SATenstein: Automatically Building Local Search SAT Solvers From Components

Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, Kevin Leyton-Brown
Department of Computer Science
University of British Columbia
{ashiqur, xulin730, hoos, kevinlb}@cs.ubc.ca

Abstract

Designing high-performance algorithms for computationally hard problems is a difficult and often time-consuming task. In this work, we demonstrate that this task can be automated in the context of stochastic local search (SLS) solvers for the propositional satisfiability problem (SAT). We first introduce a generalised, highly parameterised solver framework, dubbed SATenstein, that includes components gleaned from or inspired by existing high-performance SLS algorithms for SAT. The parameters of SATenstein control the selection of components used in any specific instantiation and the behaviour of these components. SATenstein can be configured to instantiate a broad range of existing high-performance SLS-based SAT solvers, and also billions of novel algorithms. We used an automated algorithm configuration procedure to find instantiations of SATenstein that perform well on several well-known, challenging distributions of SAT instances. Overall, we consistently obtained significant improvements over the previously best-performing SLS algorithms, despite expending minimal manual effort.[1]

1 Introduction

In Mary Shelley's classic novel *Frankenstein; or, The Modern Prometheus*, a brilliant and obsessed scientist, Victor Frankenstein, sets out to create a 'perfect' human being from scavenged body parts. The approach we follow in this work is based on the same general idea: we build new algorithms using components scavenged from existing high-performance solvers for a given problem. Unlike Frankenstein, we use an automated construction process, which enables us to build algorithms whose performance is optimised for specific types of tasks (i.e., problem instances) with minimal human effort.

In the traditional approach to building heuristic algorithms, most design choices are fixed at development time, with a small number of parameters exposed to the user. In contrast, we advocate an approach in which the designer fixes as few choices as possible, instead exposing a vast number of design elements as parameters that can be configured at runtime. This spares the designer the burden of making early decisions without knowing how they will interact with other algorithm components on instance distributions of interest. Instead, the designer can include as many alternate approaches to solving the same subproblem as seem promising, drawing on elements of known algorithms as well as novel mechanisms. Of course, to solve actual problems, such a framework must be made to instantiate a particular algorithm by setting its parameters. We use a black-box algorithm configuration procedure to make this choice for an instance distribution of interest.

Of course, we are not the first to propose building algorithms by using automated methods to search a large design space. To the contrary, our work can be seen as part of a general and growing trend, fueled by an increasing demand for high-performance solvers for difficult combinatorial problems in practical applications, by the desire to reduce the human effort required for building such algorithms, and by an ever-increasing availability of cheap computing power that can be harnessed for automating parts of the algorithm design process (see also [Hoos, 2008]). There are many examples of work along these lines [Gratch and Dejong, 1992; Minton, 1993; Carchrae and Beck, 2005; Xu *et al.*, 2008; Gagliolo and Schmidhuber, 2006; Fukunaga, 2002; Oltean, 2005; Westfold and Smith, 2001; Monette *et al.*, 2009; Gaspero and Schaerf, 2007]. We discuss the most closely-related work in detail in Section 2.

In broad strokes, our approach is distinguished in three key ways. First, it explicitly separates the specification of a vast combinatorial design space from the search for high-performance algorithms in that space. Second, it uses a completely automated procedure for the latter stage, which makes it possible to obtain specialised solvers for various types of input instances using essentially exclusively computational power rather than human effort. Finally, it is validated by the fact that we obtain new state-of-the art solvers for one of the most challenging and widely-studied problems in computer science: the propositional satisfiability problem (SAT). We also note that our approach differs from automated algorithm selection methods such as SATzilla [Xu *et al.*, 2008], which select one of several solvers to be run on a given problem instance based on characteristics of that instance. Indeed, the two approaches are complementary: methods like SATzilla can take advantage of solvers obtained using the automated design approach pursued in this work.

Although the approach we have outlined is not limited to a particular domain, this paper focuses on the automated con-

[1]We thank Frank Hutter for valuable suggestions regarding automatic configuration, experimental setup, and algorithm design, and Dave Tompkins for his help with extending the UBCSAT framework.

struction of stochastic local search (SLS) algorithms for SAT. This domain is a challenging one for automated algorithm design, since a broad and sustained research effort has gone into the (manual) design of high-performance SLS algorithms for SAT since the late 1980s. SLS-based solvers continue to represent the state of the art for solving various types of SAT instances; they also play an important role in state-of-the-art portfolio-based automated algorithm selection methods for SAT [Xu *et al.*, 2008].

This paper introduces SATenstein, a flexible new framework for designing SLS SAT solvers. SATenstein incorporates components from over two dozen existing SLS algorithms for SAT, as well as on a variety of novel strategies. In total, this design space comprises more than 2×10^{11} distinct SAT solvers, including most state-of-the-art SLS-based SAT solvers known from the literature as well as a vast number of completely novel designs. It therefore spans a much wider range of distinct algorithms than previous SAT solver frameworks such as our own UBCSAT [Tompkins and Hoos, 2004], which we used as a starting point for this work.

We rely on a black-box algorithm procedure to choose an actual SAT solver from the rich design space defined by our framework. In principle, any sufficiently-powerful configuration procedure could be used to identify SATenstein configurations that perform well on given distributions of SAT instances. We used ParamILS [Hutter *et al.*, 2007b; 2008], a tool developed in our own group, for this purpose.

We demonstrate experimentally that our new, automatically-constructed solvers outperform the best SLS-based SAT solvers currently available by a significant margin on six well-known SAT instance distributions, ranging from hard random 3-SAT instances to SAT-encoded factoring and software verification problems. Because SLS-based SAT solvers are the best known methods for solving four of these six benchmark distributions, our new solvers represent a substantial advance in the state of the art for solving the respective sub-classes of SAT. On the remaining two types of instances, in particular on SAT-encoded software verification problems, our new solvers narrow the gap between the performance of the best SLS algorithms and the best DPLL-based solvers.

The remainder of this paper is organised as follows. After a discussion of related work (Section 2), we describe the design of SATenstein (Section 3). This is followed by a description of the setup we used for empirically evaluating SATenstein (Section 4) and a presentation and discussion of the results from these experiments (Section 5). We end with some general conclusions and an outlook on future work (Section 6).

2 Related Work

There is a large body of literature in AI and related areas that deals with automated methods for building effective algorithms. This includes work on automatic algorithm configuration (see, e.g., [Gratch and Dejong, 1992; Minton, 1993; Hutter *et al.*, 2007b]), algorithm selection (see, e.g., [Guerri and Milano, 2004; Carchrae and Beck, 2005; Xu *et al.*, 2008]), algorithm portfolios (see, e.g., [Gomes and Selman, 2001; Gagliolo and Schmidhuber, 2006]), and, to some extent, genetic programming (see, e.g., [Fukunaga, 2002; 2004; Oltean, 2005]) and algorithm synthesis (see, e.g., [Westfold and Smith, 2001; Monette *et al.*, 2009; Gaspero and Schaerf, 2007]). In what follows, we restrict our discussion to research efforts that are related particularly closely to our approach.

Notably, Fukunaga automatically constructed variable selection mechanisms for a generic SLS algorithm for SAT by means of genetic programming [Fukunaga, 2002; 2004]. The design space considered in his approach is potentially unbounded, and his genetic programming procedure is custom-tailored to searching this space. Only GSAT-based and WalkSAT-based SLS algorithms up to the year 2000 were considered, and candidate variable selection mechanisms were evaluated solely on Random-3-SAT instances with at most 100 variables. In contrast, we consider a huge but bounded combinatorial space of algorithms based on components taken from 25 of the best SLS algorithms for SAT currently available, and we use an off-the-shelf, general-purpose algorithm configuration procedure to search this space. During this meta-algorithmic search process, candidate solvers are evaluated on SAT instances with up to 4 878 variables and 56 238 clauses. The best automatically-constructed solvers obtained by Fukunaga achieved a performance level similar to the best WalkSAT variants available in 2000 on moderately-sized SAT instances, but did not consistently improve on the performance of the best SLS-based SAT algorithms at the time. In contrast, our new SATenstein solvers perform substantially better than current state-of-the-art SLS-based SAT solvers on a broad range of challenging SAT instances. Finally, while Fukunaga's approach in principle could be used to obtain high-performance solvers for specific types of SAT instances, to our knowledge this potential was never realised. Our approach, on the other hand, is specifically designed for automatically building high-performance solvers for given instance distributions, and our empirical results clearly show that it works well in practice.

Two other pieces of work from our own group are conceptually related to the work presented here. Hutter *et al.* [2007a] used an automated algorithm configuration procedure to refine a highly-parametric DPLL-type SAT solver, SPEAR, which was subsequently tuned, using the same configuration procedure, for solving SAT-encoded hardware and software verification problems, respectively. These automatically-tuned versions of SPEAR were demonstrated to improve the state of the art in solving these types of SAT instances at the time. Similarly, Chiarandini *et al.* [2008] recently used the same automated algorithm configuration procedure to design a modular SLS algorithm for timetabling problems, which subsequently placed third in one of the categories of the 2007 International Timetabling Competition. In both cases, the automated configuration procedure was used as a tool within the human design process, while in the work presented here it provides the basis for a distinct and fully-automated phase after this process.

Another conceptual relationship exists with our own work on SATzilla [Xu *et al.*, 2008]: Both approaches use automatic methods to achieve improved performance on given distributions of SAT instances by selecting from a set of algorithms. (And, of course, both have similar names!) However, the similarity ends there. SATzilla performs per-instance selection from a fixed set of black-box SAT solvers, relying on characteristics of a given instance and substantial amounts of runtime

data for each solver. In contrast, SATenstein is an approach for automatically building novel solvers from components. Together, these components define a vast space of candidate algorithms, most of which have never been studied before. We advocate instantiating solvers from this space only on a per-distribution basis and without considering characteristics of individual problem instances. Furthermore, algorithm configuration tools that search this space are limited to gathering a very limited amount of runtime data for most configurations considered. In fact, our two approaches are highly complementary: SATenstein can be used to obtain new SAT algorithms to be used within SATzilla. Indeed, the latest version of SATzilla (recently submitted to the 2009 SAT Competition) makes use of several solvers constructed using SATenstein [Xu *et al.*, 2009].

Existing work on algorithm synthesis is mostly focused on automatically generating algorithms that satisfy a given formal specification or that solve a specific problem from a large and diverse domain (see, e.g., [Westfold and Smith, 2001; Monette *et al.*, 2009; Gaspero and Schaerf, 2007]), while our work is focused on performance optimisation in a large space of candidate solvers that are all guaranteed to be correct by construction. Clearly, there is complementarity between both approaches; at the same time, because of the significant difference in focus, the methods considered in algorithm synthesis and performance-oriented automated algorithm design, as considered here, are quite different.

Finally, our work builds fundamentally on the large body of research that gave rise to the high-performance SAT solvers from which we took the components for our SATenstein solvers (see Section 3), and on the UBCSAT solver framework for SLS-based SAT algorithms [Tompkins and Hoos, 2004], on top of which we implemented SATenstein.

Procedure `SATenstein-LS(...)`

```
   Input: CNF formula φ; real number cutoff;
          booleans performDiversification,
   singleClauseAsNeighbor, usePromisingList
   Output: Satisfying variable assignment
   Start with random Assignment A;
   Initialise parameters;
   while runtime < cutoff do
       if A satisfies φ then
           return A;
       VarFlipped ← FALSE;
       if performDiversification then
B1         if within probability diversificationProbability() then
B1             c ← selectClause();
B1             y ← diversificationStrategy(c) ;
B1             VarFlipped ← TRUE;
       if Not VarFlipped then
           if usePromisingList then
B2             if promisingList is nonempty then
B2                 y ← selectFromPromisingList() ;
               else
B2                 c ← selectClause();
B2                 y ← selectHeuristic(c) ;
           else
               if singleClauseAsNeighbor then
B3                 c ← selectClause();
B3                 y ← selectHeuristic(c) ;
               else
B4                 sety ← selectSet();
B4                 y ← tieBreaking(sety);
           flip y ;
B5         update();
```

3 SATenstein-LS

Nearly all existing SLS-based SAT solvers can be grouped into one of four broad categories: GSAT-based, WalkSAT-based, dynamic local search and G^{2}WSAT-variant algorithms. While GSAT-based algorithms are of considerable historical importance, no present state-of-the-art SAT algorithm belongs to this category. We thus constructed our solver framework to span high-performance local search algorithms from the remaining three families.

This framework is described as Procedure `SATenstein-LS`. The pseudocode is roughly divided into five building blocks, denoted B_1–B_5. A `SATenstein-LS` instantiation has the following general structure:

1. Optionally execute B_1 to perform search diversification [Pham and Gretton, 2007; Hoos, 2002];
2. Obtain a variable to flip by executing one of B_2, B_3, and B_4—thus determining whether a G^{2}WSAT-derived (B_2), WalkSAT-based (B_3), or dynamic local search algorithm (B_4) is instantiated—and then flip this variable;
3. Optionally execute B_5 to perform updates to the promising variable list, tabu attributes, clause penalties or dynamically adapted algorithm parameters.

Each of our blocks is built from one or more components, some of which are shared across multiple blocks. These components are summarised in Table 1. Each component is in turn configurable by one or more parameters (41 in total; not all of which are reflected in the high-level pseudocode), to select among different core heuristics from high-performance SLS algorithms.[2] These are the parameters that we expose on the command line and tune using our automatic configurator. Because some of these parameters conditionally depend on others, it is difficult to determine the exact number of valid `SATenstein-LS` instantiations; a conservative lower bound is 2×10^{11}.

We now give a high-level description of each building block. B_1 depends on the *selectClause()*, *diversificationStrategy()*, and *diversificationProbability()* components. Component *selectClause()* takes a categorical parameter as input and, depending on its value, selects a false clause uniformly at random or with probability proportional to its clause penalty [Tompkins and Hoos, 2004]. Component *diversificationStrategy()* can be configured by a categorical parameter to do any of the following with probability *diversificationProbability()*: flip the least recently flipped variable [Li and Huang, 2005], flip the least frequently flipped variable [Prestwich, 2005], flip the variable with minimum variable weight [Prestwich, 2005], or flip a randomly selected variable [Hoos, 2002].

[2] For detailed information, please refer to [KhudaBukhsh, 2009].

Component	Block	# param	# poss. values	Based on
diversificationProbability()	1	1	4	[Pham and Gretton, 2007; Hoos, 2002; Prestwich, 2005; Li and Huang, 2005]
SelectClause()	1, 2, 3	1	2	[Tompkins and Hoos, 2004]
diversificationStrategy()	1	4	24	[Pham and Gretton, 2007; Hoos, 2002; Prestwich, 2005; Li and Huang, 2005]
selectFromPromisingList()	2	4	$\geq$ 4341	[Li and Huang, 2005; Li et al., 2007a; 2007b; Pham and Gretton, 2007]
selectHeuristic()	2, 3	9	$\geq$ 1220800	[Hoos, 2002; Li and Huang, 2005; Li et al., 2007a; 2007b; Prestwich, 2005; Selman et al., 1994; McAllester et al., 1997]
selectSet()	4	9	$\geq$ 111408	[Hutter et al., 2002; Thornton et al., 2004]
tiebreaking()	4	1	4	[Hoos, 2002; Li and Huang, 2005; Prestwich, 2005]
update()	5	12	$\geq$ 73728	[Hoos, 2002; Li et al., 2007a; Hutter et al., 2002; Thornton et al., 2004; Pham and Gretton, 2007; Li and Huang, 2005]

Table 1: `SATenstein-LS` *components.*

Design choice	Based on
If freebie exists, use tieBreaking(); else, select uniformly at random	[Selman et al., 1994]
Variable selection from `Novelty`	[McAllester et al., 1997]
Variable selection from `Novelty`$^+$	[Hoos, 2002]
Variable selected uniformly at random	[Hoos, 2002]
Variable with best `VW1` score	[Prestwich, 2005]
Variable with best `VW2` score	[Prestwich, 2005]
Variable with best score	[Pham and Gretton, 2007; Li and Huang, 2005]
Variable selection from `Novelty++`	[Li and Huang, 2005]
Variable selection from `Novelty++`$'$	[Li et al., 2007b]
Least recently flipped variable	[Li et al., 2007a]
Variable selection from `Novelty+p`	[Li et al., 2007b]

Table 2: *Design choices for* selectFromPromisingList().

Block B_2 instantiates solvers from the G^{2}WSAT architecture, which use a "promising list" to keep track of a set variables considered for being flipped. Two strategies for selecting the variable ultimately to be flipped from the promising list have been proposed in the literature: choosing the variable with the highest score [Li and Huang, 2005] and choosing the least recently flipped variable [Li *et al.*, 2007a]. We added nine novel options based on heuristics from other solvers that, to our knowledge, have never before been applied to G^{2}WSAT-variant algorithms. (For example, we believe that variable selection mechanisms from `Novelty` variants have only been applied to unsatisfied clauses, not to promising lists.) The 11 possible values for *selectFromPromisingList()* and a reference for each heuristic are given in Table 2.

If the promising list is empty, block B_2 behaves exactly like block B_3, which instantiates WalkSAT-based algorithms. First a clause c is selected using the *selectClause()* component already described in B_1. Then a variable to be flipped is selected from c using the *selectHeuristic()* component. This component can be configured to implement 13 different variable selection strategies, including those from `WalkSAT/SKC` [Selman *et al.*, 1994], `VW1` and `VW2` [Prestwich, 2005], and various `Novelty` variants. The selection strategy in `gNovelty`$^+$ also includes an optional "flat move" mechanism; we extended this optional mechanism to the selection strategies for all the other `Novelty` variants.

Block B_4 instantiates dynamic local search algorithms. The *selectSet()* component considers the set of variables that occur in any unsatisfied clause. It associates with each such variable v a score, which depends on the *clause weights* of each clause that changes satisfiability status when v is flipped. These clause weights reflect the perceived importance of satisfying each clause; for example, weights might increase the longer a clause has been unsatisfied, and decrease afterwards [Hutter *et al.*, 2002; Thornton *et al.*, 2004]. After scoring the variables, *selectSet()* returns all variables with maximal score. Our implementation of this component incorporates three different scoring functions, including those due to McAllester *et al.* [1997], Selman *et al.* [1994], and a novel, greedier variant that only considers the number of previously unsatisfied clauses that are satisfied by a variable flip. The *tieBreaking()* component selects a variable from the maximum-scoring set according to the same strategies used by the *diversificationStrategy()* component.

Block B_5 updates underlying data structures after a variable has been flipped. Performing these updates in an efficient manner is a core issue in optimising the performance of SLS algorithms; in combination with the fact that the `SATenstein-LS` framework supports the combination of mechanisms from many different SLS algorithms, each depending on different data structures, this rendered the implementation of the *update()* technically quite challenging.

We validated the correctness of `SATenstein-LS` by carefully examining its behavior when configured to instantiate ten prominent algorithms (see [KhudaBukhsh, 2009] for details), comparing to independent reference implementations. We also compared the different implementations in terms of empirical performance, finding that in some cases they were virtually identical (for `SAPS`, `RSAPS`, and `adaptNovelty`$^+$) while in others `SATenstein-LS` was slower (the worst was 2.17 times slower, for `gNovelty`$^+$). The primary reason for observed performance differences between `SATenstein-LS` and the reference implementations was the lack of special-purpose data structure optimisations in the *update()* component.

4 Experimental Setup

We considered six sets of well-known benchmark instances for SAT, listed in Table 3. Because SLS algorithms are unable to prove unsatisfiability, we constructed our benchmark sets to include only satisfiable instances. The HGEN and FAC distributions include only satisfiable instances; for each of these distributions, we generated 2000 instances and divided these randomly into a training and test set containing 1000 instances each. For the remaining distributions, we filtered out unsatisfiable instances using complete solvers. For QCP, we generated 23 000 instances around the solubility phase transition, using the parameters given by Gomes and Selman [1997]. We solved these using a complete solver, and then randomly chose 2000 satisfiable instances. These we divided randomly into training and test sets of 1000 instances each. For SW-GCP, we generated 20 000 instances [Gent *et al.*, 1999] and used a complete solver to randomly sample 2000 satisfiable instances, which we again divided randomly into training and test sets of 1000 instances each. For R3SAT, we generated a set of 1000 instances with 600 variables and a clauses-to-variables ratio of 4.26. We identified 521 satisfiable instances using

Distribution	Description
QCP	SAT-encoded quasi-group completion [Gomes and Selman, 1997]
SW-GCP	SAT-encoded small-world graph-colouring problems [Gent et al., 1999]
R3SAT	uniform-random 3-SAT instances [Simon, 2002]
HGEN	random instances generated by generator HGEN2 [Hirsch, 2002]
FAC	SAT-encoded factoring problems [Uchida and Watanabe, 1999]
CBMC(SE)	SAT-encoded bounded model checking problems [Clarke et al., 2004], preprocessed by SatELite [Eén and Biere, 2005]

Table 3: *Our six benchmark distributions.*

Algorithm	Short Name	Reason for Inclusion
Ranov	Ranov	gold 2005 SAT Competition (random)
G^2WSAT	G2	silver 2005 SAT Competition (random)
VW	VW	bronze 2005 SAT Competition (random)
$gNovelty^+$	GNOV	gold 2007 SAT Competition (random)
$adaptG^2WSAT_0$	AG20	silver 2007 SAT Competition (random)
$adaptG^2WSAT_+$	AG2+	bronze 2007 SAT Competition (random)
$adaptNovelty^+$	ANOV	gold 2004 SAT Competition (random)
$adaptG^2WSAT_p$	AG2p	performance comparable to $adaptG^2WSAT_+$, G^2WSAT, and Ranov; see [Li et al., 2007b]
SAPS	SAPS	prominent dynamic local search algorithm
RSAPS	RSAPS	prominent dynamic local search algorithm
PAWS	PAWS	prominent dynamic local search algorithm

Table 4: *Our eleven challenger algorithms.*

complete solvers;[3] we randomly chose 250 of these instances to form a training set and 250 to form a test set. Finally, we used the CBMC generator to create 611 SAT-encoded software verification instances based on a binary search algorithm with different array sizes and loop-unwinding values. Of the instances thus obtained, we filtered out seven unsatisfiable instances and confirmed all the others as satisfiable. We randomly divided the remaining 604 instances into a training set of 303 instances and a test set of 301 instances.

In order to perform automatic algorithm configuration on the SATenstein-LS framework, we first had to quantify performance using an objective function; we chose to focus on mean runtime. However, efficient algorithm configurators terminate (or *capped*) some runs before they complete, making the mean ill-defined. Thus, following Hutter *et al.* [2007a], we define the penalised average runtime (PAR) of a set of N runs with a k-second cutoff as the mean runtime over all N runs, with capped runs counted as $10 \cdot k$ seconds.

As our algorithm configurator, we chose the FocusedILS procedure from the ParamILS framework, version 2.2 [Hutter *et al.*, 2008; 2007b], because to the best of our knowledge it is the only method able to operate effectively on extremely large, discrete parameter spaces. We set k to five seconds, and allotted two days to each run of FocusedILS. Since FocusedILS is a randomised procedure, its performance can vary significantly over multiple independent runs, in particular depending on the order it chooses for instances in the training set. We ran it 10 times on each training set, using different, randomly determined instance orderings for each run. From the 10 parameter configurations obtained from FocusedILS for each instance distribution D, we selected the parameter configuration with the best penalised average runtime on the training set, and we refer to the corresponding instantiation of SATenstein-LS as SATenstein-LS[D].

For every distribution D, we compared the performance of SATenstein-LS[D] against that of 11 high-performance SLS-based SAT solvers on the test set. We included every SLS algorithm that won a medal in any category of the SAT competition in the last five years [SAT Competition, 2009], and also included several other prominent, high-performing algorithms. These are listed in Table 4.

SATenstein-LS can be instantiated to emulate all 11 of these challenger algorithms, except that it does not support preprocessing components used by Ranov, G2 and AG20. All of our experimental comparisons are based on the original algorithm implementations, as submitted to the respective SAT competitions, except for PAWS, for which the UBCSAT implementation is almost identical to the original solver in terms of runtime. All comparisons are based on running each solver 10 times with a cutoff time of 600 seconds per run.

We carried out our experiments on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1; runtimes for all algorithms (including FocusedILS) were measured as CPU time on these reference machines.

5 Results

On every one of our benchmark distributions, we were able to find a SATenstein-LS configuration that outperformed all 11 challengers. Our results are summarised in Table 5.

In terms of penalised average runtime (the objective function that we explicitly optimised with ParamILS, albeit computed according to the 600-second cutoff we used on our test runs, rather than the 5-second cutoff we used in training), SATenstein-LS achieved markedly better performance than the challengers. For QCP, HGEN, and CBMC(SE), SATenstein-LS achieved PAR orders of magnitude better than the respective best challenger. For SW-GCP, R3SAT, and FAC, SATenstein-LS's performance advantage was still substantial, but less dramatic. It is not overly surprising that there was relatively little room for improvement on R3SAT: random instances of this type have been used prominently over the last 17 years for evaluating SLS solvers for SAT, both during development and in competitions. Conversely, CBMC(SE) is a new benchmark from a broad class of instances on which SLS algorithms are generally considered weak performers compared to state-of-the-art DPLL-based SAT solvers. We were surprised to see the amount of improvement we could achieve on QCP, a relatively widely-used benchmark, and HGEN, a hard random instance distribution quite similar to R3SAT. In both cases our results indicate that recent developments in SLS solvers for SAT have not yielded significant improvements over older solvers (ANOV and SAPS/RSAPS, respectively); we thus attribute SATenstein-LS's strong performance here to the fact that SLS-solver development over the past seven years has not taken these types of benchmarks into account, nor yielded across-the-board improvements.

We also investigated SATenstein-LS's performance according to measures other than PAR. Median-of-median runtime (the median across instances of the median across ten runs on a given instance) offers a picture of performance that disregards capped runs as long as most instances are solved in most runs. Although SATenstein-LS was not configured

[3] We identified satisfiable instances by running March_pl, each of our 11 challenger algorithms and Kcnfs-04 with cutoffs of 3600, 600 and 36 000 seconds respectively. Our solvers were able to rule out only 56 of the remaining 479 instances as unsatisfiable.

Distribution	SATenstein-LS[D]	GNOV [Pham and Gretton, 2007]	AG20 [Li et al., 2007b]	AG2+ [Li et al., 2007a]	RANOV [Pham and Anbulagan, 2007]	G2 [Li and Huang, 2005]	VW [Prestwich, 2005]	ANOV [Hoos, 2002]	AG2p [Li et al., 2007a]	SAPS [Hutter et al., 2002]	RSAPS [Hutter et al., 2002]	PAWS [Thornton et al., 2004]
QCP	**0.13**	422.33	1051	1080.29	76.22	2952.56	1025.9	28.3	1104.42	1256.2	1265.37	1144.2
	0.01	0.03	0.03	0.03	0.1	361.21	0.25	**0.01**	0.02	0.03	0.04	0.02
	100%	92.7%	80.5%	80.3%	98.7%	50.6%	82.2%	99.6%	79.4%	79.2%	78.4%	80.8%
SW-GCP	**0.03**	0.24	0.62	0.45	0.15	4103.27	159.67	0.06	0.45	3872.08	5646.39	4568.59
	0.03	0.09	0.12	0.08	0.12	N/A	40.96	**0.03**	0.07	N/A	N/A	N/A
	100%	**100%**	**100%**	**100%**	**100%**	30.5%	98.9%	**100%**	**100%**	33.2%	5%	22.1%
R3SAT	**1.51**	10.93	2.37	3.3	14.14	5.32	9.53	12.4	2.38	22.81	14.81	2.4
	0.14	0.14	0.14	0.16	0.32	0.13	0.75	0.21	0.13	1.80	2.13	**0.12**
	100%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
HGEN	**0.03**	52.87	139.33	138.84	156.96	110.02	177.9	147.53	107.4	48.31	38.51	73.27
	0.02	0.73	0.57	0.61	0.95	0.61	3.23	0.76	0.49	3.00	2.44	0.96
	100%	99.4%	98%	97.8%	97.7%	98.4%	97.5%	97.6%	98.4%	99.5%	99.7%	99.2%
FAC	**12.22**	5912.99	3607.4	1456.4	943.26	5944.6	3411.93	3258.66	1989.91	17.39	19.39	26.51
	8.03	N/A	N/A	237.50	155.77	N/A	N/A	N/A	315.48	11.60	12.88	12.62
	100%	0%	30.2%	84.2%	92.1%	0%	31.7%	37.2%	72.5%	**100%**	**100%**	99.9%
CBME(SE)	**5.59**	2238.7	2170.67	2161.59	1231.01	2150.31	385.73	2081.94	2282.37	613.15	794.93	1717.79
	0.02	0.75	0.67	0.77	0.66	0.68	0.27	5.81	3.18	0.04	0.03	19.99
	100%	61.13%	61.13%	61.13%	79.73%	64.12%	92.69%	61.79%	61.13%	90.03%	85.38%	68.77%

Table 5: *Performance summary of* `SATenstein-LS` *and the 11 challengers. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm j on distribution i as $a/b/c$, where a (top) is the the penalised average runtime; b (middle) is the median of the median runtimes over all instances (not defined if fewer than half of the median runs failed to terminate); c (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each row are indicated in bold, and additionally the best-scoring challenger(s) are indicated with an underline.*

to optimise this performance measure, using it as the basis for evaluation produced results essentially analogous to those for PAR (with R3SAT being the only benchmark on which several challengers scored slightly better). Finally, we measured the percentage of instances on which the median runtime was below the cutoff used for capping runs. According to this criterion, our `SATenstein-LS` solvers were successful on 100% of the instances in every one of our benchmark sets; on the other hand, only three of the 11 challengers solved more than 40% of the instances in every benchmark set.

The relative performance of the challengers varied significantly across different distributions. For example, the three dynamic local search algorithms (`SAPS`, `RSAPS` and `PAWS`) performed substantially better than any of the other challengers on the factoring problems (FAC); however, their relative performance on small-world graph colouring problems (SW-GCP) was weak. Similarly, GNOV (the winner of the random SAT category of the 2007 SAT Competition) performed very badly on our software verification (CBMC(SE)) and factoring (FAC) benchmarks, but solved SW-GCP and HGEN instances quite effectively. (Interestingly, on both types of random SAT instances we considered, `GNOV` did not reach the performance of some of the older SLS solvers, in particular, `PAWS` and `RSAPS`.) This suggests—not too surprisingly—that different types of SAT benchmarks are most efficiently solved using rather different solvers. The fact that our `SATenstein-LS` solvers performed better than any of the challengers for every one of our distributions clearly demonstrates that the design space spanned by the features of a large set of high-performance algorithms contains better solvers than those previously known, and that automatic exploration of this vast combinatorial space can effectively find such improved designs. Of course, solvers developed for individual benchmarks could in principle be combined using an instance-based algorithm selection technique (such as SATzilla), yielding even stronger performance.

The performance metrics we have discussed so far only describe aggregate performance over the entire test set. One might wonder whether our `SATenstein-LS` solvers performed poorly on many test instances, but compensated for this weakness on other instances. Table 6 shows that this was typically not the case, comparing each `SATenstein-LS` solver's performance to each challenger on a per-instance basis. The `SATenstein-LS` solvers outperformed the best challengers on a large majority of the test instances on all benchmark sets except for R3SAT. On that distribution, `PAWS` was the challenger that outperformed our `SATenstein-LS` solver most frequently (62% of the time). Figure 1 shows that the performance of these two algorithms was quite highly correlated, and that, while instances that are rather easy for both algorithms tend to be solved faster by `PAWS`, `SATenstein-LS[R3SAT]` performs better on harder instances. We observed the same qualitative trend for other challengers on R3SAT. This phenomenon was even more pronounced for QCP and SW-GCP, but does not appear to have occurred for CBMC(SE), HGEN and FAC, where the correlation in solver performance was also considerably weaker.

Our penalised average runtime measure is sensitive to the choice of test cutoff time, which sets the penalty. In particular, an algorithm could score well when the cutoff time is large, but could achieve much weaker PAR scores for smaller cutoffs. Reassuringly, we found that this problem did not arise for the `SATenstein-LS` solvers considered in our study. Specifically, the `SATenstein-LS` solvers outperformed their challengers in terms of PAR on all of our distributions regardless of the cutoff times used, and likewise the qualitative results from Table 5 were unaffected by cutoff time. Figure 2 gives an example of the data we examined to draw these conclusions, considering instance distribution FAC. We see that while the choice of cutoff time affected the raw PAR scores, for no cutoff would a challenger have outscored the `SATenstein-LS` solver. PAR first increased and then decreased with cutoff, because increasing the cutoff increases the penalty for unsolved instances (and thus also PAR), but decreases the score for solved instances (because the penalty for previously capped runs is replaced by the true runtime). Once all instances in a given test set are solved, PAR remains constant as cutoff increases.

To better understand the `SATenstein-LS[D]` solvers, we compared them with the `SATenstein-LS` instantiations cor-

Distribution	GNOV	AG20	AG2+	RANOV	G2	VW	ANOV	AG2p	SAPS	RSAPS	PAWS
QCP	99	87	87	100	91	92	83	81	82	79	79
SW-GCP	100	98	97	100	100	100	63	93	100	100	100
R3SAT	55	55	64	96	44	100	72	55	99	99	38
HGEN	100	100	96	99	100	100	99	100	100	100	100
FAC	100	100	100	100	100	100	100	100	73	78	80
CBME(SE)	100	100	100	100	100	100	100	100	80	75	100

Table 6: *Percentage of instances on which* `SATenstein-LS` *achieved better median runtime than each of the challengers. Medians were taken over 10 runs on each instance with a cutoff time of 600 CPU seconds/run. When both* `SATenstein-LS` *and a challenger solved a given instance with indistinguishable median runtimes, we counted that instance as 0.5 for* `SATenstein-LS` *and 0.5 for the challenger.*

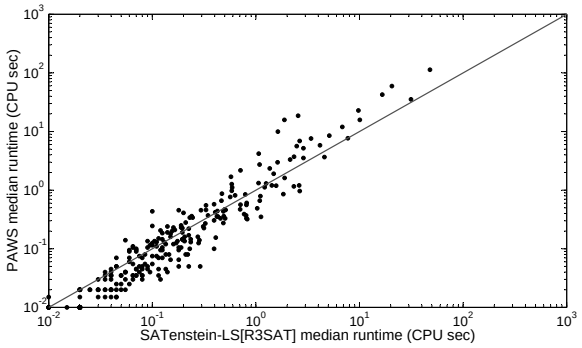

Figure 1: *Scatter plot of median runtimes of* `SATenstein-LS[R3SAT]` vs `PAWS` *on the test set for R3SAT. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run.*

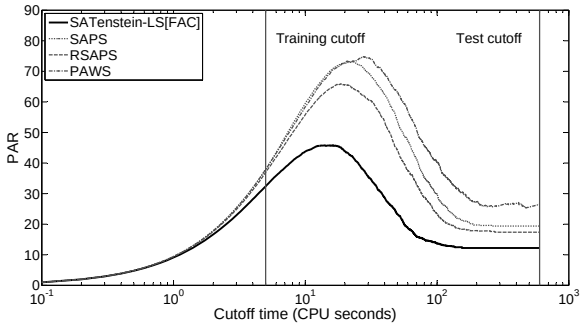

Figure 2: *Penalised average runtime for our* `SATenstein-LS` *solver and the top three challengers for FAC, as a function of cutoff, based on 10 runs on each test-set instance.*

responding to each challenger.[4] `SATenstein-LS[QCP]` uses blocks 1, 3, and 5 and an adaptive parameter update mechanism similar to that in `AG2+`. In block 3, *selectHeuristic* is based on `R-Novelty+`, and in block 1, *diversification* flips the variable with minimum variable weight as in `VW1` [Prestwich, 2005]. `SATenstein-LS[SW-GCP]` uses blocks 1 and 3, resembling `Novelty++` as used within `G2`. `SATenstein-LS[R3SAT]` uses blocks 4 and 5; it is closest to `RSAPS`, but uses a different tie-breaking mechanism. `SATenstein-LS[HGEN]` uses blocks 1, 3, and 5. In block 1 it is similar to `G2` and in block 3 is closest to `VW`. In block 5 it uses the same adaptive parameter update mechanism as `ANOV`. `SATenstein-LS[FAC]` uses blocks 4 and 5; its instantiation closely resembles that of `SAPS`, but differs in the way the score of a variable is computed. Finally, `SATenstein-LS[CBMC(SE)]` uses blocks 1, 4, and 5, drawing on elements of `GNOV` and `RSAPS`.

6 Conclusion

In this work we have advocated a new approach for constructing heuristic algorithms that is based on (1) a framework that can flexibly combine components drawn from existing high-performance solvers, and (2) a generic algorithm configuration tool for finding instantiations that perform well on given sets of instances. We applied this approach to stochastic local search algorithms for SAT and demonstrated empirically that it was able to produce new SLS-based solvers that represent considerable improvements over the previous state of the art in solving several widely studied distributions of SAT instances. Source code and documentation for our `SATenstein-LS` framework are available online at `http://www.cs.ubc.ca/labs/beta/Projects/SATenstein`.

Unlike the tragic figure of Dr. Frankenstein from Mary Shelley's novel, whose monstrous creature haunted him enough to quench forever his ambitions to create a 'perfect' human, we feel encouraged to unleash not only our new solvers, but also the full power of our automated solver-building process onto other classes of SAT benchmarks. Like Dr. Frankenstein we find our creations somewhat monstrous, recognising that our SATenstein solvers do not always embody the most elegant designs. Thus, we are currently working towards more detailed understanding of how our SATenstein solvers relate to previously-known SAT algorithms. Other interesting lines of future work include the extension of our solver framework to capture combinations of components from the `G2WSAT` architecture and dynamic local search algorithms, as well as preprocessors (as used, for example, in `Ranov`); the combination of SATenstein solvers trained on various types of SAT instances by means of an algorithm selection approach (see, e.g., [Xu *et al.*, 2008]); and the investigation of algorithm configuration procedures other than ParamILS in the context of our approach. Finally, encouraged by the results achieved on SLS-algorithms for SAT, we believe that the general approach behind SATenstein is equally applicable to non-SLS-based solvers and to other combinatorial problems. To paraphrase the words of Mary Shelley's Victor Frankenstein, we hope that ultimately many effective solvers will owe their being to this line of work.

[4] Due to space constraints, we describe our conclusions only at a high level. More detailed information about the `SATenstein-LS` configurations is given in [KhudaBukhsh, 2009].

References

[Carchrae and Beck, 2005] T. Carchrae and J.C. Beck. Applying machine learning to low knowledge control of optimization algo-

rithms. *Computational Intelligence*, 21(4):373–387, 2005.

[Chiarandini *et al.*, 2008] M. Chiarandini, C. Fawcett, and H.H. Hoos. A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In *Proc. PATAT*, 2008.

[Clarke *et al.*, 2004] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, pages 168–176, 2004.

[Eén and Biere, 2005] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT*, pages 61–75, 2005.

[Fukunaga, 2002] A.S. Fukunaga. Automated discovery of composite SAT variable-selection heuristics. In *Proc. AAAI*, pages 641–648, 2002.

[Fukunaga, 2004] A.S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *Proc. GECCO*, pages 483–494, 2004.

[Gagliolo and Schmidhuber, 2006] M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006.

[Gaspero and Schaerf, 2007] L.D. Gaspero and A. Schaerf. Easysyn++: A tool for automatic synthesis of stochastic local search algorithms. In *Proc. SLS*, pages 177–181, 2007.

[Gent *et al.*, 1999] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proc. AAAI*, pages 654–660, 1999.

[Gomes and Selman, 1997] C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. AAAI*, pages 221–226, 1997.

[Gomes and Selman, 2001] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[Gratch and Dejong, 1992] Jonathan Gratch and Gerald Dejong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proc. AAAI*, pages 235–240, 1992.

[Guerri and Milano, 2004] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proc. ECAI*, pages 475–479, 2004.

[Hirsch, 2002] E. A. Hirsch. Random generator hgen2 of satisfiable formulas in 3-CNF. `http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2-1.01.tar.gz`, 2002.

[Hoos, 2002] H.H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proc. AAAI*, pages 655–660, 2002.

[Hoos, 2008] Holger H. Hoos. Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Department of Computer Science, 2008.

[Hutter *et al.*, 2002] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. CP*, pages 233–248, 2002.

[Hutter *et al.*, 2007a] F. Hutter, D. Babić, H.H. Hoos, and A.J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proc. FMCAD*, pages 27–34, 2007.

[Hutter *et al.*, 2007b] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. AAAI*, pages 1152–1157, 2007.

[Hutter *et al.*, 2008] F. Hutter, H. H. Hoos, T. Stützle, and K. Leyton-Brown. ParamILS version 2.2. `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS`, 2008.

[KhudaBukhsh, 2009] A.R. KhudaBukhsh. SATenstein: Automatically building local search SAT solvers from components. `http://www.cs.ubc.ca/labs/beta/Projects/SATenstein/ashique_masters_thesis.pdf`, 2009.

[Li and Huang, 2005] C.M. Li and W. Huang. Diversification and determinism in local search for satisfiability. In *Proc. SAT*, pages 158–172, 2005.

[Li *et al.*, 2007a] C. M. Li, W. X. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *Proc. SAT*, pages 121–133, 2007.

[Li *et al.*, 2007b] C.M. Li, W. Wei, and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2007, 2007.

[McAllester *et al.*, 1997] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. AAAI*, pages 321–326, 1997.

[Minton, 1993] S. Minton. An analytic learning system for specializing heuristics. In *Proc. IJCAI*, pages 922–929, 1993.

[Monette *et al.*, 2009] J. Monette, Y. Deville, and P.V. Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. In *Proc. INFORMS (to appear)*, 2009.

[Oltean, 2005] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[Pham and Anbulagan, 2007] D.N. Pham and Anbulagan. Resolution enhanced SLS solver: R+AdaptNovelty+. Solver description, SAT competition 2007, 2007.

[Pham and Gretton, 2007] D. N. Pham and C. Gretton. gNovelty+. Solver description, SAT competition 2007, 2007.

[Prestwich, 2005] S. Prestwich. Random walk with continuously smoothed variable weights. In *Proc. SAT*, pages 203–215, 2005.

[SAT Competition, 2009] SAT Competition. `http://www.satcompetition.org`, 2009.

[Selman *et al.*, 1994] B. Selman, H. Kautz, , and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI*, pages 337–343, 1994.

[Simon, 2002] L. Simon. SAT competition random 3CNF generator. `http://www.satcompetition.org/2003/TOOLBOX/genAlea.c`, 2002.

[Thornton *et al.*, 2004] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira. Additive versus multiplicative clause weighting for SAT. In *Proc. AAAI*, pages 191–196, 2004.

[Tompkins and Hoos, 2004] D.A.D. Tompkins and H.H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Proc. SAT*, pages 37–46, 2004.

[Uchida and Watanabe, 1999] T. Uchida and O. Watanabe. Hard SAT instance generation based on the factorization problem. `http://www.is.titech.ac.jp/~watanabe/gensat/a2/GenAll.tar.gz`, 1999.

[Westfold and Smith, 2001] S.J. Westfold and D.R. Smith. Synthesis of efficient constraint-satisfaction programs. *The Knowledge Engineering Review*, 16(1), 2001.

[Xu *et al.*, 2008] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[Xu *et al.*, 2009] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla2009: An automatic algorithm portfolio for SAT. Solver description, 2009 SAT Competition, 2009.