Layered Clause Selection for Saturation-based Theorem Proving

Bernhard Gleiss[1]*and Martin Suda[2]†

[1] TU Wien, Austria
[2] Czech Technical University in Prague, Czech Republic

Abstract

Clause selection is one of the main heuristic decision points in navigating proof search of saturation-based theorem provers. A recently developed layered clause selection framework allows one to boost a basic clause selection heuristic by organising clauses into groups of more or less promising ones according to a specified numerical *feature*. In this work, we investigate this framework in depth and introduce, in addition to a previously presented feature (based on the amount of theory reasoning in the derivation of a clause), three new features for clause selection (tracking relatedness to the goal, the number of split dependencies in the AVATAR architecture, and closeness to the Horn fragment, respectively). We implemented the resulting clause selection heuristics in the state-of-the-art saturation-based theorem prover VAMPIRE and present an evaluation of these new clause-selection strategies and their combinations over the TPTP and SMTLIB libraries.

1 Introduction

In the context of automated theorem proving, saturation refers to the process of iteratively deriving (according to a particular inference system) logical consequences of a set of given input clauses until the empty clause is derived, which witnesses unsatisfiability. Modern saturation-based theorem provers for first-order logic typically employ some variant of a *given-clause algorithm*, in which clauses are selected for inferences one by one [14]. Clause selection, i.e. the procedure for picking in each iteration the next clause to process, is one of the main heuristic decision points in the prover, hugely affecting its performance [18].

The standard technology for implementing clause selection heuristics relies on two concepts. First, certain numerical *clause evaluation criteria* are identified, such as the number of symbols in a clause (a.k.a. clause *weight*) or the iteration when the clause was derived during search (a.k.a. clause *age*[1]), such that a small clause in terms of the given criterion is more likely to lead to a refutation than a large one. The prover then maintains a priority queue for each criterion for the efficient extraction of "the current best clause". Second, the prover *alternates* picking clauses from these priority queues using a specified ratio. For example, a heuristic may specify that every 10 selections, the prover should pick 9 clauses that are small by weight, i.e. *light*, and one clause that is small by age, i.e. *old*.

A recently developed *layered clause selection* framework [21, 6] allows one to boost a basic clause selection heuristic, such as the one just described, by organising clauses into groups of more or less promising ones according to a specified numerical *feature* and a certain *cutoff* value

*Supported by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, and the Austrian FWF research project W1255-N23

†Supported by the ERC Consolidator grant AI4REASON no. 649043 under the EU-H2020 programme and the Czech Science Foundataion project 20-06390Y.

[1] Although "age" is the common term which we keep using in this paper, "date of birth" would be more appropriate since a child clause has a higher "age" than its parents (see also Definition 1 in Section 2).

(or values). An example of a feature explored in this work is the number of positive literals of a clause. According to this feature, we can split clauses into those having at most one positive literal, i.e. those belonging to the Horn fragment, and the remaining ones. Now, the main idea of layered selection is to "instantiate" the basic clause selection heuristic separately for each such group of clauses and alternate between selecting from each group according to a new "layer two" ratio. For example, the prover may decide to pick a Horn clause every four out of five selections. Note that clause selection according to age and weight is still happening according to the original ratio on "layer one", i.e. in each group separately.

In [6], we proposed a clause feature measuring the amount of theory reasoning in the derivation of a clause, and obtained a layered clause selection heuristic that dramatically improves the performance of the automated theorem prover VAMPIRE [8] on relevant benchmarks. In this paper, we 1) present the framework of layered clause selection in more detail (Section 2) including the ideas of

- defining the groups as either disjoint or monotone with respect to set inclusion, and
- the possibility of nesting multiple layers of selections based on different features.

2) In addition to the theory reasoning feature (recalled in Section 3), we introduce three additional features:

- the first being the number of split dependencies of a clause in the AVATAR architecture [22] for clause splitting (Section 4),
- the second tracking relatedness of a clause to the goal, derived from the computation of the SInE algorithm [7, 19] (Section 5), and
- the third being the already mentioned number of positive literals in a clause, essentially measuring closeness of a clause to the Horn fragment (Section 6).

Finally, 3) we present the results of extensive experiments with the new heuristics over TPTP and SMTLIB and a specific set of benchmarks coming from program verification (Section 7).

2 Layered clause selection using multi-split-queues

We assume the reader to be familiar with the main ideas behind saturation-based theorem proving [9, 1]. Details on given-clause saturation algorithms used most often in practice can be found in [14], a comprehensive description and evaluation of clause selection heuristics predating layered selection in [18].

2.1 The Age-weight-based heuristic

The de facto standard clause selection heuristic used in modern saturation-based theorem provers is the age-weight clause selection heuristic.

Definition 1 (Age-weight clause selection heuristic)**.** For any clause C, define the age $age(C)$ as the depth of the derivation tree of C[2] and define the weight $weight(C)$ as the number of symbols[3] of C. Let further $r_a : r_w$ be a list of two positive integer values. Then the *age-weight*

[2] This corresponds to how age is defined in VAMPIRE. More precisely still, one uses only the depth with respect to generating inferences. Reductions do not alter the age of a reduced clause.

[3] Including multiplicities. As a variation, different kinds of symbols (such as the variables, the predicate symbols, or the constants) may weigh more than others [18].

clause selection heuristic $aw(r_a : r_w)$ alternates between selecting a clause C with the smallest age $age(C)$ and selecting a clause C with the smallest weight $weight(C)$ using the ratio $r_a : r_w$.

The basic understanding of age-weight selection is that it performs a blend between the breadth-first and the best-first search paradigms. Clauses of small weight are considered better, because they are closer to the ultimate goal—the empty clause of weight zero—than the larger ones. They also tend to produce small clauses as children, on average serve as stronger simplifiers, and are computationally cheaper to process. The age criterion, on the other hand, corresponds to the breadth-first aspect and helps to ensure fairness under which no generated clause (unless shown redundant) waits too long before getting selected.

2.2 Split heuristics

Definition 2 (Split heuristics)**.** Let μ be a real-valued *clause evaluation feature* such that preferable clauses have low value of $\mu(C)$, and let the *cutoffs* $c_1, \dots, c_k$ be monotonically increasing real numbers with $c_k = \infty$. Furthermore, let the *ratio* $r_1 : \dots : r_k$ be a list of positive integer values, and let finally *cs* be an arbitrary clause selection heuristic.

A *split heuristic* groups clauses into sets $G_1, \dots, G_k$, and selects clauses by alternating selection from $G_1, \dots, G_k$ using the ratio $r_1 : \dots : r_k$. The selection from each such set G_i is performed using *cs*. We define two *modes* of split clause selection heuristic, which differ in how they group clauses:

- The *monotone split heuristic* $\textit{mono-split}(\mu, c_1, \dots, c_k, r_1 : \dots : r_k, cs)$ uses sets $G_i := \{C \mid \mu(C) \leq c_i\}$ for $i = 1, \dots, k$.
- The *disjoint split heuristic* $\textit{disj-split}(\mu, c_1, \dots, c_k, r_1 : \dots : r_k, cs)$ uses sets $G_1 := \{C \mid \mu(C) \leq c_1\}$, and $G_i := \{C \mid c_{i-1} < \mu(C) \leq c_i\}$ for $2 \leq i \leq k$.

Example 1. Consider the clause selection heuristic $\textit{mono-split}(\mu, 0, 1, \infty, 3 : 1 : 1, aw(1 : 1))$. This heuristic will select 3 out of 5 times a clause C such that $\mu(C) \leq 0$, 1 out of 5 times a clause C such that $\mu(C) \leq 1$, and 1 out of 5 times an arbitrary clause. On “layer one”, e.g., 3 out of 10 times the clause C with the smallest age among the clauses C with $\mu(C) \leq 0$ is selected, or 1 out of 10 times the clause with smallest weight out of all clauses is selected.

Split heuristics allow to adapt an existing clause selection heuristic *cs* to take into account the clause feature μ. We now discuss how to pick the mode, the cutoffs, and the ratios. We observed two kinds of clause features:

First, there are features where clauses with low feature value are more likely to contribute to the proof search (this is the case for the features $dist_{th}$, $dist_{SInE}$, and $dist_{Horn}$, discussed in Section 3, Section 5, resp. Section 6). For features of this kind, one should use a monotone split heuristic. A good starting point for cutoffs and ratios is c_1, ∞ and $1 : 1$, resp., where c_1 is the feature value we expect to obtain for the empty clause $\perp$ (based on domain knowledge and experience). One can extend the cutoffs by introducing one or two additional cutoffs close to c_1, in order to smooth the transition between c_1 and ∞, and extend the ratio accordingly. It can also make sense to vary the ratio, although in our experience, it is more important to identify good cutoffs, than to fine-tune the ratio.

Secondly, there are features where clauses with low feature value are not necessarily more likely to contribute to the proof search, but are less likely to have low weight (this is the case for the feature $dist_{AV}$ discussed in Section 4). As a consequence, it does not make sense to compare clauses, which have different feature values, by weight. In such a case, one can use a split heuristic in disjoint mode. Varying the cutoffs and ratios of disjoint split heuristics has

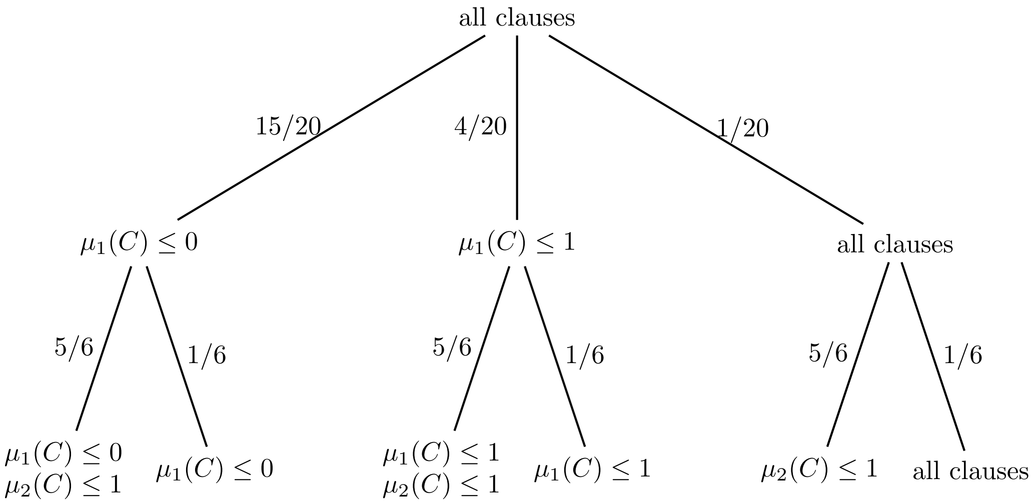

Figure 1: Demonstrating nested split heuristics.

a less predictable effect than for the monotone split heuristic and needs to be fine-tuned on a case-by-case basis.

2.3 Nesting split heuristics

As split heuristics are parameterized by an arbitrary clause selection heuristic, we are able to build clause selection heuristics containing nestings of split heuristics. Such clause selection heuristics are powerful, as they allow us to easily combine different features.

Example 2. Consider the nested split heuristic

$$\begin{aligned}&\textit{mono-split}(\mu_1, 0, 1, \infty, 15:4:1,\\&\quad \textit{mono-split}(\mu_2, 1, \infty, 5:1,\\&\qquad aw(1:1))).\end{aligned}$$

The resulting clause groupings and frequencies to pick from these groups are visualized in Figure 1. The nested split heuristics form a tree. Each leaf node of the tree represents a set of clauses, from which clauses are selected using $aw(1:1)$. We can see that the leftmost leaf node of the tree represents all clauses C with $\mu_1(C) \leq 0$ and $\mu_2(C) \leq 1$. We pick clauses from this leaf using $aw(1:1)$ in $15/20 * 5/6 = 5/8$ of the cases.

Note that each split heuristic h provides a horizontal dimension consisting of the groups of h. The nesting of different split heuristics itself provides a vertical dimension.

2.4 Implementation

In this subsection we briefly discuss how to implement clause selection heuristics. As typical runs of saturation algorithms can include several million clause selections, we strive to implement these heuristics efficiently.

An *aw*-heuristic with ratio $r_{age} : r_{weight}$ can be implemented as a container AW as follows. The container AW internally keeps two priority queues[4] Q_a, Q_w, where both Q_a and Q_w store all the clauses of AW, Q_a keeps its clauses ordered by *age* and Q_w keeps its clauses ordered by *weight*. The container AW determines whether it should select the next clause from Q_{age} or Q_{weight} using a weighted round-robin scheme with ratio $r_a : r_w$, and selection from the chosen queue proceeds by popping the first element (i.e. a clause) from that queue and deleting the corresponding record (of that clause) from the other queue.

The heuristic $\textit{mono-split}(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$, resp. $\textit{disj-split}(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$, can be implemented as a container SH as follows. Assume that cs is implemented using a container CS. The container SH keeps k instances $CS_1, \ldots, CS_k$ of CS, where container CS_i contains all clauses of group G_i of SH. The container SH determines from which of the sub-containers $CS_1, \ldots, CS_k$ it should select the next clause using a weighted round-robin scheme with ratio $r_1 : \cdots : r_k$ and then delegates clause selection to that CS_i.

2.5 Discussion

We believe that the nesting of split heuristics is a great conceptual tool for composing independent ideas on how to improve clause selection into a single compound heuristic. This can be already seen with the two layers, where the time-tested age-weight selection serves as a building block for more powerful refined heuristics, and can get further pronounced with additional nestings, as demonstrated by our experiments (see Section 7).

Nevertheless, in retrospect it is not hard to see that computationally, the layered scheme can be essentially “compiled down” to multiple level-one queues. More precisely, one needs an extension of the typical level-one queue arrangement, such as the one implemented in E [16], to allow clause queue *content filtering* by clause properties. This means that one needs to be able to set up a clause queue to only contain those clauses that satisfy a given property P. Such property P could be, e.g., $P(C) = \mu_1(C) \leq 0 \wedge \mu_2(C) \leq 1$ to define the age and weight level-one queues corresponding to the left-most leaf in Figure 1. The reason why clause queue content filtering has until now not been used (to the best of our knowledge) in saturation-based provers is probably that in the standard perspective each clause queue is meant to provide an independent view of the whole set of passive clauses and not just a subset thereof (which would complicate reasoning about completeness if left further unconstrained).[5]

3 Feature: Amount of theory reasoning

Recently, saturation-based theorem provers have increasingly been used to reason about problems requiring quantified theory reasoning [4, 2]. The standard solution to provide a prover with support for reasoning in a given theory is to extend the input axioms of the problem with an explicit axiomatization of the corresponding theory. There are two related problems caused by this approach: First, the theory axioms generate a huge number of consequences, as the theory axioms are repeatedly combined either with themselves or with other axioms, and therefore blow up the search space. Secondly, many of these generated consequences have small weight. If a standard age-weight-based heuristic is used for clause selection, those consequences are therefore often selected, as selection by weight will favor them. While manually inspecting proofs for problems of the application domains [4, 2], we observed that the amount of theory

[4] In Vampire, these priority queues are implemented as skip lists.

[5] We note that clause *priority functions* of E [17] allow the user to order clauses on a particular queue such that those clauses satisfying a given property P are all considered smaller than those that satisfy $\neg P$.

reasoning actually required to prove these problems is small. As a result, the prover spends most of its proof search in a part of the search space, where the chances to find a clause relevant for the proof are low. We are therefore facing the challenge of guiding the proof search, so that the prover does not spend too much time with theory reasoning, but at the same time still finds proofs containing a small amount of theory reasoning.

In the remainder of this section, we give an extended presentation of the solution to this challenge already presented in [6]. In a nutshell, our solution consists of a clause feature dist_{th}, which measures the amount of theory reasoning in the derivation of a clause, and a corresponding clause selection heuristic based on split heuristic and dist_{th}. We assume that the input problem is given as a set of axioms, where the axioms corresponding to the axiomatization of the theory are distinguished.

We start by formalizing the amount of theory reasoning in the derivation of a clause C as the ratio of the number of theory axioms and the number of all axioms in the derivation-DAG of C. Computing these numbers exactly for the derivation of each clause is potentially expensive, since it requires for each clause a traversal of the derivation-DAG of the clause. We instead approximate those numbers by treating the derivation-DAG as a tree, for which we can compute the numbers using running sums, as follows:

Definition 3. For a theory axiom C, define both $\mathit{thAx}(C)$ and $\mathit{allAx}(C)$ as 1. For a non-theory axiom C, define $\mathit{thAx}(C)$ as 0 and $\mathit{allAx}(C)$ as 1. For a derived clause C with parent clauses $C_1, \ldots, C_n$, define $\mathit{thAx}(C)$ as $\sum_i \mathit{thAx}(C_i)$ and $\mathit{allAx}(C)$ as $\sum_i \mathit{allAx}(C_i)$. Finally, we set $\mathit{frac}(C) := \mathit{thAx}(C)/\mathit{allAx}(C)$.

With these notations at hand, we identify proofs that only need a small amount of theory reasoning with the proofs where $\mathit{frac}(\bot)$ is at most $1/d$, for some small positive integer value d ($\bot$ here denotes the empty clause).

Next, we present a clause feature dist^d_{th} which approximates the likeliness that a given clause C occurs in a proof where $\mathit{frac}(\bot)$ is at most $1/d$. The clause selection feature dist^d_{th} is parameterized by the value d and measures the number of non-theory axioms which the derivation of C would need to contain additionally in order to achieve a ratio of at most $1 : d$.

Definition 4. Let d be a positive integer value. Then $\mathit{dist}^d_{th} : \mathit{Clauses} \to \mathbb{N}$ is defined as

$$\mathit{dist}^d_{th}(C) := \mathit{max}(\mathit{thAx}(C) \cdot d - \mathit{allAx}(C), 0).$$

The feature dist^d_{th} satisfies several properties, which we think are favorable: (i) if the derivation of a clause C consists only of several axioms, then $\mathit{dist}^d_{th}(C)$ is small, (ii) if derivations of clauses C_1, C_2 are combined into a derivation of clause C, and if both $\mathit{dist}^d_{th}(C_1) > 0$ and $\mathit{dist}^d_{th}(C_2) > 0$, then $\mathit{dist}^d_{th}(C) > \mathit{dist}^d_{th}(C_1)$ and $\mathit{dist}^d_{th}(C) > \mathit{dist}^d_{th}(C_2)$, and (iii) if derivations of clauses C_1, C_2 are combined into a derivation of clause C, and if $\mathit{frac}(C_1) = 1/d$, then $\mathit{dist}^d_{th}(C) = \mathit{dist}^d_{th}(C_2)$. Note that frac itself does not fulfill these properties.

Example 3. Consider a clause C_1, such that $\mathit{thAx}(C_1) = 3$ and $\mathit{allAx}(C_1) = 5$. Consider further a clause C_2, such that $\mathit{thAx}(C_2) = 100$ and $\mathit{allAx}(C_2) = 200$. Intuitively, C_1 is much more likely to occur in a proof with $\mathit{frac}(\bot) = 1/4$. We have $\mathit{dist}^4_{th}(C_1) = 7 < 200 = \mathit{dist}^4_{th}(C_2)$, but $\mathit{frac}(C_1) = 0.6 > 0.5 = \mathit{frac}(C_2)$.

Finally, we are able to construct a clause selection heuristic, which addresses the challenges presented at the beginning of this section, using the split heuristic from Section 2.2 as

$$\mathit{mono\text{-}split}(\mathit{dist}^d_{th}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

where d is the positive integer such that $1/d$ is the expected fraction of the proof which we want to find, μ is some clause selection strategy, $c_1, \ldots, c_{k-1}, \infty$ are cutoff values, and $r_1 : \cdots : r_k$ is a ratio. In our experience, varying d has a bigger effect than varying cutoffs and ratios, and setting d to 8 is a reasonable starting point for fine-tuning d. In our experiments, other useful values of d were between 4 and 50.

4 Feature: AVATAR-splits

AVATAR [22, 12, 10] is a theorem prover architecture in which a saturation algorithm is augmented with a SAT (or an SMT) solver to facilitate an efficient version of clause splitting [23, 13]. In a nutshell, a first-order clause C is called *splittable* if it can be written as $C = C_1 \vee \ldots \vee C_k$, $k > 1$, such that the individual *components* C_i are pairwise variable-disjoint. The main idea behind splitting is that one can reason about the individual components separately, since for every set of clauses N and every such splittable clause C, $N \cup \{C\}$ is unsatisfiable if and only if $N \cup \{C_i\}$ is unsatisfiable for every $i = 1, \ldots, k$. This is advantageous, as the individual components C_i are smaller than the original clause C and thus promise a strictly faster search.

While the exact details of how AVATAR works are out of the scope of this paper, the key aspect important here is easy to explain. First-order clauses in AVATAR need to keep track of the dependencies on splits from which they were derived. This is done by assigning to each clause C a set of dependencies D_C, denoted $C \leftarrow D_C$, where a dependency $d \in D_C$ is some identifier of a performed split registered elsewhere in the architecture. Clauses from the input have their dependency set initialised as empty and the dependency set of a derived clause is computed as the union of the dependencies of its parents. Then, when a clause such as $C_1 \vee C_2 \leftarrow D_C$ is split, with C_1 and C_2 variable-disjoint, the prover may continue reasoning with $C_1 \leftarrow D_C \cup \{[C_1]\}$ where $[C_1]$ is the identifier of the dependency on the performed split. Intuitively, each dependency $d \in D_C$ is a choice point for which the prover might need to consider alternatives in the future. This means that a clause with many dependencies corresponds to a logically weaker fact than a clause with fewer ones.

Because the basic setup of AVATAR is oblivious to the size of the dependency set of a clause, there is a danger of a strong preference for clauses of small weight (which arise easily with splitting) that nevertheless depend on many splits and are therefore not the best for closing the overall search fast. In order to potentially mitigate this effect, we propose here to use the size of the dependency set of a clause, $dist_{AV}(C \leftarrow D_C) = |D_C|$, as a feature for split heuristics.

We then construct a clause selection heuristic using the split heuristic from Section 2 as

$$\textit{disj-split}(dist_{AV}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$.

5 Feature: SInE-levels of a Clause

The Sumo Inference Engine (SInE) [7] is a well-established algorithm for *selecting premises* for first-order theorem proving, i.e. for the task of reducing—before the start of the search—the possibly large set of input axioms to a more manageable subset of those ones estimated to be most promising for proving a given conjecture. SInE is an iterative algorithm which takes the conjecture (also called the *goal*) and iteratively adds axioms that appear to be most related to the goal or to previously added axioms by a similarity metric based on sharing symbols. We define, for every input axiom A, a *heuristical distance* $dist_{SInE}(A)$ *from the goal* G as the iteration number i at which A would be added by SInE to the included axioms for proving G. By definition, $dist_{SInE}(G) = 0$ for the goal itself and typically ranges between 1 up to approximately 10 for the non-goal input axioms [19]. We will informally refer to the value $dist_{SInE}(F)$ for a particular formula F as its SInE-level.

So far, we defined SInE-levels only for the input axioms and the goal. To be able to use them as a feature of general clauses in proof search, we further define $dist_{SInE}(C)$ of a derived clause as the *minimum* of $dist_{SInE}(P_i)$ over the parents P_i of C. While the choice of the minimum operation may appear arbitrary, note that it has the nice property that $dist_{SInE}(C) = 0$ if and only if C has a goal among its ancestors. This is an important "flag" of a clause, typically tracked by a theorem prover for use in various goal-directed heuristics, and SInE-levels therefore naturally generalise this flag.

Finally, we derive a clause selection heuristic using the split heuristic from Section 2 as

$$\textit{mono-split}(dist_{SInE}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$. For instance, we can use cutoffs $0, \infty$ and ratio $1 : r_2$ to ensure that from $r_2 + 1$ clauses at least one clause is selected which has a goal among its ancestors.

6 Feature: Positive literals

Saturation-based theorem provers are known to work well on benchmarks where each axiom is a Horn clause, that is, a clause with at most one positive literal. We would like to extend the efficiency of these provers to problems which are nearly Horn, in the sense that there exists a proof of the conjecture of the problem, where the number of positive literals for each clause is small. We can formalize this as follows: The *Horn-distance* $dist_{Horn}(C)$ is defined as

$$dist_{Horn}(C) := max(posLits(C) - 1, 0),$$

where $posLits(C)$ denotes the number of positive literals of C. For a given proof P define

$$dist_{Horn}(P) := \sum_{C \text{ clause in } P} dist_{Horn}(C).$$

We claim that for many application domains, most examples are provable using a proof with a small Horn-distance. But if we run a saturation-based theorem prover on such a problem, it can still generate a lot of consequences which contain several positive literals. Such consequences would typically be classified as highly unlikely to contribute to the refutation by human inspection.

We are able to guide clause selection towards finding proofs with small Horn-distance by instantiating the split heuristic from Section 2 as

$$\textit{disj-split}(dist_{Horn}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$.

7 Experiments

We implemented the heuristics described in Sections 3–6 in the state-of-the-art theorem prover Vampire (version 4.4) [8]. Our implementation consists of about 1000 lines of C++ code and will become integrated in the next release of the prover.

Table 1: Default parameters for the heuristics presented in the first experiment.

tag	feature	d-value	cutoffs	ratio	mode of split
`th`	$dist_{th}^{d}$	8	$(0, 32, 80, \infty)$	20:10:10:1	monotone
`av`	$dist_{AV}$	–	$(0, \infty)$	1:1	disjoint
`sl`	$dist_{SInE}$	–	$(0, 1, \infty)$	1:2:3	monotone
`pl`	$dist_{Horn}$	–	$(0, \infty)$	1:4	disjoint

We evaluated the extended implementation of Vampire on two sets of problems coming from the TPTP library [20] and from SMTLIB [3], respectively. In detail, we selected all the first-order problems of the form CNF, FOF, and TF0 (including those with arithmetic) from TPTP version 7.3.0. This gave us 18 294 problems. Additionally, we picked a subset of a recent version (release 2019-05-06) of SMTLIB consisting of all the problems from the sub-logics that contain quantification and theories, such as ALIA, LRA, NRA, UFDT, . . . , except for those requiring bit-vector (BV) or floating-point (FP) reasoning, currently not supported by Vampire. For this SMTLIB benchmark we obtained 68 234 problems.

Our experiments were run on our local server with two Intel Xeon Gold 6140 Processors (i.e., with 72 processor threads) and 188GB RAM. We were running 30 instances of Vampire in parallel with no other significant load on the server. To obtain a baseline strategy, denoted as `base`, we modified the default Vampire strategy (which uses Avatar) to use the Discount saturation loop (for stability of results[6]) and the clause selection heuristic $aw(1 : 10)$ (which in our experience leads in Vampire to a good performance with Discount). All other tested strategies extend `base` by applying one or more split heuristics for clause selection on top of this setup. With the exception of Experiment 4 we used a time limit of 10 s per problem.[7]

7.1 Experiment 1: testing the initial defaults

Searching for good values of the cutoffs, the ratio and other parameters of split heuristics is rewarding, but requires some experience and a certain amount of experimental "tuning". In our previous work on layered clause selection [6], we gained some of such experience for the feature measuring the amount of theory reasoning (Section 3) and later—also with the help of an experiment reported further below—picked certain default values for parameters of the heuristics corresponding to the other features. We used these defaults (presented in Table 1) in the first experiment, the purpose of which is to demonstrate the basic improvement we obtain from using the presented heuristics and their combinations.

Table 1 assigns a *tag*, typeset in the `typewriter` font, to each of our four heuristics when understood as Vampire options used for defining a strategy. Thanks to the possibility of nesting split heuristics, these options can be turned on and off independently from one another. Although a particular fixed order is employed in Vampire to build up the nestings (namely the order, from inside out: `th`, `av`, `sl`, `pl`), we use the operator `+` to denote possible combinations to suggest that this order is actually irrelevant for the proof search (cf. Section 2.5).

The results of the first experiment are shown in Table 2, separately for TPTP and for SMTLIB. We observe that in the case of TPTP all the four new heuristics lead to an improvement in the number of solved problems. This is easiest to see from the always positive column

[6] The default Limited Resource Strategy [14] is sensitive to timing measurements and repeated runs on the same benchmark under essentially the same conditions may vary a lot.

[7] A list of the selected problems along with other information needed to reproduce our experiments can be found at https://github.com/quickbeam123/LCS4SbTP-materials.

Table 2: Results of Experiment 1: on the TPTP benchmark (left) and the SMTLIB one (right).

strategy	solved	Δbase	uniques
`base`	9108	0	9
`th`	9204	96	22
`av`	9160	52	89
`sl`	9525	417	109
`pl`	9289	181	42
`th+av+sl+pl`	9526	418	147
union	10 297	1189	

strategy	solved	Δbase	uniques
`base`	39 943	0	45
`th`	41 841	1898	321
`av`	39 906	−37	116
—	—	—	—
`pl`	40 442	499	95
`th+av+pl`	40 952	1009	287
union	43 169	3226	

Δbase, which shows the difference of the number of problems solved between the current strategy and `base`. The same success is not fully repeated on SMTLIB where `th` shows a great improvement, but `av` performs worse than `base`. (Note that we did not run `sl` on SMTLIB, since the format used in the library does not support specifying the goal.[8])

It should be pointed out that even a strategy which does not improve over `base` in terms of the number of solved problems could be valuable for the potential participation in strategy schedules, because of the problems it solves uniquely (as reported in the last column in the tables). For another view of this effect, the last line in the two tables shows the number of problems solved by at least one of the listed strategies, again also compared against `base`. We can see that the use of split heuristics allows us to solve almost 1200 TPTP problems (and more than 3200 SMTLIB problems) not solved by `base`.

7.2 Experiment 2: nesting of the heuristics

When looking in Table 2 at the performance of the combination of the four heuristics (strategy `th+av+sl+pl`) one can ask why it does not get better at achieving a combined benefit of its constituents. In Experiment 2, we look at this trend closer and especially try to estimate how much time is typically spent on computing the clause selection heuristic and how this depends on the number of heuristics combined.

The report in Table 3 is based on TPTP runs of all the 16 possible strategies which combine between zero to four of the heuristics introduced in this paper. The middle part of the table reports on their performance in terms of the number of solved problems and is comparable to (in fact, a super-set of) the results in Table 2 (left). One can notice here that combinations indeed sometimes do not outcompete their constituents. E.g., `*+av+pl` is always worse than just `*+pl`, which could indicate some unfavourable interactions of the two heuristics. (We leave a more detailed study of this phenomenon for future work.)

Our main focus in this experiment, however, is on the right part of the table. There, we took the runs on those problems which none of the strategies could solve[9] (i.e., on which they ran for the full 10 s) and measured how much time was spent (on average) on interacting with the passive clause container (this includes insertions, deletions and the popping of the selected clauses). This average time is reported in the last column, from which we can see that, indeed, the more complex combined strategies are more expensive to execute.

Additionally, for comparison, the #queues column in the table reminds us how many "layer one" clause queues each strategy maintains (recall Section 2.3 and the number of "horizontal"

[8] There is, however, interesting work on "guessing the goal" for SMTLIB problems [11], which we plan to experiment with in the future.

[9] There were 7808 such problems.

Table 3: Combined strategies run on TPTP: number of solved problems and average time spent (on the commonly unsolved problems) maintaining the passive clause container.

strategy	solved	Δbase	#queues	avg. time on unsolved
`base`	9108	0	2	0.32 s
`pl`	9289	181	2(*2)	0.32 s
`av+pl`	9179	71	2(*2)(*2)	0.32 s
`av`	9160	52	2(*2)	0.33 s
`sl`	9525	417	2*3	0.58 s
`sl+pl`	9594	486	2*3(*2)	0.58 s
`sl+av+pl`	9511	403	2*3(*2)(*2)	0.58 s
`sl+av`	9560	452	2*3(*2)	0.59 s
`th+av+pl`	9181	73	2*4(*2)(*2)	1.04 s
`th+pl`	9338	230	2*4(*2)	1.06 s
`th`	9204	96	2*4	1.06 s
`th+av`	9234	126	2*4(*2)	1.08 s
`th+av+sl+pl`	9526	418	2*4(*2)*3(*2)	1.75 s
`th+sl+pl`	9601	493	2*4*3(*2)	1.76 s
`th+av+sl`	9584	476	2*4(*2)*3	1.79 s
`th+sl`	9557	449	2*4*3	1.80 s

splits each heuristic uses, as shown in Table 1). The multiplier (*2) corresponding to `av` and `pl` is rendered in brackets, because these two heuristics use the disjoint split mode. This means that when deciding on $dist_{AV}$ (or $dist_{Horn}$), each clause is strictly inserted only into one of two possible sub-containers (rather than possibly to more than one, as with the monotone mode). Correspondingly, the strategies are clearly separated into four groups in terms of average interaction time, where the monotone splits of `sl` and `th` are the costly ones (and maintaining the 4 queues of `th` costs more than the 3 queues of `sl`) whereas the disjoint split of the other two heuristics does not seem to be adding any measurable overhead.

We remark that the reported average times are not directly proportional to speed of clause processing as each run was terminated after 10 s no matter how many selections were performed. Moreover, quite different search spaces could have been traversed by each of the strategies.

7.3 Experiment 3: parameter tunings

In this section, we shed some light on how the performance of our heuristics varies under the two possible modes of split and under changing the ratios. We focus here on the SInE-levels, AVATAR-splits, and the number of positive literals, referring the reader to [6] for more information on the behaviour of the theory reasoning heuristic. Fortunately, for these three features, we always have a canonical value for the main cutoff to try first:

- as explained in Section 5, clauses of $dist_{SInE}(C) \leq 0$ are exactly those derived with the help of the goal,
- clauses with $dist_{AV}(C) \leq 0$ do not depend on any AVATAR splits, and
- clauses with $dist_{Horn}(C) \leq 1$ are exactly the Horn clauses.

With the last feature, we also try out the "obvious" cutoff value 0.

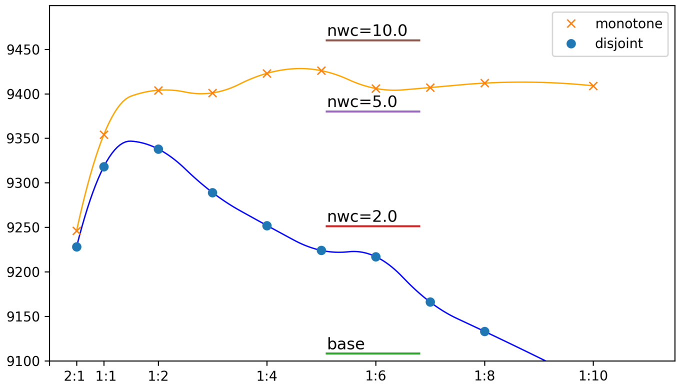

Figure 2: The number of TPTP problems solved when splitting on $dist_{SInE}$ with cutoffs $(0, \infty)$, and varying the ratio. Quadratic interpolation used to connect the obtained measurements.

Although we do not spend much effort on exploring the multi-value "horizontal" splits (since the risk of overfitting to the benchmark increases, and the parameter space becomes both too large to sample efficiently and hard to visualise), we explain how we discovered the successful default for `sl` with cutoffs $(0, 1, \infty)$ and suggest a multi-value cutoff setting for `pl`.

Tuning SInE-levels Figure 2 shows the result of varying the ratio while splitting on $dist_{SInE}$ with the cutoffs $(0, \infty)$. We can observe that with $dist_{SInE}$, the monotone mode is clearly more successful than the disjoint one.

For a comparison, the figure also includes marks for the `base` strategy and the base strategy enhanced by setting the *non-goal weight coefficient* (*nwc*) to values 2.0, 5.0, and 10.0. Changing the non-goal weight coefficient is a different way of making clause selection goal oriented which relies on multiplying the weight of each clause *not* derived from the goal by the given *nwc*, making it artificially larger and thus less likely to be selected [19]. We can see that the "$nwc =$ 10.0" strategy still beats the best possible ratio for the SInE level split queue setup here.

The method we used to further improve the SInE level split heuristic is as follows. We took the best ratio 1 : 5 for the monotone split as shown in Figure 2. Let us recall that this strategy tries to select once out of every 6 selections a clause C with $dist_{SInE}(C) \leq 0$, the remaining 5 selections being unconstrained by $dist_{SInE}$. After adding a second cutoff (of value 1), we can speculate how to split these 5 selections into those with $dist_{SInE}(C) \leq 1$ and the remaining, again unconstrained ones. This led us to an experiment with the fixed cutoffs $(0, 1, \infty)$ and the varied ratios 1:4:1, 1:3:2, 1:2:3, and 1:1:4, and with the resulting number of problems solved: 9112 (+4 over `base`), 9410 (+302), 9512 (+404), and 9501 (+393), respectively. Based on this experiment we selected the cutoffs $(0, 1, \infty)$ and ratio 1:2:3 as the default for `sl`.[10]

Tuning AVATAR-splits Figure 3 visualizes the changes in performance when varying the ratios for the AVATAR-based split heuristic with the $dist_{AV}(C) \leq 0$ cutoff and compares them

[10] Table 2 reports a slightly different number of solved problems for the default `sl`. The variation, i.e., the 9512 vs 9525 solved problems, is an artefact of rerunning the experiment (in an inherently non-deterministic environment) whose magnitude should be taken into account when interpreting the results in this whole section.

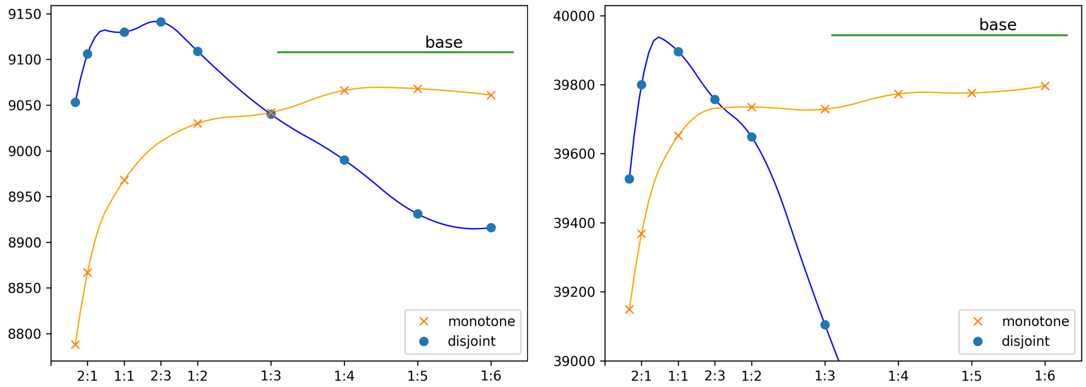

Figure 3: The number of TPTP (left) and SMTLIB (right) problems solved when splitting on $dist_{AV}$ with cutoffs $(0, \infty)$ and varying the ratio. Quadratic interpolation applied as previously.

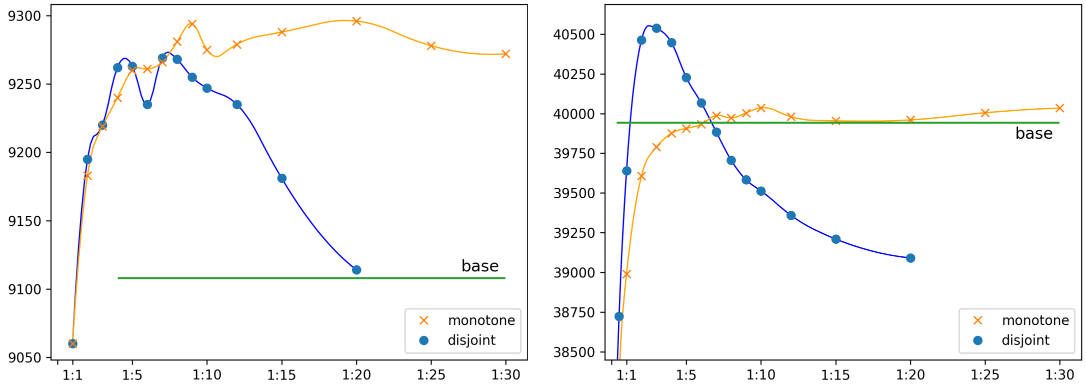

Figure 4: The number of TPTP (left) and SMTLIB (right) problems solved when splitting on $dist_{Horn}$ with cutoffs $(0, \infty)$ and varying the ratio.

to `base`. We can see that the trends are similar for TPTP and SMTLIB and that here the highest values are reached for the disjoint mode of split. Although the best ratio on TPTP is 2:3, we chose the ratio 1:1 for the default in `av` given its much better performance on SMTLIB.

Tuning the positive literals feature The situation with the positive literals feature is a bit harder to understand. Let us first have a look at the $dist_{Horn}(C) \leq 0$ cutoff, i.e. the perspective in which the "good" clauses are the clauses with no positive literals.

The performance development for this setup is visualised in Figure 4, again both for TPTP (left) and SMTLIB (right). Here, the two behaviours are quite different. Most notably, the disjoint mode, which we selected for the default of `pl`, is only dominant for SMTLIB, whereas for TPTP, better values can be achieved with the monotone mode. Moreover, the monotone

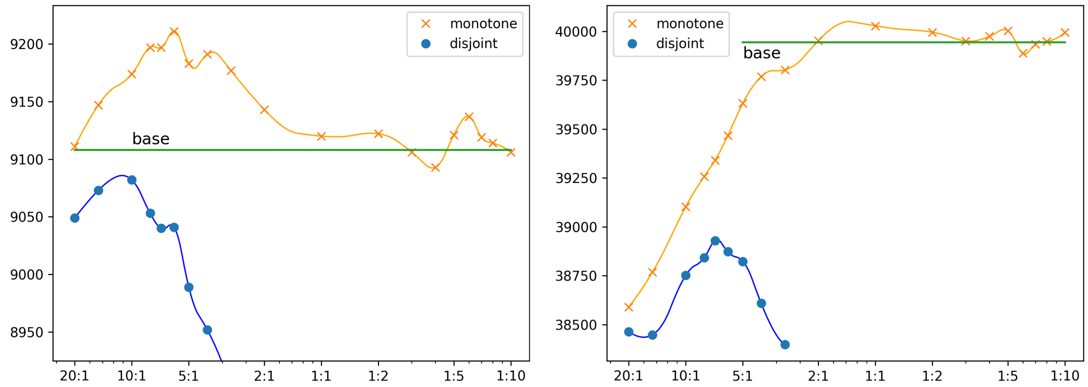

Figure 5: The number of TPTP (left) and SMTLIB (right) problems solved when splitting on $dist_{Horn}$ with cutoffs $(1, \infty)$ and varying the ratio.

mode on TPTP remains to be very successful (as compared to `base`) for up to high values of the ratio. In contrast, on SMTLIB the performance of the monotone mode is close to that of `base` from the ratio 1:5 onwards (and reaches it "from below" with the smaller values). Note that for the monotone mode, increasing x in a ratio 1:x effectively converges to turning the heuristic off. This makes the observation that this mode is still very successful on TPTP for the ratio 1:30 even more surprising. We currently do not have a good explanation for this phenomenon.

Let us with Figure 5 move on to the $dist_{Horn}(C) \leq 1$ cutoff, under which we recognise as "good" those clauses that have at most one positive literal. This "Horn fragment" perspective is both for TPTP and for SMTLIB dominated by the monotone mode of split. When looking at the actual number of problems solved, however, we observe that on neither of the two benchmarks is the improvement over `base` as pronounced as for the $dist_{Horn}(C) \leq 0$ cutoff. The best measured improvement for SMTLIB is +78 problems over `base` achieved for the ratio 1:1. On TPTP, we gain +103 problems over `base` with the ratio 6:1.

We used the same strategy for extending a good ratio from one non-trivial cutoff to two as with the SInE-levels. Focusing on the TPTP benchmark, which seems to be more "responsive" to the heuristic based on positive literals, we started from the successful ratio 1:20 from the $dist_{Horn}(C) \leq 0$ cutoff with the monotone mode of split and tested variations in which the 20 selections are distributed between a certain amount of those with $dist_{Horn}(C) \leq 1$ and the remaining, unconstrained ones. This way, we tested strategies with the cutoff $(0, 1, \infty)$ and ratios 1:15:5, 1:16:4, 1:17:3, 1:18:2, and 1:19:1, obtaining, respectively, 9365, 9383, 9368, 9368, 9277 problems solved. The best of these, 1:16:4, improves over the 1:20-ratio $(0, \infty)$-cutoff monotone-mode strategy by additional 87 problems. For comparison, however, this new strategy does not fare very well on SMTLIB, where it scores 181 fewer problems than `base`. Our tentative conclusion from tuning the positive literals feature is that the ensuing "Horn fragment" perspective is not very useful for improving the performance on SMTLIB.

7.4 Experiment 4: software verification benchmarks

A recent stream of work (started in [4]) formalizes the correctness of functional properties about programs containing loops and arrays as validity problems in quantified first-order logic modulo

Table 4: Results of Experiment 4.

strategy	refuted
`aw`	18
`split-heuristics`	78

integers and difference logic over natural numbers. It then uses superposition-based theorem proving to reason about the resulting encodings.

These encodings are extremely challenging, since they include quantifier alternations and quantified theory reasoning over both integers and difference logic and require proofs of non-trivial size. We conjecture that the key ingredient to reason in this domain efficiently is to equip the prover with domain-specific knowledge: From experience we know that proofs in this domain require only light-weight theory reasoning and light-weight reasoning with case distinctions. We furthermore know that it pays off to explore consequences related to the conjecture. Our work measures the amount of theory reasoning in a derivation of a clause using $dist_{th}$, measures the amount of case-distinctions in a derivation of a clause using $dist_{Horn}$, and keeps track of whether the conjecture occurred in the derivation of a clause using $dist_{SInE}$. We therefore are able to use nested split-heuristics with those features to guide proof search on these examples.

As a third experiment, we investigated the effect of guiding proof search using nested split heuristics with features $dist^8_{th}$, $dist_{Horn}$, and $dist_{SInE}$ on 103 benchmarks obtained from unpublished work extending and improving [4]. We adapted the default configuration of Vampire to a base configuration, by (i) turning off Avatar, (ii) turning on additional simplification rules, including backward subsumption, backward subsumption resolution, and forward- and backward subsumption demodulation [5] and (iii) additional smaller changes. Starting from this base configuration, we compared two versions `aw` and `split-heuristics`. The version `aw` combines the base configuration with Vampire's default clause selection heuristic $aw(1:1)$. The version `split-heuristics` uses a portfolio of configurations, where each configuration uses the clause selection heuristic

$$\begin{aligned}&\mathit{mono\text{-}split}(dist^8_{th}, \_, \_\\ &\quad \mathit{mono\text{-}split}(dist_{Horn}, \_, \_\\ &\qquad \mathit{mono\text{-}split}(dist_{SInE}, \_, \_\\ &\qquad\quad aw(1:1)))),\end{aligned}$$

consisting of three nested monotone split heuristics with features $dist^8_{th}$, $dist_{Horn}$ and $dist_{SInE}$, where the cutoffs and ratios of the split heuristics are varied in the portfolio. For each configuration, we imposed a timeout of 60 s. The results of the experiment are listed in Table 4. While Vampire was only able to prove 18 out of 103 examples with `aw`, it was able to prove 78 out of 103 examples while using `split-heuristics`. These results are significant, and suggest that adding domain-specific knowledge using clause selection heuristics based on the split heuristic is a key ingredient to efficient automation of the challenging software verification benchmarks obtained from extensions of [4].

8 Conclusion

We investigated the framework of layered clause selection and split heuristics for clause selection and generalized it to support arbitrary nestings and both disjoint and monotone groups. We then revisited the existing feature $dist_{th}$ and introduced three new features $dist_{AV}$, $dist_{SInE}$, and $dist_{Horn}$. Finally, we instantiated the framework of split heuristics with these four features and presented a thorough experimental evaluation. Our results suggest that split heuristics heavily improve the performance for both general domains as well as for a specific application to software verification.

Acknowledgements

We thank the anonymous reviewers for their useful comments and suggestions. We also thank Sibylle Ortner for a careful proofreading of a preliminary version of this paper.

References

[1] Leo Bachmair, Harald Ganzinger, David A. McAllester, and Christopher Lynch. Resolution theorem proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. 2001.

[2] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. Reachability analysis for aws-based networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 231–241. Springer, 2019.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In *Formal Methods in Computer Aided Design 2019 (FMCAD 2019)*, pages 170–178, 2019.

[5] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In *10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, 2020. To appear.

[6] Bernhard Gleiss and Martin Suda. Layered clause selection for theory reasoning (short paper). In *10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, 2020. To appear.

[7] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2011.

[8] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[9] Ross A. Overbeek. A new class of automated theorem-proving algorithms. *J. ACM*, 21(2):191–200, 1974.

[10] Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *2nd Global Conference on Artificial Intelligence (GCAI 2016)*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.

[11] Giles Reger and Martin Riener. What is the point of an SMT-LIB problem? In *16th International Workshop on Satisfiability Modulo Theories (SMT 2018)*, 2018.

[12] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In Amy P. Felty and Aart Middeldorp, editors, *25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *Lecture Notes in Computer Science*, pages 399–415. Springer, 2015.

[13] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. In Bernhard Nebel, editor, *17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 611–617. Morgan Kaufmann, 2001.

[14] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.

[15] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[16] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[17] Stephan Schulz. E 2.4 User Manual. http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.4/eprover.pdf (accessed January 2020), 2019.

[18] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *8th International Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 9706 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2016.

[19] Martin Suda. Aiming for the goal with SInE. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, volume 71 of *EPiC Series in Computing*, pages 38–44. EasyChair, 2020.

[20] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[21] Tanel Tammet. GKC: A reasoning system for large knowledge bases. In Pascal Fontaine, editor, *27th International Conference on Automated Deduction (CADE 2019)*, volume 11716 of *Lecture Notes in Computer Science*, pages 538–549. Springer, 2019.

[22] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification (CAV 2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.

[23] Christoph Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [15], pages 1965–2013.