broken clause as a building block was already introduced by GWSAT. The next significant advance in the performance of SAT local search algorithms came with the introduction of Novelty and R-Novelty (McAllester et al., 1997), which combined previously existing building blocks (net gain and variable age) into new combinations which were shown to significantly outperform Walksat. Finally, Novelty+ (Hoos, 1999) was developed based on a theoretical analysis which showed that Novelty and R-Novelty were essentially incomplete - these algorithms could get stuck in loops and never solve a given problem instance; Hoos observed that adding random walk eliminated this problem, and showed empirically that significant performance improvements could result.

While the development of Novelty/R-Novelty, as well as Novelty+ were motivated by careful analysis of previous algorithms and resulted in significant improvements in the state of the art performance, the actual components of these algorithms were all introduced by the time Walksat was introduced to the community in 1994. Thus, the main contributions of the Novelty variants were new, *combinations* of previously known primitives into new, effective heuristics that advanced the state of the art. It can even be argued that Walksat did not introduce any fundamentally *new* building blocks – the idea of using alternative scoring metrics such as negative and positive gain had been proposed shortly after GSAT was introduced, and the idea of a forced move is an instance of greediness (a fundamental concept in algorithm design), and is arguably a natural concept. Thus, based on our analysis of the major SAT local search algorithms developed between 1994-1999, we conclude the following:

Observation 1 *The history of SAT local search algorithms shows that significant advances do not require the invention of entirely new "ideas" – discovering a new combination of existing building blocks has resulted in some of the best known SAT local search algorithms.*

3 CLASS: A Genetic Programming System for Discovering Composite Variable Selection Heuristics

In the previous section, we showed that combining a set of existing building blocks has resulted in significant advances in the state of the art of SAT local search. We now consider how such effective combinations can be derived. Some existing SAT heuristics were designed as a result of a focused design process, which specifically addressed a weakness in an existing heuristic. GWSAT and Novelty+ added random walk to GSAT and Novelty after observing behavioral deficiencies of the predecessors (Selman and Kautz, 1993; Hoos, 1999). However, some major structural innovations involve considerable exploratory empirical effort. For example, McAllester et al (1997) note that over 50 variants of Walksat were evaluated in their study (which introduced Novelty and R-Novelty). Unfortunately, it is extremely difficult to determine *a priori* how effective any given heuristic will be. Empirical evaluation is necessary to evaluate the performance of heuristics. Seemingly subtle differences can result in very significant performance differences. For example, although there are many possible heuristics that combine some subset of the essential building blocks of Walksat (random walk, some greediness, localization of the variable domain to a single randomly selected broken clause), the performance of Walksat variants varies significantly depending on the particular choice and structural organization of these "Walksat elements". Furthermore, significant performance differences between superficially similar local search heuristics cannot be eliminated by merely tuning control parameters. See, for example, the comparison of Walksat/G, Walksat/B, and Walksat/SKC in (McAllester et al., 1997).

Based on these observations, we formulated the main hypothesis of this paper:

Hypothesis 1 *The task of combining building blocks into successful variable selection heuristics is difficult to perform manually, even for expert researchers, and is well-suited for an automated system.*

To test this hypothesis, we developed a genetic programming system for automatically discovering new SAT heuristics, **CLASS** (**C**omposite heuristic **L**earning **A**lgorithm for **S**AT **S**earch). The main components of CLASS are:

- A small language for expressing variable selection heuristics as s-expressions, and
- A meta-level, genetic programming algorithm that searches the space of possible selection heuristics by repeated application of a composition operator.

CLASS represents variable selection heuristics in a Lisp-like s-expression language. In each iteration of the local search, this s-expression is evaluated in place of a hand-coded variable selection heuristic (Figure 1, Line 4).

Tables 1 and 2 show the terminals and functions in the CLASS language. As an illustration, Figure 3 shows some standard heuristics represented as CLASS s-expressions. The language is expressive enough to implement all of the standard heuristics in Section 2 except for R-Novelty.[1]

CLASS uses a variant of strongly typed genetic programming (Koza, 1992; Montana, 1993), shown in Figure 2. 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, using Whitley's linear rank-based selection function (Whitley, 1989). The `Compose` operator (detailed below) is applied to the parents to generate a set of children, which are then inserted into the population, replacing the lowest-ranked members of the population. The best heuristic found during the course of the search is returned.

[1]The currently implementation can not express a random variables whose value is referenced more than once in an s-expression e.g., the "p" and "$2p$" in R-Novelty.

name	type	description
`+NEG-GAIN+`, `+POS-GAIN+`, `+NET-GAIN+`	gaintype	Enumerated constant which represents negative gain, positive gain, and net gain, respectively
`<, <= ,=`	comparator	less than , less than or equal, equal comparators
`+BC0+`	varset	randomly selected broken clause (`BC0` refers to the same, broken clause throughout the an expression.)
`+BC1+`	varset	another randomly selected broken clause (it is possible that `BC0` and `BC1` are assigned the same clause).
`+WFF+`	varset	the entire formula.

Table 1: CLASS Language Terminals

name	type	arguments	description
`IF-RAND-LT`	var	p float v_1, v_2 variable	If a randomly generated floating point number is less than p, then return $v1$, else return $v2$
`VAR-RANDOM`	var	vs varset	A randomly selected variable from the varset vs
`GET-VAR`	var	vs varset g gaintype	Returns variable from varset vs which has the best value of specified gaintype g. E.g.,`(GET-VAR +BC0+ +NEG-GAIN+)` returns the variable with lowest negative gain from broken clause `+BC0+`.
`GET-VAR2`	var	vs varset g gaintype	Variable from the varset vs which has the *second* best variable according to the gaintype g. E.g., `(GET-VAR2 +WFF+ +NEG-GAIN+)` returns the variable with second lowest negative gain from the set of all variables in the formula.
`OLDER-VAR`	var	v_1, v_2 variable	Least recently flipped of the variables v_1 and v_2.
`IF-TABU`	var	age integer v_1, v_2 variable	If the age of variable v_1 is less than age, then return v_2; otherwise, return v_1.
`IF-VAR-COMPARE`	var	c comparator g gaintype v_1, v_2 variable	For v_1 and v_2, compute the gain value for gaintype g. Return v_1 or v_2 depending on whether applying the comparison c returns true or false. E.g., `(IF-VAR-COMPARE < +NEG-GAIN+` v_1 v_2`)` will return v_1 if the negative gain of v_1 is less than the negative gain of v_2; otherwise, it returns v_2.
`IF-VAR-COND`	var	c comparator g gaintype n integer v_1, v_2 variable	Compute gain(v_1,g), the gain of v_1 according to gaintype g. Return v_1 if the expression [gain(v_1,g) c n] returns true; otherwise return v_2. E.g., `(IF-VAR-COND = 0 +NET-GAIN+` v_1 v_2`)` returns v_1 if the net gain of v_1 is equal to 0, and otherwise returns v_2.
`IF-NOT-MIN-AGE`	var	vs varset v_1, v_2 variable	If v_1 does *not* have minimal age among variables in a varset vs, then return v_1, else v_2.

Table 2: CLASS Language Functions

```
1.  Initialize(population,population_size)
2.  for i = 1 to MaxIterations
3.    Select parent1 and parent2 from population
4.    children = Compose(parent1,parent2)
5.    Evaluate(Children)
6.    population = Insert_and_replace(children,population)
```

Figure 2: CLASS Genetic Programming Algorithm.

```
;;;GWSAT(0.5) (GWSAT with random walk probability 0.5)
(IF-LTE 0.5
       (VAR-RANDOM +BC0+)
       (GET-VAR +WFF+ +NET-GAIN+)

;;;Walksat(0.5) (random walk probability 0.5)
(IF-VAR-COND = +NEG-GAIN+ 0
             (GET-VAR +BC0+ +NEG-GAIN+)
             (IF-RAND-LTE 0.5
                          (GET-VAR +BC0+ +NEG-GAIN+)
                          (VAR-RANDOM +BC0+)))

;;;Novelty+(0.5,0.01) (noise 0.5 and walk probability =0.01)
(IF-RAND-LT 0.01
   (VAR-RANDOM +BC0+)
   (IF-NOT-MIN-AGE +BC0+
                  (GET-VAR +BC0+ +NET-GAIN+)
                  (IF-RAND-LT 0.5
                              (GET-VAR2 +BC0+ +NET-GAIN+)
                              (GET-VAR +BC0+ +NET-GAIN+))))
```

Figure 3: GSAT with Random Walk (GWSAT), Walksat, and Novelty+ represented in the CLASS language.

3.1 Composition - A special purpose GP operator based on the historical development of SAT heuristics

The most significant difference between the CLASS learning algorithm and standard strongly typed genetic programming is how new candidate solutions (children) are created. Instead of applying the standard crossover and mutation operators, CLASS uses a method that is inspired by the way in which historical, SAT local search algorithms have been derived. Recall that GWSAT and Novelty+ were derived by adding random walk to GSAT and Novelty. This can be generalized into a general meta-heuristic for creating new variable selection strategies: Given two heuristics H_1 and H_2, combine the two into a new heuristic that chooses between H_1 and H_2 using the schema:

`If Condition` H_1 `else` H_2

where `Condition` is a Boolean expression.

We call this the *composition* operator. Intuitively, this is a reasonable meta-heuristic because it "blends" (switches between) the behavior of H_1 and H_2 according to some Boolean condition. A special case where `Condition` is a randomization function (i.e., `If (rand<`p`) then` ...) is a *probabilistic composition*, which has an interesting property. Hoos (1998) defines a SAT local search procedure to be PAC (probably approximately correct) if with increasing run-time the probability of finding a solution for a satisfiable instance approaches 1.[2] An algorithm that is not PAC is *essentially incomplete*. GSAT, Novelty, and R-Novelty are essentially incomplete; however, their performance is significantly improved by adding random walk, which was proven to make these algorithms PAC (Hoos, 1998). In other words, the historical process by which GWSAT and Novelty+ were generated can be modeled as applications of probabilistic composition. Probabilistic composition has the following useful property:

[2]Although Hoos' notion of PAC local search is inspired by the well-known idea of "Probably Approximately Correct" in machine learning, these are different definitions and should not be confused.

Property 1 *Let H_1 and H_2 be two variable selection heuristics. If either H_1 or H_2 is PAC, then the composite heuristic* `(If (rand` p`) then` H_1 `else` H_2`)`, *the result of applying the probabilistic composition operator is also PAC for all $p, 0 < p < 1$. [Proof: follows from the fact that as long as $p > 0$, there is a sequence of random choices which continues to choose H_1]*

Thus, during the process of searching for heuristics, if we have a candidate heuristic whose major deficiency is essential incompleteness, then probabilistic composition with any PAC heuristic in the population theoretically removes that deficiency.

The full composition operator used by CLASS takes two heuristics s-expressions H_1 and H_2 as input and outputs 10 new heuristics to be inserted into the population:

- Five probabilistic compositions of the form `(If (rand p)` H_1 H_2`)`, for p=0.1, p=0.25, p=0.5, p=0.75, and p=0.9
- `(OLDER-VAR` H_1 H_2`)` - evaluates H_1 and H_2, and returns the variable with maximal age.
- `(IF-TABU 5` H_1 H_2`)` - Let variable v be the result of H_1. If $age(v)$ is tabu (i.e., less than 5), then evaluate H_2.
- `(IF-VAR-COND = +NEG-GAIN+ 0` H_1 H_2`)` - Let v_1 be the result of H_1. if $NegativeGain(v_1) = 0$ return v_1, else return v_2, the result of H_2.
- `(IF-VAR-COMPARE <= +NEG-GAIN+` H_1 H_2`)` - Let v_1 be the results of H_1, v_2 the result of H_2. If $NegativeGain(v_1)$ is less than or equal to $NegativeGain(v_2)$, then return v_1, else v_2.
- `(IF-VAR-COMPARE <= +NET-GAIN+` H_1 H_2`)` - similar to (9), but uses net gain as the comparator.

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. As a simple workaround, we implemented a depth bound which works as follows.

If there is any subtree s of S rooted at depth $bound$ such that the depth of the subtree s is greater than 2 (i.e., s causes S to exceed its depth bound by more than 1), then s is replaced by a randomly generated subtree s', which has the same type as s, but is of *minimal depth*, which we define recursively to mean that if there exists a terminal of the same type as s, then such a terminal is randomly selected; otherwise, the a random function of the same type as s is generated, where the parameters are generated to be of minimal depth. For example, if a candidate heuristic `(OLDER-VAR (OLDER-VAR (GET-VAR +BC0+ +NET-GAIN+)(GET-VAR +BC0+ +NEG-GAIN+)) (GET-VAR +BC1+ +POS-GAIN+))` violates the depth bound condition by 1, then one of its subtrees could be replaced with a shallower subtree (maintaining the type constraints), resulting in `(OLDER-VAR (GET-VAR +BC1+ +NET-GAIN+) (GET-VAR +BC1+ +POS-GAIN+))`, The depth bound is "soft" because minimal depth replacement can generate a replacement subtree s' with depth greater than 1 (and thus, individuals generated in a run with depth bound d will usually have depth of $d + 1$. This depth bound also has the additional, significant effect of introducing a type of "mutation" at these new leaf nodes. Note that when this depth-bounded mutation occurs, then the PAC-preservation property of composition may not hold. A recent comparison of various other bloat control methods can be found in (Luke and Panait, 2006).

3.2 Evaluating the utility of a candidate heuristic

A candidate heuristic 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 (Mitchell, Selman, and Levesque, 1992).[3] 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 were successful, then the heuristic is run on 400, 100-variable, 430-clause random 3-SAT instances with a 20000 flip cutoff. The 50-variable instances serve to quickly filter out very poor candidates 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 candidate heuristics.

The score of an individual is: (# of 50-var successes) + 5 $\times$ (# of 100-var successes) + `1/MeanFlipsInSuccessfulRuns`

The second term for the 100-variable instances is weighted heavily because performance on large problems is much more important than performance on easier (smaller) problems. The last term serves as a tie-breaker in case all of the 50 and 100 variable instances are solved. The specific parameters (e.g., the cutoff of 130 successful descents for the 50-variable instances) were determined by a series of small-scale preliminary experiments that we performed in order to minimize the number of large instances in the set (to minimize the time required to execute the evaluation function) while still having a sufficient number of hard (large) instances to distinguish the performances of the better candidate heuristics.

3.3 Implementation Details

So far, we have focused on the mechanisms required to represent and generate SAT local search heuristics that searched for solutions with the fewest number of variable flips possible. However, our goal is to generate SAT heuristics that perform well with respect to both search efficiency (# of flips required to solve instances) and runtime. Recall that CLASS evolves the variable selection heuristic for SAT local search, and the variable selection heuristic is executed at every single variable flip 1. Thus, when we say that an evolved heuristic h is applied to a SAT instance I, it means that if a local search algorithm which uses h for variable selection requires 100,000 variable flips to solve I, then h is executed 100,000 times. In a SAT local search algorithm, the time spent executing the variable selection heuristic (as well as any related incremental data structure updates) is the computational bottleneck. An algorithm which requires too much computation per variable flip may end up running slower than a simpler heuristic which execute more flips rapidly. Thus, we carefully implemented CLASS so that it was possible to obtain fast runtimes.

The local search engine in CLASS implements the most efficient local search engine found so far in the literature, with some enhancements that yielded significant

[3]`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 the instance to be satisfiable.

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 (Fukunaga, 2004a). The standard implementation of SAT local search is the C implementation by Henry Kautz and Bart Selman (Walksat version 43).[4] We have shown that on the same machine, the CLASS local search engine, using a hand-coded s-expression for the Walksat-SKC variable selection heuristic, 90% as many variable flips per second as the Walksat-v43 implementation, compiled using `gcc` with `-O3` optimization settings, despite being implemented in Common Lisp. See (Fukunaga, 2004a) for details.

The GP component of CLASS generates an s-expression in the language specified in Section 3. The functions and terminals in the CLASS language are actually implemented as a set of macros and functions in Common Lisp. The Common Lisp compiler is used to compile the heuristic s-expression into efficient native machine code. In other words, CLASS was implemented as a *domain-specific language* that was able to fully leverage the underlying Lisp compiler and runtime environment. S-expression simplification techniques and techniques for caching subexpression values were implemented in order to minimize the amount of computation that needed to be done in order to evaluate the heuristic. Further details are in (Fukunaga, 2004b).

4 Evaluation of the Performance of Automatically Generated Heuristics

In this section, we present several experiments to evaluate the performance of the SAT local search heuristics that are generated by CLASS. We compare the heuristics evolved by CLASS to both standard local search heuristics (Section 4.6-4.7) as well as previous, direct, evolutionary approaches for solving SAT (Section 4.8).

4.1 Some Heuristics Generated by CLASS

This section describes the heuristics discovered by CLASS which are used in the performance evaluations.

We performed 10 runs of the CLASS GP, where each run consisted of 5500 individual evaluations (including the initial population), with a population of 1000. These 10 runs consisted of 2 runs each with depth bound of 2,3,4,5, and 6. The best individuals found in each of these 10 runs are used in our evaluations. They are named "depth$D$-X", where D is the depth bound (2-6), and X is 1 or 2 (run #1 or #2).

In Figure 4, we show several of these evolved heuristics. With some effort, it is possible to understand what these heuristics do. For example, depth2-2 considers two broken clauses, `BC0` and `BC1`. Then, (a) the variable in `BC1` with highest positive gain, (b) the variable in `BC0` with highest net gain, (c) the variable in `BC0` with lowest negative gain, and (d) the variable in `BC1` with lowest negative gain are computed. Out of (a)-(d), the variable with maximum age is returned. Thus, depth2-2 first selects four variables that are likely to be "good" for some reason (best positive, net, or negative gain in a broken clause), and then chooses the least recently flipped variable of the four.

[4]available at http://www.cs.washington.edu/homes/kautz/walksat.

```
;;; Heuristic: depth2-1
(IF-TABU 20
         (OLDER-VAR (GET-VAR +BC0+ +NET-GAIN+)
                    (IF-VAR-COMPARE < +POS-GAIN+
                     (GET-VAR +BC0+ +POS-GAIN+)
                     (IF-VAR-COND <= +NEG-GAIN+ 2
                                  (GET-VAR +BC0+ +NEG-GAIN+)
                                  (GET-VAR +BC0+ +NET-GAIN+))))
         (GET-VAR +BC0+ +POS-GAIN+))


;;; Heuristic: depth2-2
(OLDER-VAR (OLDER-VAR (GET-VAR +BC1+ +POS-GAIN+)
                      (GET-VAR +BC0+ +NET-GAIN+))
           (OLDER-VAR (GET-VAR +BC0+ +NEG-GAIN+)
                      (GET-VAR +BC1+ +NEG-GAIN+)))


;;; Heuristic: depth3-2
(IF-TABU 5
         (OLDER-VAR
          (IF-TABU 20 (GET-VAR +BC1+ +NEG-GAIN+)
                      (GET-VAR +BC1+ +NET-GAIN+))
          (IF-TABU 40 (GET-VAR +BC1+ +POS-GAIN+)
                      (GET-VAR +BC0+ +NEG-GAIN+)))
         (IF-RAND-LT 0.75
             (IF-TABU 5 (GET-VAR +BC1+ +NEG-GAIN+)
                        (GET-VAR +BC0+ +NET-GAIN+))
             (GET-VAR +BC0+ +NEG-GAIN+)))



;;; Heuristic: depth4-2
(IF-VAR-COMPARE <= +NEG-GAIN+
 (OLDER-VAR
  (IF-RAND-LT 0.75
      (GET-VAR +BC1+ +NET-GAIN+)
      (IF-VAR-COND = +POS-GAIN+ 0
                   (GET-VAR +BC0+ +NET-GAIN+)
                   (GET-VAR +BC0+ +NEG-GAIN+)))
  (IF-VAR-COND <= +NEG-GAIN+ 0
               (IF-VAR-COND <= +NEG-GAIN+ 0
                            (GET-VAR +BC0+ +NET-GAIN+)
                            (GET-VAR +BC1+ +NEG-GAIN+))
               (IF-TABU 40 (RANDOM-VAR +BC1+)
                            (GET-VAR +BC1+ +NEG-GAIN+))))
 (OLDER-VAR
  (OLDER-VAR
   (IF-TABU 5 (GET-VAR +BC0+ +NET-GAIN+)
              (GET-VAR +BC1+ +RANDOM+))
   (IF-VAR-COMPARE <= +NEG-GAIN+
                   (RANDOM-VAR +BC0+)
                   (RANDOM-VAR +BC1+)))
  (IF-VAR-COMPARE <= +NET-GAIN+
                  (IF-TABU 5 (GET-VAR +BC1+ +POS-GAIN+)
                             (GET-VAR +BC0+ +NEG-GAIN+))
                  (IF-VAR-COMPARE <= +NEG-GAIN+
                                  (GET-VAR +BC1+ +NEG-GAIN+)
                                  (GET-VAR +BC0+ +NET-GAIN+)))))
```

Figure 4: Some heuristics discovered by CLASS (depth2-1, depth2-2, depth, depth3-2, depth4-2)

4.2 Local Search Algorithms for Comparison

We compare the evolved heuristics with hand-coded, optimized implementations of Walksat (Selman et al., 1994) and Novelty+ (Hoos and Stutzle, 2000). The implementation we used for both algorithms is the standard Walksat/Novelty code distributed by Henry Kautz, version 43. This is a highly optimized C implementation, compiled with `gcc` using `-O3` optimization. This package implements not only Walksat, but also all of the Novelty variants. We also compare our evolved heuristics with our implementation of GWSAT, which is GSAT with Random Walk using a 0.5 walk probability (Selman and Kautz, 1993). It is well-known that the performance of SAT local search is sensitive to control parameters (Hoos and Stutzle, 2000). For Walksat, we use a walk probability of 0.5, which is the standard value in the literature. Likewise, for Novelty+, we use a noise value (p in the Novelty+ description in Section 2) of 0.7 and a walk probability of 0.01 (p_w), and a walk probability of 0.5 for GWSAT; these are all standard values.

4.3 Selection of Benchmark Instances

Two sets of benchmark instances were used. The first set consists of instances from the SATLIB benchmark suite (www.satlib.org). These instances have been frequently used as benchmarks by the SAT local search community, and include a variety of instances, including hard random satisfiable 3-SAT instances, as well as problems transformed into SAT from other problem formulations such as AI planning instances. The second set of benchmark instances is from the evolutionary computation literature. These are a set of satisfiable, random 3-SAT benchmark instances which were previously used by Gottlieb et al. in their survey on evolutionary algorithms for SAT (Gottlieb, Marchiori, and Rossi, 2002).[5]. These test suites (Suite A, Suite B, and Suite C) consist of randomly instances of various sizes (number of variables ranging from 20-100) with a clause-to-variable ratio of 4.3, which results in hard benchmark instances (Mitchell et al., 1992).

4.4 Method of Performance Measurement and Evaluation

Each algorithm was executed 100 times on each instance, with a computational limit, or *cutoff* of F flips per execution. We measured: (1) the *success rate*, which is the percentage of successful runs (where a solution is found within F variable flips). (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), and (3) the *total run time*, spent running the algorithm on all tries on all instances of the given benchmark instance(s). *Since AFS excludes the unsuccessful runs, success rate is by far the most important metric, and dominates AFS, which should be seen as a tie-breaker in case two algorithms solve all instances.* If algorithm A does not solve an instance within the time limit of F flips, then all we know is that A would take *at least* F flips, but we do not have an upper bound on the number of flips required by A.

The SR and AFS metrics are standard metrics for measuring SAT algorithm performance in both the SAT local search literature, as well as the evolutionary computing literature. The reason for this format is that requiring all solvers to solve all instances would require too much time. On the Gottlieb instances, even increasing the cutoff to 100,000,000 flips is not sufficient for all tries of all algorithms to solve all instances

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

	uf100 (1000 instances)		
Algorithm	SR	AFS	Time
GWSAT(0.5)	99.96	8636	1189.63
Walksat(0.5)	100.0	3658	399.23
Novelty+(0.7,0.01)	99.998	2317	270.32
depth2-1	99.299*+	1824	603.38
depth2-2	99.312*+	2429	781.45
depth3-1	99.413*+	1748	569.06
depth3-2	99.406*+	1800	650.06
depth4-1	100.0	1868	286.79
depth4-2	100.0	2044	405.96
depth5-1	99.456*+	1728	783.11
depth5-2	100.0	1455	332.38
depth6-1	100.0	1887	451.59
depth6-2	100.0	1753	275.87
C-Rand	99.435*+	1962	646.05

Table 3: SATLIB uf100-430 Random 3-SAT instances: Success Rate (%), Average Flips to Solution, and total Runtime (seconds) for 100 runs/instance, 500,000 flips/run.

(Fukunaga, 2004b)[6]. Furthermore, SAT is a constraint satisfaction problem, where partial solutions that violate one or more hard constraints do not necessarily have any value. For some applications of SAT, a state with 1 unsatisfied clause can be just as bad as a state with 100 unsatisfied clauses. Thus, the focus is on whether an instance is solved, and if so, how much effort it required.[7]

4.5 Statistical Tests for Significance

In Tables 3-7, we evaluate whether the success rates of the evolved algorithms are significantly different than the success rates of Walksat(0.5) and Novelty+(0.7,0.01). For each evolved heuristic, we compared the total number of successful and unsuccessful runs (the raw data from which the success rates are computed) with the number of successful and unsuccessful runs for Walksat and Novelty. These values were tested for significant differences using a standard chi-square test. In Tables 3-7, A '*' indicates a significant difference ($p < 0.01$) in the successes and failures on the benchmark set between the evolved heuristic and Walksat. Likewise, a '+' indicates a significant difference ($p < 0.01$) between the evolved heuristic and Novelty+.

4.6 Performance and Scaling on Test Instances Similar to or Smaller than the Training Instances

Recall that CLASS uses a training set of 50 and 100 variable random 3-SAT instances as the training instances during the learning process. We first evaluate the performance of the learned heuristics on test instances similar to the training instances, which are hard random 3-SAT instances generated with a clause to variable ratio of 4.3.

The results of running the standard Walksat, Novelty+, and GWSAT heuristics as well as the evolved heuristics on the on the 1000 random 3-SAT instances in the SATLIB uf100-430 benchmarks suite (100 variables, 430 clauses) are shown in Table 3. On these instances, Walksat achieved a 100% success rate, whereas Novelty+ failed on

[6]Randomized restarts would, in theory, allow the instances to be solved eventually

[7]In contrast, there are variants of SAT such as MAX-SAT where partially satisfied formula are valuable so the (weighted) number of broken clauses is used as a performance metric; however, the best MAX-SAT solvers are not necessarily the best SAT solvers, and vice versa .

Gottlieb Test Set Suite-A									
	40 vars, 3 instances			50 vars, 3 instances			100 vars, 3 instances		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
SAWEA	93	53289	-	85	60743	-	72	86631	-
RFEA2+	100	3081	-	100	7822	-	97	34780	-
FlipGA	100	17693	-	100	127900	-	87	116653	-
ASAP	100	8760	-	100	68483	-	100	52276	-
GASAT	100	1135	-	91	1850	-	95	7550	-
GWSAT(0.5)	100.0	6062	2.17	90.33	31295	19.86	71.0	10104	37.74
Walksat(0.5)	100.0	2604	0.85	99.67	22219	7.58	81.0	24855	25.75
Novelty+(0.7,0.01)	100.0	1767	0.6	100.0	3061	1.12	90.0	33148	21.12
depth2-1	100.0	1023	0.41	100.0	10307	3.79	97.67*+	34858	15.69
depth2-2	100.0	538	0.26	100.0	3634	1.6	99.67*+	18549	8.8
depth3-1	100.0	981	0.4	100.0	8219	3.24	94.0*	28916	18.42
depth3-2	100.0	746	0.32	100.0	3453	1.44	99.67*+	17771	7.84
depth4-1	100.0	722	0.34	100.0	5638	2.48	96.67*+	30864	17.7
depth4-2	100.0	825	0.48	100.0	3416	1.94	100.0*+	20754	12.1
depth5-1	100.0	817	0.47	100.0	5645	3.07	99.0*+	24787	15.29
depth5-2	100.0	766	0.53	100.0	4506	2.99	99.33*+	23871	17.05
depth6-1	100.0	1070	0.76	100.0	5225	3.67	100.0*+	22291	15.91
depth6-2	100.0	692	0.35	100.0	5552	2.56	98.67*+	28262	14.73
C-Rand	100.0	1223	0.52	100.0	6160	2.57	99.33*+	25349	12.01
Gottlieb Test Set Suite-B									
	50 vars, 50instances			75 vars, 50 instances			100 vars, 50 instances		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
RFEA2+	100	11350	-	96	39396	-	81	80282	-
FlipGA	100	103800	-	82	29818	-	57	20675	-
ASAP	100	61186	-	87	39659	-	59	43601	-
GASAT	96	2732	-	83	6703	-	69	28433	-
GWSAT(0.5)	89.56	20841	285.3	71.74	28584	641.7	56.6	17704	934.47
Walksat(0.5)	94.4	18609	185.88	82.48	32939	439.38	59.16	21824	756.55
Novelty+(0.7,0.01)	96.64	13783	133.32	89.3	25883	327.05	65.8	24655	713.48
depth2-1	93.9*+	6859	143.3	93.02*+	27295	286.67	67.52*+	29628	734.7
depth2-2	95.46*+	3341	108.48	97.38*+	20116	197.89	76.44*+	40399	755.6
depth3-1	94.0*+	7545	153.18	91.54*+	27294	330.27	67.82*+	28913	776.58
depth3-2	95.78*+	4426	115.63	96.92*+	19803	197.64	78.06*+	38894	674.05
depth4-1	98.74*+	8227	87.1	95.78*+	26176	278.95	72.64*+	35905	805.94
depth4-2	99.38*+	6590	79.49	98.48*+	19791	232.66	78.8*+	39909	926.71
depth5-1	95.0*+	5880	177.4	95.18*+	23976	338.43	73.24*+	34490	974.17
depth5-2	98.14*+	7992	148.5	96.84*+	22845	350.14	76.02*+	36465	1108.43
depth6-1	98.9*+	7866	129.17	97.26*+	23091	362.95	76.06*+	38444	1208.03
depth6-2	98.7*+	7968	91.05	96.42*+	24565	267.01	73.0*+	36429	830.2
C-Rand	95.48*+	6101	128.08	95.88*+	23263	246.75	75.42*+	36124	735.85
Gottlieb Test Set Suite-C									
	60 vars, 100 instances			80 vars, 100 instances			100 vars, 100 instances		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
SAWEA	73	47131	-	52	62859	-	51	69657	-
LSAWEA	80	37439	-	58	46337	-	57	46497	-
RFEA2+	99	19957	-	95	49312	-	79	74459	-
FlipGA	100	127520	-	73	29957	-	62	20319	-
ASAP	100	184419	-	72	45942	-	61	34548	-
GASAT	97	9597	-	66	7153	-	74	1533	-
GWSAT(0.5)	89.07	24353	654.89	65.4	28845	1562.26	58.21	19057	1812.27
Walksat(0.5)	94.72	21375	392.87	73.46	33265	1155.9	63.18	23684	1402.45
Novelty+(0.7,0.01)	98.27	15703	241.62	81.66	30991	947.0	70.04	25869	1298.68
depth2-1	97.59*+	12111	226.71	80.25*	27555	983.56	71.9*	28145	1309.69
depth2-2	98.83*+	6796	141.05	89.94*+	27638	770.54	79.86*+	36685	1330.48
depth3-1	97.38*+	13168	263.73	80.12*	27330	1045.88	71.6*	27758	1406.67
depth3-2	98.75*+	6973	146.07	89.37*+	25421	748.32	80.88*+	35079	1194.42
depth4-1	99.62*+	10410	169.33	89.1*+	32035	904.0	74.95*+	32125	1474.93
depth4-2	99.98*+	6692	129.52	94.88*+	29144	830.81	81.33*+	35966	1659.35
depth5-1	98.42*	9599	251.58	85.6*+	27778	1196.53	76.4*+	32523	1756.68
depth5-2	99.73*+	8979	215.77	90.51*+	30350	1241.74	78.95*+	32552	1978.56
depth6-1	99.86*+	8827	216.66	91.96*+	30579	1233.36	78.58*+	34478	2178.82
depth6-2	99.84*+	9375	152.21	89.45*+	30021	901.51	75.96*+	33271	1501.05
C-Rand	98.61*+	8833	177.55	87.34*+	27905	865.8	78.71*+	34054	1315.8

Table 4: Gottlieb Suites A-C: Success Rate (%), Average Flips to Solution, and total Runtime (seconds) for 100 runs/instance, 300,000 flips/run. Results for SAWEA, LSAWEA, RFEA2+, FlipGA and ASAP are from Gottlieb et al. (2002). Results for GASAT are from Lardeux et al. (2006).

some runs. In comparison, five of the evolved heuristics achieved a 100% success rate. Notably, the depth4-1 and depth6-2 heuristics performed particularly well, achieving 100% success with average runtimes of 286.79 seconds and 275.87 seconds, respectively, compared to 399.23 secondes for Walksat.

The results on the Gottlieb benchmark suite instances are shown in Table 4. On the Gottlieb instances, almost all of the evolved heuristics significantly outperformed Walksat, Novelty+, and GWSAT on the more difficult instances (the Suite-B and Suite-C instances, as well as the largest Suite-A instances).

These results on the uf100-430 instances and the Gottlieb benchmarks show that the performance of evolved heuristics is competitive with that of the standard Walksat and Novelty variants when executed on instances that are similar to (or in the case of some of the Gottlieb instances, smaller than) the instances in the training set for which they were evolved.

4.7 Generalization and Scaling of the Evolved Heuristics

Next, we evaluated the heuristics on benchmarks from problem classes that differed significantly from the training instances, in order to see how well the heuristics generalized and scaled beyond the test distribution for which they were specifically trained. Again, for each instance, we executed each algorithm 100 times with a cutoff of 500,000 variable flips. It is important to keep in mind that none of the heuristics (hand-coded or evolved) were tuned for the larger 3-SAT instances or the structured (planning, graph coloring, all-interval-series) instances; we present this data in order to show how well the heuristics discovered by CLASS generalize relative to the generalization of the hand-coded heuristics.

First, we tested the local search heuristics on larger, hard 3-SAT from SATLIB (`uf150, uf200`, and `uf250`, which are 150, 200, and 250 variable problems, respectively where the clause-to-variable ratio is 4.3). As shown in Table 5, the evolved heuristics are comparable to Walksat and Novelty+ on the uf150 instances. For the uf200 and uf250, most of the evolved heuristics perform slightly worse than the standard heuristics with respect to search efficiency and runtimes.

We also tested the evolved and standard local search heuristics on SATLIB instances from problem classes that were very different from the hard 3-SAT instances on which the evolved instances were trained. These included five AI planning instances (`medium, huge, logistics.c, bw_large.a, bw_large.b`), three all-interval-series instances (`ais6, ais8, ais10`), and 100 graph coloring instances (`flat125`).

Tables 6-7 shows that the results are quite mixed. For the `huge` planning benchmark, four of the evolved heuristics performed significantly worse than Walksat and Novelty+. For `logistics.c` and `bw_large.a`, the evolved heuristics did significantly worse than the standard heuristics. On `bw_large.b`, five of the evolved heuristics significantly outperformed both Walksat and Novelty+. The all-interval-series `ais` benchmarks are interesting because it is an unusual set of instances where Novelty+ performs significantly worse than Walksat. On these instances, the evolved heuristics significantly outperformed Novelty+, although they did not do as well as Walksat. Finally, on the 100 graph coloring instances (`flat125`), half of the evolved heuristics performed extremely poorly, and only one evolved heuristic (depth2-1) was comparable to the standard heuristics.

	uf150 (100 instances)			uf200 (100 instances)			uf250 (100 instances)		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
GWSAT(0.5)	97.9	25227	531.31	93.64	57727	1446.88	92.25	70015	1891.58
Walksat(0.5)	99.75	13395	162.85	97.22	28044	472.62	98.49	33308	473.45
Novelty+(0.7,0.01)	99.94	8282	102.42	97.58	18558	370.62	98.15	21689	382.23
depth2-1	99.29*	7480	132.8	95.57*+	15362	429.59	97.08*+	20206	411.77
depth2-2	99.24*+	15330	280.44	90.59*+	60101	1514.4	80.88*+	90641	2557.89
depth3-1	99.23*	6851	136.85	96.09*+	15354	431.35	97.68*+	19261	395.51
depth3-2	99.49*+	10979	188.06	93.4*+	45307	1049.45	86.61*+	73389	1813.98
depth4-1	99.99*	8232	125.96	97.97*	22565	484.97	98.15	30920	595.56
depth4-2	99.96*	12497	250.17	94.51*+	53219	1525.74	88.26*+	78426	2509.01
depth5-1	99.42*+	7578	191.3	96.16*+	22335	740.57	96.01*+	33553	969.2
depth5-2	100.0*	5774	129.88	98.24*+	19298	617.79	98.57+	27148	753.74
depth6-1	99.99*	8957	215.42	97.24	35952	1164.53	94.22*+	51184	1843.29
depth6-2	99.99*	8259	128.39	97.86*	25651	548.98	97.69+	33667	676.74
C-Rand	99.57+	9538	169.58	95.12*+	34997	842.06	92.13*+	55535	1356.14

Table 5: SATLIB Random 3-SAT instances: Success Rate (%), Average Flips to Solution, and total Runtime (seconds) for 100 runs/instance, 500,000 flips/run.

In general, the evolved heuristics significantly outperform GWSAT on all of the SATLIB instances except for the graph coloring instances, where half of the evolved heuristics perform very poorly. This shows that in all cases (except for graph coloring), they are all at least performing competitively with a classical heuristic which was the dominant method prior to Walksat.

Overall, these results show that the evolved heuristics scale and generalize fairly well, better than what we initially expected, given that the evolved heuristics were only trained on 50 and 100 variable random 3-SAT instances.

4.8 Comparison of the Automatically Generated Heuristics with Direct, Evolutionary Approaches

Although we have focused on the local search so far, evolutionary computation has also been applied to solve SAT *directly*, i.e., an evolutionary algorithm is applied to a population of candidate SAT solutions in order to search for a satisfying truth assignment. In contrast, CLASS is a meta-level application of a genetic programming algorithm and is an *indirect approach*, where an evolutionary algorithm is applied to a population of local search heuristics in order to generate a local search algorithm. To date, the most successful evolutionary approaches have been hybrid genetic-local search algorithms.

We briefly describe the algorithms included in the empirical comparison. A detailed comparative survey is by Gottlieb et al. (2002).All of the most successful, direct evolutionary computation approaches for SAT are based on a bit-string representation, where individuals are bit strings, where each bit represent a variable, and the value of the bit represents its current truth value. This is the same as variable assignment array that is used by SAT local search algorithms.

- SAWEA is an evolutionary algorithm where clauses have weights (Eiben and van der Hauw, 1997). The fitness function is the sum of the weights of the unsatisfied clauses. Periodically, the weights on the broken clauses are increased. LSAWEA is an extension of SAWEA which periodically applies a macro-mutation which forces some broken clauses to become satisfied (this may cause other satisfied clauses to become broken) (de Jong and Kosters, 1998).

- RFEA2+ is an evolutionary algorithms which biases a variable's value using a *refining function* (Gottlieb and Voss, 2000).
- FlipGA is a hybrid genetic local search algorithm where standard genetic operators (uniform crossover, mutation) are used to generate children which are then improved using a local search (Marchiori and Rossi, 1999). ASAP is a variant of FlipGA with a tabu mechanism (Rossi, Marchiori, and Kok, 2000).
- GASAT is a hybrid algorithm that combines tabu search and a genetic algorithm with a non-standard, problem-specific crossover operator (Lardeux et al., 2006).

Since we used the same instances and experimental parameters (cutoff) as the comparative study of Gottlieb, Marchiori, and Rossi, as well as the comparative study by Lardeux et al. (2006), we can compare our results with results previously published in the literature. Table 4 includes the results for the previous, direct, evolutionary approaches for SAT. The data for RFEA2+, FlipGA, SAWEA, LSAWEA, and ASAP are copied from Tables 3,4, and 5 in (Gottlieb et al., 2002). 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. Each run was limited to 300,000 flips. Runtimes were not reported by Gottlieb, et al. Similarly, the data for GASAT are copied from Table 8 in (Lardeux et al., 2006), and are based on 5 to 50 runs per instance (Lardeux et al. did not give more specific details regarding the number of runs for each instance), where each run was limited to 300,000 flips. Lardeux et al. did not report runtimes for GASAT on these instances. As shown in Table 4, the heuristics discovered by CLASS are quite competitive with the previous evolutionary algorithms.

4.9 Comparisons among Evolved Heuristics

Among the ten evolved heuristics (depth2-1,...,depth6-2, there is no clear, single "best" evolved heuristic which dominates others across the benchmark set. From this limited data, it is not possible to detect a clear correlation between the depth bound parameter (which controls the size limit of the evolved s-expressions) and heuristic performance. The main purpose of this present study was to show the feasibility of the CLASS approach, and Tables 3-7 indicate that CLASS is somewhat robust, in its ability to find good heuristics is not critically dependent on a specific depth bound (at least for bounds in the range 2-6). A larger-scale experiment to understand the impact of control parameters such as depth bound is a direction for future work.

5 Local Search Characteristics of Automatically Generated Heuristics

The experiments described in the previous sections have demonstrated that it is possible to discover highly effective SAT local search heuristics using CLASS. A natural question is: *Why do evolved heuristics perform well?* Despite of the good performance on the benchmarks, a possible concern is that the evolved heuristics might be "getting lucky", or exploiting some bizarre, hidden structure of the benchmarks. Are the evolved heuristics truly "good" heuristics that we should be willing to consider as a viable approach for a practical application? Given a choice between a simple, relatively well-understood heuristic such as Novelty+ and a new, relatively complex evolved heuristic such as depth4-2, a practitioner would probably prefer the simpler heuristic, if there was no justification for the evolved heuristic other than benchmark results.

	medium 1 instance			huge 1 instance			logistics.c 1 instance			bw_large.a 1 instance			bw_large.b 1 instance		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
GWSAT(0.5)	100.0	4435	0.69	100.0	74845	25.79	1.0	356127	286.67	100.0	66060	20.55	33.0	222043	241.62
Walksat(0.5)	100.0	1052	0.12	100.0	22157	3.98	44.0	228014	51.33	100.0	16173	2.23	43.0	233497	65.97
Novelty+(0.7,0.01)	100.0	480	0.07	100.0	12805	2.3	98.0	143859	21.45	100.0	10083	1.43	47.0	205571	61.82
depth2-1	100.0	1212	0.16	83.0*+	12937	13.89	44.0+	166917	53.4	73.0*+	6802	18.31	88.0*+	181273	38.59
depth2-2	100.0	388	0.08	81.0*+	13620	16.94	7.0*+	213999	80.98	84.0*+	6718	12.66	18.0*+	219645	89.91
depth3-1	100.0	1052	0.15	84.0*+	9192	13.32	33.0+	206256	64.97	76.0*+	8758	17.49	81.0*+	188747	46.29
dpeth3-2	100.0	393	0.06	87.0*+	10969	12.98	9.0*+	178684	73.96	84.0*+	5466	13.66	32.0	242740	79.07
depth4-1	100.0	546	0.09	100.0	9764	2.06	8.0*+	186419	83.14	100.0	6912	1.33	88.0*+	183345	44.73
depth4-2	100.0	389	0.09	100.0	40270	10.06	6.0*+	302547	107.81	100.0	18510	4.16	3.0*+	293659	122.56
depth5-1	100.0	711	0.15	90.0+	8322	12.02	11.0*+	240091	100.17	85.0*+	5515	15.74	89.0*+	197899	55.15
depth5-2	100.0	612	0.15	100.0	9056	2.57	22.0*+	231474	108.77	100.0	7793	1.98	64.0*	226819	87.38
depth6-1	100.0	502	0.13	100.0	18584	5.6	5.0*+	167078	126.49	100.0	7798	2.18	24.0*+	228202	128.8
depth6-2	100.0	431	0.1	100.0	10686	2.3	9.0*+	317346	84.09	100.0	6291	1.2	71.0*+	211548	59.38
C-Rand	100.0	630	0.11	80.0*+	9200	18.02	16.0*+	221815	76.16	77.0*+	5380	18.3	44.0	257543	77.96

Table 6: SATLIB Planning Instances: Success Rate (%), Average Flips to Solution, and total Runtime on benchmark set (seconds) are reported for 100 runs per instance, 500,000 flip cutoff per run.

	ais6 (1 instance)			ais8 (1 instance)			ais10 (1 instance)			flat125 (100 instances)		
Algorithm	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time	SR	AFS	Time
GWSAT(0.5)	100.0	3088	0.48	100.0	59829	12.62	55.0	207871	92.62	60.38	143128	5411.14
Walksat(0.5)	100.0	1310	0.17	100.0	34427	5.52	73.0	209207	56.88	98.32	67426	562.62
Novelty+(0.7,0.01)	100.0	14188	1.87	92.0	158556	30.78	26.0	249560	94.13	99.23	33384	270.8
depth2-1	100.0	1250	0.2	100.0	27895	5.45	62.0+	202725	73.04	98.58+	56570	532.28
depth2-2	100.0	1217	0.22	100.0	70193	15.15	42.0*+	248377	99.58	0.38*+	296300	5126.01
depth3-1	100.0	1484	0.25	100.0	63288	12.98	21.0*	246091	108.55	81.41*	137015	1930.07
depth3-2	100.0	953	0.17	100.0	52614	10.93	36.0*	238851	97.49	0.33*+	229787	4369.03
depth4-1	100.0	1241	0.23	100.0	80317	17.13	23.0*	255857	110.65	17.37*+	225017	4606.95
depth4-2	100.0	671	0.15	100.0	17573	4.58	71.0+	175182	80.22	79.03*+	141196	3154.49
depth5-1	100.0	1166	0.26	99+.0	81986	22.06	18.0*	169063	128.5	0.99*+	272501	6966.45
depth5-2	100.0	969	0.28	100.0	44556	12.98	50.0*+	224772	118.56	91.93*+	100415	2215.57
depth6-1	100.0	743	0.2	100.0	20657	6.27	73.0+	177931	88.93	74.88*+	149728	4527.96
depth6-2	100.0	998	0.18	100.0	55535	12.03	39.0*+	247451	100.46	34.11*+	203289	4000.53
C-Rand	100.0	1051	0.2	100.0	46594	10.12	26.0	220075	107.84	1.86*+	256905	5056.44

Table 7: SATLIB All-Interval-Series and Graph Coloring Instances: Success Rate (%), Avg. Flips to Solution, and total Runtime (seconds) are reported for 100 runs per instance, 500,000 flip cutoff.

As shown in Section 4.1, one can inspect evolved heuristics (Figure 4), and observe that they *appear* to encode "reasonable" behavior. This is not surprising, since the CLASS language is designed to encode combinations of reasonable building blocks. Unfortunately, as we argued in Section 2, some heuristics which appear reasonable perform very poorly, and it is extremely difficult, if not impossible, to ascertain a priori whether one seemingly good heuristic will actually outperform another Therefore, merely "understanding" the syntactic structure of an automatically generated heuristic is not sufficient. In order to be more confident that that the evolved heuristics are in fact "doing the right thing", we need to obtain a deeper understanding of the behavior of the algorithms that are generated by CLASS. Some tools are available for characterizing the behavior of local search algorithms. Schuurmans and Southey (2001) identified several metrics of local search behavior that were shown to predict the problem solving efficiency of standard SAT heuristics. The following definitions of depth, mobility, and coverage are from (Schuurmans and Southey, 2001; Southey, 2005):

The *depth* of a SAT local search algorithm at time step t is the number of broken clause at step t. We take an average of the depth over all search steps (excluding the first 100 steps).[8] A low average depth (i.e., low mean number of broken clauses) suggests that the search algorithm is spending much of its time exploring states with good objective function values.

Mobility measures how rapidly a search moves in the search space. It is calculated by computing the Hamming distance between variable assignments that are k search steps apart and averaging these distances over the entire sequence of steps to obtain average distances at time lags $k = 1, 2, 3, ..., t$. We measured the mobility over the entire run of the search algorithms. Intuitively, higher mobility means that a search algorithm is exploring new regions rapidly, which is desirable.

Coverage measures how broadly the search space is being explored by a search algorithm. Let the *gap* in the search space be the maximum Hamming distance between any unexplored assignment and the nearest explored assignment. Schuurmans and Southey's coverage metric estimates how rapidly the gap is being reduced – the rate of gap reduction indicates the rate at which an algorithm is covering the search space. Computation of the coverage rate is nontrivial; Southey (2005) presents a precise definition of the metric as well as an algorithm for approximately computing the coverage rate. It is desirable to have high coverage, because search algorithms that get stuck in local optima tend to have low coverage.

We measured depth, mobility, and coverage for three evolved heuristics C1, C2, and C3 (generated in three separate CLASS runs with an s-expression depth limit of 6), executed on 100 instances from the uf100-430 benchmarks (uf100-0001 through uf100-0100, 10 runs per instance, 10,000 flips). We also measured these metrics for several standard heuristics (GWSAT with noise probability 0.5, Walksat with noise probability 0.5, R-Novelty with noise 0.68). As shown in Table 8, the performance of C1, C2, and C3 was competitive with R-Novelty. Furthermore, Table 8 suggests that performance is correlated with mobility and coverage. In other words, it appears that the evolved heuristics and R-Novelty performed well compared to Walksat and GWSAT because they searched more broadly (higher mobility) and more systematically (higher coverage) than Walksat and GWSAT.

[8]Since the initial assignments are randomly generated, all algorithms will begin at a very poor depth. Skipping the first 100 steps focuses the metric on the behavior of the algorithms after the initial descent.

	successes	flips	depth	mobility	coverage
GWSAT	285	4471	8.00	11.683	0.00007854
Walksat	902	2151	6.97	14.504	0.0004819
R-Novelty	987	1101	8.23	23.237	0.001183
C1	958	1624	7.30	18.420	0.000789
C2	952	1771	7.08	20.505	0.000882
C3	989	1045	7.89	22.468	0.001249

Table 8: Local search characteristics of learned and standard heuristics (100 instances, 10 runs per instance (1000 runs total), cutoff=10000 flips). The *successes* column shows the number of successful runs.

Next, we conducted a large-scale experiment to measure the depth, mobility, and coverage metrics on a large sample of heuristics generated by CLASS. 1200 heuristics were chosen as follows: 400 from the population at the end of the CLASS run which produced C2, 400 from the initial (random) population of the same run, and 400 from the population at the end of the run which generated C3. This way, we sought to sample a wide range of heuristics, ranging from very poor (random) to good to very good. Each heuristic was executed on 100 instances from the uf50-215 (50 variable, 215 clause) benchmarks from SATLIB (100 runs per instance with cutoff of 1000 flips). Figures 5 shows the depth, mobility, and coverage metrics versus the heuristic's performance (number of successful runs). There is a very high correlation between performance and coverage (r=0.89), and weaker correlations with depth (r= -0.31) and mobility (r=0.138). The weak correlation between performance and the mobility metric is caused by the large number of very bad heuristics in the randomly generated subset of heuristics that perform very badly but nevertheless display high mobility. When all of the randomly generated heuristics were excluded, then the correlation between mobility and performance increased to r=0.30. The correlation between depth and performance without the random heuristics was 0.22 (note the change in the sign of the coefficient). Figure 5 suggests that high coverage is both necessary and sufficient for local search success on these random 3-SAT instances. For mobility and depth, there is apparently a range of values that are necessary (but not sufficient) for good performance.

These experiments show that good performance in heuristics generated by CLASS is correlated to their coverage and mobility, which is expected of good local search algorithms – the good performance of evolved heuristics is not just due to pure "luck" or some unexplainable exploitation of our particular benchmark instances.

6 Evaluation of the CLASS Genetic Programming Algorithm

Searching for good SAT local search 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 execution of the candidate heuristic, i.e., each call to the fitness function results in hundreds of thousands of evaluations of the candidate s-expression. The main goal of this work was to see whether it was possible to automatically discover novel SAT local search heuristics that were competitive with previous, hand-crafted heuristics. Thus, an extensive empirical comparison of meta-level evolutionary algorithms for discovering SAT heuristics is outside of the scope of this paper and left for future work. Given that each run of CLASS takes several hours, obtaining

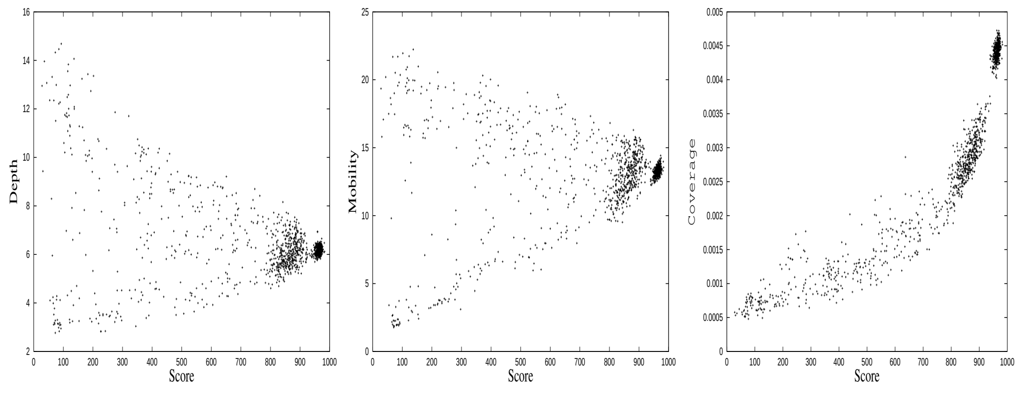

Figure 5: Average depth, mobility, and coverage rates vs. Score (# of successful runs) for 1200 automatically generated heuristics

statistically meaningful comparison of meta-level search algorithms with parameter tuning would be a nontrivial endeavor. However, the following results indicate that the current CLASS algorithm (Figure 2) is at least a reasonable approach.

We first 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 2, Line 1. We ran each algorithm 10 times. Each run involved 5500 fitness function evaluations. In each run of the GP, we generated and evaluated a population of 1000 and ran until 4500 children had been generated and evaluated. For each run of G&T, 5500 random individuals were evaluated. Both algorithms were run with an s-expression depth limit of 6. Let S_{best} denote the fitness function score (see Section 3.2) of the best individual found during each run of either the GP or G&T. We compared the mean value of S_{best} over the 10 runs, i.e., $\sum_{i=0}^{i=10} S_{best}(i)/10$, for our GP and G&T algorithms. For GP, the mean value of S_{best} was 1377 (with a standard deviation of 14.3), and for G&T, the mean was 1213 with a standard deviation of 47.3. The difference between the means is statistically significant ($p < 0.001$ according to a t-test). This shows that the GP was significantly more effective than G&T at finding higher quality heuristics.

We have also investigated the use of standard GP crossover and mutation operators, but so far, we have not been successful in evolving competitive heuristics. Since a single 5500 evaluation run of CLASS requires around 6 hours on a workstation, systematically tuning GP control parameters is a time-consuming and difficult task which we have not yet done. More experimental work is needed to determine whether there exist good control parameter settings which may allow standard GP operators to generate heuristics that are competitive with Walksat and Novelty+.

6.1 Analysis of GP Populations

In order to better understand the characteristics of the populations of heuristics evolved by the CLASS genetic algorithm, we analyzed the populations of heuristics evolved by CLASS. We first considered a single run of the CLASS GP, which we call *Run-1*. This is the run that generated the depth3-1 heuristic (population of 1000, 5500 total individuals

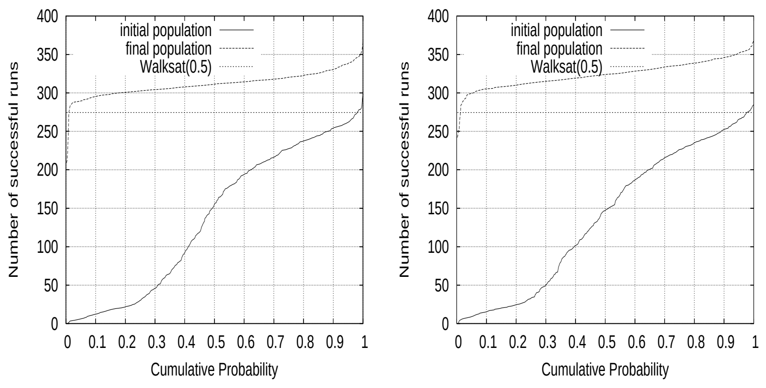

Figure 6: Analysis of initial and final populations for two CLASS GP runs.

generated, depth limit of 3). From this run, we consider the initial, randomly generated population (*initial population*), as well as the population at the end of the run (*final population*). For each member (heuristic) of the initial population, we executed the heuristic on five times on each of the 80-variable, Suite B instances with a cutoff of 30,000 flips per trial, and recorded the number of successful trials. The maximum possible score is 500 (5 trials per instance x 100 instances = 500). We then repeated this procedure for each member of the final population. This data (*Run-1* is plotted on the left side of Figure 6 as a cumulative probability distribution. For example, according to the graph on the left side of Figure 6, approximately 63% of the initial population had 200 or fewer successful trials, and approximately 20% of the final population had fewer 300 or fewer successful trials. For reference, we also executed the standard Walksat heuristic (walk probability = 0.5) five times on the same set of instances, resulting in 274 successful runs. This is shown as a straight horizontal line in Figure 6. This analysis was repeated for another sample run, *Run-2*, which used the same parameters as *Run-1*. The results for *Run-2* are shown on the right in Figure 6. Note that:

- The final populations for these two runs are clearly composed of better individuals than the initial populations
- Some of the randomly generated heuristics in the initial populations scored comparably to the standard Walksat algorithm.
- Almost all members of the final populations outscore Walksat.

6.2 On the Utility of Randomly Generated Individuals

The previous section showed that with respect to one particular evaluation function, some of the members of the initial population appeared to be comparable with the standard Walksat algorithm. This suggests that the CLASS language defines a search space of SAT local search heuristics which is rich in high-quality heuristics, compared to previously published, manually designed heuristics.

To further demonstrate the power of heuristics that could be generated by random generate-and-test, we evaluated the best individual heuristic generated by on one of

```
(IF-TABU 40
   (OLDER-VAR (GET-VAR +BC0+ +NET-GAIN+)
              (IF-VAR-COMPARE /= +POS-GAIN+
                              (OLDER-VAR (GET-VAR +BC1+ +POS-GAIN+)
                                         (GET-VAR +BC1+ +NEG-GAIN+))
                              (IF-RAND-LT 0.75 (GET-VAR +BC1+ +POS-GAIN+)
                                               (GET-VAR +BC0+ +POS-GAIN+))))
   (GET-VAR +BC0+ +NEG-GAIN+))
```

Figure 7: C-Rand: a randomly generated heuristic (best out of 5500 random individuals)

the G&T runs described in Section 6. We ran this heuristic, `C-Rand` on the Gottlieb benchmark suite. The results are included in Tables 3-7. While the search efficiency of C-Rand is not particularly outstanding when compared to the evolved heuristics, it performs quite well on random 3-SAT instances, and it also runs fast due its relative simplicity. For example, on the 100 variable problems in SuiteA, SuiteB, and SuiteC (Table 4), C-Rand significantly outperforms Walksat and Novelty+, as well as most of the direct, evolutionary approaches, and has better runtimes than the majority of the evolved heuristics. The fact that a randomly generated heuristic can perform relatively well compared to some previously published, human-designed heuristics indicates that the CLASS language successfully defines a search space where it is relatively easy to find high-quality solutions. Furthermore, the fact that C-Rand is competitive with algorithms that have been published in recent years suggests that while humans are able to invent new building blocks for SAT algorithms, human performance on generating good combinations of the building blocks is difficult for humans (i.e., not much better than random generate-and-test).

7 Related Work

CLASS uses genetic programming to generate control rules for a problem-solver. This approach has previously been used successfully in a number of domains, including the evolution of heuristics for a compiler (Stephenson, O'Reilly, Martin, and Amarasinghe, 2003) and the evolution of control rules for the PRODIGY planner (Aler, Borrajo, and Isasi, 2002). Several systems in the AI literature have learned to improve the performance of constraint satisfaction systems by modifying heuristics using what is essentially a heuristically guided generate-and-test procedure like CLASS. MULTI-TAC (Minton, 1996) adapts generalized constraint-satisfaction heuristics for specific problem classes. COMPOSER was used to configure an antenna-network scheduling algorithm (Gratch and Chien, 1996). Although CLASS clearly follows the spirit of previous systems such as MULTI-TAC, COMPOSER, and the planning system by Aler et al., we believe that CLASS extends the frontier of this line of research by demonstrating that given the same building blocks available to human researchers, an automated system could outperform well-known SAT solvers that were the result of an intensive research effort by a community of expert researchers through the 1990's.

The heuristics represented by CLASS can be viewed as a way to choose among low-level variable selection heuristics, where the functions in Table 2 with `variable` return types correspond to the low-level heuristics. From this point of view, the standard Walksat heuristic is a choice among the two low-level heuristics `(GET-VAR`

`+BC0+ +NEG-GAIN+)` and `(VAR-RANDOM +BC0+)`(see Figure 3). Thus, CLASS can be considered an instance of a *hyper-heuristic* approach to solving difficult combinatorial problems. The term "hyper-heuristic" was coined by Cowling, Kendall, and Soubeiga, (2001) to refer to the general approach of applying meta-level techniques (including learning) that "choose between a set of low-level heuristics, using some learning mechanism". A survey of hyper-heuristic approaches is by Burke et al. (2003).

CLASS works on building blocks that are lower level than most of the work in the hyper-heuristic literature. For example, a typical example of the hyper-heuristic approach is work of Ross et al. (2002), who developed a learning classifier system (XCS) which, given a bin packing problem instance, applies a set of 8 bin packing heuristics such as least-fit decreasing and next-fit-decreasing at each step, where the XCS determines which heuristic to apply next. These heuristics such as *least-fit decreasing* and *next-fit-decreasing* are relatively high-level compared to the CLASS primitives In addition, the "choice" of low-level heuristic made by the CLASS local search heuristic applies to just one single variable flip; at each flip, the evolved heuristic is applied and a new "choice" is made. This contrasts with Ross et al.'s approach, where each application of a heuristic choice results in a large number of changes to the solution state.

On one hand, the building blocks used by CLASS to generate the initial population (i.e., the functions and terminals) are lower-level components than the heuristics that are typically used in hyper-heuristic approaches, where the component heuristics are fully functional heuristics (e.g., the heuristics used by Ross et al. for bin packing).

Burke, Hyde, and Kendall, evolve bin packing heuristics using genetic programming (Burke, Hyde, and Kendall, 2006). They evolve heuristics that determine whether or not to put a given item into a given bin. The evolved function takes as input a bin and an item [9] and returns a number. If the return value is greater than 0, the item is placed in the bin; otherwise the item is not placed in the bin. In the terminology of constraint programming, this is a *value selection heuristic*, which, given a variable, decides which value to assign to it. Note that for SAT local search, value selection is trivial, because the only values for a variable are *true* or *false*, so once a variable is selected (Figure 1, there is only one other value available to assign. CLASS, on the other hand, evolves a *variable* selection heuristic. In the system of Burke et al., the variable selection heuristic is the decision about which (bin, item) pair to consider next as input to their evolved value selection heuristic (they use a static ordering which loops over bins and items).

Another technique that has been used to creating "composite" heuristics for problem solvers is the algorithm portfolio approach (Huberman, Lukose, and Hogg, 1997; Gomes and Selman, 1997), which allocates resources among a set of alternative algorithms for solving a given problem instance. As with hyper-heuristics, this differs from CLASS in that the building blocks used for portfolios are higher-level algorithms.

There is a very large body of work on hybrid algorithms that combine evolutionary computation and local search. The key difference between CLASS and hybrid genetic-local search algorithms is that while solving a particular problem instance, the learning/evolutionary component of CLASS is not used – the learned heuristic is static when it is applied to any given problem instance. The genetic programming algorithm is only

[9]This is description is highly simplified for the sake of clarity – see (Burke et al., 2006).

used off-line during the "training" phase in order to generate a local search heuristic that is likely to succeed on a given problem instance. In contrast, typical genetic-local search hybrids such as memetic algorithms (Merz and Freisleben, 2000; Krasnogor and Smith, 2000), as well as several of the direct, evolutionary approaches for SAT surveyed in Section 4.8 apply both genetic and local searches to a given problem instance. In addition, some recent memetic algorithms evolve not only the solution itself, but also evolve the local search heuristics during the run of the algorithm on a given instance (Smith, 2002; Krasnogor and Gustafson, 2004).

8 Discussion and Future Work

The main contribution of this paper is the demonstration that the discovery of SAT local search heuristics, a very difficult task that has occupied many human researchers, could be automated by a genetic programming system. A historical analysis of the SAT local search literature showed that well-known, successful SAT heuristics are composed of key building blocks. We showed that although these building blocks were all well-known by 1994, significant advances based on new combinations of building blocks were being made as late as 1999. We showed that a genetic programming system which manipulated the same building blocks that were available to the SAT research community in 1994, could discover algorithms that were competitive (with respect to both search efficiency and runtime) with the state-of-the-art SAT solvers of 1999.

When preliminary results for this work was first published based on an earlier version of CLASS (Fukunaga, 2002), Novelty+ and Walksat were the state of the art local search algorithms, despite many attempts to improve upon them. Although other algorithms had been proposed which could outperform the Walksat with respect to search efficiency (flips), none of them could significantly outperform Novelty+ with respect to overall runtime, because the mechanisms that were responsible for improved search efficiency came at the cost of a high per-flip complexity. [10] A hand-coded algorithm which could convincingly outperform Novelty+ with respect to both search efficiency and runtime was not found until the SAPS algorithm in 2002 (Hutter, Tompkins, and Hoos, 2002), which used an efficient implementation of clause weighting. This shows that in historical context, discovering algorithms competitive with the Novelty variants was clearly a very challenging task which had confounded many researchers.

Recently, the state of the art in SAT local search has improved significantly due to the discovery of very efficient implementations of clause weights (Hutter et al., 2002; Thornton, 2005). Furthermore, Pham, Thornton, and Sattar have recently developed a technique which represents dependencies between variables and significantly improves SAT local search performance on structured instances from CAD and AI applications (Pham, thornton, and Sattar, 2007). A promising direction for future research is to improve CLASS by adding clause weighting and variable dependencies. Since we have shown in this paper that CLASS is able to generate heuristics that are competitive with the Walksat and Novelty variants, given all the building blocks that were available to the Walksat and Novelty inventors, there is good reason to be optimistic that it is possible to generate algorithms that are competitive with the current state of the art algorithms, given the same key, additional building blocks.

[10]E.g., it was well-known since 1993 that clause weighting resulted in significant improvements in search efficiency (Selman and Kautz, 1993). However, overall runtimes of clause weighting tended to underperform Walksat and Novelty variants.

There are many other avenues for future work. In Section 4.7, we showed that a CLASS heuristic which was evolved using only random 3-SAT instances in the training set generalized fairly well on more structured problem instances which were generated using different problem generators. An area for future work is to further improve generalization by extending the CLASS evaluation function 3.2 to include instances from a wider range of problem classes.

There is much more work to be done in exploring alternate mechanisms for exploring the space of heuristics. We showed that the current CLASS GP algorithm is *sufficient* for generating heuristics which are competitive with standard local search heuristics, and that the CLASS language (functions/terminals) define a search space where it is relatively easy to find high-quality heuristics. As shown in Section 6.2, even randomly generated heuristics in this language can compete with some previously published, human-designed heuristics; this further supports our hypothesis that generating good combinations of the building blocks is very difficult for humans (i.e., not much better than brute-force, generate and test). This suggests that an important area for future work is to improve the process of developing new heuristic algorithms by combining the strengths of humans (invention of building blocks) and machines (brute-force search). One approach is to integrate more human guidance into CLASS. We have done some preliminary work in this direction by implementing CLASS-L, which incorporates a *library* of hand-coded fragments and higher level building blocks (Fukunaga, 2002). Finally, although we showed that the current CLASS GP significantly outperforms random sampling of this space, we have only begun to explore the space of GP algorithms, and future work will more systematically investigate alternative GP search strategies (e.g., using standard crossover and mutation operators).

9 Conclusion

This paper described and evaluated a system for automatically discovering SAT local search variable selection heuristics. We have shown that a simple genetic programming algorithm based on a single operator (composition) suffices to generate effective SAT local search heuristics that are competitive with efficient implementations of standard Walksat and Novelty variants, as well as previous evolutionary approaches. The evolved heuristics scale and generalize fairly well on random instances as well as more structured problem classes. The history of SAT heuristics suggests that human researchers excel at finding and classifying the relevant building blocks for problem solving. However, the task of combining these features into effective composite heuristics appears to be a combinatorial problem that is difficult for humans. We have shown that this difficult problem can be solved by a simple evolutionary approach, which discovered new heuristics that are competitive with some of the best human-designed heuristics for SAT. Our results demonstrate that evolutionary computation is a very promising approach for discovering successful heuristics for difficult problems.

Acknowledgments

Thanks to Rich Korf and Jason Fama for helpful discussions, and Jim Clune for helpful comments on a draft of this paper. Finnegan Southey provided valuable clarifications regarding the implementation of the coverage metric measurements. Thanks to the anonymous reviewers for helpful comments that significantly improved this paper.

References

Aler, R., Borrajo, D., and Isasi, P. (2002). Using genetic programming to learn and improve control knowledge. *Artificial Intelligence*, *141*(1-2), 29–56.

Burke, E., Hyde, M., and Kendall, G. (2006). Evolving bin packing heuristics with genetic programming. In *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN)*, Lecture Notes in Computer Science Vol 4193, pp. 860–869. Springer.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003). Hyper-heuristics: An emerging direction in modern search technology. In Glover, F. and Kochenberger, G. (Eds.), *Handbook of Meta-heuristics*, chap. 16, pp. 457–474. Kluwer.

Cowling, P., Kendall, G., and Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. In Burke, E. and Erben, W. (Eds.), *Selected Papers of the Third International Conference on the Practice and Theory of Automated Timetabling (PATAT 2000)*, Lecture Notes in Computer Science, pp. 176–190.

de Jong, M. and Kosters, W. (1998). Solving 3-SAT using adaptive sampling. In Poutré, H. and van den Herik, J. (Eds.), *Proceedings of the Tenth Dutch/Belgian Artificial Intelligence Conference*, pp. 221–228.

Eiben, A. and van der Hauw, J. (1997). Solving 3-SAT with adaptive genetic algorithms. In *Proc. Fourth IEEE Conference on Evolutionary Computation*, pp. 81–86.

Fukunaga, A. (2002). Automated discovery of composite SAT variable-selection heuristics. In *Proc. AAAI*, pp. 641–648.

Fukunaga, A. (2004a). Efficient implementations of SAT local search. In *Proceedings of Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*, pp. 287–292, Vancouver, British Columbia.

Fukunaga, A. (2004b). Evolving local search heuristics for SAT. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, Vol. 3103 of *Lecture Notes in Computer Science*, pp. 483–494. Springer-Verlag.

Gent, I. and Walsh, T. (1993). Towards an understainding of hill-climbing procedures for SAT. In *Proceedings of National Conf. on Artificial Intelligence (AAAI)*, pp. 28–33.

Gomes, C. and Selman, B. (1997). Algorithm portfolio design: theory vs. practice. In *Proc. Uncertainty in Artificial Intelligence (UAI)*.

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

Gottlieb, J. and Voss, N. (2000). Adaptive fitness functions for the satisfiability problem. In *Proceedings of the Conference on Parallel Problem Solving from Nature*, Vol. 1917 of *Lecture Notes in Computer Science*, pp. 621–630. Springer-Verlag.

Gratch, J. and Chien, S. (1996). Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, *4*, 365–396.

Hoos, H. (1998). *Stochastic local search - methods, models, applications*. Ph.D. thesis, TU Darmstadt.

Hoos, H. (1999). On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of AAAI*, pp. 661–666.

Hoos, H. and Stutzle, T. (2000). Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, *24*, 421–481.

Huberman, B., Lukose, R., and Hogg, T. (1997). An economics approach to hard computational problems. *Science*, *275*(5269), 51–4.

Hutter, F., Tompkins, D., and Hoos, H. (2002). Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. Eighth International Conference on the Principles and Practice of Constraint Programming (CP'02)*, pp. 233–248.

Kautz, H. (2006). Deconstructing planning as satisfiability. In *Proc. Twenty-first National Conference on Artificial Intelligence (AAAI-06)*, pp. 1524–1526.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers By the Means of Natural Selection*. MIT Press.

Krasnogor, N. and Gustafson, S. (2004). A study on the use of "self-generation" in memetic algorithms. *Natural Computing*, *3*(1), 53–76.

Krasnogor, N. and Smith, J. (2000). A memetic algorithm with self-adaptive local search: TSP as a case study. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2000)*, pp. 987–994. Morgan Kauffman.

Lardeux, F., Saubion, F., and Hao, J.-K. (2006). GASAT: A genetic local search algorithm for the satisfiability problem. *Evolutionary Computation*, *14*(2), 223–253.

Luke, S. and Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evol. Comput.*, *14*(3), 309–344.

Marchiori, E. and Rossi, C. (1999). A flipping genetic algorithm for hard 3-SAT problems. In et al., W. B. (Ed.), *Proceedings of Genetic and Evolutionary Computation Conference*, pp. 393–400. Morgan Kaufmann.

McAllester, D., Selman, B., and Kautz, H. (1997). Evidence for invariants in local search. In *Proceedings of National Conf. on Artificial Intelligence (AAAI)*, pp. 459–465.

Merz, P. and Freisleben, B. (2000). Fitness landscapes, memetic algorithms, and greedy operators for graph bipartitioning. *Evolutionary Computation*, *8*(1), 61–91.

Minton, S. (1996). Automatically configuring constraint satisfaction problems: a case study. *Constraints*, *1*(1).

Mitchell, D., Selman, B., and Levesque, H. (1992). Hard and easy distributions of SAT problems. In *Proceedings of National Conf. on Artificial Intelligence (AAAI)*, pp. 459–65.

Montana, D. (1993). Strongly typed genetic programming. Tech. rep., Bolt, Beranek and Neuman (BBN).

Pham, D., thornton, J., and Sattar, A. (2007). Building structure into local search for SAT. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2359–2364.

Ross, P., Schulenburg, S., Marin-Blazques, J., and Hart, E. (2002). Hyper-heuristics: learning to combine simple heuristics in bin-packing problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pp. 942–948.

Rossi, C., Marchiori, E., and Kok, J. (2000). An adaptive evolutionary algorithm for the satisfiability problem. In *Proceedings of ACM Symposium on Applied Computing*, pp. 463–469, New York, New York. ACM.

Schuurmans, D. and Southey, F. (2001). Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, *132*, 121–150.

Selman, B. and Kautz, H. (1993). Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proc. Intl. Joint Conf. Artificial Intelligence (IJCAI)*.

Selman, B., Kautz, H., and Cohen, B. (1994). Noise strategies for improving local search. In *Proceedings of National Conf. on Artificial Intelligence (AAAI)*.

Selman, B., Levesque, H., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of National Conf. on Artificial Intelligence (AAAI)*, pp. 440–446.

Smith, J. (2002). Co-evolution of memetic algorithms: initial results. In et al., J. M. (Ed.), *Proceedings of the 7th International Conference on Parallel Problem Solving from nature (PPSN)*, Lecture Notes in Computer Science Vol.4193, pp. 537–548. Springer.

Southey, F. (2005). Constraint metrics for local search. In *Proc. 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, pp. 269–281.

Stephenson, M., O'Reilly, U., Martin, M., and Amarasinghe, S. (2003). Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA.

Thornton, J. (2005). Clause weighting local search for SAT. *Journal of Automated Reasoning*, *35*(1-3), 97–142.

Velev, M. and Bryant, R. (2003). Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, *35*(2), 73–106.

Whitley, D. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proc. International Conf. on Genetic Algorithms (ICGA)*, pp. 116–121.