Captain Jack: New Variable Selection Heuristics in Local Search for SAT

Dave A.D. Tompkins[1], Adrian Balint[2], and Holger H. Hoos[1]

[1] Department of Computer Science
University of British Columbia, Canada
{davet,hoos}@cs.ubc.ca
[2] Institute of Theoretical Computer Science,
Ulm University, Germany
adrian.balint@uni-ulm.de

Abstract. Stochastic local search (SLS) methods are well known for their ability to find models of randomly generated instances of the propositional satisfiability problem (SAT) very effectively. Two well-known SLS-based SAT solvers are SPARROW, one of the best-performing solvers for random 3-SAT instances, and VE-SAMPLER, which achieved significant performance improvements over previous SLS solvers on SAT-encoded software verification problems. Here, we introduce a new highly parametric algorithm, CAPTAIN JACK, which extends the parameter space of SPARROW to incorporate elements from VE-SAMPLER and introduces new variable selection heuristics. CAPTAIN JACK has a rich design space and can be configured automatically to perform well on various types of SAT instances. We demonstrate that the design space of CAPTAIN JACK is easy to interpret and thus facilitates valuable insight into the configurations automatically optimized for different instance sets. We provide evidence that CAPTAIN JACK can outperform well-known SLS-based SAT solvers on uniform random k-SAT and 'industrial-like' random instances.

1 Introduction

The propositional satisfiability problem (SAT) is one of the most prominent problems in computer science, not only because it is a prototypical $\mathcal{NP}$-complete problem, but also because of its simplicity, expressiveness and practical relevance. Problem instances from domains such as software verification can be easily encoded into SAT, and there is much interest in developing SAT solvers that can solve these practical problems effectively. There is also much interest in random instances; they have been frequently studied and are underlying one of the three categories in the SAT competition.

Two popular approaches for solving SAT are conflict driven clause learning (CDCL) and stochastic local search (SLS), and in this work we focus on the latter. SLS solvers are usually incomplete, *i.e.*, they cannot determine with certainty that a given propositional formula is unsatisfiable. SLS algorithms for SAT typically start by randomly assigning to every variable appearing in a given formula a value of either true or false. Then, in each subsequent search step a variable is selected to have its truth assignment *flipped* from true to false or vice versa.

K.A. Sakallah and L. Simon (Eds.): SAT 2011, LNCS 6695, pp. 302–316, 2011.

Because SLS is the most effective approach for solving random satisfiable instances, there has been much interest in studying the performance of SLS-based solvers on random instances from the so-called uniform random k-SAT distribution [6], especially 3-SAT instances at, or near, the solubility phase transition, where there is an equal probability of generating a satisfiable or unsatisfiable instance [12]. One of the best known SLS solvers for large random 3-SAT instances near the phase transition is SPARROW [3], which replaced the variable selection heuristic in GNOVELTY$^+$ [13] with a probabilistic distribution-based mechanism. We will describe SPARROW in Section 2.1.

While SLS is currently not competitive with CDCL on large satisfiable industrial instances, there has been some recent success in closing the gap. The VE-SAMPLER algorithm [18] was able to achieve a significant improvement in this area by solving the CBMC software verification benchmark instances over ten times faster than the previous best known SLS algorithm SATENSTEIN-LS [10]. In Section 2.2, we will describe some of the specifics of VE-SAMPLER.

SATENSTEIN-LS and VE-SAMPLER are examples of *highly parametric* algorithms that are designed to be *automatically* configured, using an automated algorithm configurator that takes as inputs an algorithm, its configuration space and an instance set and then attempts to find the best-performing configuration of the algorithm on the given instance set. Following this approach, which is a prominent special case of computer-aided algorithm design [7], the traditional role of an algorithm designer is redefined to be more focused on constructing rich and interesting *spaces* of algorithms.

It is well established that SAT instances drawn from different sources and distributions have different characteristics, and the efficacy of a solver on an instance often depends on those characteristics. Portfolio-based approaches, such as SATZILLA [19], exploit this phenomenon by selecting one or more solvers to be used for solving a given instance based on characteristics of that instance. One of our goals in this work was to create a highly parametric algorithm that can help explore the algorithmic differences (or similarities) between configurations that achieve good results on different types of instances (*e.g.*, random *vs* application or random 3-SAT *vs* 7-SAT). To achieve this goal, our algorithm should have a parameter space that is not only easy to understand, but also contains configurations that achieve good performance on a wide variety of instance types; without good performance, the resulting configurations are not very meaningful, and without intuitively understandable configurations, it is difficult to draw conclusions from automatically optimized configurations. In Section 4, we introduce CAPTAIN JACK, a new, highly parametric algorithm that attempts to strike a balance between these two objectives. It was named for the fictional pirate Captain Jack Sparrow, because it incorporates elements from SPARROW, as well as because it can achieve good performance on a wide variety of instances, and is hence a *jack-of-all-trades*.

In Section 6, we present evidence that CAPTAIN JACK does achieve good performance on nine different instance sets, and is now the best known SLS-based SAT algorithm for large random 3-SAT instances and a class of recently proposed 'industrial-like' instances [1]. Moreover, we show how the resulting configurations found by CAPTAIN JACK provide interesting insight into the configurations found for each of the instance sets; for example, we found evidence that the CAPTAIN JACK configuration

optimized for the previously mentioned industrial-like instances exhibits characteristics consistent with those obtained for some practical software verification instances.

Additional information and experimental data, including source code and instance sets, can be found at `www.cs.ubc.ca/research/captain-jack`.

2 Background

Throughout this work, we use the approach for modeling and representing SLS algorithms introduced by Tompkins and Hoos [18], according to which each search step involves three heuristic stages. First, the variables are *filtered* so that only a subset of variables are considered as flip candidates. Next, the candidates are evaluated according to one or more *variable expressions (VEs)*, where each VE is a mathematical expression that can include *properties* of the variables. Finally, once the VEs have been evaluated, a *variable selection mechanism (VSM)* is employed to select the variable to be flipped. A *controller* determines for each search step which filter, VEs and VSM are used.

Some variable properties are defined via a VE that contains other properties, such as *score*, which is equivalent to the VE $\langle make - break \rangle$, where the properties *make* and *break* measure the number of clauses that would become satisfied and unsatisfied, respectively, if the variable were to be flipped. The value of a property can depend on the specific context in which the variable is selected and additional state information of the algorithm. For example, algorithms with dynamic clause penalties, such as PAWS [15] and GNOVELTY$^+$ [13], use a *penalized* property *penScore*, whose value depends on the full variable assignment and on the clause penalties (weights).

Variable properties can be loosely classified as either *greedy* properties, which tend to increase the number of satisfied clauses during search, such as *make* and *break*, or *diversification* properties, which tend to better explore the search space and avoid local minima, such as *age* and *flips*. The *age* property is defined as the number of search steps that have occurred since the given variable was last flipped. The *flips* property (*a.k.a. flipcount*) measures how many times a variable has been flipped. In Section 4, we will describe several new greedy and diversification properties.

2.1 SPARROW

The SPARROW algorithm [3], named after the city of Ulm's mascot, is based on the GNOVELTY$^+$ algorithm [13]. GNOVELTY$^+$ combines a clause penalty-based scheme similar to PAWS [15] with the promising variable scheme of G^{2}WSAT [11] (see [13] and the GNOVELTY$^+$ source code, version 1.2, from the 2009 SAT Competition for more details). The behaviour of SPARROW differs from GNOVELTY$^+$ only when there are no (penalized) promising variables, in which case a novel VE and a probabilistic VSM is used instead of the NOVELTY-based component in GNOVELTY$^+$. The VE used by SPARROW is $\langle sparrowScore \cdot sparrowAge \rangle$, where *sparrowScore* and *sparrowAge* are defined as follows[1]:

$$sparrowScore = \begin{cases} c_b^{penScore} & \text{if } penScore < 0 \\ 1 & \text{otherwise} \end{cases}, \qquad sparrowAge = 1 + \left(\frac{age}{c_d}\right)^{c_e}.$$

[1] The definition and notation we use differs slightly from the published version of SPARROW [3], but accurately reflects the source code implementation.

SPARROW uses a *distribution-based VSM*, where each variable is selected proportionally to the VE $\langle sparrowScore \cdot sparrowAge \rangle$. When we use a similar approach to select an element from a set where the elements are assigned fixed weights, we will refer to it as *weighted selection*.

The full parameter space for SPARROW includes the three constants mentioned above and the smoothing probability (ps) inherited from GNOVELTY$^+$. Balint and Fröhlich proposed $(c_b, c_d, c_e, ps) := (2, 4, 10^5, 0.347)$ as a good configuration for large 3-SAT instances, which was found by manual tuning on selected 3-SAT competition benchmark instances [3].

2.2 VE-SAMPLER and PARAMILS

VE-SAMPLER [18] was developed to demonstrate the power of using new and innovative variable properties and VEs in SLS algorithms. It was inspired by the VW2 algorithm [14], which was observed to be very effective on the CBMC software verification instances with the VE $\langle break + c \cdot flips \rangle$. Based on the WALKSAT architecture, VE-SAMPLER uses the selection of an unsatisfied clause as a filter. In each search step, a controller selects one of six VEs using weighted selection; one of these VEs corresponds to a (freebie) random walk step, and the remaining five are all of the general form:

$$\|greedy\|^{a_1} + \mathrm{clw}(s, m, l) \cdot \|diversification\|^{a_2},$$

where $\|p\|$ indicates a property p that is normalized to values between zero and one, and $\mathrm{clw}(s, m, l)$ is a simple mechanism that selects between three coefficients (s, m, l) depending on whether the clause length is respectively less than, equal to, or greater than three. Each VE uses one greedy property (chosen from a set of five) and one diversification property (chosen from a set of thirteen), except for one VE that uses two greedy properties (see [18] for details). VE-SAMPLER uses *maximum VSM*, where the variable with the maximum evaluated VE is selected. At the time of this writing, VE-SAMPLER is the fastest SLS-based SAT solver on the CBMC and SWV software verification instances [18]; it has subsequently been shown to have good performance on random 3-SAT instances [16], although on these, it does not reach the performance of SPARROW or SATENSTEIN-LS.

The configuration space of VE-SAMPLER is enormous, with over 10^{50} unique configurations. The configurator used on VE-SAMPLER was PARAMILS [9,8], an SLS-based procedure that searches the configuration space of a given algorithm. The primary inputs to PARAMILS are a target solver (binary), a set of target (training) instances, a solver cutoff time, an evaluation function and a configuration file that specifies the configuration space (each solver parameter along with a set of possible values). The evaluation function used was the penalized average runtime (PAR-10), where instances not solved within the cutoff are counted as ten times the given cutoff time. The primary output of PARAMILS is the best configuration of the target algorithm that PARAMILS found by the search process, *i.e.*, the configuration that achieves the best PAR-10 performance on the instances in the given set.

To ensure that the results from PARAMILS generalize to instances other than those used during the optimization process, we use a set of test instances to report final results that is disjoint from the training set used when running PARAMILS.

3 Design Considerations underlying CAPTAIN JACK

Many SLS for SAT algorithms switch between greedy (intensification) steps and diversification steps, or use diversification properties as tie-breakers in greedy steps. SPARROW and VE-SAMPLER have *mixed steps* that combine a greedy and a diversification property in a VE; the likelihood of a variable being selected can be dominated by the greedy property, the diversification property or neither of them. CAPTAIN JACK allows for all three types of steps (greedy, diversification and mixed), and we introduce parameters to control the balance between the three. Mixed steps are rarely used in SLS algorithms for SAT, and we were interested in observing the proportion of mixed steps PARAMILS would select, and how that proportion would depend on the target instance set. We were also curious whether PARAMILS would select a distribution-based VSM as in SPARROW or a maximum VSM as in VE-SAMPLER; therefore, CAPTAIN JACK supports both types of VSMs.

One of the objectives of our earlier work had been to encourage the use of more diverse variable properties [18]; this was achieved in VE-SAMPLER by means of categorical parameters that select properties from a given set. One problem with this approach is that a configuration only has a few selected properties, which may include duplicates, and it is hard to assess the viability of each individual property. To help avoid this problem in CAPTAIN JACK, each property has a parameterized weight that controls how frequently it is selected. This makes it easier to assess which properties are important for a given instance set. It also renders CAPTAIN JACK an excellent framework for introducing and testing new variable properties; after these are added to the configuration space, the automated configurator can gradually introduce them into configurations by means of modifying their weights relative to other properties.

One significant departure in CAPTAIN JACK from SPARROW is the absence of penalized clause weights. In preliminary experiments, we observed that with penalized clauses the proportion of (greedy) promising steps was significantly higher, reducing the impact of the core CAPTAIN JACK components we were interested in exploring. In addition, we found the penalized promising variable mechanism as implemented in GNOVELTY$^+$ (and, hence, SPARROW) problematic and memory intensive for large instances.

Finally, when designing a highly parametric algorithm with the aim of configuring it automatically, it is important to understand the limitations of the automatic algorithm configurators currently available. The state-of-the-art configurator PARAMILS, which we used in this work, tends to have difficulties with configuration spaces like that of VE-SAMPLER, characterized by many categorical parameters (*e.g.*, property selection) and complex interaction between parameters (*e.g.*, the normalization and non-linear transformation for each property). The same holds for all other configurators we are currently aware of. To render CAPTAIN JACK more easily configurable, we decided to use very few categorical parameters, no conditional parameter dependencies and smoother interactions between parameters (as introduced by the previously mentioned property weights).

4 Captain Jack

In each search step of CAPTAIN JACK, the controller makes four core algorithmic decisions that determine the behaviour of the solver:

1. if promising steps are enabled and promising variables exist, select one; otherwise,
2. determine if a greedy, diversification or mixed step will occur;
3. select the greedy and/or diversification properties to use; and
4. determine the VSM (maximum or distribution-based) to be used and select the variable accordingly.

First, if promising variables exist, a straightforward G^2WSAT greedy search step [11] is taken, in which the promising variable with the best *score* is selected, breaking ties by the *age* property; this step is skipped if promising variables are turned off, which is one of the many configurable parameters. If no promising variables exist, an unsatisfied clause is selected at random (*i.e.*, the same filter as in WALKSAT is used).

The second decision in CAPTAIN JACK is which *type* of step to take: a greedy step, a diversification step or a mixed step. Each type of step has a parameterized weight, and the type of step is determined by weighted selection. For instances with variable clause lengths, the weights also depend on the length of the selected clause. CAPTAIN JACK uses a clause length range classification similar to VE-SAMPLER, where each clause falls into one of the following four ranges: $\{\leq 2, 3, 4 \ldots 9, \geq 10\}$; thus, for instances with variable clause lengths, there are 12 weights that determine the step type.

The next decision is to select the greedy and/or diversification properties. There are 9 greedy and 17 diversification properties, as described in Table 1. Each property is assigned a parameterized weight and is selected by weighted selection. For greedy and diversification steps, only one property is selected, while for mixed steps one greedy and one diversification property is selected, and the product of those two is computed.

The final decision in CAPTAIN JACK is whether to use a distribution-based VSM or a maximum VSM. The probability of using a maximum VSM is a parameterized value that is determined by the clause size and the type of the search step, resulting in 12 total probabilities for instances with variable clause lengths.

Table 1 gives an overview of the variable properties used in CAPTAIN JACK. As in VE-SAMPLER, when using interchangeable properties, special care must be taken to adjust the values of the *score* properties that can have negative values, and the *break* and *flips* properties, where large property values indicate undesirable choices. In CAPTAIN JACK, we opted for simplicity over potentially more effective normalizations and used very straightforward adjustments. For example, a constant is added to the *score* property values so the minimal candidate variable *score* is one (see website for more details).

In addition to the five greedy properties used in VE-SAMPLER, we introduced four new greedy properties (see upper part of Table 1). The *sparrowScore* property assigns a constant value of one to non-promising variables with a positive score; $sparrowScore_2$ replaces this constant with a new parameter (c_s). The *scoreRatio* has two obvious forms, and we added a separate Boolean parameter (b_s) to determine which of the two ratios should be used. *relMake* and *relBreak* were already used in VE-SAMPLER, as the *relative* number of clauses affected, normalized by the number of occurrences of the variable (*numOcc* and *numOcc'* are the number of clauses the variable currently appears in

Table 1. Variable properties in CAPTAIN JACK *Top:* greedy, *bottom:* diversification

make	number of clauses that become satisfied if the variable is flipped
break	number of clauses that become unsatisfied if the variable is flipped
score	increase in the number of satisfied clauses if the variable is flipped $(make - break)$
$sparrowScore_2$	from the SPARROW algorithm: if $score \leq 0$, ${c_b}^{score}$, otherwise c_s
scoreRatio	if b_s, $(make/(make + break))$, otherwise $(make/(break + \epsilon))$
relMake	$(make/numOcc)$: *make* adjusted by the number of occurrences of the variable (see text)
relBreak	$(break/(numOcc' + \epsilon))$
relScore	$(relMake - relBreak)$
relScoreRatio	same as *scoreRatio*, with *relMake* and *relBreak*
rand	a random number between zero and one
flat	no property value, *i.e.*, one
fair	1 for the 'next' variable in the clause, otherwise 0 (see text)
last	0 if the variable was flipped the last time the clause was selected, otherwise 1
age	number of search steps since the variable was last flipped
age_1	number of search steps since the flip prior to the most recent flip
age_5	number of search steps since the fifth most recent flip
ageRange	the *age* property with less sensitivity $\lfloor age/c_r \rfloor$
sparrowAge	from the SPARROW algorithm: $1 + (age/c_d)^{c_e}$
tabu	0 if the variable is tabu $(age < c_t)$, otherwise 1
flips	number of times the variable has been flipped
flops	number of times the variable appeared in a selected clause, but was not flipped
normFlops	similar to *flops*, but is increased by $1/clauseLen$ each time it is not flipped
resetFlops	same as *flops*, but reset to zero whenever the variable is flipped (excl. promising steps)
relFlips	$(flips/numOcc)$
relFlops	$(relFlops/numOcc)$
relNormFlops	$(normFlops/numOcc)$

as false and true, respectively). The properties *relScore* and *relScoreRatio* are defined similarly.

Furthermore, we introduced ten new diversification properties (see lower part of Table 1). Whenever a clause is selected and the *fair* property is selected, the 'next' variable in the clause is assigned a property value of one, and all others are zero; this is implemented by maintaining a counter for each clause and simply selecting the next variable in sequence. The *last* property is zero for the variable that was flipped at the most recent time the same clause was selected, regardless of the property based on which that previous selection was made. In VE-SAMPLER, the *age*' property was used to keep track of the number of steps since the flip prior to the most recent flip. Here, we call this property age_1 and generalize it to the age_k family of properties. In CAPTAIN JACK we wanted to explore larger values of k and added the age_5 property. The *ageRange* property uses a divisor parameter (c_r) and a floor function to achieve a coarser evaluation of *age*. In VE-SAMPLER, the *filtCount* property was used to maintain how often a variable appears in the filtered variables (*i.e.*, the selected clause). The *flops* property is very similar, but is instead only incremented when the variable appears in the selected clause and is not flipped. The *normFlops* property is related to *flops*, but is incremented by $(1/clauseLen)$ when the variable is not flipped. The *resetFlops* property is the same as *flops*, but is reset to zero whenever the variable is selected (excluding when it is flipped as a promising variable). Finally, the *relFlips*, *relFlops* and *relNormFlops* properties are all normalized analogously to *relMake*.

5 Experimental Setup

For our experiments we used nine benchmark sets: six random uniform k-SAT sets, one random industrial-like set and two sets of SAT-encoded software verification problems.

For the random instances, we generated two sets each for $k = 3, 5, 7$; one set for smaller instances at the solubility phase transition and one set of larger, slightly underconstrained instances. For the phase transition sets, we used clauses/variables ratios of 4.26, 21.11 and 87.79 for 3-, 5-, and 7-SAT, respectively, as specified by Mertens *et al.* [12]. For the underconstrained sets, we chose ratios of 4.2, 20 and 85, as previously used in the SAT competition. To select the instance size n for these sets, we took into consideration both the sizes used in the competition and the time required to solve instances. We selected $n = 1\,000$ (1k) and $10\,000$ (10k) for 3-SAT, 100 and 500 for 5-SAT, and 60 and 90 for 7-SAT. For each set, we generated instances with the 2009 SAT competition generator and removed instances that were not solved by TNM within $10\,000$ seconds; we randomly selected 250 instances for training and 250 for testing.

The CBMC and SWV software verification instance sets have been previously studied in the literature [10,8,18]. While PICOSAT [4] can solve any of the instances in these sets in less than two seconds, they are known to be challenging for SLS-based solvers; in fact, about 50% of the instances in SWV cannot be solved consistently by any SLS-based SAT solver we are aware of.

Finally, we used the double power-law generator provided by Ansótegui *et al.* to generate a set of random industrial-like instances we dub IL50k; we chose this generator since it produces variable length clauses that have properties similar to industrial problems [1]. Our set contains satisfiable instances with the same characteristics as the instances used by Ansótegui *et al.* $(\frac{m}{n}, \beta, \overline{k}) := (2.650, 0.75, 5)$, but with $5 \cdot 10^4$ (50k) variables instead of 500k; we randomly chose 50 training and 50 test instances.

We compared the performance of CAPTAIN JACK against six state-of-the-art SLS solvers. These include the three top SLS solvers from the 2009 SAT Competition random category, TNM, GNOVELTY$^+$2 and ADAPTG^{2}WSAT2009^{++} (henceforth, as AG^{2}2009^{++}), for which we used the parameterless competition versions (see the competition booklet for details). We also selected the UBCSAT [17] implementation of SPARROW [3]; it behaves exactly like the original implementation, but is more efficient and exposes additional parameters. The final two solvers are the highly parametric VE-SAMPLER [18] and SATENSTEIN-LS [10] solvers. For VE-SAMPLER, we used the three native (*i.e.*, non-interpreted) implementations configured for CBMC, SWV and random 3-SAT instances at the phase transition (R3SAT in [10]). We chose not to reconfigure the interpreted version of VE-SAMPLER on our new sets, which would require developing native implementations for a fair comparison.

We used the FOCUSEDILS 2.3.5 variant of the PARAMILS framework to configure SPARROW, SATENSTEIN-LS and CAPTAIN JACK on each of the nine sets. In each of the configuration experiments for SPARROW and SATENSTEIN-LS, we performed 24 independent PARAMILS runs of at least 7 CPU days each, from which we selected the one with the best performance on the respective training set. For CAPTAIN JACK, we used a training protocol comprising three sequential stages, designed to deal with successively harder instances and larger fractions of the given training set. These three

Table 2. Solver Evaluation on Test Sets. Each cell summarizes the test-set performance of the solver for 10 runs on each instance in the set with a cutoff of 600 seconds. The top row shows the penalized average runtime (PAR-10): the mean of all runs over all instances with timeouts replaced with a penalized value of 6 000 ($= 10 \cdot 600$) seconds. The second row shows the mean of the median runtimes for each instance in the set, where if any instances has a median at the cutoff time, the median is included but marked ($^+$). The third row indicates the percentage of all runs completed within the timeout. The algorithms indicated ([$*$]) have been optimized by PARAMILS on each target set. Unfortunately, GNOVELTY$^+$2 crashed on the CBMC and SWV instances and could therefore not be evaluated on those.

Algorithm	3-SAT 1k	3-SAT 10k	5-SAT 100	5-SAT 500	7-SAT 60	7-SAT 90	IL50k	CBMC	SWV
	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.	PAR-10 m.m. % sol.
CAPTAIN JACK[$*$]	69.0 15.4$^+$ 99.0%	**43.3** **24.4** **99.8%**	0.31 0.23 100%	12.1 9.2 100%	1.5 1.1 100%	68.9 22.3$^+$ 99.4%	**0.83** **0.78** 100%	0.35 0.31 100%	4 533 464$^+$ 24.5%
SPARROW[$*$]	63.0 11.9$^+$ 99.1%	220 58.3$^+$ 97.1%	**0.18** **0.14** 100%	12.9 9.1 100%	**0.81** **0.59** 100%	32.4 12.9 99.7%	1.4 1.3 100%	69.4 3.9 99.0%	4 413 446$^+$ 26.5%
VE-SAMPLER[$*$]	82.4 15.0$^+$ 98.8%	n/a	n/a	n/a	n/a	n/a	n/a	**0.08** **0.07** 100%	**2 856** **295**$^+$ **52.5%**
SATENSTEIN[$*$]	**33.5** **8.2**$^+$ **99.6%**	72.1 30.0 99.3%	0.21 0.15 100%	**3.3** **2.6** 100%	0.92 0.68 100%	34.1 13.6 99.7%	1.2 1.1 100%	0.62 0.54 100%	4 640 464$^+$ 22.7%
TNM	75.4 14.2$^+$ 98.9%	691 154$^+$ 90.3%	0.22 0.17 100%	27.2 17.5 99.9%	1.9 1.6 100%	30.6 13.3 99.8%	350 44.4$^+$ 94.4%	525 73.5$^+$ 91.8%	4 640 464$^+$ 22.7%
GNOVELTY$^+$2	78.6 16.3$^+$ 98.9%	2 604 382$^+$ 58.8%	0.27 0.20 100%	11.2 8.3 100%	1.1 0.78 100%	22.1 **10.3** 99.9%	1 901 295$^+$ 70.4%	n/a	n/a
AG^{2}2009^{++}	76.3 12.4$^+$ 98.8%	2 373 331$^+$ 62.1%	0.21 0.17 100%	24.4 15.2 99.9%	1.5 1.4 100%	**15.6** 11.9 **100%**	6.9 6.5 100%	3 267 383$^+$ 46.7%	4 217 440$^+$ 30.0%

stages consisted of 24 independent PARAMILS runs for one, three and three CPU days, respectively (see website for further details).

The PARAMILS training was conducted on Westgrid clusters (see website for details), but otherwise all solver evaluations and times reported were performed using the EDACC framework [2] running on bwGRiD [5] (Intel two socket 4-Core Xeon E5540 CPUs 2.83 GHz, with 16GB RAM running Linux).

6 Results and Discussion

In Table 2, we present the results from our evaluation of CAPTAIN JACK and several state-of-the-art algorithms on the instance sets described in Section 5. As previously mentioned, CAPTAIN JACK was designed to perform well on a wide variety of instances, and this is reflected in the results. We note that the configurations evaluated here are, for the most part, not the best that exist in the CAPTAIN JACK design space. For example, in results not reported here, but available online, we ran PARAMILS for an

additional three days on each set and were able to obtain configurations that achieved modest improvements in PAR-10 (6%-9%) on 3sat10k, 5sat500 and IL50k and more significant improvement (20%) on CBMC and SWV. Furthermore, as we will discuss later in this section, we found evidence that in some cases, the best CAPTAIN JACK configuration on a given type of instances was not the one optimized for that type, which clearly indicates that PARAMILS at least in some cases produced sub-optimal configurations.

The relative performance of CAPTAIN JACK is the best on 3sat10k and IL50k, and CAPTAIN JACK is now the best known SLS solver for those sets. On the sets 5sat100, 7sat60 and 7sat90 CAPTAIN JACK performs significantly worse than SPARROW. We conjecture that this is connected to the relatively small number of variables in these instances, but further investigation is needed to explain this phenomenon.

Finally, we were quite surprised by the strong performance of SPARROW on IL50k, as results obtained for the default configuration of SPARROW were much worse. Interestingly, the configuration of SPARROW found by PARAMILS on IL50k uses no smoothing, a rare situation for high-performance SLS algorithms using clause penalties – a phenomenon that might warrant further investigation.

Next, we investigated the CAPTAIN JACK configurations found by PARAMILS for each of our instance sets (see Table 3). As mentioned earlier, we do not believe that these configurations are optimal, and longer runs of PARAMILS could produce rather different configurations. Nevertheless, we believe that qualitative differences between these configurations, which are based on multiple, long runs of PARAMILS, are likely meaningful.

In CAPTAIN JACK we introduced several new variable properties, and we were curious to see which role these would play in the configurations found by PARAMILS. Clearly, the age_5 property is very effective, which was surprising to us, especially since the value of 5 was selected somewhat arbitrarily; this suggests that the age_k family of properties merits further study. The slight modification we introduced in $sparrowScore_2$, which allows for positive non-promising variables to have a parameterized value (c_s), also appeared to be quite effective, and most configurations had a value of c_s slightly greater than one. Considering that the original *sparrowScore* property had been developed for solving random 3-SAT, it is perhaps not surprising that $sparrowScore_2$ is prominently used only in the random k-SAT configurations. Conversely, our new properties *scoreRatio* and *relScoreRatio* appeared to be useful only on the software verification benchmarks, both of which had b_s set to true (see Table 1). Our intuition was that the *fair* and *flops*-based properties would be good for diversification, but that they should be used sparingly; our results are consistent with this intuition, but further study is warranted. Overall, the diversification properties most often selected are the *age* variants, which is also the most prominent diversification property in the literature. It is also very clear that the *flips* properties are very important for instances with variable-length clauses and non-uniform variable distributions.

We were wondering whether PARAMILS would prefer a 'traditional' approach for SLS-based SAT solving with mostly greedy steps and a few diversification steps, as it did for 7sat90; however, most of the optimized configurations turned out to favour mixed steps. This supports previous evidence that exploring new and innovative

Table 3. Parameter Settings in CAPTAIN JACK. *(Top:)* The weight for each search step type (greedy / mixed / diversification), and for each type, the percent of steps where the maximum VSM was selected (as opposed to a distribution-based VSM). For instances with variable length clauses, the values depend on the selected clause length $\{\leq 2, 3, 4 \ldots 9, \geq 10\}$. *(Bottom:)* The weight for each variable property (greedy and diversification properties are selected independently). All weights have been normalized to appear as percentages. Non-applicable values are shown as a dash (-), and weights with value zero are blank. The most significant properties (with a combined weight of $\geq 75\%$) are in bold. As an example, for 3sat10k, 94% of the (non-promising) search steps are mixed steps, and in 90% of those steps the maximum VSM is used. *make* and age_5 are each independently selected as the greedy and diversification property for 40% of the mixed steps. This means that for 13.5% $(0.94 \cdot 0.9 \cdot 0.4 \cdot 0.4)$ of the (non-promising) search steps the variable with the maximum value of $\langle make \cdot age_5 \rangle$ is selected.

step type /	3-SAT		5-SAT		7-SAT		IL50k				CBMC				SWV			
	1k	10k	100	500	60	90												
VSM	%	%	%	%	%	%	≤ 2	3	4-9	≥ 10	≤ 2	3	4-9	≥ 10	≤ 2	3	4-9	≥ 10
greedy step		3	6	67		93	10			8	8	67	89	33	31		32	94
mixed step	100	94	94	33	100	1	80	100	67	31	62	17		66	62	100	64	3
div. step		3				6	10		33	62	31	17	11	1	8		4	3
greedy: % max	-	80	30	50	-	50	80	-	-	100	80	0	40	0	100	-	0	90
mixed: % max	10	90	90	10	90	0	90	10	0	70	60	70	-	10	60	100	0	30
div.: % max	-	0	-	-	-	20	10	-	20	0	30	10	80	0	10	-	50	0

property	3-SAT		5-SAT		7-SAT		IL50k	CBMC	SWV
	1k	10k	100	500	60	90			
	%	%	%	%	%	%	%	%	%
make		**40**	2.8			0.7	0.5	13	
break	1.2	10	2.8		**50**	**47**		6.3	6.2
score	9.9	10	2.8	20		3	**70**	**25**	0.4
$sparrowScore_2$	**79**	**40**	**90**	**78**	**50**	**47**	0.5	0.8	3.1
scoreRatio	9.9		1.4	2.4		1.5		0.8	**25**
relMake	-	-	-	-	-	-	**18**	1.6	0.4
relBreak	-	-	-	-	-	-	8.8		3.1
relScore	-	-	-	-	-	-	1.1	1.6	12
relScoreRatio	-	-	-	-	-	-	1.1	**51**	**50**
rand	**15**	0.6	0.5	0.3	0.6	**52**	1.3		1.4
flat		**9.9**	1	**44**	0.3			0.3	0.3
fair	0.2		0.5	1.4	0.3		0.7	5.2	0.7
last	**15**	4.9	4.1	0.7	10	**13**	0.3		0.3
age	3.6	**9.9**	**33**	1.4	2.5	**13**		1.3	2.8
age_1	0.5	**20**	**16**	1.4			**43**		
age_5	**15**	**40**	4.1	0.7	**20**			**42**	**44**
ageRange	7.3	0.6	1	2.7	0.6		0.3	1.3	
sparrowAge	**29**		0.3	0.7	**20**	1.6	2.7	0.3	
tabu	3.6	4.9	**33**	**44**	**41**	3.2	0.7	0.7	
flips	3.6		2	0.7	1.3		**43**	0.3	**44**
flops	7.3		0.5		2.5	1.6	0.3	0.3	0.3
normFlops	0.5		0.3	2.7	0.3	**13**			
resetFlops	0.5	**9.9**	4.1	0.3	0.3	3.2		0.3	0.7
relFlips	-	-	-	-	-	-	5.4	**42**	1.4
relFlops	-	-	-	-	-	-	2.7	5.2	0.3
relNormFlops	-	-	-	-	-	-		0.7	2.8

methods for mixing properties in VEs is a promising area of research. We introduced clause-length-dependent behaviour in CAPTAIN JACK to see if we could observe interesting trends, *e.g.*, we hypothesized that mixed or diversification steps could be more important for larger clause lengths, but the results are inconclusive. However, it appears

that IL50k and CBMC benefit from more diversification steps than the random k-SAT sets, and – as we will observe in Table 4 – it appears that SWV does as well. For random k-SAT, the configurations for the underconstrained sets allow for more greedy steps, which is consistent with the understanding that these instances are relatively easier to search than those at the phase transition. The proportion of solely greedy steps is actually higher (12%) for 3sat10k if we consider that *flat* is selected as the diversification property for 10% of the mixed steps; this is an example of the kind of inter-parameter dependency that we were attempting to minimize in CAPTAIN JACK, but is impossible to eliminate completely.

There is no clear preference between maximum *vs* distribution-based VSMs. It would be interesting re-run PARAMILS on CAPTAIN JACK with two different restricted configuration spaces that force all steps to be either a maximum or distribution-based VSM; we could then observe which VSM would achieve better performance, and study the differences between the resulting configurations. We note that CAPTAIN JACK is well suited for this kind of analysis, which we are confident will lead to an improved understanding of the performance potential inherent in various algorithm components, and of their interaction when solving various types of SAT instances.

There are a few additional parameter settings, such as the parameters c_s, c_r and b_s in Table 1, that are not shown in Table 3 and can be found online. One of these is the Boolean parameter to control if G^{2}WSAT promising steps are taken; only the CBMC and SWV configurations do not take promising steps. This may suggest that promising steps may be more suited for randomly generated instances (including IL50k), which is consistent with the observation that AG^{2}2009^{++}, which relies heavily on promising steps, is the worst performer on CBMC in Table 2.

Another method for evaluating the differences between our configurations is to cross-test each of the configurations on each of the sets, and we present the results of these experiments in Table 4. These results also indicate how 'specialized' each of our optimized configurations are. The most surprising result is that in several cases, the best configuration for an instance set is actually *not* the one optimized for the respective training set. As previously stated, this clearly indicates that PARAMILS does not always find optimal configurations within the design space of CAPTAIN JACK. The most interesting such configuration is the one obtained for IL50k, which performs very well on the SWV set. This suggests that the industrial-like instances could indeed be very useful for optimizing performance on harder industrial instances. This similarity is reinforced further as the SWV configuration is the second-best configuration (by a large margin) on the IL50k set. Finally, because the IL50k set has an *average* clause length of 5, we hypothesized that the 5-SAT configurations might perform well on IL50k or vice-versa, but this appears not to be the case. This further highlights the fact that the structural aspects of real verification instances captured by the industrial-like instances, albeit simplistic, are at least to some degree informative.

One final experiment we performed was to test if algorithms trained on the IL50k instances would be able to solve larger instances from the same distribution. Ansótegui *et al.* [1] generated larger (500k) instances, and demonstrated that GNOVELTY$^+$ was unable to solve these. We generated a set of ten such instances (IL500k) that, aside from the number of variables, have the same characteristics as the IL50k set. Because

Table 4. Cross-Testing of CAPTAIN JACK configurations. Each configuration of CAPTAIN JACK was run once on each instance in each test set with a cutoff of 600 seconds. We report the ratio of the resulting PAR-10 to the PAR-10 for the targeted configuration. Configurations that outperform the targeted configuration for the set are in bold.

Configuration	3-SAT		5-SAT		7-SAT		IL50k	CBMC	SWV
	1k	10k	100	500	60	90			
CJ [3sat1k]	1	61.5	1.38	95.7	1.08	1.03	157	5 876	1.02
CJ [3sat10k]	2.65	1	1.41	545	1.99	3.99	167	1 890	1.02
CJ [5sat100]	2.56	135	1	93.2	1.18	**0.72**	170	7 108	1.03
CJ [5sat500]	24.3	200	1.35	1	1.00	**0.97**	1 271	10 014	1.00
CJ [7sat60]	99.1	200	**0.82**	539	1	2.33	786	9 989	1.02
CJ [7sat90]	105	200	1.82	12.1	1.44	1	1 929	3 088	**0.98**
CJ [IL50k]	16.6	200	4.50	567	2.20	15.8	1	1 106	**0.83**
CJ [CBMC]	19.9	200	6.71	483	2.97	7.70	1 236	1	1.02
CJ [SWV]	148	200	17.6	567	9.47	79.2	2.29	2.43	1

the instances are so large, we observed that many SLS solvers (*e.g.*, from the SAT competition, but also SATENSTEIN-LS) encounter technical problems when trying to solve them. We ran the IL50k configurations of CAPTAIN JACK and SPARROW and the UBCSAT implementation of ADAPTG^{2}WSAT$^+$ on each of the 10 instances with a cutoff of 12 hours per instance. ADAPTG^{2}WSAT$^+$ solved only 5 instances and SPARROW was able to solve 9. However, CAPTAIN JACK was able to solve all 10 instances in a combined time of 77 minutes. For perspective, PICOSAT [4] solved all 10 instances in a combined time of 2 minutes and showed little variation in runtime per instance compared to the SLS solvers.

7 Conclusions and Future Work

In this work, we have introduced CAPTAIN JACK, a highly parametric SLS algorithm that can be automatically configured to perform well on various types of SAT instances and is currently the best known SLS algorithm for solving large random 3-SAT and 'industrial-like' instances. We designed CAPTAIN JACK in a way that would aid us in exploring which components and heuristic mechanisms give rise to strong performance on different types of SAT instances and made several interesting observations in this respect. We also introduced several new variable properties and provided evidence these can be very effective; in particular, our results suggest that the family of age_k properties merits further investigation. Finally, we provided preliminary evidence that training on smaller industrial-like instances may be a viable approach to improving SLS algorithm performance on larger industrial problems.

Our results reported here provided further evidence that mixed VEs can be very effective; while CAPTAIN JACK, SPARROW and VE-SAMPLER combine only two variable properties, we believe that it may be interesting to investigate more complex combinations. We also see potential in developing an *adaptive* CAPTAIN JACK that adjusts its balance between diversification and intensification throughout the search and incorporates a mixed VE that combines property values accordingly. Ultimately, we hope that CAPTAIN JACK will provide further insight into SLS algorithm development, and that algorithm developers will be able to gain insight and inspiration from examining

CAPTAIN JACK configurations that are effective for solving particular instance sets. We believe that such work will lead to new and specialized lightweight algorithms, similar to SPARROW, that improve the state-of-the-art for solving SAT.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This research has been enabled by the use of computing resources provided by the bwGRiD project [5], WestGrid and Compute/Calcul Canada. Furthermore, HH acknowledges funding received through the NSERC Discovery Grants Program.

References

1. Ansótegui, C., Bonet, M.L., Levy, J.: Towards industrial-like random SAT instances. In: IJCAI 2009, pp. 387–392 (2009)
2. Balint, A., Diepold, D., Gall, D., Gerber, S., Kapler, G., Retz, R.: EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In: LION-2011 (to appear)
3. Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 10–15. Springer, Heidelberg (2010)
4. Biere, A.: PicoSAT essentials. JSAT 4, 75–97 (2008)
5. bwGRiD: Member of the German D-Grid initiative, funded by the Ministry of Education and Research and the Ministry for Science, Research and Arts Baden-Württemberg
6. Chvátal, V., Szemerédi, E.: Many hard examples for resolution. Journal of the ACM 35(4), 759–768 (1988)
7. Hoos, H.H.: Computer-aided design of high-performance algorithms. Tech. Rep. TR-2008-16, University of British Columbia (2008)
8. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (2009)
9. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI 2007, pp. 1152–1157 (2007)
10. KhudaBukhsh, A.R., Xu, L., Hoos, H.H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: IJCAI 2009, pp. 517–524 (2009)
11. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 158–172. Springer, Heidelberg (2005)
12. Mertens, S., Mézard, M., Zecchina, R.: Threshold values of random k-SAT from the cavity method. Random Structures & Algorithms 28, 340–373 (2006)
13. Pham, D.N., Thornton, J., Gretton, C., Sattar, A.: Combining adaptive and dynamic local search for satisfiability. JSAT 4, 149–172 (2008)
14. Prestwich, S.: Random walk with continuously smoothed variable weights. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 203–215. Springer, Heidelberg (2005)
15. Thornton, J., Pham, D.N., Bain, S., Ferreira Jr., V.: Using cost distributions to guide weight decay in local search for SAT. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 405–416. Springer, Heidelberg (2008)
16. Tompkins, D.A.D.: Dynamic Local Search for SAT: Design, Insights and Analysis. Ph.D. thesis, University of British Columbia (2010)

17. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In: H. Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 306–320. Springer, Heidelberg (2005)
18. Tompkins, D.A.D., Hoos, H.H.: Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 278–292. Springer, Heidelberg (2010)
19. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)