On the Quality and Quantity of Random Decisions in Stochastic Local Search for SAT

Dave A.D. Tompkins and Holger H. Hoos

Department of Computer Science, University of British Columbia
2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada
{davet, hoos}@cs.ubc.ca

Abstract. Stochastic local search (SLS) methods are underlying some of the best-performing algorithms for certain types of SAT instances, both from an empirical as well as from a theoretical point of view. By definition and in practice, random decisions are an essential ingredient of SLS algorithms. In this paper we empirically analyse the role of randomness in these algorithms. We first study the effect of the quality of the underlying random number sequence on the behaviour of well-known algorithms such as Papadimitriou's algorithm and Adaptive Novelty$^+$. Our results indicate that while extremely poor quality random number sequences can have a detrimental effect on the behaviour of these algorithms, there is no evidence that the use of standard pseudo-random number generators is problematic. We also investigate the amount of randomness required to achieve the typical behaviour of these algorithms using derandomisation. Our experimental results indicate that the performance of SLS algorithms for SAT is surprisingly robust with respect to the number of random decisions made by an algorithm.

1 Introduction

The Propositional Satisfiability Problem (SAT) is the prototypical $\mathcal{NP}$-complete problem and a prominent hard combinatorial decision problems. Some of the best known methods for solving certain types of SAT instances are Stochastic Local Search (SLS) algorithms; these are typically incomplete, i.e., they cannot determine that a formula is unsatisfiable, but they often find models of satisfiable formulae surprisingly effectively. Many SLS algorithms are probabilistically approximate complete (PAC) and will solve a soluble instance with arbitrarily high probability when allowed to run long enough [1].

A typical SLS algorithm for SAT consists of an initialisation phase, in which a truth value is assigned to each variable, and a search phase, during which the values of individual, heuristically selected variables are changed in an attempt to reach a satisfying assignment. The search phase is a sequence of search steps known as *flips* because in each step typically one variable's assignment is changed *(or flipped)*. Stochastic (random) decisions are typically used in both phases, and in the following we describe the most common ways SLS algorithms for SAT make use of random decisions:

Variable initialisation is heavily randomised in most SLS algorithms for SAT; typically, the initial variable assignment is obtained by assigning each variable a truth value $\{\top, \bot\}$ chosen uniformly and independently at random.

Heuristic tie-breaking occurs when a choice needs to be made between several alternatives that are ranked identically by a given heuristic evaluation function; many SLS algorithms for SAT break these ties randomly.

L. Lamontagne and M. Marchand (Eds.): Canadian AI 2006, LNAI 4013, pp. 146–158, 2006.

Variable selection often includes randomised choices; examples include the noise mechanisms in Novelty and variable selection in Simulated Annealing.

Neighbourhood selection occurs when an algorithm narrows the list of flip candidates to a subset of all the variables. For example, in the WalkSAT algorithms, in each step an unsatisfied clause is selected uniformly at random, and then only variables in occurring in this clause are considered as flip candidates.

Random walk steps involve flipping randomly selected variables; they can help to increase search diversification, to avoid stagnation, and to render an algorithm PAC. In a uniform random walk all variables can be selected with uniform probability. In a conflict-directed random walk only variables occurring in currently unsatisfied clauses can be selected, such as in Papadimitriou's algorithm [2] and WalkSAT [3].

Random restarts cause an algorithm to randomly re-initialise all variables; most SLS algorithms for SAT, including algorithms of purely theoretically interest, such as Schöning's algorithm [4], perform periodic random restarts.

Search control mechanisms can also make use of randomised decisions; examples include the probabilistic smoothing mechanism in the SAPS algorithm [5] and the random selection of the tabu tenure parameter in Robust Tabu Search [6].

The prominent use of random decisions in many components of SLS algorithms raises some interesting questions: Why are most algorithms so heavily randomised? How important are those random decisions? How important is the quality of the underlying random numbers? How much randomness is necessary? Can randomness be eliminated altogether? In this paper, we attempt to shed light on some of these questions.

Some of these questions have been addressed in previous work. Gent and Walsh investigated the role of random decisions in GSAT [7]. They found that random decisions were neither important in the initialisation phase nor for tie breaking, and that deterministic substitutions could be made in both cases. Much of their analysis revolved around the ability of the algorithm to diversify the search during re-initialisation. They did not study the impact of the quality of random decisions, and it is not clear to which extent their observations apply to more powerful SLS algorithms for SAT that do not require restart mechanisms and their application to a broad range of SAT instances.

There has been a large body of work dedicated to the quest for increasingly higher quality random number generators. In the Monte Carlo simulation literature, there has been evidence that even *good* random number generators can produce very undesirable errors in their results [8, 9]. In work related to this paper, Ribeiro *et al.* recently surveyed random number generators to find a good candidate for randomised algorithms [10]. In our previous work [11] we investigated the role of random decisions in the SAPS algorithm, which we will develop further in Section 4.

The remainder of this paper is structured as follows: In Section 2 we briefly introduce the algorithms and problem instances used in our computational experiments reported later. In Section 3 we investigate how important the quality of the random decisions are, while in Section 4 we explore the quantity of random decisions required to achieve the typical behaviour of these algorithms. Finally, Section 5 contains a brief discussion of our main findings and points out some directions for future work.

2 Preliminaries

The first algorithm we consider in this work is conflict-directed random walk (CRWALK). After randomly initialising all variables, in each search step this algorithm selects a currently unsatisfied clause and flips a randomly chosen variable from that clause. This algorithm was first studied by Papadimitriou, who proved that it solves 2-SAT in expected quadratic time [2]. Extending it with a simple periodic restart mechanism leads to Schöning's algorithm, whose run-time on 3-SAT instances was proven to be bounded from above by $O(1.334^n)$ [4]. More recently, Iwama and Tamaki's have extended Schöning's algorithm to improve this bound to $O(1.324^n)$ [12].

We chose to include CRWALK in our study because it is a prominent, yet very simple algorithm that is purely based on random decisions. Originally, we had decided to include Schöning's algorithm in our study because of its provably excellent worst-case behaviour, but in preliminary experiments on a large set of instances from SATLIB we found no empirical evidence for any differences between its behaviour and that of CRWALK (which, given well-known empirical results on the behaviour of WalkSAT algorithms [13, 1] is not surprising). As will also be apparent from the results reported later in this paper, CRWALK performs quite poorly when compared against high-performance SLS algorithms for SAT, because it completely lacks heuristic guidance.

The two other algorithms we used in this study, Adaptive Novelty$^+$ (ANOV$^+$) [14] and SAPS [5] are amongst the best performing SLS algorithms for SAT currently known. ANOV$^+$, a member of the WalkSAT family, placed first in the random category of the 2004 SAT competition [15]. In addition to random initialisation it uses randomised neighbourhood selection, randomised heuristic variable selection, and conflict directed random walk steps. ANOV$^+$ employs a deterministic mechanism for adapting its noise setting p during the search and therefore requires no parameter tuning.

Scaling and Probabilistic Smoothing (SAPS) changes the space it is searching by dynamically modifying penalty weights associated with the clauses of the given CNF formula [5]. In addition to random initialisation it uses randomised heuristic tie-breaking, randomised search control mechanisms, and uniform random walk steps. SAPS shows performance that is competitive with ANOV$^+$. We mainly included it in this study because (as we will discuss in more detail later) in long search trajectories SAPS approaches deterministic behaviour [11]. In all experiments reported in this study we used the default parameters for SAPS ($\alpha = 1.3, \rho = 0.8, P_{smooth} = 0.05, wp = 0.01$).

All three algorithms (CRWALK, ANOV$^+$ and SAPS) are available as part of the UBCSAT software package [16] which is available for download from the UBCSAT website[1]. Unless otherwise stated (as in Section 3) all experiments have been conducted using the default random number generator in UBCSAT, Mersenne Twister [17].

For our experiments, we have used individual satisfiable instances obtained from SATLIB [18]. We provide brief descriptions here, while more detailed information is available from the SATLIB website[2]. The uniform random 3-SAT instance sets (`ufN-*`) are all randomly generated with N variables at the phase transition. The hardest, median and easiest instance from these sets are referred to as `-hard`, `-med` and

[1] `http://www.satlib.org/ubcsat`
[2] `http://www.satlib.org`

`-easy`, respectively. The `flatN-*` instances are encodings of randomly generated flat graph 3-colouring problems with N vertices; these instances share structure induced by the SAT-encoding. The `ii` and `ssa` instances are from the DIMACS challenge set, and are a formulation of Boolean function synthesis problem and encodings from circuit fault analysis, respectively. The `bw` instances are encodings of a blocks world planning problem and have been popular instances in the literature. The `ferry` instances are from the SAT 2003 competition industrial category. The `anov10M-struct` set contains over two thousand instances and includes all structured (non-random) instances currently available on SATLIB where ANOV$^+$ has a median run-time between 1 000 and 10 000 000 steps.

3 The Quality of Random Decisions

When implementing SLS algorithms, all random decisions are realised using a random number generator (RNG). In principle, a true random number generator (TRNG), which obtains a sequence of random numbers from a truly random source could be used. Hardware implementations of TRNGs that obtain random data from physical phenomae, such as atmospheric noise or radioactive decay, are available and are popular in applications such as gambling[3] and cryptography [19]. However, most computer implementations use pseudo-random number generators (PRNG) instead. A PRNG is a finite state machine with memory, and performs deterministic mathematical operations on the state information to generate a sequence of numbers. Once a PRNGs is initialised with a numerical *seed*, it will produce a series of numbers that may have the appearance of being random, but in fact can all be deterministically calculated from the seed. The quality of a PRNG is solely determined by the mathematical operations it performs. Ideally, sequences will be uniform and unbiased (*i.e.*, equal fractions of numbers from the sequence should fall into equal intervals), uncorrelated (*i.e.*, the numbers in the sequence should be statistically independent of one another), and have long periods (because the state information in a PRNG is finite, all PRNGs will eventually cycle, but the period between cycles should be very large) [1].

Because of the importance of high quality random numbers in cryptography and other applications, tests have been developed that measure the quality of a sequence of random data. The American National Institute of Standards and Technology (NIST) has produced a document [20] with companion software[4] to test the quality of random data. The NIST software includes 16 groups of tests that cover a wide variety of statistical properties. Another popular software tool for quickly analysing the quality of random numbers is known as `ent` and was developed by John Walker at Fourmilab[5].

There are numerous PRNGs available that use a wide variety of mathematical methods. We have selected a few characteristic PRNGs to test, in addition to data generated by a TRNG. The following are brief descriptions of the RNGs we used:

True Random Data. This data was obtained from `random.org` and was generated by a hardware device measuring atmospheric noise.

[3] `http://www.first.fraunhofer.de/owx_download/keno-engl.pdf`
[4] `http://csrc.nist.gov/rng`
[5] `http://www.fourmilab.ch/random`

Table 1. Randomness quality tests on 160MB of data generated by various RNGs. The Bias value is the average value of all bits (the ideal value is 0.5). The χ^2 analysis from `ent` shows the distribution value and a percentage which indicates how frequently a TRNG would have a larger distribution value, where values $> 95\%$ or $< 5\%$ are highly suspect. The Monte Carlo π analysis from `ent` gives an estimated value of π and the respective error. For the NIST tests, we report the overall percentage of the tests passed by the respective data, where each of the 16 groups of tests was weighted equally.

	Bias	χ^2 Analysis	Monte Carlo π	NIST %
True random	0.5000290	235.9 (75%)	3.14094 (0.021%)	97.80
Unix C random()	0.4999988	224.6 (90%)	3.14148 (0.004%)	99.50
LCG	0.5000000	0.0 (99.99%)	3.14123 (0.011%)	93.53
LFG	0.5000129	237.3 (75%)	3.14139 (0.007%)	96.69
MT	0.5000204	278.5 (25%)	3.14203 (0.014%)	98.37
Random: Skewed 1.25:1	0.5554831	2165538.1 (0.01%)	2.76998 (11.829%)	16.39
Random: Cycled 16k	0.5000086	4327.3 (0.01%)	3.14631 (0.150%)	59.18

'C' random(). We chose the linux gcc 'C' `random()` function because it is the default PRNG for many programmers, and is also currently the default PRNG for the original WalkSAT software package by Kautz [3] when compiled under Linux. We used gcc v3.3.3 on SuSE Linux v9.1.

LCG. The Linear Congruential Generator (LCG) we chose was based on the ANSI 'C' specification: $I_{j+1} = (I_j \times 1103515245 + 12345)$ except that only one byte (bits 11-18) of random data was collected per iteration, a common practice to improve the quality of this particular PRNG.

LFG. The Lagged Fibonacci Generator (LFG) we chose was from the book by Knuth [21], and the source code is available from his website[6].

MT. The Mersenne Twister (MT) we chose is the MT19937 algorithm [17], which has an astounding period of $(2^{19937}-1)$. This is the default PRNG in the current release of the UBCSAT software package [16].

In Table 1 we examine the relative quality of the some of these RNGs. There is little difference between the results for the PRNGs and the TRNG, with the exception of LCG, which is clearly the worst of the tested PRNGs. It is often the case that individual sequences of TRNGs fail more tests than individual sequences of PRNGs [20]. The bottom two rows of Table 1 will be discussed later.

We now investigate to which extent the quality of the source of randomness affects SLS behaviour. Intuitively, bias in the random number sequence can be expected to have a negative impact on SLS performance for the following reason. For most random decisions made within an SLS algorithm, there are more bad choices (that increase the length of the current run) than good choices. Most forms of bias would therefore tend to increase the relative probability of making a bad choice. However, note that even when using a TRNG with extreme bias, as long as the probability of generating 0 or 1 at any position of the sequence is greater zero, the PAC property of a given SLS algorithm

[6] `http://www-cs-faculty.stanford.edu/~knuth/programs/rng.c`

Table 2. The effect of different types of random data streams on the CRWALK algorithm. For the true random data, the mean number of search steps (run-length) required to find a solution is given, while for all other sources the mean search steps is given as a fraction of the number required for the true random source. The *c.v.* is calculated as the standard deviation divided by the mean ($\sigma/\bar{x}$). Note that *c.v.* $= 1$ characterises an exponential run-length distribution, which is typical for high-performance SLS algorithms for SAT. All experiments results are based on 500 runs with a maxiumum run-length of 2^{32} (4.3B) steps. For the cycled streams with a reported ∞ mean, we confirmed cyclic behaviour by examining the respective search trajectories.

	ii8c2		ssa7552-159		flat50-med		uf100-med		uf50-hard	
	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*
True random	300k	0.99	2.21M	0.97	631k	0.97	76.1M	0.94	372k	0.97
Unix C random()	1.13	0.94	1.02	0.91	0.96	0.93	1.10	0.99	1.10	0.99
LCG	1.13	1.02	1.02	1.00	0.95	0.98	1.02	0.96	1.02	0.96
LFG	1.15	0.99	1.05	1.02	0.94	1.03	0.98	0.97	0.98	0.97
MT	0.97	0.95	0.99	0.98	0.90	0.93	0.93	0.96	0.93	0.96
Skewed 1.25:1	0.48	0.97	3.39	1.08	0.93	0.97	0.97	1.03	0.97	1.03
Skewed 1.5:1	0.29	0.92	**15.27**	0.97	0.85	1.04	1.10	0.96	1.10	0.96
Skewed 2:1	0.13	0.94	$>$ **368**	0.97	0.93	1.03	1.03	0.99	1.03	0.99
Skewed 4:1	**0.06**	1.00	$>$ **2 000**	0.02	0.88	1.02	0.96	1.05	0.96	1.05
Cycled 16k	1.28	0.86	0.66	0.96	0.92	0.87	0.82	1.16	0.82	1.16
Cycled 4k	1.23	0.85	0.82	1.15	0.89	0.83	0.61	1.11	0.61	1.11
Cycled 1k	0.89	0.76	2.17	0.91	0.55	0.83	0.52	1.00	0.52	1.00
Cycled 512	0.68	1.22	∞	**0**	0.10	0.75	0.63	1.12	∞	**0**
Cycled 256	2.38	0.56	∞	**0**	0.41	0.70	0.41	0.69	∞	**0**

would remain intact, since the required sequence of 'correct decisions' would still occur (albeit with much lower probability).

The effect of correlation in the random number sequence, as long as it does not involve deterministic dependencies, would be very similar for analogous reasons. (Note that correlation, in this context, corresponds to bias for certain subsequences.)

Deterministic cycles in the random number sequence, on the other hand, could easily lead to a loss of the PAC property, because in combination with the finite state information held by the algorithm (which in addition to the search position may include search control variables, such as tabu status information or dynamic penalty weights), they could cause cycles in the search trajectory that do not include any solutions to the given problem instance. Note that all PRNGs are periodic; whether or not this leads to observable stagnation of a given SLS algorithms depends on the period of the PRNG as well as on the amount and nature of state information used by the SLS algorithm.

In order to empirically study the effect of poor quality RNGs on SLS algorithms, we generated some intentionally bad random number sequences by manipulating the data we had from the TRNG. First, we introduced a skew s in our data by converting 32-bits of our random data to obtain fixed-point binary values in the range [0,1), generating a 1 if the value was greater than $s/(s+1)$. Next, we generated cycled data where we simply truncated the random data at a fixed number of bytes and repeated the same sequence. We ran our new poor streams through the same tests we performed on the PRNGs, and from Table 1 it is clear that our poor RNGs do not meet very high standards of quality.

Table 3. The effect of different sources of random data streams on the ANOV$^+$ algorithm *(above)* and the SAPS algorithm *(below)* on the same instances. See Table 2 for details.

Random Data	uf100-med		uf250-hard		bw-large.c		ferry9u	
	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*	$\overline{x}/\overline{x}_{rand}$	*c.v.*
Random Source	998	0.63	3.00M	0.96	10.0M	0.99	880k	0.88
Skewed 1.25:1	1.17	0.61	1.29	1.06	0.91	1.05	0.57	0.87
Skewed 2:1	1.61	0.65	4.16	1.01	0.99	0.95	0.68	0.90
Skewed 4:1	3.02	0.76	**96.31**	0.62	1.30	1.00	$>$ **3 122**	0.75
Cycled 16k	1.06	0.80	0.85	0.95	0.93	1.17	0.98	0.40
Cycled 512	1.26	0.50	∞	**0**	0.13	1.61	1.03	0.80
Cycled 256	0.33	0.79	∞	**0**	0.66	1.33	∞	**0**
Random Source	1.06k	1.01	304k	1.07	14.6M	0.99	1.92M	1.01
Skewed 1.25:1	1.31	0.97	1.33	1.01	0.54	1.04	0.39	0.97
Skewed 2:1	1.89	1.16	**3.03**	1.08	0.34	0.97	0.26	0.97
Skewed 4:1	**2.37**	1.09	**5.45**	1.04	**0.42**	1.02	**0.11**	0.90
Cycled 16k	1.10	0.99	0.99	1.00	0.95	0.97	0.78	0.90
Cycled 512	0.55	0.72	0.96	0.49	0.88	1.18	2.18	0.89
Cycled 256	1.39	0.89	1.44	0.83	1.26	0.99	0.39	1.23

In what follows, we made the streams progressively worse, and so the data in Table 1 can be considered the best of the bad streams we generated.

To examine the effects of different RNGs on our selected algorithms, we ran CRWALK, ANOV$^+$ and SAPS with the different sources of random data, and present the results in Tables 2, 3 (top), and 3 (bottom) respectively. We provided the PRNG comparison for CRWALK, and we can see the algorithm was very robust *w.r.t.* the selection of the PRNGs; analogous observations were made for ANOV$^+$ and SAPS.

For the skewed data, the data streams had an increasing amount of ones, and we shall consider what effect it would have on the specific implementations of the algorithms. For CRWALK, the bias would be toward arbitrarily specific clauses and literals. For the ANOV$^+$ algorithm, the same bias would exist for clause selection, but more importantly the frequency of random walk steps and noisy heuristic decisions would decrease. For SAPS, the only significant change is a decrease in the smoothing frequency. Not all of the changes were negative, and in some cases such as the CRWALK algorithm on the `ii8c2` instance, the skew greatly improved the performance of the algorithm.

For the cycled data, we continued to shorten the length of the cycles and thereby increased the likelihood that the algorithms would cycle. In Tables 2 and 3 we present results from situations where both the CRWALK and the ANOV$^+$ algorithm became stuck in endless loops. Note that although CRWALK and ANOV$^+$ are both PAC, our empirical results show that these algorithms can become essentially incomplete when using cyclic random number streams. The fact that all finite PRNGs eventually cycle suggests that *no conventional implementation* of an SLS algorithm is truly PAC. (An implementation may be PAC for a given instance, but with a countably infinite number of SAT instances there is no hope of guaranteeing that an implementation will be PAC for any arbitrary instance.)

Given this conclusion, it might seem wise to implement algorithms with TRNGs. If efficient TRNGs were readily available it would be an ideal solution. However, TRNGs are far from efficient when compared to PRNGs. We must add perspective to this discussion and consider how incredibly unlikely the aforementioned circumstances are with a good PRNG. For example, the Mersenne Twister PRNG has a period of $(2^{19937} - 1)$, which makes it very unlikely to ever encounter cycling behaviour in practice. Rather, if cyclic behaviour is observed for an algorithm using a PRNG of this type, the cyclic behaviour is far more likely due to a design flaw, an implementation error, or simply because (even when using true random numbers) the algorithm is not PAC.

When implementing an SLS algorithm and selecting a PRNG, there are several factors to be considered. To assess the quality of a given PRNG, one of the many available test suites can be used; however, any reasonable PRNG will have sufficient quality *w.r.t.* bias and correlation to render impacts on the performance of typical SLS algorithms very unlikely. However, in order to minimise the chance of encountering cycling behaviour of an SLS algorithm in practice, it is generally advisable to chose a PRNG with a large period. Another potentially important factor is the efficiency of a PRNG; this is particularly relevant in the context of highly randomised SLS algorithms that make random decisions in every (or almost every) search step. Finally, especially in the context of scientific research, the use of platform-independent PRNGs makes it possible to reproduce unusual algorithm behaviour exactly across different hardware and operating systems. The previously mentioned Mersenne Twister has *all* of the qualities that are desirable for a PRNG and overall appears to be the best choice in the context of implementing SLS algorithms.

4 Quantity of Randomness

In the previous section, we examined how the quality of random numbers can affect SLS behaviour. In this section, we will study the quantity of random decisions made by SLS algorithms, and consider how many random decisions are truly required. We first investigate random decisions in the SAPS algorithm and give a quick review of our previous work [11]. It has been observed that high-performance dynamic local search algorithms, such as ESG or SAPS, become essentially deterministic after an initial search phase [22]. Intuitively, the clause penalties become unique after numerous scaling and smoothing steps, and so there is no heuristic tie breaking necessary. To further investigate the role of randomness in these algorithms, we have previously created and studied a mostly derandomised variant of SAPS known as SAPS/NR [11].

SAPS/NR does not perform any random walk steps at local minima, uses periodic smoothing after every $(\lfloor 1/P_{smooth} \rfloor)$ local minima, and breaks all ties by selecting the variable with the smallest index. At first glance, it may seem that SAPS/NR is completely deterministic, but we must emphasise that the initialisation of SAPS/NR is identical to the initialisation in SAPS, and consequently the initial starting position for each run of SAPS/NR is completely random. In Figure 1 we compare the performance differences between SAPS and SAPS/NR. The `ferry9u` instance is one of the few cases in which we have found significant performance differences; in the overwhelming majority of cases, both algorithms show no significant performance differences.

Instance	SAPS		SAPS/NR	
	Mean	*c.v.*	Mean	*c.v.*
uf100-med	1 075	0.95	1 041	1.01
uf250-hard	287 907	0.98	292 488	0.96
bw-large.c	13 413 962	0.98	14 510 361	1.05
ferry9u	1 883 606	1.03	3 179 808	1.06

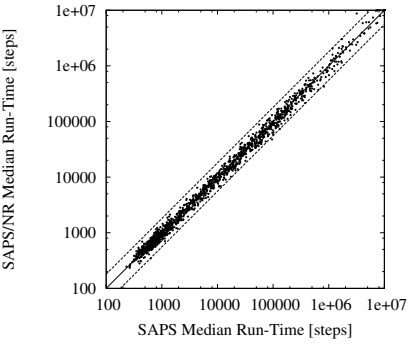

Fig. 1. Performance comparison of SAPS and SAPS/NR. *Left:* For each instance, SAPS and SAPS/NR were run 1000 times. For a description of the *c.v.*, see Table 2. *Right:* Correlation of median run-time over 100 runs on each instance of set `anov10M-struct`. Using the Mann-Whitney U-test with sample size 100, performance ratios below 1.8 (corresponding to data points inside the band drawn around the main diagonal of the plot) are not statistically significant at standard significance and power levels [1].

After restricting all of the random decisions to the initialisation phase, we will next consider what happens when we remove the random decisions from the initialisation phase as well. If we deterministically initialise the variable assignments, SAPS/NR will always take the same number of steps to solve an instance, reducing the variability in the run-time to zero, which can be seen in Figure 3 *(left)* as a vertical line. The deterministic initialisation method we used was a simple greedy approach: for each variable, if the positive literal appears more frequently than the negative, the variable is assigned a value of $\top$, otherwise $\bot$. When variables with an equal number of positive and negative literals are encountered, they are deterministically assigned $\top$ or $\bot$, alternating between variables.

We next consider what happens if between the initialisation and the search phase we select one variable uniformly at random and flip it [7]. Remarkably, as can be seen in Figure 3 *(left)*, the variability introduced by just that one random decision is close to the full variability seen by the regular, fully randomised version of SAPS. Because this instance has 250 variables, there are 250 discrete levels in the curve, corresponding to each of the 250 variables that could have been flipped. It is quite remarkable and rather counter-intuitive that flipping just one variable between the initialisation and search phase could have such a dramatic effect on the run-time behaviour of the algorithm. We note that this phenomenon is very reminiscent of the extremely sensitive dependence on initial conditions found in chaotic dynamic systems.

Next, we consider similar derandomisations for CRWALK and ANOV$^+$, two algorithms that depend on random decisions to a much greater extent than SAPS. It should be noted that the derandomised versions of these algorithms described in the following were chosen for their simplicity rather than for their performance or their exceptionally strong correlation to the original algorithms. We did not invest time in tuning and engineering our algorithms with different derandomisation strategies to meet higher quality

[7] Parameters `-varinitgreedy -varinitflip 1` in UBCSAT.

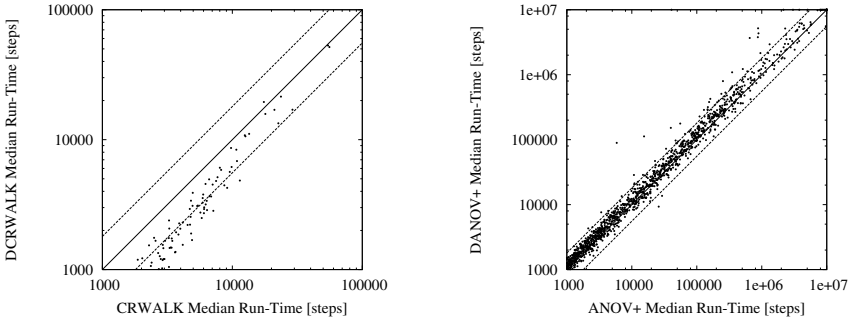

Fig. 2. Comparison of *left:* CRWALK and DCRWALK and *right:* ANOV$^+$ and DANOV$^+$. Instance sets *left:* `flat30-*` and *right:* `anov10M-struct`. For each instance, 100 runs were performed. See Figure 1 for further details.

standards. Our goal was to illustrate that our simple, straightforward approach works reasonably well for most instances.

Recall that CRWALK uses random decisions to select unsatisfied clauses and to decide which variable in a selected clause is to be flipped. To implement clause selection in DCRWALK, our deterministic version of CRWALK, we keep track of the number of times each clause has been selected (count) and the number of steps that each clause has been unsatisfied (unsat) and we simply select the clause that has the smallest (count: unsat) ratio, breaking ties by selecting the clause with the smallest index. This method ensures that clauses are selected in a uniform, fair, and deterministic manner. For literal selection, we simply keep a counter for each clause, selecting the first literal the first time the clause is selected, the second literal the second time, and so on, returning to the first literal when all have been exhausted. Thus, DCRWALK removes all of the randomness from the heuristic search phase, while still allowing for random decisions at the initialisation phase. Note that our approach differs substantially from some of the published theoretical methods for derandomising Schöning's algorithm [23], which use Hamming balls to eliminate randomness from the initialisation phase and depart from traditional SLS by using backtracking in the local search phase.

To derandomise the ANOV$^+$ algorithm, we need to replace three types of random decisions: clause selection, random walk steps, and noisy variable selection. For clause selection, we maintain a list of the currently false clauses and simply step through that list, selecting the clause in the list that is the current search step number modulo the size of the list. Instead of random walk steps, every ($\lfloor 1/wp \rfloor$) steps a variable is selected to be flipped using the same variable selection scheme used by DCRWALK. For the noisy variable selection, we use two integer variables n and d. If the ratio ($\frac{n}{d}$) is less than the current noise setting p a noisy decision is made and n is incremented, conversely, if ($\frac{n}{d}$) is greater than p the greedy decision is made and d is incremented. Whenever the adaptive mechanism modifies the noise parameter p, the values of n and d are reinitialised to $\lfloor 256 \cdot p \rfloor$ and $(256 - n)$, respectively.

In Figure 2 we compare the performance of DCRWALK and DANOV$^+$ with their fully randomised versions. In general, we do not see the same tight correlation observed for SAPS/NR, however, for the most part our derandomised algorithms show very

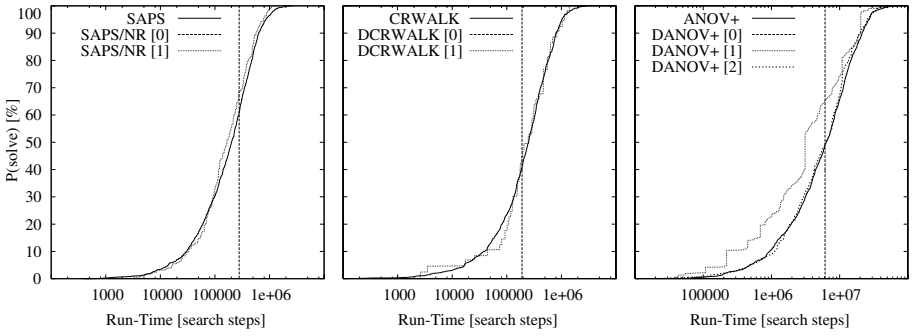

Fig. 3. A run-length distribution comparison of *left:* SAPS and SAPS/NR, *centre:* CRWALK and DCRWALK and *right:* ANOV$^+$ and DANOV$^+$ (with [N] total random decisions per run) based on 1000 runs. See the text for the deterministic initialisation method used for the derandomised algorithms. The vertical bar ([0]) reflects when *all* random decisions have been replaced, while the [1] curve shows the behaviour when one single random variable has been flipped after the deterministic initialisation in each run. Instances are *left:* `uf250-hard`, *centre:* `uf50-hard` and *right:* `bw-large.c`.

similar behaviour. Our DCRWALK algorithms seems to outperform CRWALK for the vast majority instances, possibly because the clause selection scheme is fair and unbiased. Gent and Walsh observed similarly improved behaviour for a fair deterministic version of GSAT [7]. Our DANOV$^+$ algorithm suffers from slightly worse performance on average, and there are significant outliers that indicate some inherent problems with our derandomisation approach on some specific instances, but for most instances the performance of DANOV$^+$ resembles that of ANOV$^+$.

In Figure 3 we see evidence that the same 'chaotic' behaviour observed for SAPS/NR is also present for DCRWALK and DANOV$^+$. Using the same deterministic initialisation as in SAPS/NR, we obtain the same behaviour: with just one simple random decision in DCRWALK and two in DANOV$^+$, the full variability found in the run-time distributions of the original, heavily randomised versions of these algorithms is achieved. What makes this observation remarkable is not so much that in principle, the amount of random decisions can be drastically reduced without any substantial effect on the behaviour of the algorithm (after all, any implementation of an SLS algorithm using a PRNG is fully deterministic), but rather that it can be done using very simple derandomisation schemes.

5 Conclusions

In this paper we have investigated the role of random decisions in SLS algorithms for SAT. Most of these algorithms heavily use various types of random decisions, and we have argued that from a theoretical point of view, their performance can be expected to be severely compromised by some of the features associated with poor-quality random number sequences. Nevertheless, our our empirical results indicate that in practice, the behaviour of these algorithms is remarkably robust with respect to the quality of the

RNG used to implement these random decisions. This is in contrast to some other types of randomised algorithms, such as algorithms used for Monte Carlo simulations. As a consequence, there is no reason to consider the use of true random number generators (which have the disadvantage of typically being rather slow), or to worry about minor differences in the quality of readily available pseudo-random number generators, especially if their period is high. Because of its extremely high period, efficiency and platform-independent availability, we recommend to use the Mersenne Twister PRNG for the implementation of SLS algorithms.

We have also found that at least the three prominent SLS algorithms for SAT we studied (SAPS, ANOV$^+$, CRWALK) can be almost completely derandomised using very simple mechanisms to replace the random decisions without significantly changing their behaviour. In particular, versions of these algorithms that use only a single random decision during intialisation exhibit basically the full variability in the run-time required to solve a given SAT instance as the original, fully randomised algorithms. Eliminating this last random decision leads to completely deterministic algorithms which may often perform similarly well as their fully randomised versions on average. At the same time, these deterministic algorithms can no longer benefit from easy and efficient parallelisation by means of performing multiple independent tries in parallel [1].Additionally, at least for the deterministic version of ANOV$^+$ we observed substantially degraded performance on a very small number of instances. Therefore, we see no practical advantages in using completely or partially derandomised SLS algorithms.

Overall, our results are fully consistent with the widely held view that the role of random decisions in SLS algorithms is primarily to provide search diversification. Therefore, neither the quality of the RNG nor the quantity of random decisions used by an SLS algorithm is of crucial importance to its behaviour.

In future work, it would be interesting to conduct a detailed empirical analysis on the implementation costs of various PRNGs and the difference in run-time behaviour between randomised and derandomised algorithms. With respect to the quantity of random decisions, more algorithms can be tested for straightforward derandomisation, and more robust derandomisation methods should be explored. For individual instances on which derandomised algorithms are found to perform poorly (*i.e.* `ferry9u` for SAPS/NR), it would be interesting to further explore the reasons underlying the loss of performance, and to investigate which specific type of derandomisation is causing the problem; this information could be used to help identify how to use random decisions more effectively. Finally, it would be very worthwhile to extend our methods to other combinatorial problem domains (*e.g.*, constraint satisfaction or travelling salesperson problems) to test the generality of our observations.

References

1. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann (2004)
2. Papadimitriou, C.H.: On selecting a satisfying truth assignment. In: Proc. of the 32nd Symp. on Foundations of Computer Science. (1991) 163–169
3. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proc. of the 12th Nat'l Conf. on Artificial Intelligence (AAAI 94). (1994) 337–343

4. Schöning, U.: A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: Proc. of the 40th Symp. on Foundations of Computer Science. (1999) 410–414
5. Hutter, F., Tompkins, D.A., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: Proceedings of CP-02. (2002) 233–248
6. Éric D. Taillard: Robust taboo search for the quadratic assignment problem. Parallel Computing **17** (1991) 443–455
7. Gent, I.P., Walsh, T.: Towards an understanding of hillclimbing procedures for SAT. In: Proc. of the 11th Nat'l Conf. on Artificial Intelligence (AAAI 93). (1993) 28–33
8. Ferrenberg, A.M., Landau, D.P., Wong, Y.J.: Monte Carlo simulations: Hidden errors from "good" random number generators. Physical Review Letters **69** (1992) 3382–3384
9. Bauke, H., Mertens, S.: Pseudo random coins show more heads than tails. Journal of Statistical Physics **114** (2004) 1149–1169
10. Ribeiro, C.C., Souza, R.C., Vieira, C.E.C.: A comparative computational study of random number generators. Pacific Journal of Optimization **1** (2005) 565–578
11. Tompkins, D.A.D., Hoos, H.H.: Warped landscapes and random acts of SAT solving. In: Proc. of the 8th Symp. on Artificial Intelligence and Mathematics. (2004)
12. Iwama, K., Tamaki, S.: Improved upper bounds for 3-SAT. In: Proc. of the 15th ACM-SIAM Symp. on Discrete algorithms (SODA 04). (2004) 328–328
13. Parkes, A.J., Walser, J.P.: Tuning local search for satisfiability testing. In: Proc. of the 13th Nat'l Conf. on Artificial Intelligence (AAAI 96). (1996) 356–362
14. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proc. of the 18th Nat'l Conf. on Artificial Intelligence (AAAI 02). (2002) 655–660
15. Le Berre, D., Simon, L.: 55 solvers in Vancouver: The SAT 2004 competition. In: Proc. of 7th Conf. on Theory & Applications of Satisfiability Testing (SAT 04). (2005) 321–345
16. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In: Proc. of (SAT 04). (2005) 305–320
17. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Modeling & Comp. Simulation **8** (1998) 3–30
18. Hoos, H.H., Stützle, T.: SATLIB: An online resource for research on SAT. In: SAT2000: Highlights of Satisfiability Research in the year 2000. (2000) 283–292
19. Bagini, V., Bucci, M.: A design of reliable true random number generator for cryptographic applications. In: Workshop Cryptographic Hardware & Embedded Syst. (1999) 204–218
20. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report 800-22, NIST (2000)
21. Knuth, D.E.: The Art of Computer Programming, Volume 2. Addison-Wesley (1969)
22. Schuurmans, D., Southey, F., Holte, R.C.: The exponentiated subgradient algorithm for heuristic boolean programming. In: Proc. of (IJCAI 01). (2001) 334–341
23. Dantsin, E., Goerdt, A., Hirsch, E.A., Schöning, U.: Deterministic algorithms for k-SAT based on covering codes & local search. In: Automata, Languages & Prog. (2000) 236–247