Evaluating CDCL Variable Scoring Schemes

Armin Biere(✉) and Andreas Fröhlich

Johannes Kepler University, Linz, Austria
{armin.biere,andreas.froehlich}@jku.at

Abstract. The VSIDS (variable state independent decaying sum) decision heuristic invented in the context of the CDCL (conflict-driven clause learning) SAT solver Chaff, is considered crucial for achieving high efficiency of modern SAT solvers on application benchmarks. This paper proposes ACIDS (average conflict-index decision score), a variant of VSIDS. The ACIDS heuristics is compared to the original implementation of VSIDS, its popular modern implementation EVSIDS (exponential VSIDS), the VMTF (variable move-to-front) scheme, and other related decision heuristics. They all share the important principle to select those variables as decisions, which recently participated in conflicts. The main goal of the paper is to provide an empirical evaluation to serve as a starting point for trying to understand the reason for the efficiency of these decision heuristics. In our experiments, it turns out that EVSIDS, VMTF, ACIDS behave very similarly, if implemented carefully.

1 Introduction

The application track of SAT competitions [1,2] is dominated by *conflict-driven clause learning* (CDCL) [3] solvers. Beside *learning* [4], the most important feature of these solvers is the *variable state independent decaying sum* (VSIDS) decision heuristic [5], actually in its modern variant *exponential VSIDS* (EVSIDS) [6], as first implemented in the MiniSAT solver [7]. The EVSIDS heuristic allows fast selection of decision variables and adds focus to the search, but also is able to pick up long-term trends due to a "smoothing" component, as argued in [6].

On the practical side, there have been various attempts to improve on the EVSIDS scheme. These include the *variable move-to-front* (VMTF) strategy of the Siege SAT solver [8], the BerkMin strategy [9], which is focusing on recently learned clauses, and the *clause move-to-front* (CMTF) strategies of HaifaSAT [10] and PrecoSAT [11]. In this paper, we suggest another new decision heuristic, called *average conflict-index decision score* (ACIDS). Our main contribution, however, is to show that EVSIDS, VMTF, and ACIDS empirically perform equally well, if implemented carefully. Beside allowing simpler implementation, these empirical results further shed light on what EVSIDS actually

Supported by Austrian Science Fund (FWF), national research network RiSE (S11408-N23). Builds on discussions from the 2014 workshop on Theoretical Foundations of Applied SAT Solving (14w5101), hosted by Banff International Research Station, and Dagstuhl Seminar 15171 (2015), Theory and Practice of SAT Solving.

M. Heule and S. Weaver (Eds.): SAT 2015, LNCS 9340, pp. 405–422, 2015.
DOI: 10.1007/978-3-319-24318-4_29

means. They open up new directions for treating practically successful decision heuristics formally, for instance in the context of proof complexity.

Regarding alternative decision schemes, we refer to the cube-and-conquer approach [12]. It combines CDCL with classical look-ahead [13] solving, and is particularly effective for solving hard combinatorial benchmarks (in parallel). The rest of the paper will focus on decision heuristics for CDCL solving, related to VSIDS. This paper also complements recent developments which try to relate and explain VSIDS with community structure [14–16].

2 Decision Heuristics

Following the same decision order in every branch of a DPLL [17] search tree amounts to a simple *static* decision heuristic, as in ordered binary decision diagrams (BDDs) [18], which even with dynamic variable reordering are restricted to one variable order along each path from root to a leaf. The freedom of being able to pick an arbitrary variable in every node "dynamically" is generally considered an advantage of SAT over BDDs, e.g., in the context of bounded model checking [19]. These *dynamic* decision heuristics originally only took the current partial assignment in a search node into account when selecting the next decision variable. They did not consider how the search progressed to reach this point in the search space. We call this set of restricted dynamic heuristics *first-order dynamic decision heuristics*. A typical example is the dynamic literal individual sum heuristic (DLIS). It selects as next decision literal one with the largest DLIS score, which is computed as the number of still unsatisfied clauses in which a literal occurs. A well-known and often applied variant of DLIS is the Jeroslow-Wang heuristic [20], which for instance is discussed in [21], together with other related early decision heuristics, including Bohm's, MOM's, etc.

With the introduction of learning in Grasp [4], these first-order heuristics implicitly became *second-order dynamic heuristics*, since learned clauses were used in computing scores too, and they do capture the history of the search progress. An early evaluation [21] of decision heuristics, originally designed as first-order heuristics but then applied as second-order heuristics together with clause learning, showed that variants of DLIS actually perform quite well.

In principle, one has to distinguish between selecting a *decision variable* and selecting a *decision phase*, i.e., the Boolean constant to which the selected variable is assigned. However, almost all modern CDCL solvers implement *phase saving* [22], which always reassigns the decision variable to the last phase it was previously assigned. Modulo initialization, typically based on (one-sided) Jeroslow-Wang's heuristic [20], phase saving turns the decision heuristic into a variable selection heuristic. Accordingly, we focus on variable selection, which in turn will be based on selecting a variable with the highest *decision score*.

Using learned clauses for computing scores is actually quite expensive, since it requires either to traverse the whole clause data base, which is growing fast due to adding learned clauses, or requires expensive book keeping of scores during propagation of assigned variables. The latter became expensive after it was possible to reduce propagation effort through lazy clause watching techniques [5,23],

particularly since learned clauses tend to be large [24]. Thus, one of the most important observations in the seminal Chaff paper [5] was that it is possible and even beneficial to replace DLIS by an even more aggressive dynamic scoring scheme, the VSIDS (variable state independent decaying sum) scheme, which does not require to traverse the clause data base at decision variable selection, nor to use expensive full occurrence list traversal for accurate score updates.

VSIDS. The *variable state independent decaying sum* (VSIDS) of Chaff [5] maintains a *variable score* for each variable. The basic idea is that variables with large score are preferred decisions. The original VSIDS implementation in Chaff worked as follows. Variables are stored in an array used to search for a decision variable. After learning a clause, the score of its variables is incremented. Further, every 256th conflict, all variable scores are divided by 2, and the array is sorted w.r.t. decreasing score. This process is also called *variable rescoring*. Moreover, note that the order of decision variables is not changed between rescores.

The process of updating scores of variables is also referred to as *variable bumping* [7]. Note, however, that in modern solvers and also in our experiments we not only bump variables of the learned clause, but all *seen* variables occurring in antecedents used to derive the learned clause through a regular input resolution chain [25] from existing clauses.

The *decide* procedure selects the next decision variable, by searching for the first unassigned variable in the ordered array, starting at the lower end, e.g., the variable with the highest score during sorting. An essential optimization in Chaff is to cache the position of the last found decision variable with maximum score in the ordered array. This position is used as starting point for the next search. If a variable in the array with a position smaller than the cached maximum score position becomes unassigned then the maximum score position is updated to that position. During rescoring, similar updates might be necessary.

The first part of VSIDS, e.g., only incrementing scores, constitutes an approximation of dynamic DLIS. It counts occurrences of variables in clauses, ignoring whether a clause is satisfied or not, or even removed during learned clause deletions [3] (called clause database *reduction* in the following). This restricted version of VSIDS without smoothing is denoted INC (or inc in the experiments).

As an alternative to using frequent rescoring, we propose that the smoothing part of VSIDS can also be approximated by adding the *conflict-index* to the score instead of just incrementing it. The conflict-index is the total number of conflicts that occurred so far. We call this scheme SUM (or sum in our experiments).

At each conflict, a new clause is learned, except for instance if on-the-fly subsumption [26,27] is employed. This might trigger additional conflicts, through strengthening existing clauses, without learning a new clause. Our implementation does not bump variables in this case, nor does it increase the conflict-index.

EVSIDS. If variables are rescored at each conflict, a variant of VSIDS, called *normalized VSIDS* (NVSIDS) [6], is an *exponential moving average* on how often a variable occurred in antecedents of learned clauses [6]. For NVSIDS, the score s of a bumped variable is computed as $s' = f \cdot s + (1 - f)$, using a damping

factor f with $0 < f < 1$. The score of other variables, which are not bumped, still have to be "rescored", e.g., $s' = f \cdot s$.

At each conflict, NVSIDS requires to update the score of *all* variables. A more efficient implementation, which we called *exponential VSIDS* (EVSIDS) in [6], was originally proposed by the authors of MiniSAT [7]. It updates only scores of (the much smaller set) of bumped variables by adding an exponential increasing score increment g^i, with i denoting the conflict-index and $g = 1/f$, thus $g > 1$. As the relative order of variables for NVSIDS and EVSIDS is identical [6], the notion of NVSIDS is only of theoretical interest (for the purpose of this paper).

Typical values for g are in the range of 1.01 to 1.2. Small values have been shown to be useful for hard satisfiable instances (like cryptographic instances). Large values are useful with very frequent restarts, particularly in combination with the *reuse-trail* technique [28]. In Glucose 2.3, even without reusing the trail, it was thus suggested to slowly decrease g over time from a large value to a small one.[1] In the (new) version of Lingeling used in our experiments, g is kept at 1.2.

Instead of rescoring variables explicitly, MiniSAT uses a priority queue, which is implemented as a binary heap. This data structure allows fast insertion and removal of variables and also updating scores, all in logarithmic time. If this priority queue was updated eagerly to contain exactly all the unassigned variables, then searching for an unassigned variable with maximal score would even be possible in constant time. However, the number of propagated variables per decision can be quite large (on average, 323 propagations per decision for 275 benchmarks in the evsids column in Tab. 2). Removing them *eagerly* is too costly.

A *lazy* alternative, as first implemented in MiniSAT [7] and now being the default implementation of modern CDCL solvers, is to remove variables with maximum score from the priority queue until the removed variable turns out to be unassigned. It is then used as the next decision variable. Note that, during backtracking, this lazy scheme still requires to insert variables back into the priority queue, as they are unassigned, in order to make sure that the priority queue contains all unassigned variables (but assigned ones are not eagerly removed).

While the original implementation of VSIDS in Chaff [5] can be considered to be lazy too, variable selection is still imprecise, since rescoring is delayed. An attempt to provide a more efficient implementation of rescoring with precise variable selection was implemented in the JeruSAT solver [29]. It still uses counters, i.e., inaccurate integer scores, but instead of using one sorted array for all variables, partitions them into doubly linked lists of variables with the same score. This allows faster insertion, removal, update, and rescoring.

Another invention in MiniSAT, particularly important for EVSIDS, is to use a precise floating-point representation instead of integers as in previous solvers. Even though we do not have separate experimental evidence in this paper, our experience suggests that using integer scores dramatically deteriorates performance compared to using floating-point scores. Even fixed-point scores (as in PrecoSAT [11]) need additional techniques like clause based decision heuristics in order to be competitive with floating-point based EVSIDS.

[1] Every 5000th conflict, f is increased by 0.01, starting at 0.8 until 0.95 is reached.

However, g^i usually grows very fast: Note that $1.01^{4459}, 1.2^{244} > 2^{64}$, and, more severely, $1.01^{71333}, 1.2^{3894} > 1.797 \cdot 10^{308}$ ($\approx$ maximum value in 64 bit IEEE double floating-point representation). Thus, even for EVSIDS with floating-points, the variable scores and the score increment have to be rescored occasionally, as in the VSIDS scheme. This also becomes necessary if the score of a bumped variable would overflow during an update. We will report how often this occurs and how much time is spent on rescoring in our experiments.

VMTF. Variable selection heuristics can be seen as online sorting algorithms of variable scores. This view suggests to use online algorithms with efficient amortized complexity, such as *move-to-front* (MTF) [30]. A similar motivation was given in the master thesis of Lawrence Ryan [8], which precedes MiniSAT [7] and introduced the Siege SAT solver as well as the *variable move-to-front* (VMTF, or vmtf in the experiments) strategy. As in Chaff, the restriction in Siege's VMTF bumping scheme was to only move variables in the learned clause. Actually, only a small subset of those variables, e.g., of size 8, was selected, according to [8].

The restriction in Siege to move only a small subset of variables might have been partially motivated by the cost of moving many. It is not uncommon that tens of thousands variables occur in antecedents of a learned clause, which also are rather long for some instances. In our experiments in Sect. 4, the default decision heuristic (evsids in Tab. 2) bumped on average 276 literals per learned clause of average length 105 (on 275 considered instances). Unfortunately, details on how even this restricted version of VMTF is implemented in Siege were not provided. The source code is not available either. We give details for a fast implementation of *unrestricted* VMTF in Sect. 3.

ACIDS. As further extension to the proposed SUM heuristic we want to introduce the *average conflict-index decision score* (ACIDS, or acids in our experiments). While SUM realizes a certain amount of smoothing (compared to INC) by giving a larger weight to later conflicts, this effect is rather small when compared to the exponential kind of smoothing that is applied in VSIDS and EVSIDS. However, as smoothing is conjectured to be an important part for variable score heuristics [6], the latter kind of smoothing might be preferable. We realize this as follows. In the ACIDS scheme, in the same way as for INC, SUM, VSIDS, and EVSIDS, we keep a score for each variable. Whenever a variable is bumped, its score is updated to be $s' = (s+i)/2$, with i being the conflict-index. Compared to SUM, much stronger smoothing is realized by ACIDS. In addition to giving a larger weight to later conflicts, the influence of earlier conflicts decreases exponentially in the number of times the variable is bumped.

To compare the influence of the current conflict with that of earlier ones, we can represent the score of the variable by $s = s_c + s_p$, with s_c and s_p representing the contribution of the current conflict and the previous conflicts, respectively. As before, we define i to be the current conflict-index. Further, I_p is the set of indices of all previous conflicts the variable was involved in. For SUM, $s_c = i$ and $s_p = \Sigma_{I_p} i_p$, with i_p being the elements of I_p. By definition, this will lead to $s_p > s_c$ in most cases, particularly after a certain number of conflicts occurred.

Table 1. Summary of considered variable scoring schemes, where s and s' denote current and updated variable scores, i the conflict-index, and f a damping factor with $0 < f < 1$, used in our reformulation NVSIDS of VSIDS as exponential moving average [6]. For EVSIDS, we use the inverse $g = 1/f$ of f (thus $g > 1$). For the VSIDS version implemented in Chaff, we set $h_i^m = 0.5$ if m divides i, and $h_i^m = 1$ otherwise.

	variable score s' after i conflicts		
	bumped	not-bumped	
STATIC	s	s	static decision order
INC	$s + 1$	s	increment scores
SUM	$s + i$	s	sum of conflict-indices
VSIDS	$h_i^{256} \cdot s + 1$	$h_i^{256} \cdot s$	original implementation in Chaff [5]
NVSIDS	$f \cdot s + (1 - f)$	$f \cdot s$	normalized variant of VSIDS [6]
EVSIDS	$s + g^i$	s	exponential dual of NVSIDS [6,7]
ACIDS	$(s + i)/2$	s	average conflict-index decision scheme
VMTF	i	s	variable move-to-front [8]

Similarly for INC, $s_c = 1$ and $s_p = |I_p|$, which already implies $s_p > s_c$ as soon as a variable is bumped twice. However, for the ACIDS heuristic, we obviously have $s_p < s_c$ at every point in the search.

Note that, in contrast to VSIDS and NVSIDS, scores of variables that are not bumped do not change for ACIDS. This not only allows to keep track of accurate scores in each step, but also avoids (delayed) variable rescoring. Additionally, compared to EVSIDS, the scores of variables grow much slower when using the ACIDS heuristic. In particular, the score of a variable in ACIDS is bounded by the conflict-index i, instead of being exponential in the number of conflicts, as it was the case for EVSIDS. Thus, also rescoring of variables to prevent overflow does not occur in practice. Considering overall performance, our experiments in Sect. 4 show that ACIDS works as well as EVSIDS and VMTF.

Clause Based Decision Heuristics. There also is related work on using recently learned clauses in variable selection, such as the BerkMin heuristic [9], or clause-move-to-front (CMTF) strategies [10,11]. In our experience, they are inferior to variable scoring schemes as considered in this paper, and we leave it to future work for a more detailed comparison. The same applies to one-sided schemes which select literals instead of variables (without phase saving).

3 Implementation

We describe how the VMTF scheme can be implemented efficiently, as well as how these techniques can be lifted to implement a generic priority queue, which (empirically) is efficient for all the considered scoring schemes. This new implementation of a priority queue for variable selections combines ideas originally implemented in Chaff [5] and JeruSAT [29], but adds additional optimizations

and works with arbitrary precise floating-point scores, in contrast to an imprecise earlier version implemented in Lingeling [31].

Variable scores play a role while (a) *bumping* variables participating in deriving a learned clause, (b) *deciding* or searching for the next decision variable, (c) *unassigning* variables during backtracking, (d) *rescoring* variable scores either for explicit smoothing in VSIDS or due to protecting scores from overflow during bumping, and (e) comparing past decisions on the trail to maximize *trail reuse* [28]. First, we explain a fast implementation for VMTF, focusing on (a)-(c). Next, we address its extension to precise scoring schemes using floating-point numbers, which in previous implementations followed the example set by MiniSAT to use a binary heap data structure. Last, we discuss (d) and (e).

3.1 Fast Queue for VMTF

According to Sect. 2, the score of a variable in VMTF is the conflict-index, e.g., the number of conflicts at the point a variable was last bumped. With this score definition, VMTF can be simulated with a binary heap. However, every bump then needs a logarithmic number of steps to "bubble-up" a bumped variable in the heap. Instead, a queue, implemented as doubly linked list which holds all variables, only requires two simple constant time operations for bumping: dequeue the variable and enqueue it back at the end of the list, which we consider as head. Even storing the score seems to be redundant.

To find the next decision variable in the queue, we could start at the end (head) of the queue and traverse it backwards until an unassigned variable is found. Unfortunately, this algorithm has quadratic accumulated complexity. For example, consider an instance with 10000 variables and a single clause containing all variables in default phase. However, we can employ the same[2] optimization as used in Chaff (see Sect. 2) and remember the variable up to which the last search proceeded until finding an unassigned variable. Since the solver will restart the next search at this variable, we call this reference *next-search*.

During backtracking, variables are unassigned and (as in Chaff) next-search potentially has to be updated to such an unassigned variable if it sits further down the queue closer to head than the next-search variable. In order to achieve this, we could use the scores of the variables for comparing queue position. However, in VMTF, variables bumped at the same conflict all get the same score, and thus simply using the score leads to violation of the following important invariant: variables right of next-search (closer to head) are assigned.

To fix this problem, we globally count enqueue operations to the queue with an *enqueue-counter* and remember with each variable the value of the enqueue-counter at the point the variable was enqueued as *enqueue-time*. Thus, the enqueue-time precisely captures the order of the elements in the queue and can be used to precisely compare the relative positions of variables in the queue. In the actual implementation, we use a 32-bit integer for the enqueue-counter,

[2] But in reverse order, e.g., while we prefer the variable with largest score at the end of the queue, Chaff had the variable with largest score at the first array position.

which occasionally, e.g., after billion enqueue operations, requires to reassign enqueue-times to all queue elements in a linear scan of the queue. Note that, in a dedicated queue implementation for VMTF (like `queue` in our experiments), the scores become redundant again, after adding enqueue-times.

3.2 Generic Queue for all Decision Heuristics

For other schemes, it is tempting to also just use a queue implemented as doubly linked list as for VMTF, maintaining both scores and enqueue-times. Every operation remains constant time except for bumping. We have to ensure that the queue is sorted w.r.t. score. However, only for VMTF, bumped variables are guaranteed to be enqueued at the end (head) of the queue, i.e., in constant time. For other scoring schemes, a linear search is required to find the right position, which risks an accumulated quadratic bumping effort. To reduce enqueue time, we propose three optimizations and two modifications to the bumping order.

The **first optimization** is inspired by bucket sort and already gives acceptable bumping times for EVSIDS. It is motivated by the following observation. For EVSIDS, rescoring to avoid floating-point overflow of scores and score increment occurs quite frequently, e.g., roughly every 2000 conflicts, as Tab. 2 suggests. Thus, the exponents of variable scores represented as floating-point numbers will tend to span the whole range of possible values[3]. So instead of a single queue, we keep a stack of queues, indexed by the exponent of the scores of variables. Variables belong to the queue of the floating-point exponent of their score. As the motivation on rescoring shows, this stack will soon grow to its maximum size for EVSIDS, but for other scoring schemes (particularly for VMTF or INC) it will only have very few elements or even just one.

Note that, since exponents can be negative, the actual index to access the stack is obtained after adding the negation of the minimum negative exponent. Furthermore, Lingeling uses its own implementation of floating-points, in order to make execution of Lingeling deterministic across different hardware, compilers, and compiler flags. These software floats have a 32 bit exponent, but we restrict exponents to 10 bits including a sign bit, by proper rescoring of large scores and truncation of small scores. MiniSAT/Glucose use 10^{100} as an upper score limit, which is only a slightly smaller maximum limit than ours $2^{512} \approx 10^{154}$, but then does not use any truncation for small scores, which means that the minimum score exponent in MiniSAT is (roughly) 2^{-10}. So Lingeling uses 9 bits for positive scores and 9 bits for negative scores, while MiniSAT uses slightly less than 9 bits for positives scores and (almost) full 10 bits for negative scores.

When searching for decisions as well as during backtracking, more specifically during unassigning variables, we additionally have to maintain the highest exponent of an unassigned variable. This follows the same idea as for next-search in a single queue and only adds constant time effort for all considered operations.

During conflict analysis, variables participating in resolutions to derive a learned clause are collected on a *seen-variables* stack, before they are bumped

[3] Almost 2048 values for an 11-bit exponent in IEEE representation of 64 bit doubles.

(or discarded if on-the-fly subsumption succeeds). The analysis traverses the trail of assigned variables in reverse order. Thus, there is a similarity between the order of variables on the seen-variables stack and the reverse order of assignments. However, this is not guaranteed, particularly for variables with smaller decision-level. The order of bumping these variables then follows this order too.

At a conflict, it can happen that thousands of variables with different score are bumped and end up in almost random order w.r.t score order on the seen-variables stack (or worse, in reverse order) before they are bumped. For many of these variables, even for EVSIDS, the new updated score might end up having the same exponent and all those variables have to be enqueued to the same queue. However, since their scores still differ, enqueueing them degrades to insertion-sort. There are instances where bumping leads to a time-out due to this effect.

A **first modification** to the order in which variables are bumped prevents this problem. Before actually first dequeuing a bumped variable, then updating its score, and finally enqueueing it back, we sort the seen-variables stack w.r.t. increasing score. However, a similar problem occurs if all bumped variables have the same score exponent, which also does not change during update. This is for instance almost always the case for INC. The **second modification** prevents this corner case by first dequeuing all variables on the seen-variables stack, and only then updating their score and enqueueing them back in score order.

While EVSIDS exponents of variable scores are more or less spread out, other schemes do not have this property, clearly not INC, but probably also SUM and ACIDS to a smaller extent. For these schemes, score exponents might cluster around some few values. Thus, our **second optimization** repeats the bucket sort argument w.r.t. some fixed number of highest bits of the mantissa of a variable score. For each queue (indexed by exponent), we add another cache-table (indexed by highest bits of mantissa) of references pointing to the last element in the queue with matching highest mantissa bits. This ensures that these variables referenced in the cache-table have the maximum score among variables in this queue with the same highest bits of the mantissa of their score. In our implementation, we use the highest 8 bits and thus a cache-table of size 256. This cache is only used for fast enqueue and can be ignored otherwise.

If bumping individual variables is done in the order of their scores, as suggested by the first modification above, there is a high chance that consecutively bumped variables end up in the same queue one after each other or at least close to each other. Thus, as a **third optimization**, we propose to additionally cache the *last-enqueued* variable for each (sub) queue consisting of variables with the same highest mantissa bits. In an enqueue operation, we first check whether the corresponding cache-table entry of the second optimization points to a variable with smaller (or equal) score. If this is the case, we enqueue right next to it. Otherwise, we obtain the last-enqueued variable and start searching for the proper enqueue position from there towards the end, e.g., towards larger scores. This might fail if the score of the last-enqueued variable is larger or if the last-enqueue reference is not valid, e.g., if the variable is already dequeued. We then search backwards from the cache-table reference (towards smaller scores).

Altogether, these optimizations and modifications seem to avoid the most severe worst-case corner cases. We track this by profiling relative and total decide and particularly bump time per instance. Total time summed for these over all instances are shown in Tab. 2. Further distribution plots are included in the additional material, mentioned in the results in Sect. 4.

3.3 Rescore, Reuse-Trail and Complexity

For the original array based VSIDS implementation, *rescoring* requires sorting variables. For a binary heap implementation, one would expect that the heap does not change, since rescoring does not change the relative order of variables. However, due to finite precision of scores, even when using floating-points, rescoring will make the score of some variables the same, even though they differed in score before rescoring. Moreover, scores of many variables will become zero after a few rescores (particularly in EVSIDS). In this situation, the binary heap will only remain unchanged after rescoring if the actual scores are the only mean to compare variables (and for instance the variable index is not used as a tie breaker for comparing variables with the same score). The same argument applies to our improved queue based implementation.

The reuse-trail optimization [28] is based on the following observation. After a restart, it often happens that the same decisions are taken and the trail ends up with the same assigned variables. Thus, the whole restart was useless. By comparing scores of assigned previous decisions with the score of the next decision variable before restarting, this situation can be avoided. With some effort, this technique can be lifted to our generic queue implementation. To simplify the comparison in favor of a clean experiment, the results presented in Sect 4 are without reuse-trail (except for sc14ayv, the old 2014 version of Lingeling).

While we do not have a precise complexity analysis for this new data structure, our empirical results show that it performs almost as good as a dedicated binary heap for EVSIDS (heap) and as a dedicated simplified queue for VMTF (queue). This makes our empirical comparison of decision heuristics more accurate since they all use the same implementation. This data structure should also allow to experiment with new scoring schemes without the need to implement dedicated data structures. It might also be possible to improve it further, while our binary heap implementation is close to being as fast and compact as possible.

4 Results

The variants of Lingeling used in the experiments evolved from the SAT competition 2014 version *ayv* [32] (sc14ayv)[4]. This old 2014 version of Lingeling solved the largest number of instances in the SAT+UNSAT application track. This success of Lingeling can be contributed to the rather long time limit of 5000 seconds as used in the competition. For shorter time limits, Glucose version 2.3 [33,34]

[4] Acronyms in sans serif font denote SAT solver versions and configurations.

(glucose-2.3) from 2013 and particularly its 2014 derivative SWDiA5BY A26 [35] (swdia5bya26) show much better performance, despite lacking many effective preprocessing and inprocessing techniques [36].

Our post competition analysis showed that this effect can be contributed to two different aspects. On the one hand, the benchmark selection scheme used in the SAT competition 2014 (and already in 2013) had a strong influence on those results. Benchmarks were selected in such a way to level out performance of solvers. The goal of the organizers was to make the competition as interesting as possible, with the unfortunate effect, however, that unique solving capabilities, such as inprocessing [36], are deemphasized. On the other hand, our analysis showed that there is indeed an algorithmic feature implemented in all the Glucose variants taking part in the competition, which on these competition benchmarks is quite effective: the Glucose restart strategy [37].

This strategy uses the glucose level of learned clauses, which is the number of different decision levels [33] in the learned clause. It compares current short term average glucose level of learned clauses with a long term average. If short term average is substantially larger than long term average (say 25%), a restart is triggered, unless a restart happened very recently (less than 50 conflicts earlier).

To derive this conclusion, we implemented all techniques used in Glucose 2.3 and SWDiA5BY A26 previously not available in Lingeling, and compared their effect on the considered SAT competition 2014 application track benchmarks. Without being able to give more details, which is also not the focus of this paper, implementing a variant of the Glucose dynamic restart scheme [37] had the largest impact and allowed us to solve a comparable number of benchmarks as the aforementioned Glucose variants even with much smaller time limits.

Beside incorporating effective techniques from Glucose and SWDiA5BY, the base line version *b7ztzu* of Lingeling (evsids), as used in this evaluation, differs from the 2014 version sc14ayv mainly in the implementation of the priority queue used for selecting decision variables as detailed in Sect. 3. In other solvers, and previously in Lingeling, the priority queue was implemented with a binary heap data structure, as pioneered by MiniSAT [7]. This change was necessary to avoid slowing down the decision selection procedure for certain decision heuristics, particularly the variable move-to-front strategy (VMTF), which does not require the overhead of a binary heap. It is also slightly faster than using a binary heap.

As Glucose (and thus SWDiA5BY) is based on MiniSAT [7] (minisat), we also include in our comparison the latest version of MiniSAT from *git-hub*, which essentially has not changed since 2011. For all these considered MiniSAT derivatives, we use the default configuration with the internal MiniSAT version of SatELite style preprocessing [38] enabled.

The experiments were performed on our benchmark cluster, consisting of 30 nodes with Intel Q9550 Core 2 Quad CPUs running at 2.83GHz and 8 GB of main memory. Each job, e.g., pair of solver (configuration) and benchmark, had exclusive access to one node and CPU, respectively. The time limit was set to 1000 seconds, which is substantially smaller than the original competition

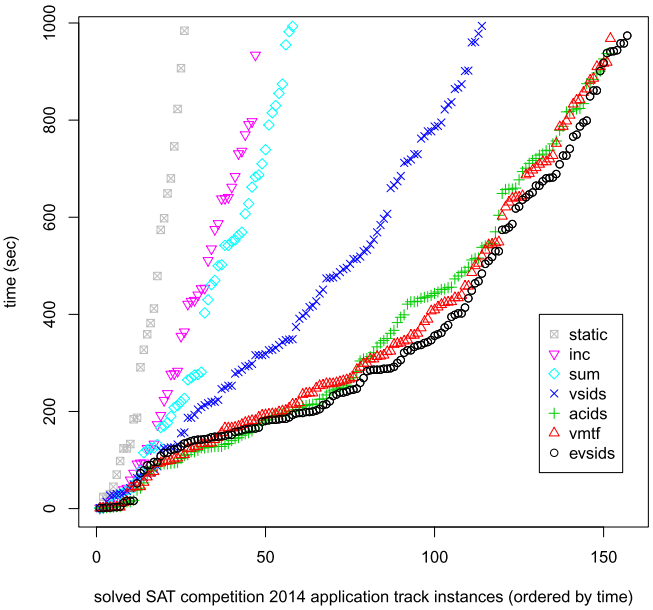

Fig. 1. Lingeling with variable scoring schemes of Sect. 2 on SAT competition 2014 application track benchmarks using the generic priority queue implementation of Sect. 3.

time-out of 5000 seconds (competition hardware was further roughly 1.2 times faster). As memory limit, we used 7GB.

In this paper, we focus on the 300 instances of the SAT+UNSAT application track of the SAT competition 2014, but exclude 25 instances, which were solved by the new Lingeling base line version evsids, without producing any conflicts. Among those excluded, there are 13 satisfiable "argumentation" instances [39] submitted 2014, with name prefix "*complete...*". These excluded 13 instances have a simple solution, with all variables set to false. In contrast, if this is not detected and a more sophisticated phase initialization heuristic like Jeroslow-Wang [20] is triggered before switching to phase saving [22], they become very hard. The old SAT competition 2014 version of Lingeling sc14ayv fails to solve 7 within 1000 seconds in our set-up.

The other excluded 12 instances are unsatisfiable combinational hardware equivalence checking "miter" benchmarks [40] submitted 2013. They are solved by our base line version evsids, and all other considered new variants of Lingeling, during the first preprocessing phase, without any search. Within 1000 seconds, the three MiniSAT/Glucose variants easily solve the 13 excluded satisfiable "argumentation" instances, due to initializing the saved phase to false, but need more effort than Lingeling to solve the unsatisfiable "miters". Both glucose-2.3 and swdia5bya26 fail on benchmark *6s151*, and minisat even fails on 11 "miters" (but does solve *6s165-non*). Note that, altogether, there were 30

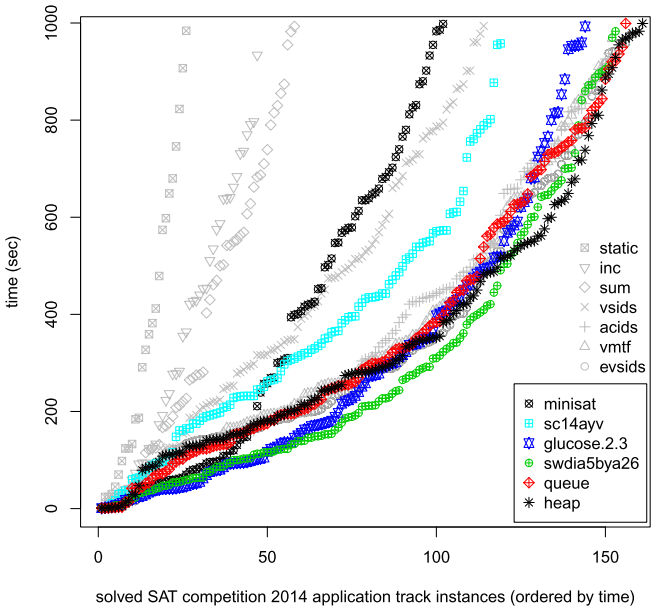

Fig. 2. Additional variants of Lingeling on SAT competition 2014 application track benchmarks as well as other state-of-the-art SAT solvers for this set-up.

"miters" in the competition. Thus, 17 "miters" remained in our subset of 275 actually compared instances, as well as 7 out of the 20 original "argumentation" benchmarks.

We describe additional specifics of the configurations used in our experiments on top of what has been explained in detail in previous sections and further summarize conclusions which can be drawn from the data provided in the tables and cactus plots. All experimental data including source code is available at http://fmv.jku.at/evalvsids/evalvsids.7z (27MB).

The main result of the paper is documented in Fig. 1. The cactus plot shows, that EVSIDS, VMTF, as well as our new ACIDS scheme, perform equally well. This is supported by the data in the upper part of Tab. 2, which corresponds to the same experiment. In the last three rows, we see that our generic priority queue is still somewhat optimized for EVSIDS and VMTF. For instance, ACIDS needs more time during bumping, which applies even more to INC and SUM.

In Fig. 2 and the lower part of Tab. 2, we compare against two variants of the new Lingeling, one using a dedicated optimized binary *heap* implementation for EVSIDS on one side, and the other one using a dedicated optimized *queue* implementation for VMTF. Both are slightly faster. Decision plus bumping time decreases. Otherwise, they show very similar behavior. We also compare against the state-of-the-art on these benchmarks, which for this small time-out of 1000 seconds, consists of SWDiA5BY A26 and also to some extent its "parent" Glucose

Table 2. Additional statistics for runs in Fig. 1 (top) and Fig. 2 (bottom). Columns correspond to the various considered configurations as discussed in the main text. Each of the two tables consists of three parts. In the first three rows, below the configuration names, the number of solved instances (out of 275) are listed, then split into unsatisfiable and satisfiable instances. The next 5 rows sum up statistics over all 275 runs. First, there is the overall number of reductions (learned clause deletions), number of restarts, number of times variables were rescored, followed by the number of conflicts and decisions. In the last 5 rows, the table shows the total time spent in pre- and inprocessing (simp), the CDCL loop (search), for bumping, searching for the next decision (decide), and rescoring (again over all 275 benchmarks). To give a concrete example, consider the "evsids" column. For all the considered 275 benchmarks, this configuration restarted 5.8 million times and used 3.7 billion decisions. In total, it used roughly 143.1 thousand seconds in search, among which it spent 2.3 thousand seconds selecting the next decision variable, and 7.8 thousand seconds for bumping. Altogether, it solved 157 instances (out of 275), from which 87 were unsatisfiable and 70 satisfiable.

	evsids	vmtf	acids	vsids	sum	inc	static
solved	157	152	151	114	58	47	26
unsatisfiable	87	85	82	51	22	17	9
satisfiable	70	67	69	63	36	30	17
reductions (1e3 #)	8	8	8	10	8	8	8
restarts (1e3 #)	5826	6000	5678	4491	2612	2387	5593
rescored (1e3 #)	253	0	0	2338	0	0	0
conflicts (1e6 #)	488	476	444	604	527	540	463
decisions (1e6 #)	3691	3581	3889	4263	2603	2567	21503
simp (1e3 sec)	29.7	30.0	29.4	32.6	34.5	34.1	31.2
search (1e3 sec)	143.1	146.4	147.9	174.9	203.9	209.7	226.7
bump (1e3 sec)	7.8	6.2	16.0	16.9	34.6	37.2	0.0
decide (1e3 sec)	2.3	2.5	2.6	2.8	1.7	1.7	12.9
rescore (1e3 sec)	0.2	0.0	0.0	2.6	0.0	0.0	0.0

	heap	queue	swd ia5by a26	glu cose 2.3	sc14 ayv	mini sat
solved	161	156	153	144	119	101
unsatisfiable	90	86	81	79	60	41
satisfiable	71	70	72	65	59	60
reductions (1e3 #)	8	8	59	10	30	—
restarts (1e3 #)	5870	6003	3210	3846	7948	1782
rescored (1e3 #)	241	0	—	—	393	—
conflicts (1e6 #)	463	474	650	728	760	1090
decisions (1e6 #)	3874	3566	5868	6818	5002	8388
simp (1e3 sec)	29.2	29.7	0.8	0.8	32.4	2.2
search (1e3 sec)	141.8	144.6	165.4	172.5	164.4	206.5
bump (1e3 sec)	3.8	4.9	—	—	3.3	—
decide (1e3 sec)	4.9	2.5	—	—	6.4	—
rescore (1e3 sec)	0.1	0.0	—	—	0.0	—

2.3. We also include MiniSAT 2.2, e.g., the "grandparent" of SWDiA5BY A26, and version *ayv* of Lingeling of the SAT Competition 2004 (sc14ayv).

5 Conclusion

In this paper, we evaluated several important CDCL decision schemes, including VSIDS [5] and the related EVSIDS [6] heuristic, which are considered to be one of the major reasons for good performance of modern SAT solvers on application benchmarks. While some reasons for the efficiency of VSIDS have been conjectured before [6], there is still a lot of ongoing research on finding good explanations for its performance, particularly related to problem structure [14–16]. Understanding VSIDS and related decision heuristics in a better way would help us to further improve performance of SAT solvers from a practical point of view, as well as open up possibilities for formal analysis in a theoretical sense.

To take a major step into that direction, we gave a detailed evaluation, comparing VSIDS and EVSIDS to several other heuristics, including static decision heuristics, a non-smoothing version of VSIDS and approximations of smoothing versions. We also proposed ACIDS, a new decision heuristic with similar properties as VSIDS, and revisited the VMTF scheme [8], which is easy to implement and also offers an alternative perspective on the meaning of the decision order of variables. We further provided a formalization of the score update as a function for each heuristic to capture its effect in a clear way.

In our experiments, it turned out that EVSIDS, VMTF, and ACIDS perform very similarly. Since efficient implementation is crucial and non-trivial for all those heuristics, we pointed out differences in underlying data structures and discussed important aspects of implementation in detail. We further provided detailed results, allowing us to analyze the effect variations in heuristics and implementations cause on the time spent in the individual steps of a search.

In addition, our results also shed new light on the performance of decision heuristics from an algorithmic point of view, as well as on many beliefs about decision heuristics that have been held previously. For instance, EVSIDS, VMTF, and ACIDS have in common that they put a very strong focus on variables that participated in the most recent conflicts. This is in contrast to heuristics, such as INC and SUM, where the occurrence in earlier conflicts also contributes significantly to the score of a decision variable throughout the whole progress of the search. While VSIDS, EVSIDS, and ACIDS implement explicit smoothing schemes to realize this kind of focus, the good performance of VMTF in our experiments shows that this is not necessarily required when directly using a more aggressive bumping strategy for recent conflict variables.

For future work, it will be interesting to analyze the contribution of the individual components in detail. Having provided a formal way of describing general scoring schemes and given several implementations of flexible data structures in a simpler way, the next steps could be motivated by theory as well as practice. For instance, combining aggressive bumping strategies in combination with particularly adapted smoothing schemes could yield even more efficient decision heuristics. Similarly, more refined functions for updating the variable scores

could be beneficial as well. On the other hand, simple but yet efficient heuristics, such as VMTF, might allow us to analyze CDCL more formally, e.g., in the context of proof complexity.

References

1. Balint, A., Belov, A., Heule, M.J.H., Järvisalo, M. (eds.): Proceedings of SAT Competition 2013. Volume B-2013-1 of Department of Computer Science Series of Publications B. University of Helsinki (2013)
2. Belov, A., Heule, M.J.H., Järvisalo, M. (eds.): Proceedings of SAT Competition 2014. Volume B-2014-2 of Department of Computer Science Series of Publications B. University of Helsinki (2014)
3. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. [41], 131–153
4. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5), 506–521 (1999)
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, pp. 530–535. ACM, Las Vegas, June 18–22, 2001
6. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University (2004)
9. Goldberg, E.I., Novikov, Y.: Berkmin: a fast and robust sat-solver. In: 2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), pp. 142–149. IEEE Computer Society, Paris, March 4–8, 2002
10. Gershman, R., Strichman, O.: Haifasat: A new robust SAT solver. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) Hardware and Software Verification and Testing. LNCS, vol. 3875, pp. 76–89. Springer, Heidelberg (2006)
11. Biere, A.: P{re, i}coSAT@SC 2009. In: SAT 2009 Competitive Event Booklet, pp. 42–43 (2009)
12. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer, Heidelberg (2012)
13. Heule, M., van Maaren, H.: Look-ahead based SAT solvers. [41], 155–184
14. Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of SAT formulas. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 410–423. Springer, Heidelberg (2012)
15. Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., Simon, L.: Impact of community structure on SAT solver performance. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 252–268. Springer, Heidelberg (2014)
16. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J.: The fractal dimension of SAT formulas. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 107–121. Springer, Heidelberg (2014)
17. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)

18. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
19. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)
20. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence **1**(1–4), 167–187 (1990)
21. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) EPIA 1999. LNCS (LNAI), vol. 1695, pp. 62–74. Springer, Heidelberg (1999)
22. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
23. Zhang, H.: SATO: an efficient propositional prover. In: McCune, William (ed.) CADE 1997. LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
24. Biere, A.: PicoSAT essentials. JSAT **4**(2–4), 75–97 (2008)
25. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. (JAIR) **22**, 319–351 (2004)
26. Han, H., Somenzi, F.: On-the-fly clause improvement. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 209–222. Springer, Heidelberg (2009)
27. Hamadi, Y., Jabbour, S., Sais, L.: Learning for dynamic subsumption. In: 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, ICTAI 2009, pp. 328–335. IEEE Computer Society, November 2–4, 2009
28. van der Tak, P., Ramos, A., Heule, M.J.H.: Reusing the assignment trail in CDCL solvers. JSAT **7**(4), 133–138 (2011)
29. Nadel, A.: Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University (2002)
30. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)
31. Biere, A.: Lingeling and friends entering the SAT challenge 2012. In: Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., Sinz, C. (eds.) Proceedings SAT Challenge 2012: Solver and Benchmark Descriptions. Volume B-2012-2 of Department of Computer Science Series of Publications B., University of Helsinki, pp. 33–34 (2012)
32. Biere, A.: Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. [2], 39–40
33. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, Pasadena, California, USA, pp. 399–404, July 11–17, 2009
34. Audemard, G., Simon, L.: Glucose 2.3 in the SAT 2013 Competition. [1], 42–43
35. Oh, C.: MiniSat HACK 999ED, MiniSat HACK 1430ED and SWDiA5BY. [2], 46–47
36. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
37. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 118–126. Springer, Heidelberg (2012)

38. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
39. Wallner, J.P.: Benchmark for complete and stable semantics for argumentation frameworks. [2], 84–85
40. Biere, A., Heule, M.J.H., Järvisalo, M., Manthey, N.: Equivalence checking of HWMCC 2012 circuits. [1], 104
41. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)