EagleUP: Solving Random 3-SAT using SLS with Unit Propagation

Oliver Gableske[1] and Marijn Heule[2]

[1] Institute of Theoretical Computer Science, Ulm University, Germany
[2] Algorithmics Group, Delft University of Technology, The Netherlands

Abstract. This paper introduces a novel approach which combines SLS SAT solving and unit propagation to realize fast performance on huge satisfiable uniform random 3-CNF formulas. Currently, applying unit propagation within local search solvers was only beneficial for structured formulas. We show how unit propagation can be combined with SLS solvers to realize speed-ups on random formulas as well. Our novel approach includes techniques of systematic SAT solving, like a variable selection heuristic that originates from a look-ahead solver. Additionally, we developed a scheme using the Cauchy probability distribution to decide when to perform unit propagation.
We present our SLS solver `Eagle`, which is three times faster compared to `TNM`, the winner of the last SAT competition on satisfiable random formulas. We also show that `EagleUP`, the enhanced version with unit propagation, experiences an additional speed-up of about 15% to 22%.

Keywords: SLS, unit propagation, Cauchy distribution, RW heuristic

1 Introduction

SAT is one of the most studied combinatorial problems, and while the interest in it was once only academic, the progress in the construction of practically applicable SAT solvers has brought much attention to the topic of solving real-world application and industrial problems with it. However, not all designs of SAT solvers can compete equally well on all types of problems. While systematic approaches that perform considerable amounts of reasoning are often more useful on application and crafted instances, local search approaches that perform no reasoning at all are superior on large satisfiable random instances that usually do not contain any exploitable structure.

The most simple form of reasoning is called *unit propagation* (or Boolean constraint propagation) [20], abbreviated with *UP*. It is based on propositional logic and exploits the fact that in order to create a satisfying assignment for a formula in conjunctive normal form (CNF), it is required to satisfy all the clauses of such a formula. These formulas can contain unit clauses, i.e., clauses that have all but one literal falsified. A unit clause can only be satisfied if its last unassigned literal is made true, i.e., the assignment to the corresponding variable is directly implied. UP discovers implications given by unit clauses and forces the assignments to the corresponding variables.

While the application of UP is of vital importance in systematic search solvers to solve structured problems of the SAT competitions [17], its application in stochastic local search (SLS) solvers is rare. One of the few examples is the `UnitWalk` solver [7] which assigns variables based on a random order and only flips the truth values in case a variable occurs in a unit clause. In comparison to pure SLS solvers, `UnitWalk` shows strong performance on many structured formulas, yet it cannot compete on random ones. This observation inspired the development of the `QingTing` [11] solver which switches between SLS and UnitWalk. A different approach of stochastic systematic search has been proposed in [13], and applies randomization within backtrack search algorithms. Nevertheless, the application of UP in SLS solvers to solve random k-CNF formulas only seemed to weaken the performance.

The goal of our work was to show how UP can be combined with SLS solvers to boost their performance on random k-CNF formulas as well.

To do so, we apply UP if the SLS cannot improve the current assignment, similar to the `SatHys` approach [2], which combines SLS with conflict-driven clause learning (CDCL) solving. Furthermore, the order in which variables are assigned is based on the recursive weight (RW) heuristic [1, 14], which is used to solve small unsatisfiable random k-CNF formulas. Moreover, we modify the SLS solvers current assignment by overriding it with the propagated assignments of the UP. This partial modification is similar to both the approach of `UnitWalk` [7] and the phase-saving heuristic, a value ordering heuristic used in CDCL solvers [16]. Additionally, in contrast to other approaches, we directly return to SLS when UP finds a conflict. Last, as opposed to most related work, we do not perform UP after every flip of the SLS solver. Instead, we force the SLS solver to search without another call to UP until a given number of flips, referred to as cool-down periods, has passed. The computation of the lengths of the cool-down periods is done using the Cauchy probability distribution. The application of cool-down periods that are computed using a probability distribution has, to our best knowledge, never been realized before to combine UP with SLS.

To provide evidence for the feasibility of our approach, we implemented an SLS solver called `Eagle` following the Sparrow scheme [3]. Its extension `EagleUP` is combined with UP based on the ideas above. We tested both solvers on numerous large satisfiable uniform random 3-CNF formulas with a clause-to-variable ratio of 4.2 (the hardest ratio from the SAT competition used for such formulas). The tests show that `EagleUP` is about 15% to 22% faster than `Eagle`, thus showing that combining UP with SLS solvers can boost their performance on large satisfiable random 3-CNF formulas. Moreover, we show that both `Eagle` solvers clearly outperform `TNM` which was the best solver for satisfiable uniform random formulas during the SAT 2009 competition.

The remainder of the paper is structured as follows. Section 2 briefly explains SAT and UP. Section 3 gives an overview of the SLS solver `Eagle`. Section 4 explains how UP is combined with `Eagle` to yield `EagleUP`. It also explains the RW heuristic, as well as the cool-down periods along with the Cauchy distribution. Section 5 presents an empirical study that compares `Eagle`, `EagleUP` and `TNM`. Section 6 concludes the paper.

2 Preliminaries

2.1 *k*-SAT

Given a set of n Boolean *variables* $\mathcal{V} = \{x_1, \ldots, x_n\}$ and the set of corresponding *literals* $\mathcal{L} = \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$ we use the logical operation OR ($\vee$) to create *clauses*, i.e., $c_i = (x_1 \vee \neg x_3 \vee \ldots \vee x_8)$. In k-SAT, each clause contains exactly k different literals. The set of clauses is denoted $\mathcal{C}$. Using clauses and the logical operation AND ($\wedge$) we can construct formulas, i.e., $F = c_1 \wedge \ldots \wedge c_m$. Formulas of that type are in *conjunctive normal form* (k-CNF for short). Formulas can be understood as sets of clauses, while clauses can be understood as sets of literals. The number of clauses in a given formula is denoted m, and $r = m/n$ is its ratio.

A (possibly partial) mapping $\alpha : \mathcal{V} \longrightarrow \{0, 1\}$ is called an *assignment*. If α maps *all* variables to a Boolean value it is called *total*. We write $c(\alpha) = 1$ if assignment α *satisfies* clause c. We say α *satisfies* F if it satisfies all clauses in F, and denote this by $F(\alpha) = 1$. A formula that has at least one satisfying assignment is called *satisfiable*, and *unsatisfiable* otherwise. Assignments can be understood as sets of assignments to single variables, e.g., $\alpha = \{x_2 = 0\}$ which assigns only variable x_2 to false. We say a *literal is assigned by* α if the corresponding variable is assigned by α. A clause is called *falsified* if it has all its literals evaluate to false under a given assignment. A clause is called *unsatisfied* if it has no literal evaluate to true under the given assignment but contains a not yet assigned variable. The k-SAT problem is then the task to detect whether a given k-CNF formula is satisfiable.

Before presenting our SLS solver `Eagle`, we discuss the `iUP` algorithm which is used by systematic search SAT solvers.

2.2 Iterative Unit Propagation `iUP`

Let F be a k-CNF formula. For the remainder of this paper, let β be an assignment that is used by the iterative unit propagation algorithm `iUP` (Fig. 1) as follows. `iUP` starts with an empty assignment β and tries to extend it to satisfy F. The extension of β is done in two ways.

In case unit clauses are present (yet unsatisfied clauses with only one unassigned literal), `iUP` will force an assignment to the last remaining variable in that clause such that the clause becomes satisfied.

In case no unit clause is present, `iUP` selects an unassigned variable according to a given variable selection heuristic `VAR`. There exists a vast body of work on variable selection heuristics [9]. Most of these heuristics have been optimized for random (unsatisfiable) k-SAT instances. Probably the most effective one, based on the SAT 2009 competition, is the RW heuristic used in the look-ahead based `march` solver [14]. Therefore, we decided to use this heuristic in our approach. Details about this heuristic are discussed in Section 4.1.

After a variable is selected, a value selection heuristic `VAL` should decide the preferred truth value for that variable. These value selection heuristics are potentially very powerful: perfect value selection heuristics, which are computable in polynomial time, would solve any SAT problem in a single `iUP` run – showing

that $\mathcal{P} = \mathcal{NP}$. However, little is known about effective value selection heuristics. Currently the most commonly used heuristic, both in local search and complete search, is known as phase-saving [16]. It selects the truth value based on a reference assignment. This assignment stores for each variable the last truth value to which the variable was assigned. The local search solver `UnitWalk` [7] as well as most CDCL solvers use this value selection heuristic, and we decided to use it too. In the context of our work, the reference assignment for phase-saving is provided by the SLS solver when a call to `iUP` is performed.

Ideally, `iUP` propagates all variables without detecting a conflict, i.e., it finds a satisfying assignment. However, in most cases, conflicts arise. There are two possible ways to deal with them. First, as done in `UnitWalk` [7], one can ignore conflicts and keep propagating other variables. Second, as done by most complete SAT solvers, one can stop UP and unassign those variables that are causing the conflict. Additionally, CDCL solvers add a conflict clause to avoid hitting the same conflict again. We decided to stop UP as soon as a conflict is detected, but we do not add conflict clauses because that does not seem to work on random formulas. Fig. 1 summarizes the functioning of `iUP`.

`iUP`(CNF formula F, variable sel. heur. `VAR`, value sel. heur. `VAL`, conflictStopFlag)
  Initialize $\beta := \{\}$; //start with an empty assignment
  REPEAT
    IF there is an unsatisfied clause in F that has all but one literal assigned under β
    THEN assign the corresponding variable in β such that it satisfies the clause;
    ELSE use `VAR` to select a variable unassigned in β; use `VAL` to assign it in β;
  UNTIL β assigns all variables **OR** (conflictStopFlag **AND** conflict is found)
  return β;

Fig. 1. The algorithm performs iterative unit propagation on the CNF formula F using heuristics `VAR`, `VAL` and a flag that determines whether to stop once conflicts emerge.

3 The Sparrow-like SLS Solver `Eagle`

The major goal of our work was to see if `iUP`, as explained in Fig. 1, is capable of improving the performance of a given SLS solver. The SLS solver we used for our studies is called `Eagle` and is a re-implementation of the `Sparrow` solver as presented in [3]. In comparison, our from-scratch implementation `Eagle` applies improved data structures for computing performance critical values like the Sparrow probabilities, but for the sake of simplicity we will not give any implementation details here. This section covers the general functioning of a Sparrow-like solver and gives an overview of how `Eagle` performs search.

A Sparrow-like SLS solver is quite similar to G2WSAT solvers. G2WSAT solvers follow the approach in [10] and work in *two modes*. Starting from a random total assignment α, they compute variable scores that basically reflect how big the improvement in terms of satisfied clauses in the formula will be when they invert a single variable assignment in α. The inversion of a single variable assignment from $x_i = 1$ to $x_i = 0$ or vice versa is called a flip. The computation of variable scores along with clause weighting schemes like PAWS [18] is complex and requires numerous data structures. For the sake of simplicity we will not

cover these schemes here and refer the reader to [10] and [18]. Variables with positive score can be *promising variables*. Briefly stated, a promising variable is a variable that has a positive score because of flips made to *other* variables. A variable with negative score that is flipped has positive score afterwards, but re-flipping it will only undo a step made before and so these variables are not considered promising. For a more detailed description see [10].

If promising variables are present, G2WSAT solvers will pick the one with highest score and flip it (*greedy mode*), breaking ties in favor of the least recently flipped variable. If no promising variable is present, the solver uses a heuristic to decide which variable to flip despite the fact that this will not yield any immediate improvement (*random mode*). We call this a *dead end*. There are numerous heuristics that G2WSAT solvers can use when working in random mode. A well-known one is *Novelty+* [8] as used in the solver `gNovelty+` [15].

Sparrow introduced a new heuristic that helps G2WSAT solvers to escape from dead ends. Let us assume the following situation in order to explain the Sparrow heuristic. The solver resides in a dead end α. At least one clause is now falsified, and the solver picks one of the falsified clauses at random. Let this clause be $u_i = (x_{i_1} \vee \ldots \vee x_{i_k})$. For all the variables contained in u_i, the solver computes a probability to flip this variable as follows:

$$p(x_{i_j}) = \frac{p_s(x_{i_j}) \cdot p_a(x_{i_j})}{\sum_{l=1}^{k} p_s(x_{i_l}) \cdot p_a(x_{i_l})} \text{ with } p_s(x_{i_l}) = a_1^{\mathrm{s}(x_{i_l})}, \text{ and } p_a(x_{i_l}) = \left(\frac{\mathrm{f}(x_{i_l})}{a_3}\right)^{a_2}$$

In the above equations, $\mathrm{s}(x_{i_l})$ is the current score of variable x_{i_l} according to the G2WSAT scheme. Furthermore, $\mathrm{f}(x_{i_l})$ is the number of flips that passed since variable x_{i_l} was flipped last. The constants $a_1 = 4, a_2 = 4$, and $a_3 = 26500$ have been determined experimentally (used in `Eagle` and `EagleUP`). It is worth noting that this randomized heuristic makes the application of *tries* and *cutoffs* obsolete. Fig. 2 summarizes the functioning of `Eagle`.

```
Eagle(CNF formula F, timeout t)
    Initialize random total assignment α; Set flips := 0;
    WHILE α does not satisfy F AND timeout t is not yet reached
        calculate scores for all variables;
        IF promising variables exist
        THEN //greedy mode
            x := pick promising variable with highest score,
                 breaking ties in favor of the least recently flipped;
            flip x; flips++;
        ELSE //random mode
            pick random unsatisfied clause u; compute sparrow probab. for all x ∈ u;
            randomly pick a variable according to the computed probabilities;
            flip the picked variable; flips++; adapt clause weights(smooth-prob. 0.347);
    ENDWHILE
    IF α satisfies F THEN output α; ELSE output UNKNOWN;
```

Fig. 2. The functioning of `Eagle` following the approach from [3].

Sparrow-like solvers have proven to be very competitive on large satisfiable random 3-SAT instances. Both Sparrow implementations, the one given in [3] and our own implementation `Eagle`, were about three times faster than `TNM`, which was the winner of the SAT 2009 competition in that category. The Sparrow scheme is therefore a good starting point if a fast new solver is to be implemented. Additionally, improving algorithms that are very good by themselves is usually much harder than improving algorithms with less good performance. Therefore, improving the performance of a Sparrow-like solver with `iUP` is considered to be a non-trivial task and a useful result. How this is done exactly will be covered in the next section.

4 Enhancing `Eagle` with `iUP`

As already stated in the introduction, UP has the ability to improve the performance of SLS solvers. In order to enhance a given SLS solver with UP, one must answer five questions regarding `iUP`:

- Questions regarding the "What" (further covered in Section 4.1):
  1. What variable selection heuristic `VAR` should one use? We decided to use the RW heuristic to create a static ordering of the variables, and pick the first variable according to this ordering that has not yet been propagated during the current call of `iUP`.
  2. What value selection heuristic `VAL` should one use? We picked phase-saving using a dead end from the SLS as reference assignment.
  3. What about conflicts? We decided to stop `iUP` if a conflict emerges.
- Questions regarding the "When" (further covered in Section 4.2):
  4. When to perform `iUP`? We call `iUP` if the SLS is in a dead end. Furthermore, we use cool-down periods to control the frequency of such calls. The lengths of the cool-down periods are computed using the Cauchy probability distribution.
- Questions regarding the "How" (further covered in Section 4.3):
  5. How to use the resulting assignment β of an `iUP` call? We modify the current dead end assignment α of the SLS solver by partially overriding it with β.

More elaborate answers to the above questions are given in the next sections.

4.1 Regarding the "What" – the `VAR`, `VAL`, and RW heuristics

This section presents the exact functioning of `iUP` in our solver `EagleUP`. It explains how the variable selection heuristic `VAR` and the value selection heuristic `VAL` are realized using the *recursive weight (RW) heuristic*.

The general idea of recursive weight heuristics is to help systematic search SAT solvers identify variables with strong impact on the formula. Such solvers usually pick a single variable in every node of their search tree and assign it to a not yet explored value. Picking variables with strong impact will then give a large reduction of the remaining formula, and therefore, a strong reduction in the size of the remaining search space.

RW is based on the work by Mijnders et. al. [14] as well as Athanasiou et. al. [1]. Before we explain how and why we use it, we will give a thorough explanation on how it is computed.

Given a 3-CNF formula F with n variables in $\mathcal{V}$ and $2n$ literals in $\mathcal{L}$. We write $(l \vee l' \vee l'')$ to denote a clause containing *literal* $l \in \mathcal{L}$ and *two other arbitrary literals* $l', l'' \in \mathcal{L}$. We compute the RW-score of a *variable* $x \in \mathcal{V}$ using the functions $h_i : \mathcal{L} \mapsto \mathbb{R}$ and $\mathrm{RW} : \mathcal{V} \mapsto \mathbb{R}$ up to a recursion depth of 5 as follows:

$$
\begin{aligned}
h_0(l) &= 1 \\
s_{i+1} &= \frac{1}{2n} \cdot \sum_{l \in \mathcal{L}} h_i(l) \\
h_{i+1}(l) &= \frac{1}{s_i^2} \cdot \sum_{(l \vee l' \vee l'') \in F} h_i(\neg l') \cdot h_i(\neg l'') \\
\mathrm{RW}_5(x) &= h_5(\neg x) \cdot h_5(x)
\end{aligned}
$$

We create a variable ordering θ_{RW_5} such that $\theta_{\mathrm{RW}_5}(x_i) < \theta_{\mathrm{RW}_5}(x_j) \Leftrightarrow \mathrm{RW}_5(x_i) > \mathrm{RW}_5(x_j)$. The ordering θ_{RW_5} has to be computed exactly once (i.e., it is static) and prioritizes variables with high impact on F.

When no unit clause is present, the `iUP` algorithm must decide what variable to propagate in the current iteration. This decision is made in the variable selection heuristic `VAR` which will pick the first variable according to θ_{RW_5} that has not yet been propagated by `iUP` (i.e., it is not assigned in β, see Fig. 1).

The reason why we prefer variables with high impact is that assigning these variables first creates new unit clauses sooner. Creating new unit clauses sooner then means that `iUP` relies less often on the reference assignment in future iterations. Given a satisfiable formula, this is helpful since this reference, currently not satisfying all clauses of the formula, must contain erroneous variable assignments. The less often `iUP` relies on it, the smaller the chance of propagating one of the contained errors. Fig. 3 summarizes the functioning of `VAR`.

`VAR`(variable ordering θ_{RW_5})
  pick first variable x according to θ_{RW_5} that is unassigned in β;
  return x; //This x has not yet been propagated in the current call of `iUP`.

Fig. 3. The functioning of `VAR`. See Fig. 1 for details on how it is used in `iUP`.

After the variable was selected by `VAR` it must be assigned a value to complete the current iteration of `iUP`. The value selection heuristic `VAL` we use here is phase-saving with the current dead end of the SLS solver as a reference. Fig. 4 summarizes the functioning of `VAL`.

`VAL`(reference assignment α)
  for variable x that was picked by `VAR`, assign $\beta(x) := \alpha(x)$;

Fig. 4. The functioning of `VAL`. See Fig. 1 for details on how it is used in `iUP`.

The realization of `iUP` is now completely determined by the above definitions of `VAR`, `VAL`, and the rule to stop as soon as a conflict emerges (see Fig. 1). Before we can explain how this realization of `iUP` is embedded into `Eagle` to yield `EagleUP`, we must clarify when and how often we want to call `iUP` during the local search. This is done in the next sections.

4.2 Regarding the "When" – Cauchy distributed cool-down periods

In order to embed `iUP` into an SLS solver, it must be clarified when `iUP` is to take place during the local search. A reasonable approach would be to have the SLS solver perform its local search and call for `iUP` as soon as it gets stuck in a dead end. Following this approach has two advantages. First, the SLS solver is allowed to continue its search towards a satisfying assignment without interruption as long as this search seems promising, e.g., is done greedily. Second, a call to `iUP` is only performed in situations where the SLS solver could make use of additional support in order to advance the search again, e.g., to escape from a dead end.

However, on 3-CNF instances with a ratio of 4.2, `Eagle` will encounter a dead end in about every third flip (determined experimentally). Calling `iUP` in every third flip would mean that multiple `iUP` calls rely on almost identical reference assignments. Given the static variable ordering θ_{RW_5}, the chance for different outcomes of these `iUP` calls is extremely small. Therefore, two calls of `iUP` should be separated by more than three flips in order to not waste computational time. We call this number of separating flips the *cool-down period* and denote it by $\mathfrak{c}$.

In short, a cool-down period $\mathfrak{c}$ is to be computed after every call to `iUP`. The necessity for cool-down periods raises the questions on how long they have to be, what they depend on, and how they get computed. We have tried several approaches in order to answer these questions. We first used fixed values for $\mathfrak{c}$. This resulted in `EagleUP` having a performance that is worse than the performance of `Eagle`. We then tried to compute $\mathfrak{c}$ based on the number of variables n. This worked for some formulas and we came to the conclusion that the formula size must be part of any calculation for $\mathfrak{c}$. We then defined an interval, based on n, from which $\mathfrak{c}$ is drawn uniformly at random and were able to increase the number of formulas where speedups could be found. Additionally, we realized (empirically) that the performance gains for `EagleUP` where quite stable independent from the specific sizes of the intervals checked (we investigated $\mathfrak{c} \in [n \pm 0.5n, 2.5n \pm 0.5n]$). However, due to space constraints we will not present this here in detail. As a result of this, we conjectured that the size of the interval is of less importance to the performance than the distribution of the values picked from it. This lead us to the question whether any other distribution than the uniform random one might work better.

We then tried the binomial and the Cauchy distribution with different intervals, and extensive tests revealed that the Cauchy distribution gives the best results. In comparison, the uniform distribution resulted in about 12% speedup, the Cauchy distribution resulted in about 17% speedup. The remainder of this section explains the details of the Cauchy distribution and how we use it to compute the cool-down period $\mathfrak{c}$.

The *Cauchy distribution* is defined as follows. Let $\gamma \in \mathbb{R}, \gamma > 0$ and $\omega \in \mathbb{R}$. The Cauchy distribution is given by the *probability density function* (PDF)

$$c : \mathbb{R} \mapsto \mathbb{R},\ c(z) = \frac{1}{\pi} \cdot \frac{\gamma}{\gamma^2 + (z - \omega)^2}$$

Its *cumulative distribution function* (CDF) is

$$C : \mathbb{R} \mapsto \mathbb{R},\ C(z) = P(Z < z) = \frac{1}{2} + \frac{1}{\pi} \cdot \arctan\left(\frac{z - \omega}{\gamma}\right).$$

Since the density c is symmetric and has its mean in ω, the parameter ω is called the center of the distribution. The parameter γ is called the *width* parameter as it describes how strong the decay around the center is in the PDF. A width parameter close to 0 will give a strong decay in the PDF and a strong increase in the CDF (see Fig. 5). For more details about the Cauchy distribution see [12]. Actual computations use a discretized CDF of the Cauchy distribution.

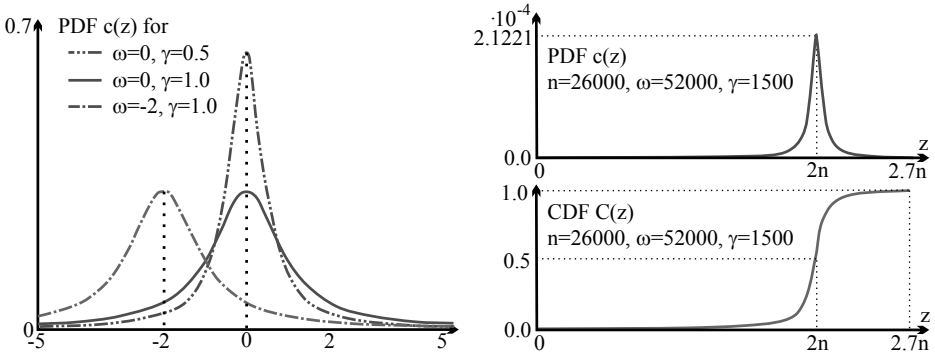

Fig. 5. Left hand side: the shapes of PDFs of the Cauchy distribution given different values for ω and γ. Right hand side: examples for the PDF and CDF of the Cauchy distribution for a formula with $26,000$ variables and optimal values for ω and γ.

The idea is to pick a number $a \in [0, 1)$ uniformly at random and identify the smallest cool-down length $\mathfrak{c}$ with $C(\mathfrak{c}) \geq a$. The values for $\mathfrak{c}$ are then distributed according to the Cauchy distribution.

Our tests revealed that the interval $[0, 2.7n]$ is reasonable for discretization, meaning that the cool-down periods have lengths ranging from 0 flips up to $2.7n$ flips. Further, we determined the values $\gamma = 1500$ and $\omega = 2n$ to be suitable (see Fig. 5, right hand side). We have also studied the stability of performance gains under modification of these parameters and checked the interval $[0, 2.7n \pm 0.3n]$ with $\gamma = 1500 \pm 300, \omega = 2.0n \pm 0.5n$. However, no significant changes in performance gains could be found.

In summary, whenever `iUP` has been performed, a random number $a \in [0, 1)$ is picked. The length of the next cool-down period then is $\mathfrak{c} = \min\{z | C(z) \geq a\}$.

4.3 Regarding the "How" – modifying the reference assignment

As stated in the previous section, it is a viable approach to call `iUP` if the local search is stuck in a dead end α and then use the resulting assignment β of the

`iUP` call to help the SLS solver escape from it. The question is how exactly β is used to help modify α in order to relocate the local search out of the dead end.

The resulting `iUP` assignment β hosts all assignments that could be propagated without running into a conflict. Comparing α and β on all variables assigned in β provides a set $\mathcal{M}$ of variables that have changed their assignment during unit propagation. In short $\mathcal{M} = \{x \in \mathcal{V} \mid \beta(x) \text{ is defined}, \alpha(x) \neq \beta(x)\}$.

After a call to `iUP`, we create $\mathcal{M}$ and override all variables from $\mathcal{M}$ in α with the assignments from β. This effectively changes the current reference assignment and thereby places the SLS solver out of the current dead end. We refer to this as *partially overriding α with β*.

Sometimes, `iUP` does not run into a conflict, thereby producing a satisfying assignment β. However, this happened in less than 0.01% of the `iUP` calls. So in general, applying UP helps the SLS solver to escape from the current dead end to an assignment that has increased consistency regarding the searched formula.

4.4 EagleUP: Embedding `iUP` into `Eagle`

The previous sections explained how the SLS solver performs its search, how `iUP` is performed, when `iUP` is performed, and how the result of a call to `iUP` is used. The following section summarizes these explanations and presents how `iUP` is embedded into `Eagle` to yield `EagleUP`.

`EagleUP` initializes a set of data structures before the actual search starts. Given the 3-CNF formula F with n variables, it first initializes a total random assignment α for the formula that is used by the local search. Furthermore, it initializes the static variable ordering θ_{RW_5} using the RW heuristic (see Section 4.1). Additionally it pre-computes the CDF of the Cauchy distribution in $[0, 2.7n]$ (see Section 4.2) using the experimentally determined parameters $\omega = 2n$, $\gamma = 1500$. The first cool-down period is set to $\mathtt{c} = \omega$.

After the initialization phase, `EagleUP` performs local search as explained in Section 3. Local search continues in greedy mode until it gets stuck in a dead end, where it switches to random mode. In contrast to `Eagle`, the solver now checks if the current cool-down period `c` is over. If not, it continues in random mode as done in `Eagle`, following the Sparrow heuristic to decide what variable to flip next. If the cool-down period is over, a call to `iUP` is performed.

A call to `iUP` is performed using the current dead end α as a reference assignment (see Section 4.1). After the call to `iUP`, the solver performs two tasks. First, the length of the next cool-down period is computed using the Cauchy distribution (see Section 4.2). Second, the current dead end α is partially overridden with the result β from the `iUP` call (see Section 4.3).

The above scheme is repeated until the solver either finds a satisfying assignment for the formula, or if a given timeout is reached (see Fig. 6).

5 Empirical Study

We performed an empirical study in order to test the feasibility of our approach. We will first explain the general setup of this study, then present the results, and finally give a brief discussion of the results.

```
EagleUP(CNF formula F, timeout t)
    Initialize random total assignment α; Set flips := 0; Set lastIUPCall := 0;
    Initialize θ_RW5 using the RW heuristic; //θ_RW5 is not modified again, i.e., is static
    Compute Cauchy CDF C(z), z ∈ [0, 2.7n] with ω := 2n and γ = 1500; Set c := ω;
    WHILE α does not satisfy F AND timeout t is not yet reached
        calculate scores for all variables;
        IF promising variables exist
        THEN //greedy mode
            x := pick promising variable with highest score,
                breaking ties in favor of the least recently flipped;
            flip x; flips++;
        ELSE //random mode
            IF flips > lastIUPCall + c
            THEN //do iUP
                randomly pick a ∈ [0, 1) and set c := min{z|C(Z) ≥ a};
                lastIUPCall := flips;
                α := iUP(F, VAR(θ_RW5), VAL(α), true); //partially override α with β
            ELSE //do Sparrow
                pick random unsatisfied clause u; compute sparrow prob. for all x ∈ u;
                randomly pick a variable according to the computed probabilities;
                flip the picked variable; flips++; adapt clause weights(sp=0.347);
    ENDWHILE
    IF α satisfies F THEN output α; ELSE output UNKNOWN;
```

Fig. 6. The functioning of `EagleUP`. See Fig. 1,3,4 for details on `iUP`, `VAR` and `VAL`.

Setup of the empirical study. Our empirical study was carried out on the bwGRiD [4], which provided us with 64 Intel Harpertown quad-core CPUs with 2.83Ghz and 8GB RAM each. The operating system on the bwGRiD is Scientific Linux. The support software to carry out the study is the EDACC system [6].

The solvers we used in our study were `TNM`, in the version that was presented at the SAT 2009 competition [17], as well as our new SLS solver `Eagle` and its `iUP` enhanced version `EagleUP` (sources are available at [5]). The parameters for all solvers are kept fixed for all experiments: for `TNM`, the setting from the SAT competition is used. For the `Eagle` solvers we use $a_1 = 4, a_2 = 4, a_3 = 26500$ and the smoothing-prob. for clause weighting is 0.347. For `EagleUP`, the additional parameters for computing the cool-down periods are as explained in Section 4.2.

The benchmark is divided into four parts. Part A contains 600 satisfiable 3-CNF formulas with a clause-to-variable ratio of 4.2. Their sizes range from $20,000$ variables to $30,000$ variables in increments of $2,000$ (100 formulas each). For the sizes of $20,000$ to $26,000$ we took all formulas from the SAT 2009 competition, 10 for each size, and created new formulas according to the fixed clause length model (no tautologies, no duplicate clauses, no duplicate literals in a clause), 90 for each size. For the sizes of $28,000$ and $30,000$ we had to create all formulas ourselves. Part B contains 1300 3-CNF formulas with $26,000$ variables and various ratios (4.14 to 4.26 in increments of 0.01, 100 formulas each, all generated using the fixed clause length model). Part C contains 83 satisfiable

crafted formulas, and Part D contains 53 satisfiable application formulas, all from the SAT 2009 competition. Each of the three tested solvers had to perform 50 runs with different seeds on each formula and they all used the same seed for a given formula and run. The timeout was set to 2,000 seconds.

Results. The following results are available at [5] in more detail. Fig. 7 presents the success rate, the average runtime of the successful runs, and the average standard deviation of the runtime of the successful runs. Furthermore, it states the speedup in percent of `Eagle` over `TNM`, and `EagleUP` over `Eagle`.

Part A Solver	succ. rate [%]	avg. run time [s]	avg. std. dev. [s]	speed up [%]	succ. rate [%]	avg. run time [s]	avg. std. dev. [s]	speed up [%]	succ. rate [%]	avg. run time [s]	avg. std. dev. [s]	speed up [%]
	v20,000, r4.2				v22,000, r4.2				v24,000, r4.2			
TNM	77.90	708.09	389.61	76.7	68.34	899.15	434.65	76.7	58.00	899.64	401.62	68.9
Eagle	99.70	164.71	138.51	21.2	99.70	209.47	173.56	18.7	98.42	279.40	213.44	22.4
EagleUP	99.72	129.76	97.81		99.96	170.13	129.28		99.28	216.64	155.78	
	v26,000, r4.2				v28,000, r4.2				v30,000, r4.2			
TNM	49.90	1017.95	374.88	70.7	47.86	1062.19	383.74	69.9	30.32	1192.53	314.95	62.8
Eagle	97.64	297.37	229.84	16.9	97.70	318.93	234.06	15.4	95.82	443.45	310.29	16.3
EagleUP	98.18	247.07	185.92		98.76	269.73	190.01		97.94	371.05	261.43	

Part B Solver	avg. run time [s]	speed up [%]	avg. run time [s]	speed up [%]	avg. run time [s]	speed up [%]	avg. run time [s]	speed up [%]	avg. run time [s]	speed up [%]	avg. run time [s]	speed up [%]
	r4.14		r4.16		r4.18		r4.20		r4.22		r4.24*	
Eagle	9.36	6.5	29.85	11.1	94.97	16.6	297.37	16.9	763.49	6.7	1107.28	5.7
EagleUP	8.75		26.53		79.24		247.07		712.04		1043.27	

Fig. 7. Part A: The table presents the results for each solver and formula size. First, the success rate of the solver in percent. Second, the average runtime of the successful runs of the solver in seconds. Third, the average standard deviation of the run times of the successful runs in seconds. Fourth, the amount of time in percent that is saved when using `Eagle` instead of `TNM`, or `EagleUP` instead of `Eagle`. **Part B:** The table presents the average runtime of the `Eagle` solvers as well as the speedup on formulas with 26,000 variables and variable ratios. For details on Part C and D see [5].

Discussion (Part A). When taking a look at the results for `TNM` and `Eagle` one can clearly see that `Eagle` is the superior solver. The success rate of `Eagle` is at least 20% larger than the one of `TNM`. At the same time, `Eagle` can save up to 75% of the runtime of `TNM`. Furthermore, the total average standard deviation of the runtimes is about half for `Eagle` in comparison to `TNM`. Given the fact that TNM is the winner of the SAT 2009 competition random benchmark, we deem our solver to be one of the best solvers for large random 3-SAT formulas.

The UP enhanced solver `EagleUP` is able to improve the results even further. `EagleUP` has an increased success rate over `Eagle` and it reduces the average runtime of the successful runs as well as the average standard deviation of the times of the successful runs. It is worth noting that `EagleUP` experiences a near 100% success rate on formulas with 30,000 variables, while `TNM` has a success rate of about 31%. All together, we managed to improve the runtime of `Eagle` by about 15% to 22% through the application of UP, and thereby successfully applied this simple form of reasoning on the bastion of large uniform random 3-SAT formulas. All together, this empirically proves the feasibility of our approach.

The time `EagleUP` spent to perform `iUP` was about 13% of its run-time. The average number of propagated variables in a single call to `iUP` was about 42% of

all variables (10% as decision variables picked by `VAR/VAL` and 32% as implied by unit clauses). The average number of variables that got their assignments overridden after a single call to `iUP` was about 1% of all variables. Executed on a formula with 30,000 variables, `Eagle` uses about 10 MB RAM and `EagleUP` uses about 21 MB RAM.

Discussion (Part B). The results indicate that it is possible to change the ratio of formulas and still have `EagleUP` outperform `Eagle`. If the ratio is decreased, we observe a drop in speedup. This can be explained by the fact that on underconstrained formulas, numerous solutions exist: `Eagle` starts search right away and needs only a small amount of flips until it finds a solution. `EagleUP` suffers from the overhead of pre-computing the Cauchy distribution and θ_{RW_5}, even though this information is hardly used in the short search afterwards. Increasing the ratio results in a drop in speedup as well. This can be explained by the fact that comparatively few solutions exist on critically constrained formulas, which increases the chances for `iUP` to propagate wrong variable assignments. This in turn decreases its ability to increase the consistency of the current SLS assignment, resulting in less favorable positions for the following search.

Discussion (Part C and D). Given the results from part C and D (not presented here, see [5]) we can state that `EagleUP` outperforms `Eagle` on both the crafted and the application formulas. On crafted formulas `EagleUP` had a 8% higher success rate than `Eagle`, and on application formulas the success rate was about 3% higher. Additionally, on application formulas, `EagleUP` achieved the higher success rate in less computational time than `Eagle`. The studies on crafted and application formulas were only preliminary (without any parameter tuning). We therefore conclude that our scheme to combine UP with SLS has potential on these type of formulas, but it is yet too soon to make any definitive statement about the performance gains possible.

6 Conclusions and Future Work

In this paper we presented a novel approach to implement UP into SLS solvers. In contrast to the related work, our approach relies on a fixed variable ordering for performing UP that is computed using the RW heuristic. The usage of a fixed variable ordering made it mandatory to use cool-down periods in between the calls of UP. These cool-down values are computed using the Cauchy probability distribution, an approach that has, to our knowledge, never been tried before.

The results we presented show a significant improvement of `EagleUP`, our UP enhanced SLS solver, over `Eagle`, our plain SLS implementation following the Sparrow scheme. `EagleUP` is about four times faster than `TNM`, which is the winner of the SAT 2009 competition for uniform random 3-SAT formulas. On the basis of these results, we have shown that UP can successfully be applied to SLS solvers for solving large uniform random 3-SAT formulas.

The future work will include whether the static variable ordering θ_{RW_5} used by unit propagation can be replaced by a more dynamic ordering. A full dynamic ordering — i.e., recomputing RW every time before selecting a variable — is too costly to be useful in practice. Instead, we want to pre-compute a few variable orderings using the binary search tree that arises using RW as variable selection

heuristic. Additionally, we want to adapt our approach to work on structured instances. The preliminary results from the crafted and application formulas give hope that performance gains for structured formulas are also possible. Adding clause learning in `EagleUP` for structured formulas might be worthwhile too.

Acknowledgements. The authors would like to thank Julian Rüth, Martin Bader, Adrian Balint, Dominikus Krüger, Marcus Bombe, and Jacobo Torán. The first author is supported by the Graduate School *Mathematical Analysis of Evolution, Information and Complexity* in Ulm. The second author is supported by the Dutch Organization for Scientific Research NWO under grant 617.023.61.

References

1. Athanasiou, D., Fernandez, M.A.: Recursive Weight Heuristic for Random k-SAT. Technical report from Delft University. `http://www.st.ewi.tudelft.nl/sat/reports/RecursiveWeightHeurKSAT.pdf`, 2010.
2. Audemard, G., Lagniez, J.-M., Mazure, B., Saïs, L.: Boosting local search thanks to CDCL. In LPAR-17, LNCS 6397:474-488. Springer 2010.
3. Balint, A., Fröhlich, A.: Improving Stochastic Local Search for SAT with a New Probability Distribution. In SAT'10, LNCS 6175:10-16. Springer 2010.
4. bwGRiD (`www.bw-grid.de`), member of the German D-Grid initiative, funded by the *Ministry for Education and Research*, Germany.
5. `Eagle(UP)` benchmark, sources, results: `http://www.uni-ulm.de/~s_ogable/`.
6. Balint, A., Gall, D., Kapler, G., Retz, R.: Experiment Design and Administration for Computer Cluster for SAT-solvers (EDACC). JSAT 7:77-82, 2010.
7. Hirsch, E.A., Kojevnikov, A.: UnitWalk: A New SAT Solver that Uses Local Search Guided by Unit Clause Elimination. AMAI 43(1-4):91-111. Kluwer 2005.
8. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In AAAI'02:635-660, 2002.
9. Kullmann, O.: Fundaments of Branching Heuristics. Chapter 7 of Handbook of Satisfiability:205-244. IOS Press 2009.
10. Li, C.M., Huang, W.Q.: Diversification and Determinism in Local Search for Satisfiability. In SAT'05, LNCS 3569:158-172. Springer 2005.
11. Li, X.Y., Stallmann, M.F., Brglez, F.: QingTing: A fast SAT solver using local search and efficient unit propagation. In SAT'03, LNCS 2919:452-467. Springer 2003.
12. Lupton, R.: Statistics in Theory and Practice. Section 3.7, Cauchy Distribution, p. 21-22. Princeton University Press, ISBN 0-691-07429-1, 1993.
13. Lynce, I., Marques-Silva, J.: Random backtracking in backtrack search algorithms for satisfiability. Discrete Applied Math., 155(12), p. 1604-1612. Elsevier 2007.
14. Mijnders, S., De Wilde, B., Heule, M.J.H.: Symbiosis of search and heuristics for random 3-SAT. In LaSh'10, 2010.
15. Pham, D.N., and Gretton, C.: gNovelty+. Solver description for the SAT 2007 competition. `http://www.satcompetition.org/2007/gNovelty+.pdf`, 2007.
16. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In SAT'07, LNCS 4501:294-299. Springer 2007.
17. The SAT competition homepage: `http://www.satcompetition.org`.
18. Thornton, J., Pham, D.N., Bain, S., Ferreira, V.: Additive versus multiplicative clause weighting for SAT. In AAAI'04:191-196. AAAI Press 2004.
19. Wei, W., Li, C.M.: Switching Between Two Adaptive Noise Mechanisms in Local Search for SAT. TNM solver description for the SAT 2009 competition. `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.
20. Zabih, R., McAllester, D.A.: A rearrangement search strategy for determining propositional satisfiability. In AAAI'88:155-160. AAAI Press 1988.