Artificial Intelligence 232 (2016) 20–42

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

# SATenstein: Automatically building local search SAT solvers from components

Ashiqur R. KhudaBukhsh [a,*,1], Lin Xu [b,1], Holger H. Hoos [b], Kevin Leyton-Brown [b]

[a] Department of Computer Science, Carnegie Mellon University, United States
[b] Department of Computer Science, University of British Columbia, Canada

ARTICLE INFO

*Article history:*
Received 18 December 2013
Received in revised form 5 November 2015
Accepted 7 November 2015
Available online 2 December 2015

*Keywords:*
SAT
Stochastic local search
Automatic algorithm configuration

ABSTRACT

Designing high-performance solvers for computationally hard problems is a difficult and often time-consuming task. Although such design problems are traditionally solved by the application of human expertise, we argue instead for the use of automatic methods. In this work, we consider the design of stochastic local search (SLS) solvers for the propositional satisfiability problem (SAT). We first introduce a generalized, highly parameterized solver framework, dubbed SATenstein, that includes components drawn from or inspired by existing high-performance SLS algorithms for SAT. The parameters of SATenstein determine which components are selected and how these components behave; they allow SATenstein to instantiate many high-performance solvers previously proposed in the literature, along with trillions of novel solver strategies. We used an automated algorithm configuration procedure to find instantiations of SATenstein that perform well on several well-known, challenging distributions of SAT instances. Our experiments show that SATenstein solvers achieved dramatic performance improvements as compared to the previous state of the art in SLS algorithms; for many benchmark distributions, our new solvers also significantly outperformed all automatically tuned variants of previous state-of-the-art algorithms.


## 1. Introduction

In Mary Shelley's classic novel *Frankenstein; or, The Modern Prometheus*, a brilliant scientist, Victor Frankenstein, set out to create a perfect human being by combining scavenged human body parts. We pursue a similar idea: scavenging components from existing high-performance algorithms for a given problem and combining them to build new high-performance algorithms. Our idea is inspired by the fact that many new solvers are created by augmenting an existing algorithm with a mechanism found in a different algorithm (see, e.g., [33,53]) or by combining components of different algorithms (see, e.g., [62]). Unlike Victor Frankenstein's creation, we propose to use an automated construction process that enables us to optimize performance with minimal human effort.

* Corresponding author.
*E-mail addresses:* akhudabu@cs.cmu.edu (A.R. KhudaBukhsh), xulin730@cs.ubc.ca (L. Xu), hoos@cs.ubc.ca (H.H. Hoos), kevinlb@cs.ubc.ca (K. Leyton-Brown).
[1] We thank Paul Cernek for helping to run experiments and contributing to the SATenstein codebase. Ashiqur R. KhudaBukhsh and Lin Xu contributed equally to this work.

http://dx.doi.org/10.1016/j.artint.2015.11.002
0004-3702/

Traditionally, high-performance heuristic algorithms are designed through an iterative, manual process in which most design choices are fixed at development time, usually based on preliminary experimentation, leaving only a small number of parameters exposed to the user. In contrast, we propose a new approach to heuristic algorithm design in which the designer fixes as few design choices as possible, instead exposing all promising design choices as parameters. This approach removes from the algorithm designer the burden of making early design decisions without knowing how different algorithm components will interact on problem distributions of interest. Instead, it encourages the designer to consider many alternative designs, drawing from known solvers as well as novel mechanisms. Of course, such flexible, highly parameterized algorithms must be instantiated appropriately to achieve good performance on a given instance set. With the availability of advanced automated parameter configurators and cheap computational resources, finding a good parameter configuration from a huge parameter space becomes practical (see, e.g., [40,9,13]). Of course, we are not the first to propose building algorithms by using automated methods to search a large design space. Rather, 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 [34]). There are many examples of work along these lines [25,56,12,79,20,18,60,77,57,21].

Although our general approach is not specifically tailored to a particular domain, in this work we address the challenge of constructing stochastic local search (SLS) algorithms for the propositional satisfiability problem (SAT): an NP-complete problem of great interest to the scientific and industrial communities alike. SLS-based solvers are important because they have exhibited consistently dominant performance for several families of SAT instances; they also play an important role in state-of-the-art portfolio-based automated algorithm selection methods for SAT [79]. Substantial research and engineering effort has been expended in building SLS algorithms for SAT since the late 1980s (see, e.g., [67,33,62]), with new solvers being introduced every year.

We leveraged this rich literature (discussed in detail later) to design `SATenstein-LS`. This algorithm draws mechanisms from 25 high-performance SLS SAT solvers and also incorporates many novel strategies. The resulting design space contains a total of $2.01 \times 10^{14}$ candidate solvers, and includes most existing, state-of-the-art SLS SAT solvers that have been proposed in the literature. We demonstrate experimentally that our new, automatically-constructed solvers dramatically outperform the best SLS-based SAT solvers currently available (with the default parameter configurations manually tuned by their authors) on six well-known SAT instance distributions, ranging from hard random 3-SAT instances to SAT-encoded factoring and software verification problems. In most cases, our new solvers also significantly outperform the best SLS-based SAT solvers even when we automatically tune the originally exposed parameters of every one of these incumbent solvers. Because SLS-based SAT solvers are the best known methods for solving most of our benchmark distributions, our new solvers represent a substantial advance in the state of the art for solving the respective sub-classes of SAT. On one of the two instance families for which this is not the case—SAT-encoded number factoring problems—our new solvers narrow the gap between the performance of the best SLS algorithms and the best DPLL-based solvers.

This paper[2] is organized as follows. Section 2 discusses related work; we describe the design and implementation of `SATenstein-LS` in Section 3. We then describe the setup we used for empirically evaluating `SATenstein-LS` (Section 4) and present the results from our experiments (Section 5). Section 6 presents some general conclusions and an outlook on future work.

## 2. Related work

The propositional satisfiability problem (SAT) asks, for a given propositional formula $F$, whether there exists a complete assignment of truth values to the variables of $F$ under which $F$ evaluates to true (see, e.g., [8]). $F$ is called satisfiable if there exists at least one such assignment and unsatisfiable otherwise. A SAT instance is usually represented in conjunctive normal form (CNF), i.e., as a conjunction of disjunctions of literals, where each literal is a propositional variable or the negation of variables. Each disjunction of literals is called a clause. In this case, the goal for a SAT solver is to find a variable assignment that satisfies all clauses of a given CNF formula or to prove that no such assignment exists.

### 2.1. Local-search SAT solvers

Over the past decades, considerable research and engineering effort has been invested into designing and optimizing algorithms for SAT. State-of-the-art SAT solvers include tree-search algorithms (see, e.g., [69,28,7,16,2,30]), local search algorithms (see, e.g., [42,61,50,64,62,11]) and resolution-based preprocessors (see, e.g., [70,15,3]). Every year, competitions are held, in which new state-of-the-art solvers emerge. The trend of continuing performance improvement in SAT competitions suggests that there is room for even further enhancements of current solver technology.

[2] An early version of the work described in this article was published at IJCAI [45]. This article substantially extends that work in five key ways. (1) It describes `SATenstein-LS`'s architecture in considerably more detail, and (2) presents all-new experiments based on longer configuration runs, albeit on the same distributions. (3) Our comparison with tuned versions of challengers (Section 5.2) is entirely new, as is (4) our comparison with complete solvers (Section 5.3). (5) We extended `SATenstein-LS` with a local search strategy found in the recent high-performance SAT solver, `Sattime`, and compared the performance of the augmented `SATenstein-LS` with three recent SLS-based SAT solvers, including `Sattime`.

Stochastic local search (SLS) algorithms represent the state of the art in solving certain types of SAT instances and have been the subject of an intense and sustained interest since the early 1990s (see, e.g., [67,76]). A typical SLS algorithm for SAT consists of an initialization phase and a local search phase. In the initialization phase, all variables are assigned truth values. At each step of the local search phase, the truth value of a single, heuristically chosen variable is changed. Exceptions include SLS solvers based on evolutionary algorithms (e.g., [47]) that maintain a population of candidate solutions and use recombination techniques. The search process is terminated when either a satisfying assignment is found or a given bound on the runtime or run length is reached or exceeded. Almost all SLS algorithms for SAT are incomplete, i.e., they cannot establish the unsatisfiability of a given formula.

The vast majority of existing SLS-based SAT solvers can be grouped into four broad categories: GSAT-based [67], WalkSAT-based [66], dynamic local search algorithms [42,71], and $G^2$WSAT variants [50]. Almost all of the recent high-performance SLS SAT solvers are based on WalkSAT, dynamic local search, or $G^2$WSAT. `SATenstein-LS` thus draws deeply on these families of solvers, which we discuss in more detail in Section 3. GSAT-based algorithms are mostly of historical importance, but ideas that originated in GSAT remain important in more modern solvers. Partly for that reason, we briefly describe its architecture here.

At each step, `GSAT` evaluates each variable using a scoring function, then flips the variable with the highest score. The score of a variable is determined from two quantities, *MakeCount* and *BreakCount*. The *MakeCount* of a variable with respect to an assignment is the number of previously unsatisfied clauses that will be satisfied if the variable is flipped. Similarly, the *BreakCount* of a variable with respect to an assignment is the number of previously satisfied clauses that will be unsatisfied if the variable is flipped. The scoring function of `GSAT` is *MakeCount* – *BreakCount*.

### 2.2. UBCSAT

UBCSAT [74] is an SLS solver implementation and experimentation environment for SAT. It provides implementations of many existing high-performance SLS algorithms from the literature. These implementations generally match or exceed the efficiency of the respective implementations made available by the original authors. UBCSAT implementations have therefore been widely used as reference implementations for many well-known local search algorithms (see, e.g., [64,46]). In addition, UBCSAT also provides a rich interface that includes numerous statistical and reporting features facilitating empirical analysis of SLS algorithms.

Many existing SLS algorithms for SAT share common components and data structures. The general design of UBCSAT allows for the reuse and extension of such common components and mechanisms. This made UBCSAT an ideal environment for the implementation of `SATenstein-LS` (described below). However, UBCSAT and `SATenstein-LS` are quite different at a conceptual level. UBCSAT implements many well-known solvers in a stand-alone fashion; it does not provide for the creation of new solvers by combining existing solver components.

### 2.3. Automated algorithm design

There is a large body of literature in AI and related areas that deals with automated methods for building heuristic algorithms. This includes work on automatic algorithm configuration (see, e.g., [25,56]), algorithm selection (see, e.g., [48,12,79,78]), parallel portfolios (see, e.g., [24,20]), and, to some extent, genetic programming (see, e.g., [18,19,60]), hyper-heuristics (see, e.g., [54]), autonomous search (see, e,g., [27]), and algorithm synthesis (see, e.g., [77,57,21]). In what follows, we restrict our discussion to research efforts that are related particularly closely to our approach.

#### 2.3.1. Automated construction of algorithms

Here we consider three closely related lines of previous work in more detail, contrasting them with our own. First, Minton [56] used meta-level theories to produce distribution-specific versions of generic heuristics, and then found the most useful combination of these heuristics by evaluating their performance on a small set of test instances. He focused on producing distribution-specific versions of candidate heuristics and only considered at most 100 possible heuristics. The performance of the resulting algorithms was comparable to that of algorithms designed by a skilled programmer, but not an expert. In contrast, our work lays out a generalized, highly parameterized framework that can be instantiated to yield many trillions of distinct candidate solvers. We achieved performance exceeding the current state of the art on most of the instance distributions we considered.

Second, Gratch and Dejong [25] presented a system that starts with a STRIPS-like planner and augments it by incrementally adding search control rules. In contrast, `SATenstein` does not augment an existing solver; rather, our goal is to design a method for automatically building new solvers by combining components from as many existing solvers as possible.

Finally, and most closely related to our work, Fukunaga's [18,19] genetic programming approach has a similar goal to our own: the automated construction of local search heuristics for SAT. Fukunaga considered a potentially unbounded design space, based only on GSAT-based and WalkSAT-based SLS algorithms up to the year 2000. His candidate variable selection mechanisms were evaluated on uniform random 3-SAT and graph coloring instances with at most 250 variables. While Fukunaga's approach could in principle be used to obtain high-performance solvers for specific types of SAT instances, to the best of our knowledge, this potential has never been realized; the best automatically-constructed solvers obtained by

Fukunaga only achieved a performance level similar to that of the best WalkSAT variants available in 2000, based on an evaluation on moderately-sized SAT instances. In contrast, as mentioned above, we consider a huge but bounded combinatorial space of algorithms, based on components taken from two dozen of the best SLS algorithms for SAT currently available, and we employ an off-the-shelf, general-purpose algorithm configuration procedure to search this space. The solvers thus obtained perform substantially better than current state-of-the-art SLS-based SAT solvers on a broad range of challenging SAT instances with up to 4978 variables.

#### 2.3.2. Automated algorithm configuration

Recently, considerable attention has been paid to the problem of automated algorithm configuration. F-Race [9,4,10] uses a non-parametric statistical test to iteratively filter out configurations that are significantly worse than others ("racing"), continuing until a cutoff time is reached and only a small number of good configurations are left. ParamILS [40,39] is a model-free method based on iterated local search and with a "challenge incumbent" procedure somewhat akin to racing. GGA [1] is a model-free method based on a genetic algorithm. Finally, SMAC [37] performs sequential model-based optimization, iterating between fitting models and using them to make choices about which configurations to investigate. The experiments in this paper make use of ParamILS; this method offers the advantages of scalability to large parameter spaces, stability, and previous success in applications (see, e.g., [36,13]). However, in principle, `SATenstein-LS` could be configured using any state-of-the-art algorithm configuration procedure.

#### 2.3.3. Programming by optimization

`SATenstein` advocates designing new solvers by inducing a single parameterized solver from distinct examples in the literature, and then searching this parameter space automatically [45]. This approach is an example of—and indeed was part of the inspiration for—a design philosophy we call Programming by Optimization (PbO) [35]. In general, PbO means seeking and exposing design choices during a development process, and then automatically finding instantiations of these choices that optimize performance in a given use context. `SATenstein-LS` can be seen as an example of PbO in which the algorithm design space has been obtained by unifying a large number of local search schemes for SAT into a tightly integrated, highly parametric algorithm framework. However, the PbO philosophy goes further and is ultimately more general: it emphasizes encouraging developers to identify and expose design choices as parameters, rather than merely recovering parameters from existing, fully implemented examples. Because of its emphasis on changing the software development process, the PbO paradigm is also supported by programming language extensions that allow parameters and design choices to be exposed quickly and transparently (for further details, see www.prog-by-opt.net).

#### 2.3.4. Algorithm selection and SATzilla

To address a potential source of confusion, we contrast `SATenstein` with our similarly-named—but rather different—previous work on `SATzilla`. For a given problem instance or problem distribution, we often have to solve an "algorithm selection problem" [65]: which algorithm(s) should be run in order to minimize some performance objective, such as expected runtime? Different machine learning techniques can be applied to solve this problem (see, e.g., [49,26,12,79,44,43]). `SATzilla` [59,79] instantiates such an approach, using predictive models to select among a portfolio of existing algorithms on a per-instance basis. In contrast, `SATenstein` is an approach for automatically building solvers from components, yielding a huge candidate set of solvers, most of which have never been studied before. Indeed, the two approaches are complementary: methods like `SATzilla` can take advantage of solvers obtained using the automated design approach pursued in this work. In fact, `SATzilla2009_R` and `3S`, which both performed extremely well in the random category of the 2009 and 2012 SAT Competitions (each winning a gold medal for random SAT+UNSAT), both make use of multiple `SATenstein-LS` solvers [80,43]. Indeed, the synergy between `SATenstein` and `SATzilla` runs even deeper: an approach dubbed `Hydra` [78] automatically builds portfolio-based algorithm selectors, based only on a single, highly parameterized algorithm such as `SATenstein`. Experiments show that Hydra outperformed `SATzilla` based on 17 state-of-the-art SLS solvers, even when restricted only to multiple different instantiations of `SATenstein-LS`.

#### 2.3.5. Further related work

Frankenstein was also used as a metaphor for algorithm design in work by Montes de Oca et al. [58], where a Particle Swarm Optimization (PSO) algorithm is created by combining algorithm components drawn from existing high-performance PSO algorithms. These component designs were hand-picked by the algorithm designer; in contrast, we specify a combinatorial design space from which we use an automated algorithm configurator to find a good design for a given problem distribution. Frankenstein's PSO can thus be seen an example of manual algorithm design, whereas our goal is to automate the algorithm-building process.

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., [77,57,21]). In contrast, like other research that falls under the PbO umbrella [35], our work is focused on finding an efficient solver from a huge space of candidate solvers that are all guaranteed to be correct by construction.

**Procedure** SATenstein-LS(...).

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

## 3. SATenstein-LS

SATenstein-LS is a highly parameterized, stochastic local search (SLS) SAT solver that not only draws components from several high-performance SLS-based SAT solvers, but also incorporates several novel mechanisms. SATenstein-LS can be configured to instantiate dozens of well-known SLS solvers, along with many trillions of others that have never been studied before. In this section, we present a high-level outline of SATenstein-LS and explain the functionality of the major building blocks used in our design. We also give a detailed description of the parameters exposed by SATenstein-LS.

### 3.1. Design

As discussed in Section 2.1, most SLS algorithms for SAT fall into one of four broad categories: GSAT-based, WalkSAT-based, dynamic local search, and G$^2$WSAT variants. Since no recent, state-of-the-art SLS solver is GSAT-based, we constructed SATenstein-LS by drawing components from algorithms belonging to the three remaining categories.

As shown in the high-level algorithm outline (Procedure SATenstein-LS), SATenstein-LS is comprised of five major building blocks, $B1$–$B5$. Any instantiation of SATenstein-LS follows the same high-level structure:

1. Optionally execute $B1$, which performs search diversification.
2. Execute either $B2$, $B3$ or $B4$, thus performing a WalkSAT-based, dynamic local search or G$^2$WSAT-based procedure, respectively.
3. Optionally execute $B5$ to update data structures such as promising list, clause penalties, dynamically adaptable parameters or tabu attributes.

Each of our building blocks consists of one or more components (listed in Table 2); some of these components are shared across different building blocks. Each component is configurable by one or more parameters. Out of 42 parameters overall, 6 of SATenstein-LS's parameters are integer-valued (listed in Table 11), 19 are categorical (listed in Table 12), and 17 are real-valued (listed in Table 13). All of these parameters are exposed on the command line so that they can be optimized using an automatic configurator. After fixing the domains of integer- and real-valued parameters to between 3 and 16 values each (as we did in our experiments, reported later) the total number of valid SATenstein-LS instantiations was $2.01 \times 10^{14}$.

We now give a high-level description of each of the building blocks. In particular, we provide detailed descriptions of SATenstein-LS's three key building blocks, $B2$, $B3$ and $B4$, which map to three broad categories of SLS-based SAT solvers.

#### 3.1.1. Block $B1$

$B1$ is constructed using the *SelectClause()*, *DiversificationStrategy()* and *DiversificationProbability()* components. *SelectClause()* is configured by one categorical parameter and, depending on its value, either selects an unsatisfied clause uniformly at random or selects a clause with probability proportional to its clause penalty [74]. Component *diversificationStrategy()* can be configured by a categorical parameter to do any of the following with probability *diversificationProbability()*: flip the least recently flipped variable [50]; flip the least frequently flipped variable [64]; flip the variable with minimum variable weight [64]; or flip a randomly selected variable [33].

#### 3.1.2. Block $B2$ (WalkSAT-based algorithms)

Block $B2$ instantiates WalkSAT-based algorithms, which—unlike GSAT or its variants—select in each step a single unsatisfied clause (typically uniformly at random from the set of all currently unsatisfied clauses), and consider only the variables appearing therein as candidates for flipping; the variable to be flipped is chosen using a heuristic. WalkSAT/SKC [66], one of the earliest and most prominent algorithms from this family, uses a scoring function that only depends on *BreakCount* (see Section 2.1) for variable selection.

As previously described in the context of $B1$, component *SelectClause()* is used to select an unsatisfiable clause $c$. The *SelectHeuristic()* component selects a variable from $c$ for flipping. Depending on a categorical parameter, *SelectHeuristic()* can instantiate any of the thirteen well-known WalkSAT-based heuristics, notably including Novelty variants, VW1 and VW2. Table 3 lists these heuristics and related continuous parameters. We also extended the Novelty variants with an optional "flat move" mechanism, as found in the selection strategy in gNovelty$^+$ [71,62].

WalkSAT/Tabu [55] is an extension of WalkSAT/SKC that forbids variables that have been flipped within the last $t$ steps from being flipped again, where $t$ is a parameter called the *tabu tenure*. If all variables in all unsatisfied clauses are *tabu*, then the *tabu list* is ignored. Tabu variants of WalkSAT algorithms can be configured in SATenstein-LS by setting the categorical parameter *performTabuSearch*.

Novelty [55] and its variants are also very prominent WalkSAT algorithms. Novelty scores the variables in the selected clause using the same scoring function as GSAT. If the variable with the highest score is not the most-recently-flipped variable within the clause, then it is deterministically selected for flipping. Otherwise, it is selected with probability $(1 - p)$, where $p$ is a parameter called the *noise setting* (with probability $p$, the second-best variable is selected). To prevent search stagnation, Novelty has been augmented with a probabilistic conflict-directed random walk mechanism, leading to the Novelty$^+$ algorithm [32]. Later Novelty variants (e.g., adaptNovelty$^+$; [33]) also use a dynamic mechanism for changing the *noise* parameter during the search process; this mechanism has since been extended to many other SLS-based SAT solvers (e.g., [53]) and can be instantiated in SATenstein-LS by setting the parameter *useAdaptiveMechanism* to 1 (for further details, see, Table 12).

#### 3.1.3. Block $B3$ (dynamic local search algorithms)

Block $B3$ instantiates dynamic local search algorithms. The most prominent feature of dynamic local search (DLS) algorithms is the use of penalties (or weights) associated with the clauses of the given CNF formula. DLS algorithms typically use a GSAT-like variable selection mechanism, but calculate scores taking clause penalties into account, reflecting the perceived importance of satisfying each clause. At each step, penalties associated with unsatisfied clauses are increased (additively [71] or multiplicatively [42]); this enables the local search process to escape from local minima of the objective function defined by the sum of the penalties of unsatisfied clauses. In order to ensure that the penalty values do not increase unboundedly and to appropriately emphasize recent search history, occasional *smoothing* steps are performed to reduce penalties.

In SATenstein-LS, the task of pruning the set of variables based on clause weights is accomplished by the *selectSet()* component. *selectSet()* first considers the set of variables that occur in any unsatisfied clause and 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. 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. [55], Selman et al. [66], 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.

#### 3.1.4. Block $B4$ ($G^2$WSAT variants)

Block $B4$ instantiates G$^2$WSAT-based algorithms that combine key features of the GSAT and WalkSAT architectures and use a data structure *promising list* containing *promising decreasing variables*. (The definition of a *promising decreasing variable* is somewhat technical; interested readers should refer to Appendix A.) Like GSAT, G$^2$WSAT has a deterministic greedy component that looks at the *promising list* first. If this list contains at least one variable (*promising decreasing variable*), G$^2$WSAT deterministically selects the variable with the best score for flipping, breaking ties in favor of the least recently

**Table 1**
Design choices for *selectFromPromisingList()*.

| Param value | Design choice | Based on |
|---|---|---|
| 1 | If freebie exists, use tieBreaking(); else, select uniformly at random | [66] |
| 2 | Variable with best score | [50] |
| 3 | Least-recently-flipped variable | [53] |
| 4 | Variable with best VW1 score | [64] |
| 5 | Variable with best VW2 score | [64] |
| 6 | Variable selected uniformly at random | [32] |
| 7 | Variable selection from Novelty | [55] |
| 8 | Variable selection from Novelty++ | [50] |
| 9 | Variable selection from Novelty$^+$ | [32] |
| 10 | Variable selection from Novelty++$'$ | [52] |
| 11 | Variable selection from Novelty+p | [52] |

**Table 2**
SATenstein-LS components.

| Component | Block | Parameters | Instantiations | Detailed Info |
|---|---|---|---|---|
| diversificationStrategy() | 1 | searchDiversificationStrategy | 4 | Table 12 |
| SelectClause() | 1, 2, 4 | selectClause | 2 | Table 12 |
| diversificationProbability() | 1 | rdp, rfp, rwp | 216 | Table 13 |
| selectFromPromisingList() | 4 | selectPromVariable | 4312 | Table 1, 12 |
| | | promDp, promWp, promNovNoise | | Table 13 |
| selectHeuristic() | 2, 4 | heuristic | | Table 3, 12 |
| | | performAlternateNovelty | $1.83 \times 10^6$ | Table 12 |
| | | wp, dp, wpWalk, novNoise, s, c | | Table 13 |
| selectSet() | 3 | scoringMeasure, smoothingScheme | | Table 12 |
| | | maxinc | 24 576 | Table 11 |
| | | alpha, rho, sapsthresh, pflat | | Table 13 |
| tiebreaking() | 3 | tieBreaking | 4 | Table 12 |
| update() | 5 | useAdaptiveMechanism, adaptivenoisescheme, | | Table 12 |
| | | adaptWalkProb, performTabuSearch, | | Table 12 |
| | | useClausePenalty, adaptiveProm, | | Table 12 |
| | | adaptpromwalkprob, updateSchemePromList, | $1.76 \times 10^8$ | Table 12 |
| | | tabuLength, phi, theta, promPhi, promTheta, | | Table 11 |
| | | ps | | Table 13 |

flipped variable. If the promising list is empty, the stochastic component of G$^2$WSAT is employed, a Novelty variant that belongs to the WalkSAT architecture.

In SATenstein-LS, selection of *promising variable* is performed by the *selectFromPromisingList()* component. For this component, in addition to two existing strategies found in the G$^2$WSAT literature (see, e.g., [50,53]), we added nine novel strategies based on variable selection heuristics from other solvers. These, to the best of our knowledge, have never been used before in the context of *promising variable* selection for G$^2$WSAT-based algorithms. For example, in previous work, variable selection mechanisms used in Novelty variants were only applied to variables of unsatisfiable clauses, not to promising lists. Table 1 lists the eleven possible strategies for *selectFromPromisingList*. If *promising list* is empty, *B*4 behaves exactly as *B*2, which instantiates WalkSAT-based algorithms.

Except for G$^2$WSAT [50], all G$^2$WSAT variants use the reactive mechanism found in adaptNovelty$^+$ [32]. gNovelty$_+$ [62], the winner of the 2007 SAT Competition in the random satisfiable category, also uses clause penalties and a smoothing mechanism found in dynamic local search algorithms [71] which can be activated in SATenstein-LS by setting the categorical parameter *useClausePenalty* to 1. As already mentioned in the context of *B*2, the reactive mechanism for is activated by setting the categorical parameter *useAdaptiveMechanism* to 1.

#### 3.1.5. Block *B*5

Block *B*5 updates data structures required by the previously mentioned mechanisms, (e.g., dynamic local search) after a variable has been flipped. Performing these updates in an efficient manner is of crucial importance for the performance of many SLS algorithms. As the SATenstein-LS framework supports the combination of mechanisms from many different SLS algorithms, each depending on different data structures, the implementation of the *update()* function was technically quite challenging.

### 3.2. Implementation and validation

As already mentioned, SATenstein-LS is built on top of UBCSAT [74]. UBCSAT makes use of a trigger-based architecture that facilitates the reuse of existing mechanisms. While designing and implementing SATenstein-LS, we not only

**Table 3**
List of heuristics chosen by the parameter *heuristic* and dependent parameters.

| Param. value | Selected heuristic | Dependent parameters |
|---|---|---|
| 1 | Novelty [55] | novnoise |
| 2 | Novelty$^+$ [33] | novnoise, wp |
| 3 | Novelty++ [50] | novnoise, dp |
| 4 | Novelty++′ [52] | novnoise, dp |
| 5 | R-Novelty [55] | novnoise |
| 6 | R-Novelty$^+$ [33] | novnoise, wp |
| 7 | VW1 [64] | wpwalk |
| 8 | VW2 [64] | s, c, wpwalk |
| 9 | WalkSAT-SKC [66] | wpwalk |
| 10 | Noveltyp [52] | novnoise |
| 11 | Novelty$^+$p [52] | novnoise, wp |
| 12 | Novelty++p [52] | novnoise, dp |
| 13 | Novelty++′p [52] | novnoise, dp |

studied existing SLS algorithms, as presented in the literature, but we also analyzed the SAT competition submissions of such algorithms. We found that the published pseudocode of VW2 [64] differed from its 2005 SAT Competition version, which includes a reactive mechanism; we included both versions in SATenstein-LS's implementation. We also found that in the SAT competition implementation of gNovelty$^+$, Novelty used a PAWS-like [71] "flat move" mechanism. We implemented this alternate version of Novelty in SATenstein-LS and exposed a categorical parameter to choose between the two implementations. While examining the implementations of various SLS solvers, we noticed that certain key data structures were implemented in different ways. In particular, different G$^2$WSAT variants use different realizations of the update scheme of *promising list*. We included all these update schemes in SATenstein-LS and declared parameter *updateSchemePromList* to select between them.

Since SATenstein-LS is quite complex, we took great care in validating its implementations of existing SLS-based SAT solvers. We compared our SATenstein-LS implementation with ten well-known algorithms' reference implementations (specifically, every algorithm listed in Table 5 except for Ranov), measuring running times as the number of variable flips.[3] These ten algorithms span G$^2$WSAT-based, WalkSAT-based, and dynamic local search procedures, and also make use of all the prominent SLS solver mechanisms discussed earlier. Our validation results showed that in every case, reference solvers and their SATenstein-LS implementations have the same run-length distributions on a small set of 10 validation instances chosen from block world and software verification, based on a Kolmogorov–Smirnov test (5000 runs per solver–instance pair with significance threshold 0.05).

## 4. Experimental setup

In order to study the effectiveness of our proposed approach for algorithm design, we configured SATenstein-LS on training sets from various distributions of SAT instances and compared the performance of the SATenstein-LS solvers thus obtained against that of several existing high-performance SAT solvers on disjoint test sets.

### 4.1. Instance distributions

We considered six sets of well-known benchmark instances for SAT (see Table 4). These six distributions can be grouped into three broad categories: industrial (CBMC(SE), FAC), handmade (QCP, SW-GCP), and random (R3SAT, HGEN). Because SLS algorithms are unable to prove unsatisfiability, we constructed our benchmark sets to include only satisfiable instances.

The instance generators for HGEN and FAC only produce satisfiable instances. For each of these two distributions, we generated 2000 instances with the generator settings shown in Table 4. 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 suggested by Gomes and Selman [23]. We first filtered out unsatisfiable instances and then chose 2000 satisfiable instances uniformly at random. For SW-GCP, we generated 20 000 instances following [22] and then drew a sample of 2000 satisfiable instances uniformly at random from this set. 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 complete solvers, then chose 500 of these uniformly at random. 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. We preprocessed these instances using SatELite [17], identifying 604 of them as satisfiable and the remaining 7 as unsatisfiable.

Finally, we randomly split each of the six instances sets thus obtained into training and test sets of equal size.

[3] SATenstein-LS does not support preprocessing, as a consequence of being built on top of UBCSAT. We thus manually disabled the preprocessing steps of G2, AG2p, AG2+, and AG20 when performing this validation.

**Table 4**
Our six benchmark distributions.

| Distribution | Description | Generator parameters | Train/test size |
|---|---|---|---|
| QCP | SAT-encoded quasi-group completion problems [23] | order $O \in [10, 30]$; holes $H = h * O^{1.55}$, $h \in [1.2, 2.2]$ | 1000/1000 |
| SW-GCP | SAT-encoded small-world graph-colouring problems [22] | ring lattice size $S \in [100, 400]$; nearest neighbors connected: 10; rewiring probability: $2^{-7}$; chromatic numbers: 6 | 1000/1000 |
| R3SAT | uniform-random 3-SAT instances [68] | variable: 600; clauses-to-variables ratio: 4.26 | 250/250 |
| HGEN | random instances generated by HGEN2 [31] | variable $n \in [200, 400]$ | 1000/1000 |
| FAC | SAT-encoded factoring problems [75] | prime number $\in [3000, 4000]$ | 1000/1000 |
| CBMC(SE) | SAT-encoded bounded model checking [14], preprocessed by SatELite [17] | array size $s \in [1, 2000]$; loop unwinding $n \in 4, 5, 6$ | 302/302 |

### 4.2. Configuration protocol

In order to perform automatic algorithm configuration, we first had to quantify performance using an objective function. Consistent with most previous work on SLS algorithms for SAT, we chose to focus on mean runtime. In order to deal with runs that had to be terminated at a given cutoff time, following Hutter et al., [39], we used a variant of mean runtime known as PAR-10, defined as the average runtime over a given set of runs, where timed-out runs are counted as 10 times the given cutoff time. Unless explicitly stated otherwise, all runtimes reported in this article were measured using PAR-10 over the respective set of instances.

To perform automated configuration, we used the FocusedILS procedure from the ParamILS framework, version 2.3 [41]. We chose this method because it has been demonstrated to operate effectively on many extremely large, discrete parameter spaces (see, e.g., [40,38,72,63]), and because it supports conditional parameters (discussed below). FocusedILS takes as input a parameterized algorithm (the so-called target algorithm), a specification of domains and (optionally) conditions for all parameters, a set of training instances, and an evaluation metric. It outputs a parameter configuration of the target algorithm that approximately minimizes the given evaluation metric.

As just mentioned, FocusedILS supports conditional parameters, which are important to `SATenstein-LS`. For example, condition $A|B = b$ means that $A$ is activated if $B$ take the value $b$. When more than one such condition is given for the same parameter $A$, these are interpreted as being connected by logical 'and'. For example, the two conditions, $A|B = b$ and $A|C = c$, are interpreted as $A|(B = b) \wedge (C = c)$. Some parameters in `SATenstein-LS` can be activated in more than one way. While this cannot be directly specified in the input to FocusedILS, we can express such disjunctive conditions using dummy parameters, as illustrated in the following example. Consider an algorithm `S` with four parameters, $\{A, B, C, D\}$, and where $A$ is activated if $B = b$ or $C = c$, while $D$ is activated if $A = a$. As it is impossible to express the condition $A|(B = b) \vee (C = c)$ directly in the input to FocusedILS, we introduce two dummy parameters, $A^*$ and $D^*$. Using these additional parameters, the given conditions can be expressed as $A|B = b$; $A^*|C = c$; $A^*|B \neq b$; $D|A = a$; $D^*|A^* = a$. Since only one of $(A, A^*)/(D, D^*)$ is activated, we can simply map $A^*$ to $A$ and $D^*$ to $D$ when instantiating `S` with a parameter configuration found by FocusedILS.

We used a cutoff time of 5 CPU seconds for each target algorithm run, and allotted 7 days to each run of FocusedILS; we note that, while 5 CPU seconds is unrealistically short for assessing the performance of SAT solvers, using short cutoff times during configuration is important for the efficiency of the configuration process and typically works well, as demonstrated by our `SATenstein-LS` results. Since ParamILS cannot operate directly on continuous parameters, each continuous parameter was discretized into sets containing between 3 and 16 values that we considered reasonable (see Table 11). Except for a small number of cases (e.g., the parameters *s,c*) for which we used the same discrete domains as mentioned in the publication first describing it [64]), we selected these values using a regular grid over a range of values that appeared reasonable. For each integer parameter, we specified 4 to 10 values, always including the known defaults (see Table 13). In all cases, these choices included the parameter values required to cover the default configurations of the solvers whose components were integrated into `SATenstein-LS`'s design space. Categorical parameters and their respective domains are listed in Table 12. As mentioned before, based on this discretization, `SATenstein-LS`'s parameter configuration space consists of $2.01 \times 10^{14}$ distinct configurations.

Since the performance of FocusedILS can vary significantly depending on the order in which instances appear in the training set, we ran FocusedILS 20 times on the training set, using different, randomly determined instance orderings for each run. From the 20 parameter configurations obtained from FocusedILS for each instance distribution `D`, we selected the parameter configuration with the best penalized average runtime on the training set. We then evaluated this configuration on the test set. For a given distribution `D`, we refer to the corresponding instantiation of a solver `S` as `S[D]`.

**Table 5**
Our eleven *challenger* algorithms.

| Algorithm | Abbrev | Reason for inclusion | Parameters |
|---|---|---|---|
| Ranov [61] | Ranov | gold 2005 SAT Competition (random) | wp |
| $G^2$WSAT [50] | G2 | silver 2005 SAT Competition (random) | novNoise, dp |
| VW [64] | VW | bronze 2005 SAT Competition (random) | c, s, wpWalk |
| gNovelty$^+$ [62] | GNOV | gold 2007 SAT Competition (random) | novNoise, wpWalk, ps |
| adaptG$^2$WSAT$_0$ [52] | AG20 | silver 2007 SAT Competition (random) | NA |
| adaptG$^2$WSAT$_+$ [53] | AG2+ | bronze 2007 SAT Competition (random) | NA |
| adaptNovelty$^+$ [33] | ANOV | gold 2004 SAT Competition (random) | wp |
| textttadaptG$^2$WSAT$_p$ [53] | AG2p | performance comparable to G$^2$WSAT [50], Ranov, and adaptG$^2$WSAT$_+$; see [52] | NA |
| SAPS [42] | SAPS | prominent DLS algorithm | alpha, ps, rho, sapsthresh, wp |
| RSAPS [42] | RSAPS | prominent DLS algorithm | alpha, ps, rho, sapsthresh, wp |
| PAWS [71] | PAWS | prominent DLS algorithm | maxinc, pflat |

### 4.3. Solvers used for performance comparison

For each instance distribution $D$, we compared the performance of SATenstein-LS[D] against that of 11 high-performance SLS-based SAT solvers on the respective test set. We included every SLS algorithm that won a medal in any category of a SAT competitions between 2002 and 2007, because those algorithms are all part of the SATenstein-LS design space.

Although dynamic local search (DLS) algorithms have not won medals in recent SAT competitions, we also included three prominent, high-performing DLS algorithms for two reasons. First, some of them represented the state of the art when introduced (e.g., SAPS [42]) and still offer competitive performance on many instances. Second, techniques used in these algorithms have been incorporated into other recent high-performance SLS algorithms. For example, the additive clause weighting scheme used in PAWS is also used in the 2007 SAT Competition winner gNovelty$^+$ [62]. We call these algorithms *challengers* and list them in Table 5. In order to demonstrate the full performance potential of these solvers, we also tuned the parameters for all parameterized challengers using the same configuration procedure and protocol as for SATenstein-LS, including the same choices of discrete values for continuous and integer parameters.

SATenstein-LS can be instantiated such that it emulates all 11 challenger algorithms (except for preprocessing components used in Ranov, AG2p, AG2plus, and AG20). However, in some cases, the original implementations of these algorithms are more efficient—on our data, by at most a factor of two on average per instance set—mostly, because SATenstein-LS's generality rules out some data structure optimizations.

Thus, we based all of our experimental comparisons on the original algorithm implementations, as submitted to the respective SAT Competitions. The exceptions are PAWS, whose implementation within UBCSAT is almost identical to the original in terms of runtime, as well as SAPS, RSAPS, and ANOV, whose UBCSAT implementations are those used in the competitions. All of our comparisons on the test set are based on running each solver 25 times per instance, with a per-run cutoff of 600 CPU seconds.

Our goal was to improve the state of the art in SAT solving. Thus, although the design space of SATenstein-LS consists solely of SLS solvers, we have also compared its performance to that of high-performance complete solvers (listed in Table 6). Unlike SLS solvers, these complete solvers are deterministic. Thus, for every instance in each distribution, we ran each complete solver once with a per-run cutoff of 600 CPU seconds.

### 4.4. Execution environment

We carried out our experiments on a cluster of 55 machines each equipped with dual 3.2 GHz Intel Xeon CPUs with 2 MB cache and 2 GB RAM, running OpenSuSE Linux 11.1. Our computer cluster was managed by a distributed resource manager, Sun Grid Engine (version 6.0). Runtimes for all algorithms (including FocusedILS) were measured as CPU time on these reference machines. Each run of any solver only used one CPU.

## 5. Results

We now present the results of performance comparisons between SATenstein-LS and the 11 challenger SLS solvers (listed in Table 5), configured versions of these challengers, and two complete solvers for each of our benchmark distribu-

**Table 6**
Complete solvers we compared against.

| Category | Solver | Reason for inclusion |
|---|---|---|
| Industrial (CBMC(SE) and FAC) | Picosat [7,6] | gold, silver 2007 SAT Competition (industrial) |
| | Minisat2.0 [69] | bronze, silver 2007 SAT Competition (industrial) |
| Handmade (QCP and SW-GCP) | Minisat2.0 [69] | bronze, silver 2007 SAT Competition (handmade) |
| | March_pl [28] | Improved, bug-free version of March_ks [29], gold in 2007 SAT Competition (handmade) |
| Random (HGEN and R3SAT) | Kcnfs_04 [16] | silver 2007 SAT Competition (random) |
| | March_pl [28] | Improved, bug-free version of March_ks [29], silver in 2007 SAT Competition (random) |

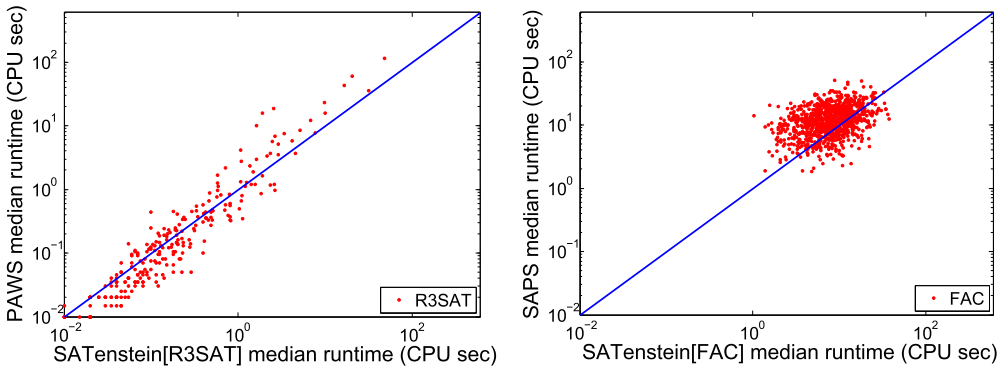


**Fig. 1.** Performance comparison of SATenstein-LS and the best challenger. Left: R3SAT; Right: FAC. Medians were taken over 25 runs on each instance with a cutoff time of 600 CPU seconds per run.

tions (listed in Table 6). Although in our configuration experiments, we optimized SATenstein-LS for penalized average runtime (PAR-10), we also examine its performance in terms of other performance metrics, such as median runtime and percentage of instances solved within the given cutoff time.

### 5.1. Comparison with challengers

For every one of our six benchmark distributions, we were able to find a SATenstein-LS configuration that outperformed all 11 challengers. Our results are summarized in Table 7.

In terms of penalized average runtime, the performance metric we explicitly optimized using ParamILS (with a cutoff time of 5 CPU seconds rather than the 600 CPU seconds used here for testing, as explained in Section 5.2), our SATenstein-LS solvers achieved better performance than every challenger on every distribution. For QCP, HGEN, and CBMC(SE), SATenstein-LS achieved a PAR-10 that was orders of magnitude better than the respective best challengers. For SW-GCP, R3SAT, and FAC, there was substantial, but less dramatic improvement. The modest improvement in R3SAT was not very surprising (Fig. 1: Left); R3SAT is a well-known SAT distribution on which SLS solvers have been evaluated and optimized for decades. Conversely, on a new benchmark distribution, CBMC(SE), where DPLL solvers represent the state of the art, SATenstein-LS solvers performed markedly better than every SLS-based challenger. We were surprised to see the amount of improvement we obtained for HGEN, a hard random SAT distribution very similar to R3SAT, and QCP, a widely-known SAT distribution. We noticed that on HGEN, some older solvers such as SAPS and PAWS performed much better than more recent medal winners such as GNOV and AG20. Also, for QCP, a somewhat older algorithm, ANOV, turned out to be the best challenger. These observations led us to believe that the strong performance of SATenstein-LS was partly due to the fact that the past seven years of SLS SAT solver development have not taken these types of distributions into account and have not yielded across-the-board improvements in SLS solver performance.

We also evaluated the performance of SATenstein-LS solvers using two other performance metrics: median-of-median runtime and percentage of solved instances. If a solver finishes most of the runs on most instances, the capped runs will not affect its median-of-median performance, and hence the metric does not need a way of accounting for the cost of capped runs. (When the median of medians is a capped run, we say that the metric is undefined.) Table 7 shows that, although the SATenstein-LS solvers were obtained by optimizing for PAR-10, they still outperformed every challenger in every distribution except for R3SAT, in which the challengers achieved slightly better performance than SATenstein-LS.

**Table 7**
Performance of SATenstein-LS and the 11 challengers. Every algorithm was run 25 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 $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the penalized average runtime; $b$ (middle) is the median of the median runtime over all instances (where the outer median is taken over the instances in the given test set and the inner median over the runs on each instance; this is undefined if fewer than half of the median runs failed to find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., those with median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold, and the best-scoring challenger(s) are underlined.

| SATenstein-LS[D] [45] | **0.08** **0.01** **100%** | **0.03** **0.02** **100%** | **1.11** 0.14 **100%** | **0.02** **0.01** **100%** | **10.89** **7.90** **100%** | **4.75** **0.02** **100%** |
|---|---|---|---|---|---|---|
| Solvers | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
| | 1054.99 | 0.64 | 2.14 | 137.02 | 3594.40 | 2169.77 |
| AG20 | 0.03 | 0.11 | 0.13 | 0.57 | N/A | 0.56 |
| [52] | 81.2% | **100%** | **100%** | 98.1% | 35.9% | 61.1% |
| | 1119.96 | 0.43 | 2.35 | 105.30 | 1954.83 | 2294.24 |
| AG2p | 0.02 | 0.06 | 0.14 | 0.48 | 330.26 | 2.57 |
| [53] | 80.1% | **100%** | **100%** | 98.4% | 80.6% | 61.1% |
| | 1091.37 | 0.67 | 3.04 | 148.28 | 1450.89 | 2181.92 |
| AG2+ | 0.03 | 0.08 | 0.16 | 0.59 | 238.31 | 0.64 |
| [53] | 80.3% | **100%** | **100%** | 98.0% | 91.0% | 61.1% |
| | 25.42 | 4.86 | 11.17 | 109.94 | 2897.52 | 2021.22 |
| ANOV | 0.02 | 0.04 | 0.15 | 0.50 | 588.23 | 3.10 |
| [33] | 99.6% | **100%** | **100%** | 98.6% | 51.4% | 61.1% |
| | 2942.13 | 4092.29 | 3.69 | 104.55 | 5947.80 | 2139.12 |
| G2 | 341.60 | N/A | 0.13 | 0.60 | N/A | 0.57 |
| [50] | 50.9% | 31.0% | **100%** | 98.7% | 0% | 65.4% |
| | 414.69 | 1.20 | 11.14 | 52.58 | 5935.39 | 2236.85 |
| GNOV | 0.03 | 0.09 | 0.15 | 0.71 | N/A | 0.67 |
| [62] | 93.3% | **100%** | **100%** | 99.4% | 0% | 61.5% |
| | 1127.84 | 4495.50 | 1.77 | 62.18 | 22.05 | 1693.82 |
| PAWS | 0.03 | N/A | **0.08** | 0.82 | 10.41 | 0.18 |
| [71] | 81.0% | 24.3% | **100%** | 99.4% | **100%** | 70.8% |
| | 73.38 | 0.15 | 18.29 | 151.11 | 887.33 | 1227.07 |
| RANOV | 0.1 | 0.12 | 0.36 | 0.90 | 152.16 | 0.58 |
| [61] | 99.1% | **100%** | **100%** | 98.2% | 96.8% | 79.7% |
| | 1255.94 | 5635.54 | 18.42 | 33.28 | 17.86 | 827.81 |
| RSAPS | 0.05 | N/A | 1.86 | 2.33 | 11.53 | **0.02** |
| [42] | 79.2% | 5.4% | **100%** | 99.7% | **100%** | 85.0% |
| | 1248.34 | 3864.74 | 22.93 | 40.17 | 16.41 | 646.89 |
| SAPS | 0.04 | N/A | 1.77 | 2.65 | 10.56 | **0.02** |
| [42] | 79.4% | 34.2% | **100%** | 99.5% | **100%** | 89.7% |
| | 1022.69 | 161.74 | 12.45 | 176.18 | 3382.02 | 385.12 |
| VW | 0.25 | 40.26 | 0.82 | 3.13 | N/A | 0.23 |
| [64] | 81.9% | 99.4% | **100%** | 97.8% | 35.3% | 93.4% |

Finally, we measured the percentage of instances on which the median runtime was below the cutoff used for capping runs. According to this measure, SATenstein-LS either equaled or beat every challenger, since it solved 100% of the instances in every benchmark set. In contrast, only 4 challengers managed to solve more than 50% of instances in every test set. Overall, SATenstein-LS solvers scored well on these measures for which its performance had not been explicitly optimized.

The relative performance of the challengers varied significantly across different distributions. For example, the three dynamic local search solvers (SAPS, PAWS, and RSAPS) performed substantially better than the other challengers on factoring instances (FAC). However, on SW-GCP, their relative performance was weak. Similarly, GNOV (the 2007 SAT Competition winner in the random satisfiable category) performed very poorly on our two industrial benchmark distributions, CBMC(SE) and FAC, but solved SW-GCP and HGEN instances quite efficiently.[4] This suggests that different distributions are most efficiently solved by rather different solvers. We are thus encouraged that our automatic algorithm construction process was able to find good configurations for each distribution.

[4] Interestingly, on both types of random instances we considered, GNOV failed to outperform some of the older solvers, in particular, PAWS and RSAPS.

**Table 8**
Performance of SATenstein-LS solvers, the best challengers with default configurations and the best automatically configured challengers. Every algorithm was run 25 times on each instance with a cutoff of 600 CPU seconds per run. Each table entry $\langle i, j \rangle$ indicates the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances; $c$ (bottom) is the percentage of instances solved (i.e., those with median runtime $<$ cutoff).

| Distribution | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| Best Challenger (default) | ANOV | RANOV | PAWS | RSAPS | SAPS | VW |
| | 25.42 | 0.15 | 1.77 | 33.28 | 16.41 | 385.12 |
| Performance | 0.02 | 0.12 | **0.08** | 2.33 | 10.56 | 0.23 |
| | 99.6% | **100%** | **100%** | 99.7% | **100%** | 93.4% |
| Best Challenger (tuned) | VW[D] | G2[D] | VW[D] | SAPS[D] | SAPS[D] | VW[D] |
| | 0.33 | 0.05 | 1.26 | 31.77 | **10.68** | 16.45 |
| Performance | 0.02 | 0.05 | 0.15 | 0.75 | **7.00** | **0.02** |
| | **100%** | **100%** | **100%** | 99.6% | **100%** | **100%** |
| | **0.08** | **0.03** | **1.11** | **0.02** | 10.89 | **4.75** |
| SATenstein-LS[D] | **0.01** | **0.02** | 0.14 | **0.01** | 7.90 | **0.02** |
| Performance | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |

**Table 9**
Performance summary of SATenstein-LS and the complete solvers. Every complete solver was run once (SATenstein-LS was run 25 times) on each instance with a per-run cutoff of 600 CPU seconds. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances (for SATenstein-LS, it is the median of the median runtimes over all instances. the median runtimes are not defined if fewer than half of the median runs failed to find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

| Distribution | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| Complete Solver | Minisat2.0 | Minisat2.0 | Kcnf_04 | Kcnf_04 | Minisat2.0 | Minisat2.0 |
| | 35.05 | 2.17 | 4905.6 | 3108.77 | 0.03 | 0.23 |
| Performance | 0.02 | 0.9 | N/A | N/A | **0.02** | 0.03 |
| | 99.5% | **100%** | 18.8% | 49.5% | **100%** | **100%** |
| Complete Solver | March_ pl | March_ pl | March_ pl | March_ pl | Picosat | Picosat |
| | 120.29 | 253.99 | 3543.01 | 2763.41 | **0.02** | **0.03** |
| Performance | 0.2 | 1.12 | N/A | 400.78 | **0.02** | **0.01** |
| | 98.1% | 95.8% | 42.0% | 55.2% | **100%** | **100%** |
| | **0.08** | **0.03** | **1.11** | **0.02** | 10.89 | 4.75 |
| SATenstein-LS[D] | **0.01** | **0.02** | 0.14 | **0.01** | 7.90 | 0.02 |
| Performance | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |

So far, we have discussed performance metrics that describe aggregate performance over the entire test set. One might wonder if SATenstein-LS's strong performance is due its ability to solve relatively few instances very efficiently, while performing poorly on others. We found that this is typically not the case and barring one distribution, R3SAT (detailed analysis can be found in Appendix C), although even in R3SAT SATenstein-LS[R3SAT] solved the harder instance more efficiently than PAWS, the best challenger.

### 5.2. Comparison with automatically configured versions of challengers

The fact that SATenstein-LS solvers achieved significantly better performance than all 11 challengers with default parameter configurations (i.e., those selected by their designers) admits two possible explanations. First, it could be due to the fact that SATenstein-LS's (vast) design space includes useful new configurations that combine solver components in novel ways. Second, the performance gains may have been achieved simply by better configuring existing SLS algorithms within their existing, and quite small, design spaces. To determine which of these two hypotheses holds, we compared SATenstein-LS solvers against challengers configured for optimized performance on our benchmark sets, using the same automated configuration procedure and protocol.

Table 8 summarizes the performance of our SATenstein-LS solvers, the best default challengers, and the best automatically configured challengers (for further details on individual challenger's performance, see, Table 15), and shows that our first hypothesis, the performance gain of SATenstein-LS is indeed a result of its significantly richer design space than that of the challengers, is true. For QCP, HGEN and CBMC(SE), the SATenstein-LS solvers still significantly outperformed the best configured challengers. For R3SAT and SWGCP, the performance difference was small, but still above 10%. The only benchmark where the best configured challenger outperformed SATenstein-LS was FAC. On closely examining the SATenstein-LS[FAC] configuration, we found that SATenstein-LS[FAC] was very similar to the best configured challenger, SAPS[FAC].

Overall, these experimental results provide evidence in favor of our first hypothesis: the good performance of SATenstein-LS solvers is due to combining components gleaned from existing high-performance algorithms in novel ways.

**Table 10**
Performance summary of SATenstein-LS2.0 and new set of challengers. Every solver was run 25 times on each instance with a per-run cutoff of 600 CPU seconds. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances (the median runtimes are not defined if fewer than half of the median runs failed to find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

| Distribution | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| Captain Jack | CJ[3Sat1k] | CJ[CBMC] | CJ[3Sat1k] | CJ[SWV] | CJ[7Sat90] | CJ[SWV] |
| | 7.39 | 5929.3 | 7.30 | 15.01 | 4540.40 | **4.36** |
| Performance | 0.02 | N/A | 0.33 | 0.02 | N/A | 0.09 |
| | 99.9% | 0.3% | **100%** | 99.9% | 17.7% | **100%** |
| Sattime | 178.55 | 20.81 | 2.97 | 136.87 | 650.48 | 2101.1 |
| Performance | 0.02 | 2.59 | 0.17 | 0.70 | 95.84 | 3.93 |
| | 96.8% | **100%** | **100%** | 98.3% | **100%** | 65.23% |
| Sparrow | 2.31 | 5936.8 | 11.32 | 67.79 | 4313.50 | 1544.20 |
| Performance | **0.01** | N/A | 0.21 | 1.46 | N/A | 3.51 |
| | **100%** | 0.80% | **100%** | 99.4% | 11.60% | 79.80% |
| | **0.10** | **0.03** | **1.42** | **0.03** | **15.22** | 15.94 |
| SATenstein-LS2.0[D] | **0.01** | **0.02** | **0.13** | **0.01** | **9.87** | **0.04** |
| Performance | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |

### 5.3. Comparison with complete solvers

Table 9 compares the performance of SATenstein-LS solvers and four prominent complete SAT solvers (two for each distribution). For four out of our six benchmark distributions, SATenstein-LS solvers comprehensively outperformed the complete solvers. For the other two industrial distributions (FAC and CBMC(SE)), the performance of the selected complete solvers was much better than that of either the SATenstein-LS solvers and any of our other local search solvers. The success of DPLL-based complete solvers on industrial instances is not surprising; it is widely believed to be due their ability to take advantage of instance structure (by means of unit propagation and clause learning). Our results confirm that state-of-the-art local search solvers cannot compete with state-of-the-art DPLL solvers on industrial instances. However, SATenstein-LS solvers have made significant progress in closing the gap. For example, for CBMC(SE), state-of-the-art complete solvers were five orders of magnitude better than the next-best SLS challenger, VW. SATenstein-LS reduced the performance gap to three orders of magnitude. We also obtained some modest improvements (a factor of 1.51) for FAC.

### 5.4. Configurations found

To better understand the automatically-constructed SATenstein-LS solvers, we compared their automatically selected design choices to the design of the existing SLS solvers for SAT (the full active parameter configurations of the six SATenstein-LS solvers can be found in Table 16). SATenstein-LS[QCP] uses building blocks 1, 2, and 5. Recall that block 1 is used for performing search diversification, and block 5 is used to update data structures, tabu attributes and clause penalties. In block 2, which is used to instantiate a solver belonging to the WalkSAT architecture, the *heuristic* is based on Novelty$^{++'}$, and in block 1, *diversification* flips the least-frequently-flipped variable from an UNSAT clause. SATenstein-LS[SW-GCP] is similar to SATenstein-LS[QCP] but does not use block 1. In block 2, the *heuristic* is based on Novelty++ as used within G2. SATenstein-LS[R3SAT] uses blocks 1, 3 and 5; it is closest to SAPS, but performs search diversification. A tabu list with length 3 is used to exclude some variables from the search neighborhood. Recall that block 3 is used to instantiate dynamic local search algorithms. SATenstein-LS[HGEN] uses blocks 1, 2, and 5. It is similar to SATenstein-LS[QCP] but uses a *heuristic* based on VW1 as well as a tabu list of length 3. SATenstein-LS[FAC] uses blocks 3 and 5; its instantiation closely resembles that of SAPS, but differs in the way in which variable scores are computed. SATenstein-LS[CBMC(SE)] uses blocks 1, 3, and 5; it computes variable scores using -BreakCount and employs a search diversification strategy similar to that of VW.

Interestingly, none of the six SATenstein-LS configurations we found uses a *promising list* (block 4), a technique integrated into many recent SAT Competition winners. This indicates that many interesting designs that could compete with existing high-performance solvers still remain unexplored in SLS design space. In addition, we found that all SATenstein-LS configurations differ from existing SLS algorithms (except for SATenstein[FAC], whose configuration and performance is similar to SAPS). This underscores the importance of an automated approach, since manually finding such good configurations from a huge design space is very difficult.

### 5.5. Augmenting SATenstein-LS

We now demonstrate that SATenstein-LS can be extended with strategies found in newer SLS-based SAT solvers and present results for an augmented version of SATenstein-LS (dubbed SATenstein-LS2.0), in which we integrated a Novelty variant found in the recent high-performance SAT algorithm, Sattime [51]. In addition to Sattime, we considered two additional solvers, Sparrow [5] and Captain Jack [72], for performance comparisons. We chose Sparrow, because we were curious to explore how SATenstein-LS compares with a more recent high-performance

**Table 11**
Integer parameters of SATenstein-LS and the values considered during ParamILS tuning. Multiple "active when" parameters are combined together using AND. Existing defaults are highlighted in bold. For parameters first introduced in SATenstein-LS, default values are underlined.

| Parameter | Active when | Description | Values considered |
|---|---|---|---|
| tabuLength | performTabuSearch = 1 | Specifies tabu step-length | 1, 3, 5, 7, **10**, 15, 20 |
| phi | useAdaptiveMechanism = 1<br>singleClauseAsNeighbor = 1 | Parameter for adaptively setting noise | 3, 4, **5**, 6, 7, 8, 9, **10** |
| theta | useAdaptiveMechanism = 1<br>singleClauseAsNeighbor = 1 | Parameter for adaptively setting noise | 3, 4, **5**, **6**, 7, 8, 9, 10 |
| promPhi | usePromisingList = 1<br>adaptiveProm = 1<br>selectPromVariable ∈ {7,8,9,10,11} | Parameter for adaptively setting noise | 3, 4, 5, 6, 7, 8, 9, 10 |
| promTheta | usePromisingList = 1<br>adaptiveProm = {1}<br>selectPromVariable ∈ {7,8,9,10,11} | Parameter for adaptively setting noise | 3, 4, 5, 6, 7, 8, 9, 10 |
| maxinc | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 2 | PAWS [71] parameter for additive clause weighting | 5, **10**, 15, 20 |

SLS-based SAT solver some of whose components are not present in SATenstein-LS's design space. Captain Jack is another high-performance SLS-based SAT solver with several components not included in SATenstein-LS. Furthermore, like SATenstein-LS, Captain Jack was conceived as a highly parameterized SAT solver that draws inspiration from multiple algorithms; however, its design space is smaller and more limited conceptually than that of SATenstein-LS, which unifies a broader range of local search techniques and mechanisms.

Tompkins et al. [72] described nine configurations of Captain Jack, optimized for different sets of SAT instances. In light of limited computational resources, since it was unclear which of these would perform best on any of our instance distributions, we first performed a single run of each of these configurations for all instances in each of our benchmark sets. Next, we performed 24 additional runs per instance using the configuration with the best PAR-10 score (ties were broken randomly), resulting in 25 independent runs per instance for that configuration. The implementations of Sparrow and Sattime used in our experiments were those submitted to the 2011 SAT Competition 2011.

Table 10 compares the performance of SATenstein-LS2.0 on our six benchmark distributions. Detailed descriptions of the six SATenstein-LS2.0 solvers can be found in Appendix E. We use the same notation as Tompkins et al. [72] for the Captain Jack configuration optimized for each distribution. On all distributions, in terms of PAR-10 score, SATenstein-LS2.0 outperformed both Sparrow and Sattime. However, although slightly inferior in terms of median-of-median runtime, Captain Jack outperformed SATenstein-LS2.0 on CBMC(SE) in terms of PAR-10 score. This result is not too surprising, since Captain Jack draws components from VE-Sampler [73], which, at the time it was introduced, represented a substantial improvement in the state of the art for local search techniques on these kinds of instances. In future work, SATenstein-LS2.0 could be further augmented with these components, and thus very likely achieve even better performance.

Overall, our results clearly indicate that SATenstein-LS can be augmented with components from more recent SLS-based SAT solvers, and doing so achieves performance comparing favorably even with newer high-performance algorithms.

## 6. Conclusions and future work

We have proposed a new approach for designing heuristic algorithms based on (1) a framework that can flexibly combine components drawn from existing high-performance solvers, and (2) a powerful algorithm configuration procedure for finding instantiations that perform well on given sets of instances. We have demonstrated the effectiveness of our approach by automatically constructing high-performance stochastic local search solvers for SAT. We have shown that these automatically constructed SAT solvers outperform existing state-of-the-art solvers with manually and automatically optimized configurations on a range of widely studied distributions of SAT instances.

Our original inspiration comes from Mary Shelley's classic novel, Frankenstein. One important methodological difference is that we use automated methods for selecting components for our monster instead of picking them by hand. The outcomes are quite different. Unlike the tragic figure of Dr. Frankenstein, 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, recognizing that the SATenstein solvers do not always represent the most elegant designs. Thus, desirable lines of future work include techniques for understanding the importance of different parameters to achieving strong performance on a given benchmark; the extension of our solver framework with preprocessors; and the investigation of algorithm configuration procedures other than ParamILS in the context of our approach. Encouraged by the results achieved on SLS algorithms for SAT, we believe that the general approach behind SATenstein-LS is equally applicable to non-SLS-based solvers and to other combinatorial problems. Finally, we encourage members of the SAT community to apply SATenstein-LS to their own instance distributions, and to extend SATenstein-LS with their own heuristics. Source code and documentation for our SATenstein-LS framework are freely available at http://www.cs.ubc.ca/labs/beta/Projects/SATenstein.

**Table 12**
Categorical parameters of SATenstein-LS. Unless otherwise mentioned, multiple "active when" parameters are combined together using AND.

| Parameter | Active when | Domain | Description |
|---|---|---|---|
| performSearchDiversification | Base level parameter | {0,1} | If true, block $B_1$ is performed |
| usePromisingList | Base level parameter | {0,1} | If true, block $B_2$ is performed |
| singleClauseAsNeighbor | Base level parameter | {0,1} | If true, block $B_3$ is performed<br>else, block $B_4$ is performed |
| selectPromVariable | usePromisingList = 1 | {1, 11} | See Table 1 |
| heuristic | singleClauseAsNeighbor = 1 | {1, 13} | See Table 3 |
| performAlternateNovelty | singleClauseAsNeighbor = 1 | {0,1} | If true, performs Novelty<br>variant with "flat move". |
| useAdaptiveMechanism | Base level parameter | {0,1} | If true, uses adaptive mechanisms. |
| adaptivenoisescheme | useAdaptiveMechanism = 1<br>usePromisingList = 1 | {1,2} | Specifies adaptive noise mechanisms. |
| adaptWalkProb | useAdaptiveMechanism = 1 | {0,1} | If true, walk probability or diversification<br>probability of a heuristic is adaptively<br>tuned. |
| performTabuSearch | Base level parameter | {0,1} | If true, tabu variables are<br>not considered for flipping. |
| useClausePenalty | Base level parameter | {0,1} | If true, clause penalties are computed. |
| selectClause | singleClauseAsNeighbor = 1 | {1,2} | 1 selects an UNSAT clause uniformly<br>at random.<br>2 selects an UNSAT clause with a<br>probability proportional to its<br>clause penalty. |
| searchDiversificationStrategy | performSearchDiversification = 1 | {1,2,3,4} | 1 randomly selects a variable from an<br>UNSAT clause.<br>2 selects the least-recently-flipped<br>-variable from an UNSAT clause.<br>3 selects the least-frequently-flipped<br>variable from an UNSAT clause.<br>4 selects the variable with least<br>VW2 weight from an UNSAT clause. |
| adaptiveProm | usePromisingList = 1 | {0,1} | If true, performs adaptive versions of<br>Novelty variants to select variable<br>from promising list. |
| adaptpromwalkprob | usePromisingList = 1<br>adaptiveProm = 1 | {0,1} | If true, walk probability or diversification<br>probability of Novelty variants used<br>on promising list is adaptively tuned. |
| scoringMeasure | usePromisingList = 0<br>singleClauseAsNeighbor = 0 | {1,2,3} | Specifies the scoring measure.<br>1 uses MakeCount - BreakCount<br>2 uses MakeCount<br>3 uses -BreakCount |
| tieBreaking | usePromisingList = 1<br>selectPromVariable $\in$ { 1,4,5 }<br>*or* singleClauseAsNeighbor = 0 | {1,2,3,4} | 1 breaks ties randomly.<br>2 breaks ties in favor of the<br>least-recently-flipped variable.<br>3 breaks tie in favor of the<br>least-frequently-flipped variable.<br>4 breaks tie in favor of the<br>variable with least VW2 score. |
| updateSchemePromList | usePromisingList = 1 | {1,2,3} | 1 and 2 follow $G^2$WSAT.<br>3 follows gNovelty+. |
| smoothingScheme | useClausePenalty = 1 | {1,2} | When singleClauseAsNeighbor = 1 :<br>1 performs smoothing for only random<br>3-SAT instances with 0.4 fixed<br>smoothing probability.<br>2 performs smoothing for all instances.<br>When singleClauseAsNeighbor = 0 :<br>1 performs SAPS-like smoothing.<br>2 performs PAWS-like smoothing. |

## Appendix A. Definitions

**Definition 1.** Promising Decreasing Variable: A variable $x$ is said to be *decreasing* with respect to an assignment $A$ if its GSAT-score is positive, i.e., if flipping it causes a net decrease in the number of unsatisfied clauses. A *promising decreasing variable* is defined as follows:

1. For the initial random assignment $A$, all *decreasing* variables with respect to $A$ are *promising*.
2. Let $x$ and $y$ be two different variables where $x$ is not *decreasing* with respect to $A$. If, after $y$ is flipped, $x$ becomes *decreasing* with respect to the new assignment $A'$, then $x$ is a *promising decreasing variable* with respect to $A'$.
3. As long as a *promising decreasing variable* is *decreasing*, it remains *promising* with respect to subsequent assignments in local search.

**Table 13**
Continuous parameters of SATenstein-LS and values considered during ParamILS tuning. Unless otherwise mentioned, multiple "active when" parameters are combined together using AND. Existing defaults are highlighted in bold. For parameters first introduced in SATenstein-LS, default values are underlined.

| Parameter | Active when | Description | Discrete values considered |
|---|---|---|---|
| wp | singleClauseAsNeighbor = 1<br>heuristic ∈ {2,6,11}<br>useAdaptiveMechanism = 0<br>*or* smoothingScheme = 1<br>singleClauseAsNeighbor = 0<br>useClausePenalty = 0 | Randomwalk probability for Novelty$^+$ | 0, **0.01**, 0.03, 0.04, 0.05, 0.06, 0.07, 0.1, 0.15, 0.20 |
| dp | singleClauseAsNeighbor = 1<br>heuristic ∈ {3,4,12,13}<br>useAdaptiveMechanism = 0 | Diversification probability for Novelty++ and Novelty++′ | 0.01, 0.03, **0.05**, 0.07, 0.1, 0.15, 0.20 |
| promDp | usePromisingList = 1<br>selectPromVariable ∈ {8,10}<br>adaptiveProm = 0 | Diversification probability for Novelty variants used to select variable from promising list | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| novNoise | singleClauseAsNeighbor = 1<br>heuristic ∈ {1,2,3,4,5,6,10,11,12,13}<br>useAdaptiveMechanism = 0 | Noise parameter for all Novelty variants | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| wpWalk | singleClauseAsNeighbor = 1<br>heuristic ∈ {7,8,9}<br>useAdaptiveMechanism = 0 | Noise parameter for WalkSAT and VW1 | 0.1, 0.2, 0.3, 0.4, **0.5**, 0.6, 0.7, 0.8 |
| promWp | usePromisingList = 1<br>selectPromVariable ∈ {9,11} | Randomwalk probability for Novelty variants used to select variable from promising list | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| promNovNoise | usePromisingList = 1<br>selectPromVariable ∈ 7,8,9,10,11 | Noise parameter for all Novelty variants used to select variable from promising list | 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 |
| alpha | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | 1.01, 1.066, 1.126, 1.189, **1.3**, 1.256, 1.326, 1.4 |
| rho | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | 0, 0.17, 0.333, 0.5, 0.666, **0.8**, 0.83, 1 |
| sapsthresh | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | **−0.1**, -0.2, -0.3, -0.4 |
| ps | useClausePenalty = 1<br>singleClauseAsNeighbor = 1<br>*or* singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>useAdaptiveMechanism = 0<br>smoothingScheme = 1 | Smoothing parameter for SAPS, RSAPS, and gNovelty$^+$ | 0, 0.033, **0.05**, 0.066, 0.1, 0.133, 0.166, 0.2, 0.3, **0.4**, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 |
| s | singleClauseAsNeighbor = 1<br>useAdaptiveMechanism = 0<br>*or* singleClauseAsNeighbor = 0<br>tieBreaking = 4<br>useAdaptiveMechanism = 0 | VW parameter for smoothing | **0.1**, 0.01, 0.001 |
| c | singleClauseAsNeighbor = 1<br>useAdaptiveMechanism = 0<br>*or* singleClauseAsNeighbor = 0<br>tieBreaking = 4<br>useAdaptiveMechanism = 0 | VW parameter for smoothing | 0.1, **0.01**, 0.001, 0.0001, 0.00001, 0.000001 |
| rdp | performSearchDiversification = 1<br>searchDiversificationStrategy ∈ {2,3} | Parameter for search diversification | 0.01, 0.03, **0.05**, 0.07, 0.1, 0.15 |
| rfp | performSearchDiversification = 1<br>searchDiversificationStrategy = 4 | Parameter for search diversification | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15 |
| rwp | performSearchDiversification = 1<br>searchDiversificationStrategy = 1 | Parameter for search diversification | **0.01**, 0.03, 0.05, 0.07, 0.1, 0.15 |
| pflat | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 2 | Parameter for PAWS that controls "flat-moves" | 0.05, 0.10, **0.15**, 0.20 |

## Appendix B. SATenstein-LS parameters

This section lists all SATenstein-LS parameters along with a short description on each parameter's function, when it is active, and the values we considered for our tuning experiments. Tables 11, 12, 13 list the integer, categorical and continuous parameters of SATenstein-LS, respectively.

## Appendix C. Per-instance performance comparison with challengers

Table 14 summarizes the performance of each SATenstein-LS solver compared to each challenger on a per-instance basis and shows that SATenstein-LS's superior aggregate performance over challengers is not a result of better performance on few harder instances and worse or equal performance on the rest. Except for R3SAT, SATenstein-LS solvers

**Table 14**
Percentage of instances on which SATenstein-LS achieved better (equal) median runtime than each of the 11 challengers. Medians were taken over 25 runs on each instance with a cutoff time of 600 CPU seconds per run.

| Challengers | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| AG20 | 76.1 (23.3) | 95.8 (4.2) | 45.6 (17.6) | 98.0 (1.5) | 100.0 (0.0) | 100.0 (0.0) |
| AG2p | 70.6 (28.6) | 88.9 (10.7) | 47.6 (15.2) | 98.2 (1.1) | 100.0 (0.0) | 100.0 (0.0) |
| AG2+ | 75.4 (24.1) | 94.3 (5.7) | 61.6(12.4) | 98.5 (1.1) | 100.0 (0.0) | 100.0 (0.0) |
| ANOV | **57.7** (**40.4**) | 68.5 (27.2) | 57.2 (8.0) | 97.6 (1.3) | 99.9 (0.0) | 100.0 (0.0) |
| G2 | 81.4 (18.6) | 100.0 (0.0) | 34.0 (15.2) | 98.0 (1.4) | 100.0 (0.0) | 100.0 (0.0) |
| GNOV | 97.5 (2.4) | 99.6 (0.4) | 48.8 (16.4) | 99.4 (0.4) | 100.0 (0.0) | 100.0 (0.0) |
| PAWS | 69.0 (30.1) | 100.0 (0.0) | **19.6** (**3.2**) | 100.0 (0.0) | 68.8 (0.0) | 100.0 (0.0) |
| RANOV | 100.0 (0.0) | **100.0** (**0.0**) | 99.2 (0.0) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) |
| RSAPS | 71.5 (28.0) | 99.8 (0.2) | 96.8 (3.2) | **100.0** (**0.0**) | 81.1 (0.0) | 42.2 (54.5) |
| SAPS | 70.9 (28.5) | 100.0 (0.0) | 96.8 (2.4) | 100.0 (0.0) | **73.7** (**0.2**) | 48.8 (48.5) |
| VW | 85.3 (14.7) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) | **100.0** (**0.0**) |

**Table 15**
Performance summary of the automatically configured versions of 8 challengers (three challengers have no parameters). Every algorithm was run 25 times on each problem instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the penalized 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 find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

| Solvers | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| | 26.13 | 0.06 | 2.68 | 119.75 | 1731.16 | 994.94 |
| ANOV[D] | **0.02** | **0.04** | 0.12 | **0.54** | 296.84 | 0.50 |
| [33] | 99.6% | **100%** | **100%** | 98.2% | 90.1% | 83.4% |
| | 514.29 | **0.05** | 3.64 | 98.70 | 617.83 | 1084.60 |
| G2[D] | 0.03 | 0.05 | 0.15 | 0.75 | 110.42 | 0.58 |
| [50] | 91.4% | **100%** | **100%** | 99.1% | 97.8% | 81.4% |
| | 417.33 | 0.22 | 8.87 | 68.24 | 5478.75 | 2195.76 |
| GNOV[D] | 0.03 | 0.09 | 0.17 | 0.62 | N/A | 0.19 |
| [62] | 92.9% | **100%** | **100%** | 99.4% | 0.3% | 61.8% |
| | 68.06 | 0.70 | 1.91 | 64.48 | 22.01 | 1925.56 |
| PAWS[D] | **0.02** | 0.35 | **0.09** | 0.83 | 10.39 | 0.50 |
| [71] | 99.2% | **100%** | **100%** | 99.4% | **100%** | 67.7% |
| | 75.06 | 0.15 | 13.85 | 141.61 | 336.27 | 1223.83 |
| RANOV[D] | 0.1 | 0.12 | 0.24 | 0.77 | 95.53 | 0.47 |
| [61] | 98.9% | **100%** | **100%** | 98.1% | 100% | 80.4% |
| | 868.37 | 0.19 | 1.32 | 42.99 | 12.17 | 67.59 |
| RSAPS[D] | 0.04 | 0.15 | 0.11 | 0.64 | 7.86 | **0.02** |
| [42] | 85.2% | **100%** | **100%** | 99.5% | **100%** | 99.0% |
| | 27.69 | 0.31 | 1.54 | **31.77** | **10.68** | 62.63 |
| SAPS[D] | 0.06 | 0.21 | 0.16 | 0.75 | **7.00** | **0.02** |
| [42] | 99.8% | **100%** | **100%** | 99.6% | **100%** | 99.0% |
| | **0.33** | 417.71 | **1.26** | 57.44 | 32.38 | **16.45** |
| VW[D] | **0.02** | 8.43 | 0.15 | 1.00 | 17.60 | **0.02** |
| [64] | **100%** | 94.8% | **100%** | 99.6% | 100% | **100%** |

outperformed the respective best challengers for each distribution on a per-instance basis. R3SAT was an exception: PAWS outperformed SATenstein-LS[R3SAT] most frequently (77.2%), but still achieved a lower PAR-10 score, indicating that SATenstein-LS[R3SAT] achieved dramatically better performance than PAWS on a relatively small number of hard instances.

## Appendix D. Performance comparison with configured challengers

Table 15 summarizes the performance of configured challengers, and Fig. 2 shows the PAR-10 ratios of SATenstein-LS solvers over the default and configured challengers. Compared to challengers with default configurations (see Table 7), the specifically optimized versions of the challenger solvers often achieved significantly better performance, reducing their performance gaps to SATenstein-LS solvers. For example, automatic configuration of G2 led to a speedup of 5 orders of magnitude in terms of PAR-10 on SWGCP and solved 100% of the instances in that benchmark set within a 600 second cutoff (vs. 31% for G2 default). However, it is worth noting that the configured challengers sometimes also exhibited worse performance than the default configurations (in the worst case, VW[SWGCP] was 2.58 times slower than VW default in terms of PAR-10 with a cutoff of 600 CPU seconds). This was caused by the short cutoff time used during the configuration process, as motivated in Section 5.2; had we used the same 5 CPU second cutoff time for computing PAR-10 (recall that

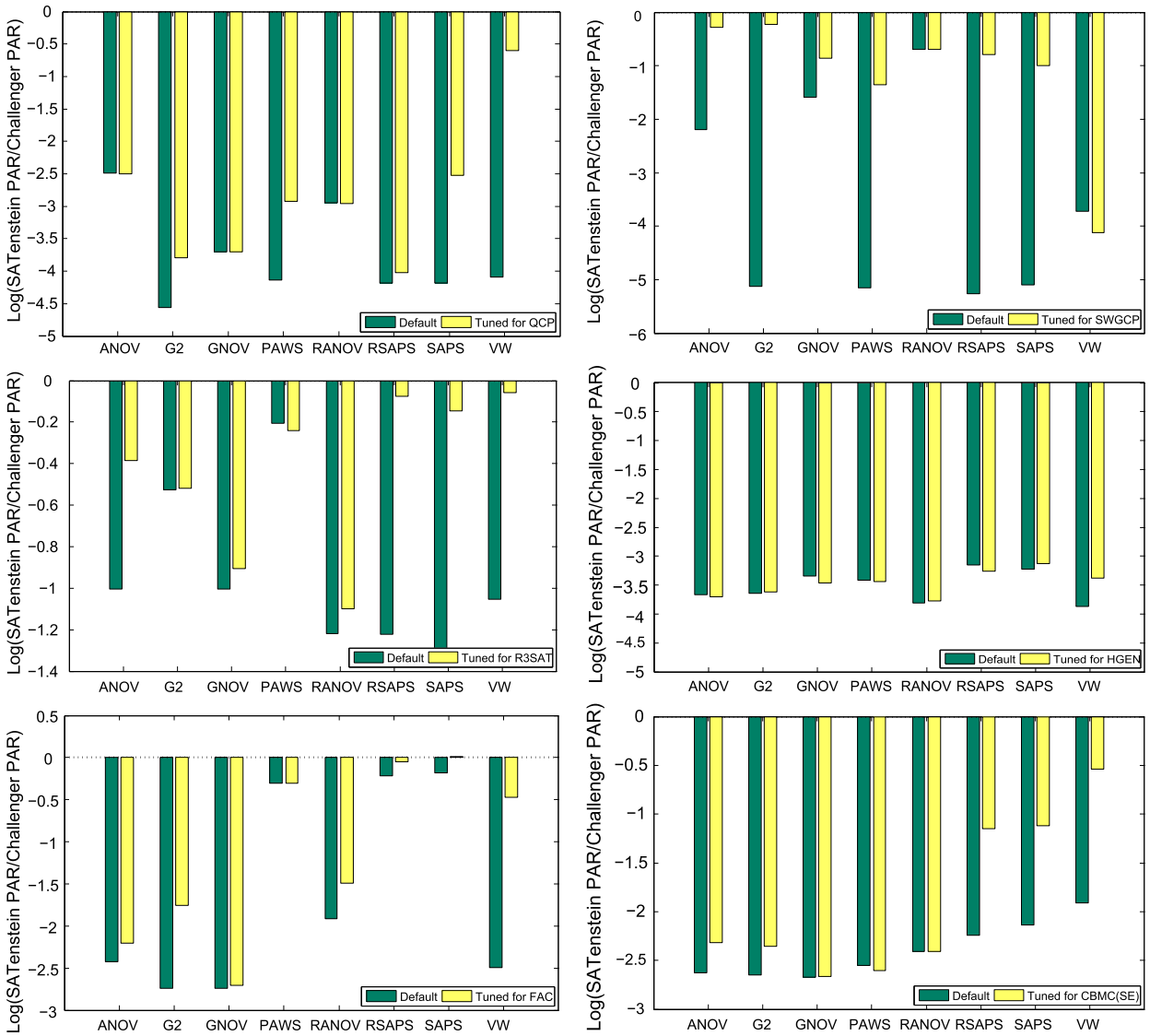


**Fig. 2.** Performance of `SATenstein-LS` solvers *vs* challengers with default and optimized configurations. For every benchmark distribution `D`, the base-10 logarithm of the ratio between `SATenstein[D]` and one challenger (default and optimized) is shown on the y-axis, based on data from Tables 7 and 15. Top-left: QCP; Top-right: SWGCP; Middle-left: R3SAT; Middle-right: HGEN; Bottom-left: FAC; Bottom-right: CBMC(SE).

we used a cutoff time of 5 CPU seconds for every ParamILS tuning experiment, and we always computed the PAR-10 of the test performance based on a cutoff of 600 CPU seconds), the configured challengers would have always outperformed the default versions.

Examining benchmark distributions individually and ranging over our 8 challengers, we observed average and median speedups over default configurations of 396 and 3.58 (for `QCP`), 15 900 and 3240 (for `SWGCP`), 5.84 and 2.74 (for `R3SAT`), 1.23 and 1.01 (`HGEN`), 15.4 and 1.61 (`FAC`), 6.61 and 2.00 (`CBMC(SE)`). We were surprised to observe only small speedups for all challengers on `HGEN`. Considering challengers individually and ranging over our 6 benchmark distributions, average and median PAR-10 improvement was 15.0 and 1.85 (for `ANOV`), 13 200 and 3.84 (for `G2`), 1.74 and 1.05 (for `GNOV`), 1070 and 0.98 (for `PAWS`), 1.33 and 1.03 (for `RANOV`), 4870 and 6.85 (for `RSAPS`), 2080 and 12.3 (for `SAPS`), 539 and 16.6 (for `VW`). `RANOV` showed the smallest performance improvement as a result of automated configuration across all benchmarks; this is likely due to `RANOV`'s small parameter space (it has only one parameter).

## Appendix E. `SATenstein-LS` parameter configurations found

Tables 16 and 17 present the `SATenstein-LS` and `SATenstein-LS2.0` parameter configurations found for each distribution considered in this paper; we omit inactive parameters. In what follows, we describe these parameter configurations in detail.

**Table 16**
SATenstein-LS parameter configuration found for each distribution.

| Distribution | Parameter configuration |
|---|---|
| QCP | -useAdaptiveMechanism 0 -performSearchDiversification 1 -usePromisingList 0 -singleClauseAsNeighbor 1 -adaptWalkProb 0 -selectClause 1 -useClausePenalty 0 -performTabuSearch 0 -heuristic 4 -performAlternateNovelty 0 -searchDiversificationStrategy 3 -dp 0.07 -c 0.0001 -novNoise 0.5 -rfp 0.1 -s 0.1 |
| SW-GCP | -useAdaptiveMechanism 0 -performSearchDiversification 0 -usePromisingList 0 -singleClauseAsNeighbor 1 -adaptWalkProb 0 -selectClause 1 -useClausePenalty 0 -performTabuSearch 0 -heuristic 3 -performAlternateNovelty 0 -dp 0.01 -c 0.01 -novNoise 0.1 -s 0.1 |
| R3SAT | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 0 -scoringMeasure 3 -tieBreaking 2 -useClausePenalty 1 -searchDiversificationStrategy 1 -smoothingScheme 1 -tabuLength 3 -performTabuSearch 1 -alpha 1.189 -ps 0.1 -rho 0.8 -sapsthresh -0.1 -rwp 0.05 -wp 0.01 |
| HGEN | -useAdaptiveMechanism 0 -performSearchDiversification 1 -usePromisingList 0 -singleClauseAsNeighbor 1 -tabuLength 3 -performTabuSearch 1 -useClausePenalty 0 -searchDiversificationStrategy 4 -adaptWalkProb 0 -selectClause 1 -heuristic 7 -c 0.001 -rfp 0.15 -s 0.1 -wpWalk 0.1 |
| FAC | -useAdaptiveMechanism 0 -performSearchDiversification 0 -singleClauseAsNeighbor 0 -scoringMeasure 3 -tieBreaking 1 -useClausePenalty 1 -smoothingScheme 1 -tabuSearch 0 -alpha 1.189 -ps 0.066 -rho 0.83 -sapsthresh -0.3 -wp 0.03 |
| CBMC(SE) | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 0 -useClausePenalty 1 -smoothingScheme 1 -performTabuSearch 0 -searchDiversificationStrategy 4 -scoringMeasure 3 -tieBreaking 2 -alpha 1.066 -ps 0 -rho 0.83 -sapsthresh -0.3 -wp 0.01 -rfp 0.1 |

**Table 17**
SATenstein-LS2.0 parameter configuration found for each distribution.

| Distribution | Parameter configuration |
|---|---|
| QCP | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 1 -usePromisingList 0 -selectClause 1 -useClausePenalty 0 -searchDiversificationStrategy 3 -performTabuSearch 0 -heuristic 5 -novNoise 0.3 -rfp 0.07 |
| SW-GCP | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 1 -usePromisingList 0 -searchDiversificationStrategy 2 -selectClause 1 -useClausePenalty 0 -performTabuSearch 0 -heuristic 1 -performAlternateNovelty 0 -rdp 0.01 -novNoise 0.1 |
| R3SAT | -useAdaptiveMechanism 1 -performSearchDiversification 0 -singleClauseAsNeighbor 0 usePromisingList 0 -scoringMeasure 3 -tieBreaking 1 -useClausePenalty 1 -smoothingScheme 1 -performTabuSearch 0 -alpha 1.126 -rho 0.17 -sapsthresh -0.1 -wp 0.03 |
| HGEN | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 1 -usePromisingList 0 -performTabuSearch 1 -tabuLength 3 -useClausePenalty 0 -searchDiversificationStrategy 2 -selectClause 1 -heuristic 7 -rdp 0.07 -wpWalk 0.1 |
| FAC | -useAdaptiveMechanism 0 -performSearchDiversification 0 -singleClauseAsNeighbor 0 -usePromisingList 0 -scoringMeasure 3 -tieBreaking 3 -useClausePenalty 1 -smoothingScheme 1 performTabuSearch 0 -alpha 1.126 -ps 0.033 -rho 0.8 -sapsthresh -0.1 -wp 0.04 |
| CBMC(SE) | -useAdaptiveMechanism 0 -performSearchDiversification 0 -singleClauseAsNeighbor 1 usePromisingList 0 -useClausePenalty 0 -performTabuSearch 0 -selectClause 1 -heuristic 8 -c 0.0001 s -0.001 -wpwalk 0.1 |

SATenstein-LS2.0[QCP] uses building blocks 1, 2, and 5. Recall that block 1 is used for performing search diversification, and block 5 is used to update data structures, tabu attributes and clause penalties. In block 2, which is used to instantiate a solver belonging to the WalkSAT architecture, the *heuristic* is based on R-Novelty, and in block 1, *diversification* flips the least-frequently-flipped variable from an UNSAT clause. This configuration is the same as SATenstein-LS[QCP] at the block level but differs in the employed heuristic and search diversification strategy. SATenstein-LS2.0[SW-GCP] is similar to SATenstein-LS2.0[QCP], but flips the least-recently-flipped variable in block 1 and uses a different heuristic (Novelty). Among the challengers, SATenstein-LS2.0[SW-GCP] is closest to RANOV. The main difference to SATenstein-LS[SW-GCP] is that SATenstein-LS[SW-GCP] did not use block 1. Unlike SATenstein-LS[R3SAT], SATenstein-LS2.0[R3SAT] does not use any search diversification and only uses blocks 3 (used to instantiate dynamic local search algorithms) and 5, but both configurations are closest to SAPS. SATenstein-LS2.0[HGEN] uses blocks 1, 2, and 5. It is similar to SATenstein-LS2.0[QCP] but uses a *heuristic* based on VW1 as well as a tabu list of length 3 and mainly differs from SATenstein-LS[HGEN] in the search diversification strategy. SATenstein-LS2.0[FAC] is also very similar to SATenstein-LS[FAC] with a different tie-breaking scheme. SATenstein-LS2.0[CBMC(SE)]

uses blocks 2, and 5 and is closest to VW, a WalkSAT algorithm. This configuration is very different from what we found in SATenstein-LS[CBMC(SE)], which is a dynamic local search algorithm.

To summarize, we found that in most cases, the augmented solver found configurations that were similar but not identical to their SATenstein-LS counterparts. This indicates that the distributions we studied give rise to local search design spaces having "good regions" in which multiple, related configurations can perform well. The key exception was the case of CBMC(SE): here, we observed a substantial difference between the augmented solver configuration and SATenstein-LS[CBMC(SE)], underscoring the richness of the design space.

## References


[1] C. Ansotegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of solvers, in: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP'09, 2009, pp. 142–157.
[2] G. Audemard, L. Simon, Predicting learnt clauses quality in modern sat solvers, in: Proceedings of the 21st International Joint Conference on Artifical Intelligence, IJCAI'09, 2009, pp. 399–404.
[3] F. Bacchus, J. Winter, Effective preprocessing with hyper-resolution and equality reduction, in: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, SAT'03, 2003, pp. 341–355.
[4] P. Balaprakash, M. Birattari, T. Stützle, Improvement strategies for the f-race algorithm: sampling design and iterative refinement, in: Proceedings of the Fourth International Workshop on Hybrid Metaheuristics, MH'07, 2007, pp. 108–122.
[5] A. Balint, A. Fröhlich, Improving stochastic local search for sat with a new probability distribution, in: Theory and Applications of Satisfiability Testing—SAT 2010, Springer, 2010, pp. 10–15.
[6] A. Biere, Picosat version 535, solver description, SAT competition 2007, http://www.satcompetition.org/2007/picosat.pdf, 2007, last accessed on Sept. 16, 2013.
[7] A. Biere, Picosat essentials, J. Satisf. Boolean Model. Comput. 4 (2008) 75–97.
[8] A. Biere, M.J.H. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2009.
[9] M. Birattari, T. Stützle, L. Paquete, K. Varrentrapp, A racing algorithm for configuring metaheuristics, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002, 2002, pp. 11–18.
[10] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, Empirical Methods for the Analysis of Optimization Algorithms, Springer-Verlag, 2010, pp. 311–336, Ch. F-race and iterated F-race: an overview.
[11] S. Cai, K. Su, Configuration checking with aspiration in local search for sat, in: Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence, AAAI'12, 2012, pp. 434–440.
[12] T. Carchrae, J.C. Beck, Applying machine learning to low knowledge control of optimization algorithms, Comput. Intell. 21 (4) (2005) 373–387.
[13] M. Chiarandini, C. Fawcett, H.H. Hoos, A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract), in: Proceedings of the Seventh International Conference for the Practice and Theory of Automated Timetabling, PATAT'2008, 2008.
[14] E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2004, 2004, pp. 168–176.
[15] R. Dechter, I. Rish, Directional resolution: the Davis–Putnam procedure, revisited, in: Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning, KR'94, 1994, pp. 134–145.
[16] O. Dubois, G. Dequen, A backbone-search heuristic for efficient solving of hard 3-SAT formulae, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI'01, 2001, pp. 248–253, last accessed on September 16, 2013, http://www.laria.u-picardie.fr/~dequen/sat/cnfs_ijcai01.ps.gz.
[17] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT'05, 2005, pp. 61–75.
[18] A.S. Fukunaga, Automated discovery of composite SAT variable-selection heuristics, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence, AAAI'02, 2002, pp. 641–648.
[19] A.S. Fukunaga, Evolving local search heuristics for SAT using genetic programming, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2004, 2004, pp. 483–494.
[20] M. Gagliolo, J. Schmidhuber, Learning dynamic algorithm portfolios, Ann. Math. Artif. Intell. 47 (3–4) (2006) 295–328.
[21] L.D. Gaspero, A. Schaerf, Easysyn++: a tool for automatic synthesis of stochastic local search algorithms, in: Proceedings of the International Workshop on Engineering Stochastic Local Search Algorithms, SLS 2007, 2007, pp. 177–181.
[22] I.P. Gent, H.H. Hoos, P. Prosser, T. Walsh, Morphing: combining structure and randomness, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI'99, 1999, pp. 654–660.
[23] C.P. Gomes, B. Selman, Problem structure in the presence of perturbations, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI'97, 1997, pp. 221–226.
[24] C.P. Gomes, B. Selman, Algorithm portfolios, Artif. Intell. 126 (1–2) (2001) 43–62.
[25] J. Gratch, G. Dejong, COMPOSER: a probabilistic solution to the utility problem in speed-up learning, in: Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI'92, 1992, pp. 235–240.
[26] A. Guerri, M. Milano, Learning techniques for automatic algorithm portfolio selection, in: Proceedings of the Sixteenth European Conference on Artificial Intelligence, ECAI-04, 2004, pp. 475–479.
[27] Y. Hamadi, E. Monfroy, F. Saubion, An introduction to autonomous search, in: Autonomous Search, Springer, 2012, pp. 1–11.
[28] M. Heule, H.V. Maaren, Improved version of march_ks, http://www.st.ewi.tudelft.nl/sat/Sources/stable/march_pl, 2007, last accessed on Sept. 16, 2013.
[29] M. Heule, H.V. Maaren, March_ks, solver description, SAT competition 2007, http://www.satcompetition.org/2007/march_ks.pdf, 2007, last accessed on Sept. 16, 2013.
[30] M.J. Heule, O. Kullmann, S. Wieringa, A. Biere, Cube and conquer: guiding CDCL SAT solvers by lookaheads, in: Hardware and Software: Verification and Testing, in: Lecture Notes in Computer Science, vol. 7261, Springer, 2012, pp. 50–65.
[31] 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, last accessed on Sept. 16, 2013.
[32] H.H. Hoos, On the run-time behaviour of stochastic local search algorithms for SAT, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI'99, 1999, pp. 661–666.
[33] H.H. Hoos, An adaptive noise mechanism for WalkSAT, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence, AAAI'02, 2002, pp. 655–660.
[34] H.H. Hoos, Computer-aided design of high-performance algorithms, Tech. rep., University of British Columbia, Department of Computer Science, 2008, last accessed on Sept. 16, 2013, http://people.cs.ubc.ca/~hoos/tmp/tn-alg-design.pdf.

[35] H.H. Hoos, Programming by optimization, Commun. ACM 55 (2) (2012) 70–80.
[36] F. Hutter, D. Babić, H.H. Hoos, A.J. Hu, Boosting verification by automatic tuning of decision procedures, in: Proceedings of the Seventh International Conference on Formal Methods in Computer Aided Design, FMCAD'07, 2007, pp. 27–34.
[37] F. Hutter, H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: Proceedings of the 5th Learning and Intelligent Optimization Conference, LION'11, 2011, pp. 507–523.
[38] F. Hutter, H.H. Hoos, K. Leyton-Brown, Automated configuration of mixed integer programming solvers, in: Proc. of CPAIOR-10, 2010, pp. 186–202.
[39] F. Hutter, H.H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework, J. Artif. Intell. Res. (2009), accepted for publication.
[40] F. Hutter, H.H. Hoos, T. Stützle, Automatic algorithm configuration based on local search, in: Proceedings of the Twenty Second National Conference on Artificial Intelligence, AAAI'07, 2007, pp. 1152–1157.
[41] F. Hutter, H.H. Hoos, T. Stützle, K. Leyton-Brown, ParamILS version 2.3, http://www.cs.ubc.ca/labs/beta/Projects/ParamILS, 2008, last accessed on Sept. 16, 2013.
[42] F. Hutter, D.A.D. Tompkins, H.H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in: Eighth International Conference on Principles and Practice of Constraint Programming, CP'02, 2002, pp. 233–248.
[43] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm selection and scheduling, in: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming, CP'11, in: LNCS, vol. 6876, 2011, pp. 454–469.
[44] S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, ISAC—instance specific algorithm configuration, in: Proceedings of the 19th European Conference on Artificial Intelligence, ECAI'10, 2010, pp. 751–756.
[45] A.R. KhudaBukhsh, L. Xu, H.H. Hoos, K. Leyton-Brown, SATenstein: automatically building local search SAT solvers from components, in: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, IJCAI'09, 2009, pp. 517–524.
[46] L. Kroc, A. Sabharwal, C.P. Gomes, B. Selman, Integrating systematic and local search paradigms: a new strategy for maxSAT, in: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, IJCAI'09, 2009, pp. 544–551.
[47] F. Lardeux, F. Saubion, J.-K. Hao, Gasat: a genetic local search algorithm for the satisfiability problem, Evol. Comput. 14 (2) (2006) 223–253.
[48] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, A portfolio approach to algorithm selection, in: International Joint Conferences on Artificial Intelligence, IJCAI, 2003, pp. 1542–1543.
[49] K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the empirical hardness of optimization problems: the case of combinatorial auctions, in: Eighth International Conference on Principles and Practice of Constraint Programming, CP'02, 2002, pp. 556–572.
[50] C.M. Li, W. Huang, Diversification and determinism in local search for satisfiability, in: Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT'05, 2005, pp. 158–172.
[51] C.M. Li, Y. Li, Satisfying versus falsifying in local search for satisfiability, in: Theory and Applications of Satisfiability Testing—SAT 2012, Springer, 2012, pp. 477–478.
[52] C.M. Li, W. Wei, H. Zhang, Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition, 2007.
[53] C.M. Li, W.X. Wei, H. Zhang, Combining adaptive noise and look-ahead in local search for SAT, in: Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing, SAT'07, 2007, pp. 121–133.
[54] J. Maturana, F. Lardeux, F. Saubion, Autonomous operator management for evolutionary algorithms, J. Heuristics 16 (6) (2010) 881–909.
[55] D. McAllester, B. Selman, H. Kautz, Evidence for invariants in local search, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI'97, 1997, pp. 321–326.
[56] S. Minton, An analytic learning system for specializing heuristics, in: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, IJCAI'93, 1993, pp. 922–929.
[57] J. Monette, Y. Deville, P. Van Hentenryck, Aeon: synthesizing scheduling algorithms from high-level models, in: Proceedings of the Eleventh INFORMS Computing Society Conference, 2009, pp. 43–59.
[58] M.A. Montes de Oca, T. Stützle, M. Birattari, M. Dorigo, Frankenstein's PSO: an engineered composite particle swarm optimization algorithm, IEEE Trans. Evol. Comput. 13 (5) (2009) 1120–1132.
[59] E. Nudelman, K. Leyton-Brown, G. Andrew, C. Gomes, J. McFadden, B. Selman, Y. Shoham, Satzilla 0.9. Solver description, SAT competition, 2003.
[60] M. Oltean, Evolving evolutionary algorithms using linear genetic programming, Evol. Comput. 13 (3) (2005) 387–410.
[61] D.N. Pham, Anbulagan, Resolution enhanced SLS solver: R+AdaptNovelty+. Solver description, SAT competition, 2007.
[62] D.N. Pham, J. Thornton, C. Gretton, A. Sattar, Combining adaptive and dynamic local search for satisfiability, J. Satisf. Boolean Model. Comput. 4 (2008) 149–172.
[63] P.C. Pop, S. Iordache, A hybrid heuristic approach for solving the generalized traveling salesman problem, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 2011, pp. 481–488.
[64] S. Prestwich, Random walk with continuously smoothed variable weights, in: Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT'05, 2005, pp. 203–215.
[65] J.R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118.
[66] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI'94, 1994, pp. 337–343.
[67] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI'92, 1992, pp. 440–446.
[68] L. Simon, SAT competition random 3CNF generator, www.satcompetition.org/2003/TOOLBOX/genAlea.c, 2002, last accessed on Sept. 16, 2013.
[69] N. Sörensson, N. Eén, Minisat2007, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat, 2007, last accessed on Sept. 16, 2013.
[70] S. Subbarayan, D. Pradhan, NiVER: non-increasing variable elimination resolution for preprocessing SAT instances, in: Lecture Notes in Computer Science, vol. 3542, Springer, 2005, pp. 276–291.
[71] J. Thornton, D.N. Pham, S. Bain, V. Ferreira, Additive versus multiplicative clause weighting for SAT, in: Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI'04, 2004, pp. 191–196.
[72] D.A. Tompkins, A. Balint, H.H. Hoos, Captain Jack – new variable selection heuristics in local search for SAT, in: Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing, 2011, pp. 302–316.
[73] D.A. Tompkins, H.H. Hoos, Dynamic scoring functions with variable expressions: new SLS methods for solving SAT, in: Theory and Applications of Satisfiability Testing—SAT 2010, Springer, 2010, pp. 278–292.
[74] D.A.D. Tompkins, H.H. Hoos, UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT, in: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT'04, 2004, pp. 37–46.
[75] T. Uchida, O. Watanabe, Hard SAT instance generation based on the factorization problem, http://www.is.titech.ac.jp/~watanabe/gensat/a2/GenAll.tar.gz, 1999.
[76] B. Wah, Z. Wu, Penalty formulations and trap-avoidance strategies for solving hard satisfiability problems, J. Comput. Sci. Technol. 20 (1) (2005) 3–17.

[77] S.J. Westfold, D.R. Smith, Synthesis of efficient constraint-satisfaction programs, Knowl. Eng. Rev. 16 (1) (2001) 69–84.
[78] L. Xu, H.H. Hoos, K. Leyton-Brown, Hydra: automatically configuring algorithms for portfolio-based selection, in: AAAI, 2010, pp. 210–216.
[79] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT, J. Artif. Intell. Res. 32 (2008) 565–606.
[80] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla2009: an automatic algorithm portfolio for SAT. Solver description, SAT competition, 2009.