SATUN: A complete Hybrid SAT Solver

Oliver Gableske** and Julian Rüth

Ulm University
Institute of Theoretical Computer Science
89069 Ulm, Germany
oliver.gableske,julian.rueth@uni-ulm.de

Abstract. There are two major paradigms to solve uniform random k-SAT: SLS and DPLL, both with their own advantages and drawbacks. A hybrid solver tries to combine these paradigms in such a way, that it inherits their advantages but not their drawbacks. The first hybrid solver using partitions (`hybridGM`) was a successful combination of SLS and DPLL for satisfiable formulas, but it was also incomplete. The solver presented in this work (`SATUN`) extends the idea behind `hybridGM` resulting in a complete and fast solver for satisfiable *and* unsatisfiable formulas.

1 Introduction

The propositional satisfiability problem (SAT) is one of the most studied $\mathcal{NP}$-complete problems in computer science because it has a wide range of applications such as hardware verification, planning and scheduling. In the context of this work, we understand SAT as the task in finding a satisfying assignment for a given boolean formula, or in providing the information that the given formula has no satisfying assignment.

An algorithm solving the SAT problem for a given formula is called a SAT solver. The research on practically applicable SAT solvers brought forward two major paradigms: DPLL which is based on DP [4] and SLS which is based on GSAT [21]. The DPLL paradigm covers most solvers following a systematic search approach for SAT. These solvers usually have the advantage of being complete, i.e. they will find a solution if one exists or prove that there is none. However, they are usually comparatively slow and need considerable amounts of memory. Memory limitations often result in the DPLL solver's inability to solve formulas above a certain size. The counterpart for DPLL is the SLS paradigm. It covers all solvers following a local search approach for SAT. These approaches have the disadvantage of being incomplete, i.e. they cannot prove unsatisfiability. However, they are comparatively fast on satisfiable formulas and need just a modest amount of memory. The low memory requirements of SLS solvers allow them to solve formulas that are out of reach for DPLL solvers.

Since DPLL and SLS seem to complement each other, approaches have been made to hybridize both paradigms. A hybrid SAT solver would then inherit

** Funded by the Graduate School *Mathematical Analysis of Evolution, Information and Complexity* at Ulm University.

the best of both worlds: it would be complete and it would have an SLS-like performance on satisfiable formulas allowing it to solve even large ones.

Three major approaches to hybridize SLS and DPLL have been brought forward. The first approach uses an SLS solver to support a DPLL solver [3, 6–8, 17]. The second approach uses a DPLL solver to support an SLS solver [5, 9, 14]. The third approach is peer-like, where the SLS and DPLL solvers are supposed to benefit equally from each other [6, 15].

Up to this point in research, no truly superior hybrid SAT solver had emerged. The above cited hybrid solvers either suffered on satisfiable formulas, not being able to compete with their SLS component, or on unsatisfiable formulas, not being able to compete with their DPLL component. An approach to create a peer-like hybrid solver for uniform random k-SAT is `hybridGM` [1]. `hybridGM` is an incomplete hybrid SAT solver that outperforms its SLS component.

Briefly described, `hybridGM` works by moving the focus of the DPLL-search towards areas in which very good local minima, i.e. assignments that leave only one clause unsatisfied, are found. If a formula is satisfiable, chances are to find a satisfying assignment around such minima. `hybridGM`'s DPLL component then completely checks these areas of the search space. In comparison, the hybrid's DPLL component checks many more assignments in these areas than the SLS solver would. As a result, the hybrid solver is unlikely to "miss" solutions close to such minima. The probability for the hybrid to find a solution, as soon as its SLS component moved close to it, raises in comparison to a pure SLS approach. This in turn compensates for the additional solving time performed with the DPLL component and often grants a speed-up to the hybrid solver [1].

However, `hybridGM` does not save the information that specific parts of the search space do not contain a solution. This results in the incompleteness of this solver even though it has the ability to perform a DPLL search. Since `hybridGM` is of fundamental relevance to this work, we will explain it later in more detail.

1.1 Aim and Structure of this Work

The aim of this work is to create a complete hybrid SAT solver, called `SATUN`, that is competitive with its SLS component on satisfiable formulas, *and* that is competitive with its DPLL component on unsatisfiable formulas.

The approach behind `SATUN` is to perform local search as it is done by `hybridGM`, but for a limited amount of time. The limit is set in a way that gives local search a reasonable chance to find a satisfying assignment. In case the formula is satisfiable, this will result in a performance competitive with the hybrid's SLS component. As soon as the limit is reached, the formula is expected to be unsatisfiable. `SATUN`'s DPLL component will then perform a search on the complete search space of the formula to confirm that assumption.

However, a hybrid solver strictly following the above approach will always be inferior to its DPLL component on unsatisfiable formulas. The local search in the beginning is performed in vain because it cannot detect unsatisfiability. The hybrid's DPLL component will need to detect the unsatisfiability, but it

is started with the delay produced by the local search. The pure DPLL starts without a delay and will therefore always be faster than the hybrid.

In order to overcome that problem, SATUN will extend the above approach, by making use of the local search on unsatisfiable formulas as well. It will gather additional information on the formula during local search, which is later used by the hybrid's DPLL component. The hope is that this additional information grants a speed-up to the hybrid's DPLL, and that this speed-up is large enough to compensate for the search time of the local search. In this way, SATUN would have an SLS like performance on satisfiable formulas, while not necessarily being inferior to its DPLL component on unsatisfiable formulas.

The remainder of this paper is structured as follows. In Section 2 we will discuss the way hybridGM works and use this as a basis to explain the functioning of SATUN in Section 3. We will then present an empirical study to provide evidence for the superiority of SATUN in Section 4. A conclusion is given in Section 5.

2 hybridGM

hybridGM is an incomplete hybrid SAT solver that consists of the SLS component gNovelty+ [18] and the DPLL component March_ks [10]. For readers not familiar with these solvers we suggest [8, 10, 18].

2.1 Basic terminology for hybridGM

We will now give the necessary terminology to explain hybridGM [1].

Let F be a k-CNF formula containing n variables $\{x_1, x_2, \ldots, x_n\} = \mathcal{V}$. Let $\alpha \in \{0,1\}^n$ be a complete assignment of F. The application of the assignment α on formula F is denoted by $F(\alpha) \in \{0,1\}$. Let $\beta \in \{0,1,?\}^n$ be a partial assignment. The application of a partial assignment β on F is denoted by $F(\beta)$, where $F(\beta)$ is a new formula in which, for each $i \in \{1,\ldots,n\}$, the value of β_i is assigned to x_i if $\beta_i \in \{0,1\}$ and x_i remains unassigned if $\beta_i =?$. Additionally, we assume that the immediate simplifications have been applied to $F(\beta)$.

Example 1. For the 3-CNF formula
$F = (x_1 \vee \overline{x_2} \vee x_5) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_5}) \wedge (\overline{x_3} \vee x_4 \vee x_5) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_5})$
an example for α and β could be $\alpha = (1,0,1,1,0)$ and $\beta = (?,1,1,?,?)$. Applying α and β on F yields: $F(\alpha) = 1$ and $F(\beta) = (x_1 \vee x_5) \wedge (x_4 \vee x_5) \wedge (\overline{x_1} \vee \overline{x_5})$.

In the following, let $\mathcal{S}$ be an SLS solver, let F be a k-CNF formula with n variables from set $\mathcal{V}$, and let $\alpha^{(0)}$ be a complete assignment for F.

Definition 1 (flip trajectory). *We define a sequence of flips $T_S(F, \alpha^{(0)}) = (t_1, \ldots, t_w)$ (with $w \in \mathbb{N} \cup \{\infty\}, t_j \in \mathcal{V}$), that S performs when it searches on F starting from $\alpha^{(0)}$, as a flip trajectory of $\mathcal{S}$ on F using $\alpha^{(0)}$.*

Example 2. Given the formula from example 1 and a starting assignment $\alpha^{(0)} = (0,1,0,0,0)$, a possible flip trajectory that would lead to the satisfying assignment $\alpha^{(4)} = (1,0,1,0,1)$ could be $T_S(F, \alpha^{(0)}) = (x_1, x_5, x_3, x_2)$.

Definition 2 (search space partition). *Let $d \in \{1, \ldots, n\}$ be an arbitrary but fixed threshold constant. Let $T_S(F, \alpha^{(0)}) = (t_1, \ldots, t_w)$ be a flip trajectory, and let $\alpha^{(j)}$ be the assignment that is visited by $\mathcal{S}$ in flip j. Furthermore, let j_0 ($0 \leq j_0 \leq \min\{w-j, j-1\}$) be the largest number, such that $|\{t_{j-j_0}, \ldots, t_{j+j_0}\}| \leq d$. Define the search space partition (SSP) of $\alpha^{(j)}$ as the partial assignment $\beta \in \{0, 1, ?\}^n$, such that*

$$\beta_r = \begin{cases} ?, & \text{if } x_r \in \{t_{j-j_0}, \ldots, t_{j+j_0}\} \\ \alpha_r^{(j)}, & \text{otherwise.} \end{cases}$$

Example 3. Let F be a k-CNF formula with 10 variables. Let $d = 5$. Given the flip trajectory

$$\begin{aligned} T_S(F, \alpha^{(0)}) &= (t_1, t_2, t_3, t_4, t_5, t_6, \underline{\mathbf{t_7}}, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, \ldots) \\ &= (x_2, x_6, x_1, x_9, x_1, x_6, \underline{\mathbf{x_1}}, x_3, x_9, x_1, x_1, x_8, x_3, \ldots), \end{aligned}$$

and the complete assignment in flip $j = 7$, $\alpha^{(7)} = (0, 0, 1, 1, 0, 1, 0, 1, 1, 1)$, we create the SSP β as follows. First we set $\beta = \alpha^{(7)}$. Then we must find the largest j_0 such that $|\{t_{7-j_0}, \ldots, t_{7+j_0}\}| \leq d = 5$. We do so by starting with $j_0 = 0$ and incrementing it as long as the condition holds. With $j_0 = 5$ we have found the largest j_0 such that $|\{t_{7-5}, \ldots, t_{7+5}\}| = |\{x_6, x_1, x_9, x_3, x_8\}| \leq 5$, and these variables are set to ? in β. The resulting SSP is $\beta = (?, 0, ?, 1, 0, ?, 0, ?, ?, 1)$.

2.2 The functioning of hybridGM

`hybridGM` starts its search on F from a random complete assignment $\alpha^{(0)}$ in its SLS component. The search is performed by flipping variable assignments, yielding complete assignments $\alpha^{(j)}$, until one of the following cases occurs. Case one, $F(\alpha^{(j)}) = 1$. In this case the search is finished and $\alpha^{(j)}$ is a satisfying assignment for F. Case two, the number of unsatisfied clauses in $F(\alpha^{(j)})$ equals 1. In that case, we construct a SSP β around $\alpha^{(j)}$, as explained in the previous section.

As soon as SSP β has been constructed, $F(\beta)$ is handed over to the DPLL component which will determine if it is satisfiable. The DPLL solver will basically try to find assignments to all the unassigned variables in β to solve the sub-formula $F(\beta)$. In case the DPLL solver succeeds, it can return β', containing β and the additionally assigned variables, which is a satisfying assignment for the original formula F. In this case, the search is finished.

In case the DPLL solver can not extend β to a satisfying assignment for $F(\beta)$ (the sub-formula is unsatisfiable), the search continues in the SLS component as described above. The SSP β as well as the information that it does not contain a solution is *not* stored. Therefore, `hybridGM` is incomplete.

The reason why this works out well on satisfiable formulas is the following. In [22], it was shown that the number of unsatisfied clauses under a given assignment is correlated to the hamming distance to the nearest solution. Because a partition contains an assignment which leaves only one clause unsatisfied, chances are that a solution can be found within that partition. Since the DPLL

component is performing a complete search on that partition, it will find a solution if one exists. In contrast, the pure SLS approach is not guaranteed to find a solution from this part of the search space. Furthermore, the search of the DPLL finishes quickly because a partition significantly reduces the formula, leaving a simplified remainder that the DPLL can handle easily.

We will now continue to explain `SATUN`, the successor of `hybridGM`.

3 SATUN

Before we go into any detail about the functioning of `SATUN`, we will briefly cover the solver `March_hi` [11], as it is the DPLL component of `SATUN`. Furthermore, `March_hi` introduced a new variable selection heuristic based on a function called h_i. This function is only applicable for 3-SAT formulas, but the hybrid solver `SATUN` will need a similar function for arbitrary k. We therefore generalized the h_i function into h_i^* as described in the next section.

3.1 March_hi and the h_i^* function

The DPLL solver `March_hi` [11] is the successor of `March_ks` [10]. `March_hi` won the uniform random UNSAT category of the SAT 2009 Competition [19] and thereby proved, that it is the most successful solver on unsatisfiable uniform random k-SAT formulas. We decided to use this DPLL solver in the design of the new `SATUN` solver because the DPLL component in the new hybrid is also responsible for proving a formula to be unsatisfiable. The major improvement of `March_hi` is its new variable selection heuristic, which is based on the h_i function. The original h_i function is only applicable for 3-SAT, but `SATUN` will need to compute the h_i function for arbitrary k-SAT formulas. We therefore generalized h_i into h_i^* as follows.

Given a k-CNF formula F with n variables $\mathcal{V} = \{x_1, \ldots, x_n\}$. The set of the $2n$ literals for F is $\mathcal{L} = \{\neg x_1, x_1, \ldots, \neg x_n, x_n\}$. Define $h_i^* : \mathcal{L} \to \mathbb{R}$ recursively:

$$h_0^*(x) = 1$$

$$s_{i+1} = \frac{1}{2n} \cdot \sum_{x \in \mathcal{L}} h_i^*(x)$$

$$h_{i+1}^*(x) = \sum_{l=2}^{k} \frac{k^{k-l}}{s_{i+1}^{l-1}} \sum_{(x \vee x_{a_1} \vee \ldots \vee x_{a_{l-1}}) \in F} h_i^*(\neg x_{a_1}) \cdot \ldots \cdot h_i^*(\neg x_{a_{l-1}})$$

The inner sum is taken over all clauses $(x \vee x_{a_1} \vee \ldots \vee x_{a_{l-1}})$ of length l in the formula F that contain literal x and arbitrary literals $x_{a_1}, \ldots, x_{a_{l-1}}$. This is then multiplied by a scaling factor $\frac{k^{k-l}}{s_i^{l-1}}$. The smaller the clause-length l, the larger the scaling factor. The outer sum is taken over the different clause lengths. It is $h_i(x) = h_i^*(x)$ for a 3-SAT formula.

The more small clauses a literal occurs in, the larger its h_i^* value. Literals that often occur together in a clause will affect each other much more, than literals

that rarely occur together in a clause. h_i^* therefore respects mutual influence of literals over multiple clauses. The number i of iterations that are computed will control how much of that mutual influence is respected when h_i^* is computed for a literal x. The larger i, the more mutual influence is taken into account. We define the *score* of a variable using $i = 5$ as $score^*(x_j) = h_5^*(x_j) \cdot h_5^*(\neg x_j)$.

The idea behind $score^*$ is to measure how much (transitive) influence a variable has on the formula. The score of a variable then represents its relative strength to reduce the formula when it is branched upon. The heuristic is to branch upon the variable with highest score, yielding a strongly reduced formula which is *assumed* to have the smallest remaining search tree.

We continue to explain the functioning of `SATUN` in the next section using the above described h_i^* and $score^*$ functions.

3.2 The functioning of SATUN

In general, `SATUN` performs search in two phases. In phase one it performs initialization tasks and an SLS-like search as it is done in `hybridGM`. Phase one is also used to gather additional information about the variables of the formula. In contrast to `hybridGM`, the time for this SLS search is limited in `SATUN`. Ideally, `SATUN` will find a satisfying assignment if one exists before the limit is reached. After the limit has been reached the solver enters phase two, in which it performs a DPLL-like search using the additionally gathered information from phase one. At the end of phase two `SATUN` will be able to state whether the investigated formula has a solution or not.

PHASE 1. For a k-CNF formula F with n variables and m clauses, `SATUN` begins with some initialization tasks as follows. It will calculate $score^*$ for all variables. We denote the six *variables with highest* $score^*$ $x^1, \ldots, x^6$. For these six variables, we create a 6×6 *flip-connection-matrix* $\mathcal{M}$ in which each component represents a counter initialized to zero. Now `SATUN` starts its SLS search from a random complete assignment for F. This search is performed as in `hybridGM`.

Additionally, SATUN will observe the six variables with highest $score^*$. Whenever variable x^j is flipped *right after* x^i (with $i \neq j$) the counter $\mathcal{M}_{i,j}$ is incremented. The reason, why we have introduced the flip connection matrix with the fixed size 6×6 is that it is unnecessary to observe all possible $n \times n$ variable pairs. The SLS component prefers to flip variables with a high $score^*$ anyway. Confining the matrix to the variables with highest score therefore ignores unimportant pairs. Please note, that this has nothing to do with partitioning, i.e. the creation of SSPs and their processing by the DPLL component.

In short, phase one simply observes which of the variables with highest score are flipped as a *pair*, and how often that happens. Phase one is performed until $\mathcal{M}$ contains a specific amount of counts. This is supposed to ensure that the SLS search in `SATUN` has a reasonable chance to find a satisfying assignment, which is usually faster than using a DPLL approach as it is done in phase two. Furthermore, it ensures that a reasonable amount of counts is recorded, i.e. enough information about the six variables with highest $score^*$ has been

gathered. The amount of counts necessary is given by the *hit-count-condition*, which has been determined empirically:

$$\sum_{i=1}^{6}\sum_{j=1}^{6}\mathcal{M}(i,j) \geq 36 \cdot \frac{m}{n} \cdot \frac{10}{k-2}$$

Phase one is completed as soon as the hit-count-condition holds. Since the variables with highest score are flipped quite often by the SLS solver, and the pairwise flips indeed occur, the hit-count-condition will hold at some point. We then *expect* the formula to be unsatisfiable and continue with phase two.

PHASE 2: We define $pairsum(x^i, x^j) := \mathcal{M}_{i,j} + \mathcal{M}_{j,i}$. At the start of phase two, `SATUN` will identify $i, j \in \{1, \ldots, 6\}$ for which the $pairsum(x^i, x^j)$ is largest. This represents the pair for which the variables got flipped "together" most often (either x^i right after x^j or vice versa).

From the chosen pair (x^i, x^j), the hybrid solver now picks the variable with higher $score^*$ (let this be x^i), and creates two sub-formulas. Formula F_0 is created by assigning $x^i = 0$ in F, and formula F_1 is created by assigning $x^i = 1$ in F. Since we do this *branch* before invoking the DPLL (one could say manually) we call this technique manual branching.

The formulas F_0 and F_1 are now sequentially handed over to the DPLL component. If it can find a solution to either of them, the formula F must be satisfiable, and a solution can be presented. If both formulas F_0 and F_1 are proven to be unsatisfiable, then F must be unsatisfiable. For most formulas the second case (both unsatisfiable) will occur because the SLS component is supposed to find a solution for F within phase one.

The time needed by the DPLL to prove both sub-formulas to be unsatisfiable is supposed to be smaller than the time needed to prove the original formula to be unsatisfiable. This is supposed to compensate for the SLS search in phase one. In other words, the identification of a branching variable, that grants a speed up to the DPLL in the unsatisfiable case, allows us to try local search first to find a satisfying assignment – even on unsatisfiable formulas. For an overview of the functioning of `SATUN`, see Figure 1.

It is, however, not yet clear why manually branching on variable x^i results in a faster unsat-proof. An explanation follows in the next section.

3.3 Why manual branching works

The variable with highest $score^*$ has the most transitive influence on the formula. One could *expect* it to be the globally best choice for the first branching variable. In order to test this, we have performed an experiment as follows.

Given an unsatisfiable formula F with n variables in set $\mathcal{V}$. Create n separate manual branches on all the variables $x_j \in \mathcal{V}$. This yields two sub-formulas per branch which we call F_0^j and F_1^j. Now F is proven to be unsatisfiable by proving both F_0^j and F_1^j to be unsatisfiable using a DPLL solver. The DPLL solver will need a finite number of search nodes to show that each sub-formula F_i^j is

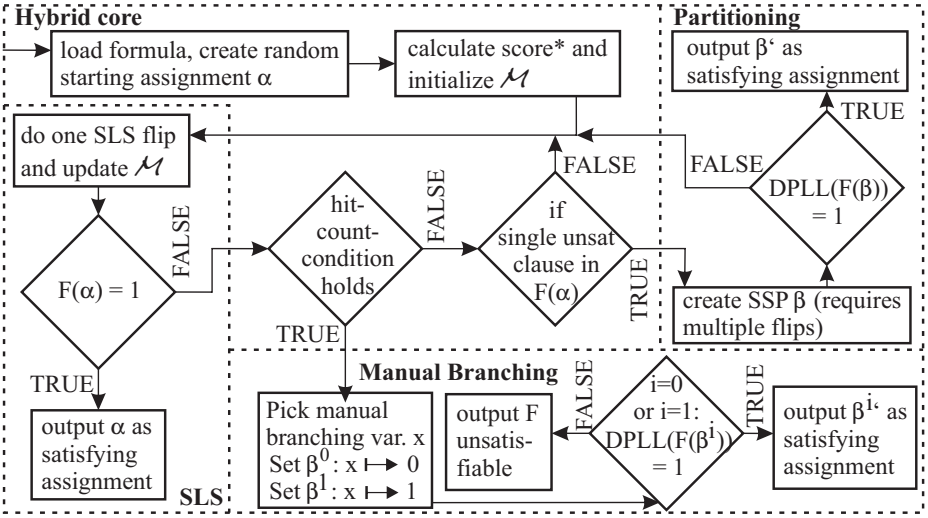

Fig. 1. An overview of the functioning of `SATUN`. See text for details. Pleas note, that a SSP β, that contains a satisfying assignment, must be extended to such an assignment by the DPLL in order to be a valid solution. We denote this extension by β'.

unsatisfiable. Let us denote this number with ν_i^j. The total amount of search nodes, which is then needed to prove F unsatisfiable when using x_j as first branching variable, is $\nu^j = \nu_0^j + \nu_1^j$.

We can now compare ν^j for all the branching variables x_j. The smaller ν^j is, the less search nodes are needed by the DPLL to prove F unsatisfiable. This in turn will result in a shorter computation time for the DPLL. One would expect the variable with highest $score^*$ to yield the smallest ν^j, but this is not always the case. In other words: the variable that is expected to reduce the formula the most when branched upon is not necessarily the one that yields the smallest remaining search tree. One is often able to identify a better branching variable among the first six variables with highest $score^*$.

$score^*$ does not state how small *further search trees* will be. It just gives relative reduction strengths for all the remaining variables for the *currently investigated formula*. The assumption, that the strongest immediate reduction of the formula yields the overall smallest search tree, is usually not true. In order to choose *the* best variable to branch upon, one would need to predict the size of the further search trees, which is, for example, represented by ν^j. However, the ν^j values are not know a priori, and their computation already requires to prove the formula to be unsatisfiable.

In general, a DPLL solver should try to identify conflicts close to the root of its search tree. This will enable it to prune a considerable amount of the search space early during the search, which in turn results in a faster search. Ideally, it should branch on variables appearing in the smallest conflict within the formula.

The flips performed by the SLS solver represent a local decision. Since the SLS solver usually deals with assignments that satisfy almost all clauses from the

formula, these decisions often represent how the SLS solver tried to overcome a local conflict, i.e. a conflict of the current variable assignments. When two variables are often flipped as a pair then they must somehow be connected. In a CNF formula, a connection between variables is given in the form of clauses. The fact that these variables have been flipped together very often is what makes the corresponding clauses, that create this connection, important. These clauses are expected to participate in a conflict that the SLS solver cannot seem to overcome. The reason for a variable pair to get flipped more often than others is *expected* to be that the respective clauses appear in relatively many conflicts, or that conflicts in which the respective clauses appear are rather small.

Therefore, branching on a variable from the mentioned variable pair is supposed to enable the DPLL solver to detect a conflict sooner. This leads to an earlier pruning of parts of the search space, which in turn reduces the necessary search nodes of the binary search tree. As a result, the DPLL will finish its computations sooner. One could say that the variables from the most often flipped variable pair represent the *locally* best choice for the first branching variable.

The observation during phase one of the manual branching is confined to the six variables with highest score, from which we pick a pair that is flipped most often. This is a compromise of the globally and locally best choice for the first branching variable that respects both intuitions explained above. A variable with a high $score^*$ will reduce the formula the most and will therefore yield a small remaining search tree, and a variable from the most often flipped pair will lead the search into a conflict sooner, yielding a smaller remaining search tree.

An empirical study of `SATUN`'s performance is presented in the next section.

4 Empirical Study

Soft- and Hardware: In order to perform the empirical study, we used the SLS solver `gNovelty+` [18] in the version of the SAT 2007 Competition, the DPLL double-lookahead solver `March_hi` [11] in the version of the SAT 2009 Competition, and the incomplete hybrid solver `hybridGM` [1] (version 7) of the SAT 2009 Competition. These solvers can be downloaded from [19], and will be the reference solvers for the new hybrid solver `SATUN`. `SATUN` can be downloaded from [20].

The hardware used was the BWgrid [2] in Baden-Württemberg, Germany. It provided us with 24 Intel Harpertown quad-core CPUs with 2.83 Ghz and 8GB RAM each. The operating system was Scientific Linux.

Benchmark: The empirical study was performed on uniform random k-SAT formulas from the SAT 2009 Competition uniform random benchmark [19]. Our benchmark consists of $3, 5, 7$-SAT formulas; some satisfiable and some unsatisfiable. For each category (sat or unsat), we picked three formulas for each constellation of k and n. This results in 57 medium-sized satisfiable formulas, 57 medium-sized unsatisfiable formulas, and 57 large-sized satisfiable formulas.

We performed 100 runs for each satisfiable formula with the randomized solvers (`gNovelty+`, `hybridGM`, and `SATUN`), as well as one run with the deterministic solver `March_hi`. Please see the following table for timeout settings.

As results of the randomized solvers, we will present the mean and median runtime in seconds "mean | median" if the solver succeeded in all runs, and the success rate in percent if the solver did not succeed in all runs. For the hybrid solver `hybridGM` we also present a tuple that states which component of the hybrid found the solution (`gNovelty+`, `March_ks`). For `SATUN`, we present a tuple (`gNovelty+`, `March_hi` with partitioning, `March_hi` with manual branching) that states which component of the hybrid found the solution, but we also distinguish whether the DPLL component found the solution during partitioning or manual branching. In case a solver did not succeed in any of its runs, we write "TO".

For each unsatisfiable formula, we have performed 1 run with `March_hi` and 5 runs with `SATUN`. It was necessary to perform several runs with `SATUN` because manual branching could identify different variables for the first branch in different runs, which results in different runtimes to prove the formula unsatisfiable.

The gain-values present a difference of the mean runtime in seconds, where the "Gain SLS" value presents the difference between `gNovelty+` and `SATUN`, and the "Gain DPLL" value presents the difference between `March_hi` and `SATUN`. We print negative gain-values whenever `SATUN` was slower, and we print bold positive gain-values whenever it was faster. In case one of the compared solvers did not achieve a success rate of 100%, we print "$\gg 0$" whenever `SATUN` had a better success rate, we print "$\ll 0$" whenever it had a worse success rate, and we print "$= 0$" whenever the solvers had equal success rates.

The line "Time difference in total" at the end of each benchmark category contains the sum over *all* runs in that benchmark category where both solvers had a 100% success rate. This line should give the reader an idea of the overall performance of `SATUN` on the respective benchmark class.

Instance	SLS gNovelty+	DPLL March_hi	hybridGM7 (gNov,March)	SATUN (gNov,March,MB)	Gain SLS	Gain DPLL
Results of the selected medium-sized satisfiable formulas (Timeout: 16000 seconds per run)						
k3-v360-S1293537826	0,05 | 0,05	10,41	0,05 | 0,04 (75,25)	0,07 | 0,06 (47,53,0)	-0,02	**10,34**
k3-v360-S144043535	0,01 | 0,01	1,27	0,01 | 0,01 (97,3)	0,02 | 0,02 (82,18,0)	-0,01	**1,25**
k3-v360-S722433227	0,01 | 0,01	4,44	0,02 | 0,02 (93,7)	0,04 | 0,02 (82,18,0)	-0,03	**4,40**
k3-v380-S1841979702	0,02 | 0,02	5,69	0,03 | 0,02 (78,22)	0,04 | 0,03 (59,41,0)	-0,02	**5,65**
k3-v380-S1912687159	0,00 | 0,01	1,59	0,01 | 0,01 (95,5)	0,02 | 0,01 (89,11,0)	-0,02	**1,57**
k3-v380-S1985172968	0,05 | 0,04	0,22	0,05 | 0,03 (84,16)	0,09 | 0,06 (60,40,0)	-0,03	**0,13**
k3-v400-S142096783	0,18 | 0,12	8,14	82% (63,19)	0,21 | 0,13 (29,71,0)	-0,03	**7,93**
k3-v400-S714125262	0,10 | 0,08	39,39	0,11 | 0,10 (75,25)	0,34 | 0,10 (61,38,1)	-0,24	**39,04**
k3-v400-S981523907	0,02 | 0,02	2,49	0,03 | 0,02 (85,15)	0,06 | 0,03 (68,32,0)	-0,03	**2,43**
k3-v420-S1362286908	0,01 | 0,01	2,62	0,01 | 0,02 (80,20)	0,03 | 0,02 (47,53,0)	-0,01	**2,59**
k3-v420-S1460100153	0,00 | 0,01	0,61	0,01 | 0,01 (92,8)	0,03 | 0,02 (82,18,0)	-0,03	**0,57**
k3-v420-S1624574519	0,13 | 0,11	25,97	0,14 | 0,08 (71,29)	0,11 | 0,08 (36,64,0)	**0,02**	**25,86**
k3-v440-S110290755	0,50 | 0,38	33,79	0,31 | 0,24 (47,53)	0,17 | 0,13 (10,90,0)	**0,32**	**33,61**
k3-v440-S1112524719	0,16 | 0,14	6,67	0,21 | 0,17 (64,36)	0,67 | 0,16 (61,35,4)	-0,51	**6,00**
k3-v440-S1263695887	0,01 | 0,01	2,71	0,02 | 0,02 (87,13)	0,02 | 0,02 (68,32,0)	-0,01	**2,69**
k3-v460-S1367181683	0,18 | 0,12	185,25	0,20 | 0,17 (74,26)	0,68 | 0,36 (75,25,0)	-0,50	**184,57**
k3-v460-S1546885609	0,11 | 0,10	0,22	0,13 | 0,10 (67,33)	0,14 | 0,08 (52,48,0)	-0,03	**0,07**
k3-v460-S1555503917	0,02 | 0,02	41,02	0,04 | 0,04 (82,18)	0,06 | 0,03 (64,36,0)	-0,03	**40,96**
k3-v480-S1924575376	0,02 | 0,02	11,56	0,03 | 0,03 (74,26)	0,05 | 0,04 (53,47,0)	-0,03	**11,51**
k3-v480-S2069223517	0,73 | 0,54	22,70	0,55 | 0,44 (72,28)	0,62 | 0,46 (51,49,0)	**0,10**	**22,08**
k3-v480-S449556655	0,04 | 0,03	63,69	0,04 | 0,03 (86,14)	0,06 | 0,04 (67,33,0)	-0,01	**63,63**
k3-v500-S602463346	0,04 | 0,03	15,44	0,05 | 0,04 (70,30)	0,06 | 0,04 (68,32,0)	-0,01	**15,38**
k3-v500-S658989873	0,06 | 0,04	126,99	0,07 | 0,06 (72,28)	0,13 | 0,07 (63,37,0)	-0,07	**126,86**
k3-v500-S95953046	0,04 | 0,03	38,31	0,05 | 0,03 (65,35)	0,06 | 0,05 (59,41,0)	-0,01	**38,25**
k3-v520-S227954525	0,07 | 0,06	685,32	0,08 | 0,06 (87,13)	0,13 | 0,08 (92,8,0)	-0,06	**685,19**
k3-v520-S247943532	0,18 | 0,14	201,59	0,21 | 0,16 (65,35)	0,14 | 0,09 (57,43,0)	**0,03**	**201,45**
k3-v520-S621134782	0,01 | 0,01	3,50	0,01 | 0,01 (79,21)	0,03 | 0,02 (52,48,0)	-0,01	**3,47**
k3-v540-S1137823545	0,17 | 0,12	37,00	0,17 | 0,14 (61,39)	0,27 | 0,20 (41,59,0)	-0,10	**36,72**
k3-v540-S2142680239	0,24 | 0,19	420,13	0,24 | 0,19 (75,25)	0,28 | 0,18 (71,29,0)	-0,04	**419,85**
k3-v540-S449511205	0,84 | 0,65	3448,97	1,22 | 0,77 (89,11)	2,60 | 1,89 (93,7,0)	-1,76	**3446,37**

Instance	SLS gNovelty+	DPLL March_hi	hybridGM7 (gNov,March)	SATUN (gNov,March,MB)	Gain SLS	Gain DPLL
k3-v560-S1429776379	0, 03 | 0, 04	147, 81	0, 04 | 0, 03 (80, 20)	0, 10 | 0, 05 (56, 44, 0)	-0,07	**147, 71**
k3-v560-S1876865608	0, 02 | 0, 02	3, 82	0, 03 | 0, 02 (78, 22)	0, 04 | 0, 03 (63, 37, 0)	-0,02	**3, 78**
k3-v560-S1879859331	0, 11 | 0, 08	331, 74	0, 14 | 0, 10 (80, 20)	0, 14 | 0, 10 (73, 27, 0)	-0,03	**331, 60**
k5-v100-S1707873242	0, 01 | 0, 01	104, 51	0, 01 | 0, 02 (99, 1)	0, 05 | 0, 04 (64, 36, 0)	-0,04	**104, 46**
k5-v100-S1772254584	0, 02 | 0, 02	32, 47	0, 02 | 0, 02 (97, 3)	0, 03 | 0, 03 (83, 17, 0)	0,00	**32, 44**
k5-v100-S282405510	0, 06 | 0, 04	28, 40	0, 06 | 0, 05 (100, 0)	0, 14 | 0, 10 (95, 5, 0)	-0,08	**28, 26**
k5-v110-S1376127278	0, 58 | 0, 42	1788, 01	0, 56 | 0, 39 (100, 0)	1, 12 | 0, 74 (100, 0, 0)	-0,54	**1786, 89**
k5-v110-S1481888618	0, 01 | 0, 02	299, 11	0, 01 | 0, 02 (99, 1)	0, 03 | 0, 02 (80, 20, 0)	-0,01	**299, 08**
k5-v110-S1492329924	1, 73 | 1, 04	1793, 71	1, 74 | 1, 04 (100, 0)	2, 63 | 2, 05 (100, 0, 0)	-0,89	**1791, 08**
k5-v120-S1017443264	0, 75 | 0, 53	51, 62	0, 77 | 0, 55 (100, 0)	3, 38 | 1, 16 (97, 0, 3)	-2,63	**48, 23**
k5-v120-S1153031014	0, 75 | 0, 65	5659, 62	0, 75 | 0, 64 (96, 4)	1, 04 | 0, 88 (88, 12, 0)	-0,29	**5658, 58**
k5-v120-S1163324878	0, 04 | 0, 03	187, 29	0, 05 | 0, 03 (95, 5)	0, 21 | 0, 12 (67, 33, 0)	-0,17	**187, 08**
k5-v90-S1040456712	0, 01 | 0, 02	2, 56	0, 02 | 0, 02 (100, 0)	0, 06 | 0, 03 (99, 1, 0)	-0,05	**2, 50**
k5-v90-S1159550149	0, 04 | 0, 03	9, 05	0, 04 | 0, 03 (100, 0)	0, 15 | 0, 10 (97, 3, 0)	-0,10	**8, 90**
k5-v90-S1568932519	0, 05 | 0, 05	0, 19	0, 06 | 0, 05 (100, 0)	0, 26 | 0, 18 (74, 26, 0)	-0,21	-0,07
k7-v60-S1514821049	0, 68 | 0, 52	124, 99	0, 69 | 0, 51 (100, 0)	1, 30 | 0, 98 (94, 6, 0)	-0,62	**123, 69**
k7-v60-S1639151107	0, 20 | 0, 13	6, 18	0, 20 | 0, 13 (100, 0)	0, 46 | 0, 25 (100, 0, 0)	-0,26	**5, 72**
k7-v60-S178043354	0, 60 | 0, 32	88, 46	0, 61 | 0, 32 (100, 0)	1, 31 | 0, 70 (100, 0, 0)	-0,71	**87, 15**
k7-v65-S1146593655	2, 01 | 1, 38	768, 20	2, 02 | 1, 25 (100, 0)	3, 65 | 2, 60 (99, 0, 1)	-1,64	**764, 55**
k7-v65-S1223878722	0, 72 | 0, 55	264, 77	0, 72 | 0, 54 (100, 0)	1, 37 | 1, 00 (98, 2, 0)	-0,65	**263, 39**
k7-v65-S1645769281	1, 34 | 0, 99	625, 16	1, 23 | 0, 87 (99, 1)	2, 01 | 1, 38 (99, 1, 0)	-0,66	**623, 15**
k7-v70-S603077945	1, 90 | 1, 30	384, 40	1, 67 | 1, 27 (100, 0)	2, 32 | 1, 64 (100, 0, 0)	-0,41	**382, 08**
k7-v70-S754967851	1, 93 | 1, 30	1263, 36	1, 99 | 1, 31 (100, 0)	3, 92 | 2, 69 (100, 0, 0)	-1,99	**1259, 43**
k7-v70-S834811574	1, 08 | 0, 77	949, 80	1, 10 | 0, 79 (100, 0)	2, 12 | 1, 48 (100, 0, 0)	-1,04	**947, 68**
k7-v75-S1402156840	8, 05 | 4, 86	2628, 17	7, 83 | 4, 68 (100, 0)	4, 28 | 3, 52 (100, 0, 0)	**3, 77**	**2623, 89**
k7-v75-S1640054673	4, 21 | 2, 88	7894, 37	4, 00 | 2, 63 (100, 0)	5, 62 | 2, 88 (98, 0, 2)	-1,41	**7888, 75**
k7-v75-S1988754311	1, 95 | 1, 28	8356, 44	1, 96 | 1, 25 (100, 0)	3, 40 | 2, 29 (100, 0, 0)	-1,45	**8353, 04**
Time differences in total for medium-sized satisfiable formulas:					−15.38	**39193.46**
Results of the selected medium-sized unsatisfiable formulas (Timeout 16000 seconds per run)						
k3-v360-S1028159446	TO	4, 65	TO	7, 07 | 7, 05 (0, 0, 5)	≫ **0**	-2,42
k3-v360-S1369720750	TO	5, 41	TO	21, 06 | 20, 42 (0, 0, 5)	≫ **0**	-15,64
k3-v360-S1906521511	TO	5, 60	TO	11, 72 | 11, 89 (0, 0, 5)	≫ **0**	-6,12
k3-v380-S1580204273	TO	39, 35	TO	46, 94 | 47, 13 (0, 0, 5)	≫ **0**	-7,58
k3-v380-S1694258328	TO	29, 23	TO	29, 62 | 29, 61 (0, 0, 5)	≫ **0**	-0,39
k3-v380-S1700363952	TO	13, 94	TO	30, 17 | 28, 93 (0, 0, 5)	≫ **0**	-16,23
k3-v400-S1039370030	TO	19, 39	TO	23, 41 | 23, 36 (0, 0, 5)	≫ **0**	-4,01
k3-v400-S104085281	TO	26, 95	TO	30, 60 | 30, 57 (0, 0, 5)	≫ **0**	-3,65
k3-v400-S125259973	TO	44, 82	TO	52, 39 | 52, 34 (0, 0, 5)	≫ **0**	-7,57
k3-v420-S1136141672	TO	63, 96	TO	70, 52 | 70, 59 (0, 0, 5)	≫ **0**	-6,55
k3-v420-S121765931	TO	150, 57	TO	134, 20 | 134, 54 (0, 0, 5)	≫ **0**	**16, 37**
k3-v420-S1394036608	TO	32, 13	TO	37, 51 | 37, 49 (0, 0, 5)	≫ **0**	-5,37
k3-v440-S1035441377	TO	90, 10	TO	98, 88 | 98, 85 (0, 0, 5)	≫ **0**	-8,78
k3-v440-S1350958473	TO	175, 57	TO	192, 62 | 192, 45 (0, 0, 5)	≫ **0**	-17,05
k3-v440-S1353993612	TO	70, 76	TO	92, 47 | 92, 40 (0, 0, 5)	≫ **0**	-21,70
k3-v460-S1159067237	TO	92, 82	TO	96, 71 | 96, 72 (0, 0, 5)	≫ **0**	-3,89
k3-v460-S1249441590	TO	367, 06	TO	422, 88 | 422, 02 (0, 0, 5)	≫ **0**	-55,81
k3-v460-S1377066099	TO	310, 57	TO	296, 22 | 294, 74 (0, 0, 5)	≫ **0**	**14, 34**
k3-v480-S1007745655	TO	321, 29	TO	345, 90 | 345, 83 (0, 0, 5)	≫ **0**	-24,60
k3-v480-S1175850950	TO	455, 76	TO	442, 90 | 450, 86 (0, 0, 5)	≫ **0**	**12, 86**
k3-v480-S1524836866	TO	407, 24	TO	410, 16 | 422, 45 (0, 0, 5)	≫ **0**	-2,92
k3-v500-S1213077873	TO	584, 34	TO	587, 89 | 587, 01 (0, 0, 5)	≫ **0**	-3,54
k3-v500-S1339364096	TO	535, 22	TO	549, 43 | 550, 10 (0, 0, 5)	≫ **0**	-14,20
k3-v500-S1731702106	TO	1419, 13	TO	1456, 54 | 1457, 41 (0, 0, 5)	≫ **0**	-37,40
k3-v520-S1048138627	TO	1901, 17	TO	1989, 58 | 1989, 61 (0, 0, 5)	≫ **0**	-88,40
k3-v520-S1089910525	TO	3195, 53	TO	3072, 96 | 3031, 26 (0, 0, 5)	≫ **0**	**122, 57**
k3-v520-S1331813943	TO	1664, 58	TO	1500, 90 | 1498, 76 (0, 0, 5)	≫ **0**	**163, 67**
k3-v540-S1077410718	TO	2529, 11	TO	2642, 32 | 2643, 71 (0, 0, 5)	≫ **0**	-113,21
k3-v540-S1250067652	TO	1436, 58	TO	1485, 81 | 1543, 50 (0, 0, 5)	≫ **0**	-49,23
k3-v540-S1404929091	TO	3892, 16	TO	3567, 78 | 3385, 63 (0, 0, 5)	≫ **0**	**324, 37**
k3-v560-S1154116462	TO	7928, 40	TO	7439, 69 | 7081, 28 (0, 0, 5)	≫ **0**	**488, 71**
k3-v560-S1750991629	TO	9150, 39	TO	10139, 75 | 10130, 64 (0, 0, 5)	≫ **0**	-989,36
k3-v560-S179568577	TO	3838, 20	TO	3571, 52 | 3444, 55 (0, 0, 5)	≫ **0**	**266, 67**
k5-v100-S1050568800	TO	347, 02	TO	356, 12 | 356, 25 (0, 0, 5)	≫ **0**	-9,10
k5-v100-S1211704640	TO	395, 39	TO	361, 10 | 362, 11 (0, 0, 5)	≫ **0**	**34, 28**
k5-v100-S1221988902	TO	373, 68	TO	386, 36 | 386, 24 (0, 0, 5)	≫ **0**	-12,68
k5-v110-S102181364	TO	1898, 77	TO	1966, 04 | 1965, 17 (0, 0, 5)	≫ **0**	-67,26
k5-v110-S1251522734	TO	1831, 63	TO	1785, 96 | 1791, 08 (0, 0, 5)	≫ **0**	**45, 67**
k5-v110-S1736056721	TO	1978, 65	TO	1920, 58 | 1953, 69 (0, 0, 5)	≫ **0**	**58, 07**
k5-v120-S1070443005	TO	8114, 91	TO	7800, 32 | 7789, 79 (0, 0, 5)	≫ **0**	**314, 59**
k5-v120-S1086008090	TO	8068, 76	TO	7966, 90 | 7938, 79 (0, 0, 5)	≫ **0**	**101, 86**
k5-v120-S1481598919	TO	8476, 37	TO	8470, 34 | 8530, 49 (0, 0, 5)	≫ **0**	**6, 03**
k5-v90-S122462268	TO	80, 69	TO	96, 09 | 96, 59 (0, 0, 5)	≫ **0**	-15,40
k5-v90-S1305875336	TO	70, 76	TO	76, 28 | 76, 73 (0, 0, 5)	≫ **0**	-5,51
k5-v90-S1331382449	TO	81, 40	TO	86, 12 | 87, 19 (0, 0, 5)	≫ **0**	-4,71
k7-v60-S108799362	TO	362, 49	TO	387, 67 | 387, 68 (0, 0, 5)	≫ **0**	-25,18
k7-v60-S1318210982	TO	351, 41	TO	369, 35 | 369, 09 (0, 0, 5)	≫ **0**	-17,93
k7-v60-S1559668863	TO	353, 51	TO	377, 49 | 380, 13 (0, 0, 5)	≫ **0**	-23,98
k7-v65-S1171422074	TO	1257, 21	TO	1284, 59 | 1298, 43 (0, 0, 5)	≫ **0**	-27,37
k7-v65-S1266438269	TO	1200, 36	TO	1231, 41 | 1226, 60 (0, 0, 5)	≫ **0**	-31,05
k7-v65-S1551099861	TO	1234, 59	TO	1260, 28 | 1255, 68 (0, 0, 5)	≫ **0**	-25,69
k7-v70-S1758422766	TO	4103, 10	TO	4111, 38 | 4114, 14 (0, 0, 5)	≫ **0**	-8,27
k7-v70-S184801254	TO	5186, 28	TO	4147, 13 | 4137, 54 (0, 0, 5)	≫ **0**	**1039, 14**
k7-v70-S1900558626	TO	5140, 10	TO	4636, 63 | 4658, 10 (0, 0, 5)	≫ **0**	**503, 47**
k7-v75-S1299934729	TO	TO	TO	TO	= 0	= 0
k7-v75-S1394630553	TO	TO	TO	40% (0, 0, 2)	≫ **0**	≫ **0**
k7-v75-S1525918493	TO	TO	TO	60% (0, 0, 3)	≫ **0**	≫ **0**
Time differences in total for medium-sized unsatisfiable formulas:					≫ **0**	**1732.92**
Results of the selected large-sized satisfiable formulas (Timeout: 1800 seconds per run)						

Instance	SLS gNovelty+	DPLL March_hi	hybridGM7 (gNov,March)	SATUN (gNov,March,MB)	Gain SLS	Gain DPLL
k3-v10000-S1012522562	322, 59 | 273, 80	TO	33, 02 | 26, 45 (23, 77)	52% (50, 3, 0)	≪ 0	≫ **0**
k3-v10000-S1618450766	493, 78 | 391, 94	TO	75, 10 | 47, 96 (37, 63)	33% (25, 8, 0)	≪ 0	≫ **0**
k3-v10000-S1806027711	56, 32 | 45, 74	TO	5, 95 | 5, 20 (44, 56)	57, 72 | 34, 15 (92, 8, 0)	-1,39	≫ **0**
k3-v12000-S1089166613	350, 50 | 225, 17	TO	21, 49 | 18, 18 (25, 75)	48% (42, 6, 0)	≪ 0	≫ **0**
k3-v12000-S1510966387	895, 42 | 855, 25	TO	715, 83 | 538, 79 (39, 61)	11% (9, 2, 0)	≪ 0	≫ **0**
k3-v12000-S15767910	608, 85 | 522, 05	TO	66, 01 | 49, 56 (35, 65)	20% (14, 6, 0)	≪ 0	≫ **0**
k3-v14000-S1195310117	936, 11 | 852, 81	TO	281, 20 | 196, 27 (32, 68)	20% (17, 3, 0)	≪ 0	≫ **0**
k3-v14000-S1256544997	856, 02 | 863, 89	TO	112, 44 | 92, 84 (29, 71)	26% (24, 2, 0)	≪ 0	≫ **0**
k3-v14000-S1294530132	79%	TO	243, 41 | 185, 55 (31, 69)	20% (18, 2, 0)	≪ 0	≫ **0**
k3-v16000-S1419235015	36%	TO	527, 29 | 386, 90 (34, 66)	5% (5, 0, 0)	≪ 0	≫ **0**
k3-v16000-S162003179	703, 80 | 654, 81	TO	47, 36 | 40, 53 (33, 67)	42% (40, 2, 0)	≪ 0	≫ **0**
k3-v16000-S1726986756	844, 87 | 770, 04	TO	65, 43 | 54, 92 (23, 77)	39% (35, 4, 0)	≪ 0	≫ **0**
k3-v18000-S1141213963	95%	TO	149, 69 | 118, 93 (27, 73)	23% (20, 3, 0)	≪ 0	≫ **0**
k3-v18000-S1279315365	78%	TO	149, 21 | 128, 40 (34, 66)	21% (18, 3, 0)	≪ 0	≫ **0**
k3-v18000-S13321468111	003, 03 | 1001, 28	TO	144, 30 | 122, 38 (31, 69)	29% (26, 3, 0)	≪ 0	≫ **0**
k3-v2000-S1075551507	0, 34 | 0, 26	TO	0, 30 | 0, 21 (55, 45)	0, 64 | 0, 44 (56, 44, 0)	-0,30	≫ **0**
k3-v2000-S1337875718	0, 84 | 0, 61	TO	0, 96 | 0, 69 (53, 47)	1, 20 | 0, 81 (54, 46, 0)	-0,36	≫ **0**
k3-v2000-S136987316	0, 64 | 0, 43	TO	0, 42 | 0, 31 (51, 49)	1, 73 | 0, 66 (54, 46, 0)	-1,08	≫ **0**
k3-v4000-S1203268705	3, 77 | 2, 36	TO	1, 27 | 1, 08 (39, 61)	11, 30 | 3, 43 (50, 50, 0)	-7,53	≫ **0**
k3-v4000-S1271018787	3, 19 | 2, 53	TO	1, 08 | 0, 85 (42, 58)	7, 59 | 3, 29 (53, 47, 0)	-4,40	≫ **0**
k3-v4000-S139794312	7, 32 | 6, 32	TO	2, 86 | 2, 23 (48, 52)	17, 52 | 9, 81 (45, 55, 0)	-10,19	≫ **0**
k3-v6000-S1119314619	26, 13 | 18, 38	TO	5, 34 | 4, 27 (47, 53)	239, 42 | 50, 40 (56, 44, 0)	-213,29	≫ **0**
k3-v6000-S1490060417	16, 65 | 11, 79	TO	2, 84 | 2, 40 (39, 61)	210, 51 | 56, 28 (53, 47, 0)	-193,86	≫ **0**
k3-v6000-S1629487320	110, 10 | 60, 74	TO	100, 55 | 63, 83 (37, 63)	260, 96 | 162, 32 (67, 33, 0)	-150,85	≫ **0**
k3-v8000-S1741784682	110, 58 | 82, 36	TO	12, 77 | 9, 71 (37, 63)	85% (56, 29, 0)	≪ 0	≫ **0**
k3-v8000-S1760662955	154, 59 | 100, 04	TO	27, 36 | 23, 69 (30, 70)	432, 16 | 214, 21 (57, 43, 0)	-277,57	≫ **0**
k3-v8000-S195213520	47, 62 | 19, 44	TO	3, 39 | 2, 94 (30, 70)	295, 97 | 32, 09 (66, 34, 0)	-248,35	≫ **0**
k5-v1000-S1040507052	3%	TO	19% (16, 3)	1% (1, 0, 0)	≪ 0	≫ **0**
k5-v1000-S1284372491	10%	TO	24% (18, 6)	1% (1, 0, 0)	≪ 0	≫ **0**
k5-v1000-S1322153318	1%	TO	11% (10, 1)	TO	≪ 0	= 0
k5-v1100-S115851319	20%	TO	68% (55, 13)	5% (4, 1, 0)	≪ 0	≫ **0**
k5-v1100-S1441840675	1%	TO	6% (5, 1)	1% (1, 0, 0)	= 0	≫ **0**
k5-v1100-S144924920	49%	TO	2087, 03 | 1662, 11 (72, 28)	7% (4, 3, 0)	≪ 0	≫ **0**
k5-v700-S1018145191	580, 18 | 474, 90	TO	464, 83 | 299, 10 (86, 14)	34% (21, 13, 0)	≪ 0	≫ **0**
k5-v700-S1192890414	49, 10 | 33, 21	TO	29, 09 | 21, 15 (82, 18)	29, 19 | 25, 84 (59, 41, 0)	**19, 91**	≫ **0**
k5-v700-S1281410666	572, 06 | 479, 54	TO	639, 42 | 392, 69 (86, 14)	31% (24, 7, 0)	≪ 0	≫ **0**
k5-v800-S1183359164	606, 63 | 487, 66	TO	636, 78 | 409, 99 (81, 19)	31% (25, 6, 0)	≪ 0	≫ **0**
k5-v800-S1185462553	12%	TO	42% (37, 5)	TO	≪ 0	= 0
k5-v800-S1353769076	47%	TO	2749, 04 | 1795, 46 (88, 12)	7% (3, 4, 0)	≪ 0	≫ **0**
k5-v900-S1119648091	47%	TO	2587, 10 | 1987, 08 (82, 18)	11% (9, 2, 0)	≪ 0	≫ **0**
k5-v900-S1253795703	62%	TO	2197, 55 | 1827, 75 (84, 16)	12% (8, 4, 0)	≪ 0	≫ **0**
k5-v900-S1334593800	7%	TO	20% (17, 3)	1% (1, 0, 0)	≪ 0	≫ **0**
k7-v140-S1017667294	194, 30 | 143, 88	TO	243, 31 | 168, 06 (100, 0)	25% (25, 0, 0)	≪ 0	≫ **0**
k7-v140-S1085378176	659, 52 | 597, 32	TO	1449, 60 | 1036, 02 (100, 0)	2% (2, 0, 0)	≪ 0	≫ **0**
k7-v140-S1818639567	125, 51 | 90, 44	TO	179, 55 | 132, 84 (98, 2)	32% (32, 0, 0)	≪ 0	≫ **0**
k7-v160-S1178729443	706, 72 | 625, 86	TO	2476, 04 | 1532, 03 (100, 0)	1% (1, 0, 0)	≪ 0	≫ **0**
k7-v160-S1543261951	790, 52 | 789, 61	TO	1954, 92 | 1478, 08 (100, 0)	4% (4, 0, 0)	≪ 0	≫ **0**
k7-v160-S1849057034	697, 18 | 626, 44	TO	1725, 58 | 990, 72 (99, 1)	TO	≪ 0	= 0
k7-v180-S1049921489	79%	TO	77% (77, 0)	4% (4, 0, 0)	≪ 0	≫ **0**
k7-v180-S1249224492	14%	TO	10% (10, 0)	TO	≪ 0	= 0
k7-v180-S1295462378	21%	TO	16% (16, 0)	1% (1, 0, 0)	≪ 0	≫ **0**
k7-v200-S1136894336	4%	TO	TO	TO	≪ 0	= 0
k7-v200-S1588298513	2%	TO	3% (3, 0)	TO	≪ 0	= 0
k7-v200-S1695726805	11%	TO	7% (7, 0)	TO	≪ 0	= 0
k7-v220-S1766907733	TO	TO	2% (2, 0)	TO	= 0	= 0
k7-v220-S193214747	TO	TO	TO	TO	= 0	= 0
k7-v220-S318878687	TO	TO	TO	TO	= 0	= 0
Time differences in total for large-sized satisfiable formulas:					≪ 0	≫ **0**

Table 1: The runtime results of the selected formulas (see text for details).

No solver actually proved a large-size unsatisfiable formula to be unsatisfiable (out-of-memory situation for both `March_hi` and `SATUN`), and therefore, we provide no results for these formulas here.

Discussion of the results: Concerning the medium-sized satisfiable formulas, we observed a slightly slower solving time of `SATUN` compared to its SLS component `gNovelty+`. `SATUN` took approximately 1.47 times as long as `gNovelty+` to finish computation on this set of formulas. However, since this results in a loss of approximately 15 seconds in total, we consider `SATUN` to be competitive with `gNovelty+` and we consider `SATUN` to have an SLS-like performance. As we can see from the result table, the manual branching is rarely performed and can therefore not explain these losses. Further investigation of the results revealed that these losses are due to the partitioning. Partitioning uses up extra calculation time for the DPLL component to find satisfying assignments close to local minima with only one unsatisfied clause. As we can see from the result table,

it succeeds to do so, but these findings do not compensate for the computation time used up on formulas from this set of formulas. The fact that `hybridGM` performs much better than `SATUN` leads to the conclusion that the way partitioning is implemented in `SATUN` must be improved. Compared to the DPLL solver `March_hi`, `SATUN` was the superior solver. `March_hi` took approximately a thousand times as long to finish the computations, which results in a time gain of 39193 seconds.

Concerning the medium-sized unsatisfiable formulas, the incomplete solvers are not able to present any results. In contrast, `SATUN` is able to outperform its DPLL component `March_hi`. It took `SATUN` about 98% of the time that `March_hi` required to finish computations for this set of formulas, which results in a time gain of 1732 seconds. We can make out two classes of formulas from the results. The first class consists of formulas where the hybrid is slower than `March_hi` (about 10% slower). An analysis of these results revealed that manual branching in the hybrid and `March_hi`'s variable selection heuristic *agreed* on the first branching variable for these formulas. In other words, the SLS component of `SATUN` failed to provide a better branching decision, resulting in the same DPLL searches for `SATUN` and `March_hi`. The time difference is then explained by the time that was uselessly put into local search. The second class of formulas is the one where the hybrid is faster than `March_hi` (about 25% faster). In total, the gains achieved in this class can compensate for the SLS search on all formulas of this set. Overall `SATUN` appears to be superior to `March_hi` for unsatisfiable formulas.

Concerning the large-size satisfiable formulas, the DPLL solver `March_hi` was unable to present any results because it experienced an out-of-memory situation on all of them. Since `SATUN` managed to present results on several formulas of this set, it is considered to be superior to `March_hi`. Compared to `gNovelty+`, `SATUN` is clearly the inferior solver. Again, an analysis of the results revealed that this is due to the way partitioning is implemented.

The whole benchmark consisted of 171 formulas. The total number of formulas solved with a success rate

- of 100% is
  90 for `gNovelty+`, 97 for `hybridGM`, 111 for `March_hi`, and 124 for `SATUN`
- greater 0% is
  111 for `gNovelty+`, 111 for `hybridGM`, 111 for `March_hi`, and 160 for `SATUN`.

5 Conclusions and Future Work

We have extended the idea behind `hybridGM` to create the complete hybrid SAT solver `SATUN`. We have done so by introducing *manual branching*, a technique that makes use of local search to gather information helpful to a DPLL solver.

`SATUN` managed to solve the most formulas during our empirical study. Concerning smaller satisfiable formulas, `SATUN` experienced minor losses in comparison to its SLS component, but it is in total superior to its DPLL component on these formulas. When it comes to unsatisfiable formulas, `SATUN` is superior to

its incomplete SLS component. Furthermore, SATUN has managed to achieve an overall better search time than its DPLL component on unsatisfiable formulas with the help of manual branching. On large-size satisfiable formulas, it outperformed its DPLL component, but experienced heavy losses compared to its SLS component. A closer investigation of the results revealed that these losses are not because of manual branching but because of the way partitioning is implemented in SATUN.

Future work includes the improvement of the way partitioning is implemented in SATUN to improve its performance. In order to do that, it might be worthwhile to use preprocessing before calling a DPLL on a partition.

It is not yet known whether the observation of combined flips for more than just two variables, i.e. for more than just pairs, could improve the selection for the first branching variable. Additionally, it could be worthwhile to not only select one but several branching variables for a manual branch.

6 Acknowledgments:

The authors would like to thank Marijn Heule, Martin Bader, and Christian Mosch for fruitful discussions and technical help.

References

1. Balint, A., Henn, M., Gableske, O.: A novel approach to combine an SLS- and a DPLL-solver for the satisfiability problem. In O. Kullmann (Ed.). Theory and Applications of Satisfiability Testing (SAT2009), LNCS 5584, 284–297, Springer 2009
2. BWgrid Homepage: http://www.bw-grid.de/
3. Crawford, J. M.: Solving satisfiability problems using a combination of systematic and local search. Second DIMACS Challenge, Rutgers University, NJ (1993)
4. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM. 7(3), 201–215 (1960)
5. Fang, H., Ruml, W.: Complete Local Search for Propositional Satisfiability. Association for the Advancement of Artificial Intelligence (AAAI-04). 161–166 (2004)
6. Fang, L., Hsiao, M.: A New Hybrid Solution to Boost SAT Solver Performance. Design, Automation, and Test in Europe. 1307–1313 (2007)
7. Ferris, B., Fröhlich, J.: WalkSAT as an Informed Heuristic to DPLL in SAT Solving. Technical report, CSE 573: Artificial Intelligence (2004)
8. Gableske, O.: Towards the Development of a Hybrid SAT Solver. Diploma Thesis, Ulm University, Germany. January 2009. http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/gableske/DA.pdf
9. Habet, D., Li, C. M., Devendeville, L., Vasquez, M.: A Hybrid Approach for SAT. Lecture Notes In Computer Science. 2470, 172–184 (2002)
10. Heule, M., van Maaren, H.: Effective incorporation of Double Look-Ahead Procedures. LNCS 4501, 258–271 (2007)
11. Heule, M., van Maaren, H.: march_hi. Solver description SAT09: http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 27–27 (2009)

12. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of AAAI 2002, 635–660 (2002)
13. Hutter, F., Tompkins, D.A., Hoos, H.H.: Scaling and probabilistic smoothing: Efficientdynamic local search for SAT. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, 233–248. Springer, Heidelberg (2002)
14. Jussien, N., Lhomme, O.: Local Search With Constraint Propagation and Conflict-Based Heuristics. 7th National Conference on Artificial Intelligence, 169–174 (2002)
15. Letombe, F., Marques-Silva, J.: Improvements to hybrid incremental SAT algorithms. Theory and Application of Satisfiability Testing. LNCS 4996, 168–181 (2008)
16. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. SAT 2005. LNCS, vol. 3569, 158–172. Springer, Heidelberg (2005)
17. Mazure, B.,Sais L., Gregoire, E.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence. 22,319–331(1998)
18. Pham, D. N., Thornton, J. R., Gretton, C., Sattar, A.: Advances in Local Search for Satisfiability. Australian Conference on Artificial Intelligence 2007: 213-222 (2007)
19. The SAT Competition homepage: `http://www.satcompetition.org`
20. The SAT research homepage at Ulm University: `http://www.uni-ulm.de/in/theo/research/sat-solving.html`
21. Selman, B., Levesque, H., Mitchell, D.: A New Method for Solving Hard Satisfiability Problems. Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI Press. 440–446 (1992)
22. Zhang, W.: Configuration landscape analysis and backbone local search. Part I: Satisfiability and maximum satisfiability. Artificial Intelligence Vol. 158, 1–26 (2004)