Decision Heuristics in a Constraint-based Product Configurator

Matthias Gorenflo[1], Tomáš Balyo[1], Markus Iser[2,3] and Tobias Ostertag[1,*]

[1]*CAS Software AG, CAS-Weg 1 - 5, 76131 Karlsruhe, Germany*
[2]*Karlsruhe Institute of Technology (KIT), KIT-Department of Informatics, Karlsruhe, Germany*
[3]*University of Helsinki, Department of Computer Science / HIIT, Helsinki, Finland*

Abstract
This paper presents an evaluation of decision heuristics of solvers of the Boolean satisfiability problem (SAT) in the context of constraint-based product configuration. In product configuration, variable assignments are searched in real-time, based on interactively formulated user requirements. Operating on user's successive input poses new requirements, such as low-latency interactivity as well as deterministic and minimal implicit product changes. This work presents a performance evaluation of several heuristics from the SAT literature along with new variants that address the special real-time requirements of incremental product configuration. Our results show that the execution time on an industrial benchmark can be significantly improved with our new heuristic.

Keywords
Configuration, Constraint-based Products, Decision Heuristics, Boolean Satisfiabiliry Problem (SAT)

1. Introduction

1.1. Motivation

In wake of an increasing globalization, the demand for customized and personalized products rises in manufacturing and service industries, which previously only utilized the advantages of mass production to offer standardized products for a good value. Shaped by Stanley Davis and his 1987 book *Future Perfect* [1], this new frontier is called *mass customization* and wants to meet the product needs of individual customers. "At its core, is a tremendous increase in variety and customization without a corresponding increase in costs. At its limit, it is the mass production of individually customized goods and services. At its best, it provides strategic advantage and economic value" [2]. This results in increasingly complex models of the product variants that can be configured, if the model shall offer many intertwined parameters a customer is allowed to choose from. Various domains are applicable for product configuration with some of the more complex product models revolving around the assembly of different vehicles. But the decision, if every request of a customer is viable, can become troublesome for even reasonably sized models. In remedying the solving process automatically, so called knowledge-based configuration systems (or simply product configurators) [3] play a key role. The product configurator is a tool to on one hand decide on a product that respects the demands of the user, which can be a customer, while conforming to the limitations and constraints of the manufacturer on the other hand. The description of the product model lies at the core and spans the solution space, which is the set of all possible product variants. This work focuses on an interactive configuration process. During the configuration of a product, a user performs adjustments to the current product by selecting from different parts or properties. Incremental requests coming from the user are called user wishes and the configurator has to check the validity of them.

A method of realizing the constraints of a product model is to utilize propositional logic [4]. Hereby, for a configuration to be valid, it has to satisfy a set of propositional formulas expressing the product configuration problem as shown in [5]. A SAT solver is then capable of checking if all user wishes can be satisfied under the propositional representation of the configurator's specifications. The truth assignment then holds the information about the concrete manifestation of the configured product.

The complexity of the SAT problem is in nondeterministic polynomial time (NP) which can lead to long computation times, exponential in the size of the problem. Contrary to that is the *interactive* nature of product configuration, where users expect a fast response regarding the feasibility of their latest demand. The shorter and faster the time spent on satisfiabilty checking, the shorter the waiting time for users. Minimizing the time of the SAT solving therefore plays an essential

ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain
*Corresponding author.
markus.iser@kit.edu (M. Iser); Tobias.Ostertag@cas.de (T. Ostertag)
0000-0003-2904-232X (M. Iser); 0000-0003-3294-3807 (T. Ostertag)

role for interactive, low-latency product configuration.

Luckily, many SAT formulas that model real world problems can be solved quickly thanks to the classic DPLL algorithm [6], which has seen many improvements and additions over the last decades. Sophisticated heuristics play a key role here. This is especially true for decision and branching heuristics, which control the ordering and values of the algorithm's truth assignments [7].

The area of product configuration tends to place different demands on a solver than the classical SAT formulas. The process of incremental product configuration generally leads to less complex computations, since we solve many simple formulas and not just one complex formula.

New challenges arise when the last user requirement cannot be met with a previously selected configuration. The possibility of stating conflicting requirements is intentional, as the user may not be aware of all the interactions between different requirements, or may be in the process of making fundamental changes. In either case, the user relies on feedback from the configurator, and it is the solver's job to calculate a new valid product configuration. This amounts to solving an optimization problem where the new configuration contains as few changes as possible while omitting as few user requirements as possible. To realize this idea, the configurator weights the user's requirements so that changes are associated with costs.

Furthermore, not only a single optimal solution is of interest but every solution with the smallest cost or within a certain delta. So the user is able to select one of the best fitting alternatives. The optimization problem can be realized as a *minimum-cost satisfiability (MinCostSAT) problem* or an equivalent *maximum satisfiability (MaxSAT) problem*. Nevertheless, the additional challenges posed by product configuration also present new opportunities to derive better heuristics for this specific use case.

1.2. Goals and Contributions

This paper lays the focal point on the goal of reaching an as optimal as possible performance of decision heuristics for SAT in the context of incremental product configuration with weighted user requirements as described in the previous section. We implemented four decision heuristics under the additional requirement of deterministic user experience, i.e., the configurator keeps producing the same results. So a user receives the same selection of alternative configurations for every repetition of a specific interactive configuration sequence. The performance of various known branching heuristics from SAT solving as well as new heuristic ideas are evaluated in the product configuration context. We examine how the different sub-formula types and literal types can be exploited effectively in these heuristics.

The implementation of the heuristics and the following evaluation is done with the product configurator *Merlin CPQ* by the *CAS Software AG*. *Merlin* has a specialized solving process of incremental problems from interactive configuration. Compared to a typical SAT solver, this configurator also supports multiple different formula terms as well as arithmetic expressions.

We evaluate the heuristics with respect to their execution time and the decision count on several product configuration benchmarks and specifically focus our efforts on speeding up the more complex to solve problems to enable fluid and user-friendly configuration of the desired product, even in these taxing cases. Experimental results show that the performance of the especially expensive benchmarks is roughly doubled by the best of the presented branching heuristics.

Interactive product configuration is a unique domain, so general-purpose heuristics do not necessarily achieve the best performance here. In this paper, we introduce new branching heuristics that achieve better performance in the domain of interactive product configuration than the well-known top dogs for more general-purpose benchmarks. It remains to be seen, how the heuristics evaluated in this paper behave in other benchmark domains.

1.3. Related Work

Most modern satisfiability solvers are based on the highly influential foundation of Davis and Putnam [8] and the shortly following DPLL algorithm [9]. Advancements were made regarding decision heuristics, efficient data structures [10], clause learning [11, 12], and search restarts [13, 14]. The effectiveness of this method caught interest in several domains taking advantage of the strong performance of SAT solvers, especially after multiple strong improvements around the turn of the last century. Probably the two largest domains using SAT are automated planning and scheduling [15, 16] and formal verification [17, 18].

Reductions to SAT are also well known in the context of product configuration [3]. Sinz, Kaiser and Küchlin show different methods in [19, 4] that can be deployed for configuration and [5] demonstrates how SAT solvers are able to be used for an interactive configuration process.

The most dominant decision heuristic in SAT solving of the last century is probably Dynamic Largest Individual Sum (DLIS) [20] found in GRASP, the algorithm that revolutionized DPLL and gave birth to the new solving paradigm Conflict-driven Clause Learning (CDCL). The predominant decision heuristic in CDCL solvers in the last two decades has been Variable State Independent Decaying Sum (VSIDS) which was first presented in Chaff [10].

Nevertheless, decision heuristics for CDCL are a vital research area [21]. Application-specific specialized heuristics are evaluated regularly [22, 23, 24]. Recently, approaches based on reinforcement learning have been successfully used to select heuristics dynamically [25, 26].

Efforts to develop specialized heuristics for product configuration were made in [27] by applying graph analysis to propositional formulas in *Merlin CPQ*. Nevertheless, the resulting heuristic based on coreness is not adaptively reacting to conflicts that will often appear in DPLL and thus could not dethrone VSIDS, which still is *Merlin*'s standard heuristic.

1.4. Overview

Chapter 2 presents the theoretical concepts and definitions covering propositional logic as well as SAT and MaxSAT solving in the context of incremental product configuration. In Chapter 3, we explain the ideas and motivations behind the branching heuristics under consideration. The implementation in *Merlin CPQ* is described in Chapter 4. Subsequently, Chapter 5 presents the results of our performance evaluation of the presented heuristics. Lastly, we provide a summary and our perspective on potential future work in Chapter 6.

2. Theoretical Preliminaries

This chapter is meant to give a brief introduction into necessary preliminaries for our work, but this information is not targeting to be an exhaustive treatise about the field of SAT solving.

2.1. Propositional Logic

In propositional logic, we have two Boolean constants to represent values of ”true” and ”false”. Propositional formulas are built from Boolean variables and operators such as negation, conjunction, and disjunction.

These operators are interpreted with respect to the usual semantics, i.e., the negation of an argument is true if and only if the argument is false, the conjunction of a set of arguments is true if and only if all arguments are true, and the disjunction of a set of arguments is false if and only if all arguments are false.

A variable assignment maps all Boolean variables of a formula to Boolean constants. The truth value of a formula under a given assignment is determined by replacing the variables with Boolean constants accordingly and by successively interpreting the truth value of all sub-formulas according to the operator semantics.

2.1.1. Normal Forms and Clause Types

The most common appearance of SAT formulas is in *conjunctive* normal form (CNF). A CNF formula is a conjunction of clauses. Each clause itself is a disjunction of one or more literals. A literal is either a Boolean variable (positive literal) or the negation of a variable (negative literal). In contrast to CNF formulas, a formula in *disjunctive* normal form (DNF) if it is a disjunction of terms, where a term is a conjunction of literals.

The *Merlin* product configurator supports formulas constructed as conjunctions of both types of normal forms, CNF and DNF formulas, as well as at-most-one (AMO) constraints over sets of literals. An AMO constraint evaluates to true if and only if at most one of its literals is true. There are several ways to encode this constraint in propositional logic but further details are unimportant for this paper.

2.2. Propositional Satisfiability

The satisfiability (SAT) problem asks the question whether it is possible to find a complete variable assignment that interprets a given formula to true (satisfiable) or declares this impossible (unsatisfiable). A SAT solver can be used to answer such problem instances. We can additionally call a propositional formula valid if it evaluates to true for every possible assignment.

The easiest way to determine satisfiability is achieved by creating a truth table for the whole formula and check whether any resulting value is true. The issue is the effort of this procedure, which grows exponentially with the number of variables.

2.2.1. DPLL Algorithm

DPLL is an enhanced depth-first search algorithm. A partial assignment is successively collected for a given CNF formula by adding literals during its search procedure. The assigned variables are then used to simplify the original formula.

Central to DPLL is unit propagation. If the CNF contains a unit clause (a clauses that consist of only a single literal), then the clause's literal is immediately used to extend the current partial assignment. All further clauses where this literal occurs can then be dropped from the formula because they are satisfied by the assignment. Furthermore, the negation of the literal is removed from all clauses in which it occurs and we call clauses where this happens ”touched” (this will be important later). The propagation stops when no more unit clauses are present.

Afterwards, one of three states is reached. If no clause remains, the instance is satisfiable and the algorithm returns a satisfying assignment. If an empty clause emerges, i.e., all literals in that clause are falsified, the instance is unsatisfied under the current partial assignment. This means that decisions have to be undone (backtracking) and if no decision can be undone the instance is unsatisfiable. Otherwise, we need to heuristically pick a decision variable which we use to extend the current partial assignment. Two recursive calls have to be made now; the variable is assigned true in one branch and false in the other.

This algorithm covers all possible branches in the worst case, which makes it sound and complete but also not better than the naive approach regarding this aspect. However, the performance on real world problems is commonly clearly superior to the worst-case performance due to unit propagation and further techniques.

2.2.2. Clause Learning

A major improvement made to DPLL-based solvers has been the concept of conflict-driven clause learning (CDCL), first shown in the GRASP solver [11, 12] and then advanced by Chaff [10]. It has been shown that CDCL p-simulates general resolution which makes CDCL strictly more powerful than classic DPLL [28]. In CDCL, each time a decision leads to an empty clause, the conflicting assignment is analyzed to derive a new clause that is added to the formula. The idea behind that is to avoid repeating mistakes across different similar branches in the search. Additionally, conflict analysis can determine a backtracking level for directly backtracking multiple decision levels instead of one at a time. A slightly closer look at this conflict analysis will be taken in the next chapter, where we discuss heuristics that operate on the implication graph that is used for conduction such a conflict analysis.

2.3. Propositional Optimization

The common formalism for describing propositional optimization problems is the weighted Maximum Satisfiability Problem [29]. A MaxSAT instance consists of hard clauses and soft clauses and associates a weight with each soft clause. A solution for a MaxSAT instance is characterized such that it satisfies all hard clauses and the sum of weights of satisfied soft clauses is maximized. In contrast, Minimum-Cost Satisfiability (MinCostSAT) associates weights with every variable to define the cost of an assignment. In that case, the target is to find a solution of lowest possible cost. The reduction of MinCostSAT to MaxSAT is straight-forward through adding the variables as negative literals in a soft unit clause with a weight according to the variable's cost.

Merlin CPQ uses such costs on the product's properties in the previous step (since we want to avoid overriding preconfigured parts as much as possible) as well as for the user wishes if they cannot all be realized together. In order to achieve this goal, *Merlin's* DPLL implementation supports best-first search, that prefers branches of minimal cost. This search follows the branches that cost the least. Should a branch appear that increases the cost above the currently optimal path's cost, then the search is interrupted and always greedily continued at another branch that sits on the path of minimal cost. A relaxed version with a reduced memory consumption is the beam search. Instead of potentially exploring all promising branches, only a maximum number of the cost-optimized children (defined by the beam width) are considered.

3. Decision Heuristics

Solving SAT instances under the given optimization goals is computationally hard. Nonetheless, DPLL in combination with good heuristics can often efficiently solve many formulas that model real world problems. Thus we are taking a look at several decision heuristics from SAT solving. In this chapter, we review and present well-known and new heuristics that implemented and evaluated in the context of product configuration in *Merlin CPQ*.

Important regarding possible heuristics for *Merlin* is also that they have to pick their decision from a predefined set of candidate literals. These candidates stem from the unit propagation's touched clauses. The touched clauses unassigned literals are candidate literals if the clauses are still unsatisfied when the heuristic is called.

3.1. Variable State Independent Decaying Sum

Variable State Independent Decaying Sum (VSIDS) is one of the most efficient decision heuristic for SAT solving since over 20 years. The original idea arose for the solver Chaff [10]. VSIDS was then refined and adjusted slightly over the years, for example in MiniSat [30]. The common idea behind the heuristic is that every variable maintains a counter. The unassigned variable with the highest counter value is chosen for the branching when the solving procedure requires a next decision. The counters are maintained as follows:

- Initially, every counter starts with a score that is typically set to zero. Alternatively, the initial score could also be the amount of occurrences of the respective variable in the propositional formula.
- The counter is incremented each time the respective variable is involved in the reasons for a conflicting assignment. With Chaff the focus was only on the learned clauses, MiniSat expanded the involvement to all clauses that appeared during conflict analysis.
- All counters are periodically decreased by a constant factor after a certain amount of conflicts occurred. This so called decaying is meant to give higher priority to variables that appeared in more recent conflicts.

3.2. Distance Heuristic

The distance heuristic pursues a promising approach as described in [31]. This branching heuristic is based on counters like VSIDS and also makes use of decaying. However, the score increment dynamically takes into account an estimate of how much a variables contributes to the conflicting assignment under analysis. This is done by taking into account the position of the variables in the implication graph that is commonly used in CDCL for analyzing the reasons for a conflicting assignment.

An implication graph is a directed acyclic graph. Each node in the implication graph either represents a clause that triggered an assignment during unit propagation (including the empty clause which triggered the conflict under analysis) or a decision literal. In the implication graph, edges represent propagated, i.e. implied, literal assignments. Each edge is rooted in the node representing the clause or decision that is the reason for the assignment and ends in the node representing the clause in which this assignment falsifies a literal. The creation process starts by representing the conflicting clause with a node and subsequently adding incoming edges for each falsified literal, rooting them in nodes representing the respective reasons for their assignment.

The authors hypothesize that the fewer clauses a variable depends on during the conflict, the higher the probability that the variable contributes to a later conflict. Unfortunately, realizing this dependence hypothesis exactly would be too computationally intensive. The distance heuristic is an approximation to this idea, in that it determines the number of vertices that are located on the longest path in the implication graph from the node representing the reason of a variable's assignment to the node representing the conflicting clause.

The distance heuristic scores literals differently depending on their responsibility for the conflict in contrast to the constant increment of VSIDS. This heuristic aims to be more precise because of the more elaborate scoring, especially during the beginning of the search where only few conflicts occurred and none of the decision heuristics is sufficiently initialized yet. However, the authors of the distance heuristic point to the computational overhead of their heuristic as compared to VSIDS. In their empirical evaluation, they found that it is better to switch from the distance heuristic to VSIDS after 50,000 conflicts. This seems especially fitting for product configuration because the focus on interactivity means that the targeted problems are typically faster to solve compared to some large SAT instances running several minutes and resolving tens of thousands of conflicts.

3.3. Conflict Heuristic

While the previously described heuristics increase the scores of variables that appear in reason clauses or learned clauses, we thought that in our context we could try something that is much simpler. The conflict heuristic is our own variant of VSIDS, where we simply increase the scores of variables in the conflicting clause. The intuition behind this procedure is that such variable assignments should be fixed as quickly as possible, ideally to satisfy as many unsatisfied clauses as possible.

3.4. Heuristics based on Pure Literals

Pure literals were already used in the original algorithm of Davis and Putnam in the affirmative-negative rule [8]. After each unit propagation, the rule searches for literals whose negated form does not occur in the formula under the current partial assignment. Such literals can be set to true without conflict, so that clauses containing them can be eliminated from the formula. This procedure was later called "pure literal elimination". However, this has disappeared from most modern CDCL solvers because the computational overhead involved is usually not compensated by the benefits of this instance simplification procedure. This is due to the advent of efficient data structures for unit propagation which are commonly used in modern SAT solvers.

Merlin uses different data structures for unit propagation, primarily due to the additional requirements imposed by incremental user interaction, which requires the ability to correct assignments deterministically and non-chronologically. But the non-chronological correction of assignments is also detrimental to the applicability of the pure literal rule. At any point in time, a user could select a property such that a previously pure literal is no longer pure. This would not be possible in classical SAT solving, where a pure literal appearing under a particular variable assignment remains a pure literal at least for all branches below it. However, assigning a pure literal explicitly by a branching decision is a valid option.

3.4.1. Pure Literal Heuristic

A first idea for a decision heuristic that utilizes this concept can thus try to select pure literals whenever possible and thus add them to the partial truth assignment. If there are no pure literals available, then the remaining variables are handled afterwards with a different strategy. In the basic variant this is done by a fixed initial ordering.

3.4.2. Pure Literal Phase

The approach of favoring pure literals can and should though be combined with other heuristics as well, where those heuristics proceed instead of the random ordering.

Additionally, preferring pure literals can be done in certain phases. So pure literals could only be preferred after a certain amount of conflicts or decisions, up to a certain amount, within a range, or by having alternating phases of favoring pure literals and just the basic heuristic.

3.4.3. State Dependent Pure Literals

We try some modifications related to keeping pure literals up-to-date during run time for potential improvement. A heuristic can remember so-called ”almost” pure literals, which we see as variables that are mostly present in one polarity, while the other polarity is rare. For every almost pure literal, the configurator also stores the clauses preventing the literal from being completely pure. If all the clauses related to an almost pure literal are marked deleted during unit propagation, then this literal is being added to the set of pure literals. This would thus increase the amount of available pure literals.

3.4.4. Pureness Variant

Pureness of literals is another variant of pure literals. The amount of positive versus negative literals of each variable determines a pureness percentage. Variables with a balanced share of positive and negative appearances lead to a low pureness, almost pure literals get a high pureness percentage, and fully pure literals would be 100 percent. The decision heuristic then picks the literal with highest pureness score. Again, different heuristics can be used as combination, either as tie breaker or to set multiple scores off against each other.

3.5. Clause-Based Heuristic

For the clause-based heuristic (CBH), all clauses are kept in an ordered list and decision literals are picked from the top-most unsatisfied clauses in that list according to [32]. To create an initial ordering of clauses, the priority of a clause is calculated based on the number of occurrences of the literals it contains. To increase the proximity of clauses that have literals in common, the ordered list is gradually populated by starting with the clause with the highest priority and then first increasing the priorities of all clauses that have literals in common with the clause just added. This process is repeated until all clauses are added to the list. The ordered list is then update on each conflict by moving the conflicting clause and the reason clauses to the front. For the decision heuristic to select a literal from the top-most unsatified clauses, each literal has additional counters to measure its contribution to (recent) conflicts, while larger scores are preferred. Further details can be found in [32].

4. Implementation

Merlin CPQ already contains implementations of VSIDS and a few other heuristics. In this section we discuss implementation details of VSIDS, CBH and CBH Simplified.

4.1. Variable State Independent Decaying Sum

VSIDS is already part of *Merlin* and the default heuristic. The concrete realization of VSIDS is quite different from the ones used in off-the-shelf SAT solvers. One difference is that Merlin initially selects variable's based on their individual occurrence counts in the formula. This is in contrast to most implementations which initialize scores with zero (cf. CaDiCaL [33], or MiniSat [30]) The variable occurrence counts and the VSIDS score of a variable are kept separately. The sum of both scores is then used to determine the final score used with decisions.

During conflict analysis, the VSIDS scores are incremented by a bump value which is initially set to 100. The purpose of VSIDS decay is to make previously bumped variables less important over time. In *Merlin*, this is realized by increasing the bump value over time. Every 15 conflicts the bump value is increased by two percent.

4.2. Clause-Based Heuristic

An exact reimplementation of the clause-based heuristic as it is described in [32] would not suit *Merlin*'s heuristics, who have to make their decision based on the given set of candidate literals. Therefore, we do not explicitly manage a global clause-list as it is done in the original CBH. Only initially, a list based on all clauses is created with the ordering described by the CBH authors. Hereby, we go through the original set of all clauses and remember for each unique literal, in which clauses it was contained. Then we create a new list with every variable, which is sorted regarding the variables' score based on the counters of the variable's two literals. The paper would now put the clauses containing the highest scoring variable into the clause list, however, we assign a priority to the literals depending on their supposed position in this list instead of explicitly storing the clause list for later usage. So in the first step, the highest scoring variable and all variables that share a clause with it receive priority -1. Each variable has two additional local literal counters which get increased for every occurrence of its respective literal as part of a clause added to the clause list. Therefore, we also increase this local score when setting the priority. The priority is afterwards decreased by one for the next bunch of clauses. This procedure now continues until every variable was selected, while the local score is always added to the variable's score.

Updates due to a conflict can then be realized by calculating the conflict responsible clauses. Each literal of these clauses receives a new priority (starting at zero) that is one higher than the previous highest priority. This procedure is equal to moving clauses to the topmost position of a clause list. Additionally, these literals get a bump to their conflict counter. The same then happens to the actual conflict clause with the next priority value.

Choosing a literal can then be done by checking the position of a literal via its priority. The ones with the highest priority (which is equal to the topmost position in a clause list) are preferred. Literals from the supposed topmost clause will receive the same priority, thus the best literal is further selected depending on a variable's counters that get bumped for conflicts. The official clause-based heuristic seems a bit over-engineered because it keeps counters that get only increased during a conflict and ones that additionally decay periodically. This decay seems of questionable importance since we already move literals that occur in recent conflicts to the top. Thus we do drop this second counter in our implementation.

CBH also operates rather complex in regard to the different counters for the variables because for every comparison of two variables, several counters need to be added and multiplied together for each of the variables. This may cause unnecessary overhead. The formulas that are used to combine the counters of a positive and negative literal into a variable's counter are the main reason behind CBH's need for this dynamic calculation. In all these formulas, the positive and negative literal get added to three times the minimum of the two literals. The reason behind this last term is to punish variables where just one of its literals is important, kind of like the inverse of pureness. But the CBH authors do not specify how strong its impact on the heuristics performance is. So we create a simplified clause-based heuristic regarding the scoring functions by removing the minimum component. Consequently, we also drop the distinction between positive and negative literals in regard to the score and directly add them up. This means we explicitly combine values from a positive and negative literal pair into one variable count, since the distinction will be unimportant for our simplified heuristic. As a welcome side effect, this combination also reduces the number of individual counters that need to be stored.

5. Evaluation

The evaluation of the heuristics is done on several problems from three rule sets in *Merlin CPQ*. These rule sets are based on the use cases of customers who use *Merlin* for product configuration. Our testset contains a total of 241 individual benchmarks across 85 methods of customizing trucks. Each test translates to one user wish

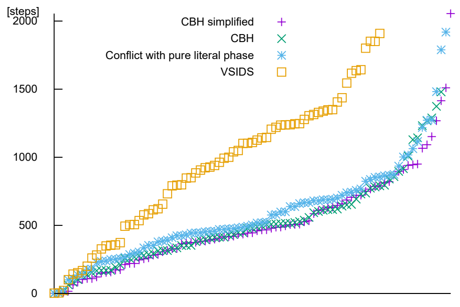

Figure 1: Evaluation with respect to decision count.

and a method with multiple tests translates to a series of user requests. The most important statistics of the formulas used in testing are as follows: 28000 variables, 127000 clauses, 5000 other constraints.

The performance metrics we consider for the problems in this evaluation are the execution time, averaged over three runs, and the sum of the amount of contradictions that were encountered plus the number of branches that were taken. The number of branches taken due to decisions is a good indicator of the heuristic's effectiveness. Contradictions happen less often, ideally a few hundreds for the most elaborate methods and at most around five to ten percent the amount of branches. Therefore we included it as a small contribution on defining the effectiveness of the decisions selected. Both numbers should be as low as possible and the sum of them is called "steps" in the following for simplicity. Their advantage is that the amount of contradictions and branches is deterministic in every run and hardware agnostic. The only downside is that potential time-consuming calculations of a heuristic remain hidden when just looking at the step count. Therefore we also measure the concrete execution time (in milliseconds) for certain comparisons where we expect overheads from parts of a heuristic.

Every metric is always plotted per test method. Cactus plots – typically seen at SAT and SMT competitions – are used whenever more than two heuristics are compared. Here, the methods are ordered by importance for each heuristic element displayed in the graph. The top and right borders represent the timeout limit.

All measurements are performed on an Intel Core i7-4710MQ CPU at 2.5 GHz with 16 GB of RAM under Windows 10, version 21H1. The *Merlin CPQ* version in use is the state of the master branch on 9. December 2021 and running JDK 11.0.13.8.

In Figure 1 we can see that the quality of the literals chosen by CBH are rather good, meaning that it takes just a few decisions to finish the algorithm for most tests

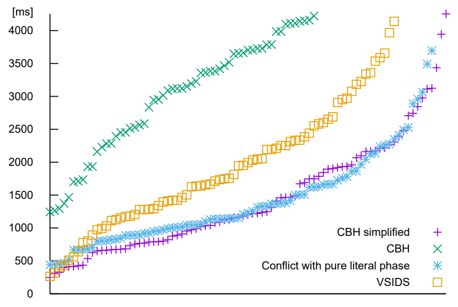

Figure 2: Evaluation with respect to execution time.

– typically even less than the pure literal modified VSIDS variants. However, looking at Figure 2 we see that the complicated summation used by CBH comes at a huge penalty for the execution time. The modified and simplified alternative eliminates this issue.

The CBH simplified heuristic works as intended. It keeps the strength of CBH's good decision making while maintaining to reach this conclusion quickly and without too much overhead. The initial priority should also be calculated more efficiently. So the execution time across the whole of rule set 1 is therefore always competitive and often even faster than the best VSIDS variants.

6. Conclusion

We presented and implemented several heuristics and tweaks that were able to improve the decision making of *Merlin CPQ*. The largest tests from an industrially used rule set could be significantly improved by an optimized version of the clause-based heuristic. Our idea of a branching heuristic that specializes on the variables causing conflicts and utilized pure literals as preference also performed almost as strong.

In a configurator, the initial weighting of variables is substantial for the performance. The main conclusion for a product configuration heuristic is the significance of selecting interrelated variables. There are several ways a configurator can group them together and take advantage of them. Keeping them united according to their occurrence in clauses is what worked best for our benchmarks. Other groupings are possible according to concepts like pure literals, the types of literals and their clause type they are contained in, or their appearance in the start configuration. The aim is to offer a few good heuristics to the people who design the rule sets that are then brought to users who can configure their product. The designers typically test several heuristics with their specific environment and are then able to choose the one that performs best for them.

There are still many promising heuristics that can be tested in the field of product configuration, as well as countless combinations and alternations. The currently popular Learning Rate Branching (LRB) [34] based on reinforcement learning is a prime candidate. However, it is important to consider the limitations of the product configurator's structure. Some information might not be directly accessible to the decision heuristic and only realizable with major changes to the whole architecture. Ideas that contain parts which are inefficient to calculate might need to be changed to perform as desired. So a heuristic has to be adopted to the system it is used in, otherwise the configuration system would be in certain aspects designed around the decision heuristic.

References

[1] S. Davis, Future Perfect, Basic Books, 1987.

[2] B. Pine, J. Pine, S. Davis, H. B. Press, Mass Customization: The New Frontier in Business Competition, Harvard Business School Press, 1993.

[3] D. Sabin, R. Weigel, Product configuration frameworks - a survey, IEEE Intell. Syst. 13 (1998) 42–49.

[4] C. Sinz, A. Kaiser, W. Küchlin, Formal methods for the validation of automotive product configuration data, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17 (2003) 75 – 97.

[5] M. Janota, Do sat solvers make good configurators?, in: SPLC, 2008.

[6] M. D. Davis, G. Logemann, D. W. Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397.

[7] H. Katebi, K. A. Sakallah, J. Marques-Silva, Empirical study of the anatomy of modern sat solvers, in: SAT, 2011.

[8] M. D. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.

[9] D. W. Loveland, A. Sabharwal, B. Selman, Dpll: The core of modern satisfiability solvers, in: Martin Davis on Computability, Computational Logic, and Mathematical Foundations, 2016.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient sat solver, Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232) (2001) 530–535.

[11] J. Marques-Silva, K. A. Sakallah, Grasp-a new search algorithm for satisfiability, Proceedings of International Conference on Computer Aided Design (1996) 220–227.

[12] J. Marques-Silva, K. A. Sakallah, Grasp: A search algorithm for propositional satisfiability, IEEE Trans. Computers 48 (1999) 506–521.

[13] G. Audemard, L. Simon, Refining restarts strategies for sat and unsat, in: CP, 2012.

[14] A. Biere, A. Fröhlich, Evaluating cdcl restart schemes, in: POS@SAT, 2018.

[15] H. A. Kautz, B. Selman, Planning as satisfiability, in: ECAI, 1992.

[16] H. A. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic and stochastic search, in: AAAI/IAAI, Vol. 2, 1996.

[17] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without bdds, in: TACAS, 1999.

[18] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Adv. Comput. 58 (2003) 117–148.

[19] C. Sinz, A. Kaiser, W. Küchlin, Detection of inconsistencies in complex product configuration data using extended propositional sat-checking, in: FLAIRS Conference, 2001.

[20] J. Marques-Silva, The impact of branching heuristics in propositional satisfiability algorithms, in: EPIA, 1999.

[21] A. Biere, A. Fröhlich, Evaluating cdcl variable scoring schemes, in: SAT, 2015.

[22] J. Rintanen, Planning as satisfiability: Heuristics, Artif. Intell. 193 (2012) 45–86.

[23] M. Iser, M. Taghdiri, C. Sinz, Optimizing minisat variable orderings for the relational model finder kodkod - (poster presentation), in: A. Cimatti, R. Sebastiani (Eds.), Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, volume 7317 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 483–484.

[24] P. Beame, H. A. Kautz, A. Sabharwal, Towards understanding and harnessing the potential of clause learning, CoRR abs/1107.0044 (2011).

[25] M. S. Cherif, D. Habet, C. Terrioux, Combining VSIDS and CHB using restarts in SAT, in: L. D. Michel (Ed.), 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021, volume 210 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:19.

[26] D. Speck, A. Biedenkapp, F. Hutter, R. Mattmüller, M. Lindauer, Learning heuristic selection with dynamic algorithm configuration, in: Proceedings of the International Conference on Automated Planning and Scheduling, volume 31, 2021, pp. 597–605.

[27] S. Haug, Graphentheoretische optimierung der satberechnung im anwendungsfall produktkonfiguration, 2021.

[28] K. Pipatsrisawat, A. Darwiche, On the power of clause-learning SAT solvers as resolution engines, Artif. Intell. 175 (2011) 512–525.

[29] C. M. Li, F. Manyà, Maxsat, hard and soft constraints, in: Handbook of Satisfiability, 2021.

[30] N. Sörensson, N. Eén, Minisat v1.13 - a sat solver with conflict-clause minimization, 2005.

[31] F. Xiao, C. Li, M. Luo, F. Manyà, Z. Lü, Y. Li, A branching heuristic for sat solvers based on complete implication graphs, Science China Information Sciences 62 (2017) 1–13.

[32] N. Dershowitz, Z. Hanna, A. Nadel, A clause-based heuristic for sat solvers, in: SAT, 2005.

[33] A. Biere, K. Fazekas, M. Fleury, M. Heisinger, CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020, in: T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions, volume B-2020-1 of *Department of Computer Science Report Series B*, University of Helsinki, 2020, pp. 51–53.

[34] J. H. Liang, V. Ganesh, P. Poupart, K. Czarnecki, Learning rate based branching heuristic for sat solvers, in: SAT, 2016.