The 481 Ways to Split a Clause and Deal with Propositional Variables

Kryštof Hoder and Andrei Voronkov

University of Manchester, Manchester, UK

Abstract. It is often the case that first-order problems contain propositional variables and that proof-search generates many clauses that can be split into components with disjoint sets of variables. This is especially true for problems coming from some applications, where many ground literals occur in the problems and even more are generated.

The problem of dealing with such clauses has so far been addressed using either splitting with backtracking (as in Spass [14]) or splitting without backtracking (as in Vampire [7]). However, the only extensive experiments described in the literature [6] show that on the average using splitting solves fewer problems, yet there are some problems that can be solved only using splitting.

We tried to identify essential issues contributing to efficiency in dealing with splitting in resolution theorem provers and enhanced the theorem prover Vampire with new options, algorithms and datastructures dealing with splitting. This paper describes these options, algorithms and datastructures and analyses their performance in extensive experiments carried out over the TPTP library [12]. Another contribution of this paper is a calculus RePro separating propositional reasoning from first-order reasoning.

1 Introduction

In first-order theorem proving, theorem provers based on variants of resolution and superposition calculi (in the sequel simply called *resolution theorem provers*) are predominant. This is confirmed by the results of the last CASC competitions.[1] The top three theorem provers Vampire [7], E-MaLeS and E [9] are resolution-based, while the fourth one iProver [4] implements both an instance-based calculus and resolution with superposition.

Resolution theorem provers use *saturation algorithms*. They deal with a search space consisting of clauses. Inferences performed by saturation algoritms are of three different kinds:

1. *Generating inferences* derive a new clause from clauses in the search space. This new clause can then be immediately simplified and/or deleted by other kinds of inference.
2. *Simplifying inferences* replace a clause by another clause that is simpler in some strict sense.
3. *Deletion inferences* delete clauses from the search space.

[1] `http://www.cs.miami.edu/~tptp/CASC/J6/`

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 450–464, 2013.

On hard problems the search space is often growing rapidly, and simplifications and deletions consume considerable time. Performance of resolution theorem provers degrades especially fast when they generate many clauses having more than one literal (*multi-literal clauses* for short) and heavy clauses (clauses of large sizes). There are several reasons for this degradation of performance:

1. The complexity of algorithms implementing inference rules depends on the size of clauses. The extreme case are algorithms for subsumption and subsumption resolutions. These problems are known to be NP-complete and algorithms implementing them are exponential in the number of literals in clauses.
2. Storing heavy clauses requires more memory. Moreover, every literal in a clause (and sometimes every term occurring in such a literal) are normally added to one or more indexes. Index maintenance requires considerable space and time and operations on these indexes slow down significantly when the indexes become large.
3. Generating inferences applied to heavy clauses usually generate heavy clauses. Generating inferences applied to clauses with many literals usually generate clauses with many literals. For example, resolution applied to two clauses containing n_1 and n_2 literals normally gives a clause with $n_1 + n_2 - 2$ literals.

To deal with multi-literal and heavy clauses, one can simply start discarding them after some time, thus losing completeness [8]. Alternatively, one can use *splitting*. There are two kinds of splitting described in the literature: splitting with backtracking (as in Spass [14]) or splitting without backtracking (as in Vampire [7]).

2 Propositional Variables in Resolution Provers

Both kinds of splitting are implemented using introduction of propositional variables denoting components of split clauses. When many such variables are introduced, they give rise to clauses with many propositional literals. Such clauses clog up search space and slow down expensive operations, such as subsumption. Therefore, the problem of dealing with propositional literals is closely related to splitting. Apart from variables arising from splitting, propositional variables are common in many applications, for example, program analysis. They may also be introduced during preprocessing when naming is used to generate small clausal normal forms.

The resolution and superposition calculus is very efficient for proving theorems in first-order logic. In propositional logic, it is not competitive to DPLL. Suppose that we have a problem that uses both propositional and non-propositional atoms. Then treating propositional atoms in the same way as non-propositional ones results in performance problems. For example, if we use the code trees technique for implementing subsumption [13] and make no special treatment for propositional variables, it will work in the worst case in exponential time even for a pair of propositional clauses, while the best algorithms for propositional subsumption are linear.

The Calculus RePro To address the problem of dealing with propositional variables, in this section we will introduce a *calculus RePro* for dealing with clauses having propositional literals, and will illustrate some options of Vampire using this calculus. The calculus separates propositional reasoning from non-propositional.

Let us call a *pro-clause* any expression of the form $C \mid P$, where C is a clause containing no propositional variables and P is a propositional formula. Logically, this pro-clause is equivalent to $C \vee P$, so the bar sign $\mid$ can be seen as simply separating the propositional and non-propositional parts of the pro-clause. We will consider a clause containing no propositional variables as a special kind of pro-clause, in which P is a false formula.

Note that a pro-clause $C \vee P$ is not necessarily a clause, since P can be an arbitrary formula. Also, any propositional formula P can be considered a special case of a pro-clause $\Box \mid P$, where $\Box$ denotes, as usual, the empty clause. We will call any pro-clause $\Box \mid P$ *propositional*.

The calculus RePro is parametrised by an *underlying resolution calculus*. That is, for every resolution calculus on clauses we will define an instance of the calculus RePro based on this resolution calculus. However, since we are not varying the underlying calculus in this paper, we will simply speak of RePro as a calculus.

Generating Inferences. For every generating inference

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

of the resolution calculus the following is an inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n}{C \mid (P_1 \vee \ldots \vee P_n)} .$$

Simplifying Inferences. Let

$$\frac{C_1 \quad \cdots \quad C_n \quad \not{D}}{C} \tag{1}$$

be a simplifying inference of the resolution calculus. Speaking the theory of resolution, this means that C is implied by $C_1, \ldots, C_n, D$ and D is redundant with respect to $C_1, \ldots, C_n, C$. If $P_1 \vee \ldots \vee P_n \rightarrow P$ is a tautology, then the following is a simplifying inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \not{D \mid P}}{C \mid (P_1 \vee \ldots \vee P_n)} . \tag{2}$$

Deletion Inferences. Let

$$C_1 \quad \cdots \quad C_n \quad \not{D}$$

be a deletion inference of the resolution calculus, that is, D is redundant with respect to $C_1, \ldots, C_n$. If $P_1 \vee \ldots \vee P_n \rightarrow P$ is a tautology, then the following is a deletion inference of RePro:

$$C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \not{D \mid P}.$$

Completeness. It is not hard to derive soundness and completeness of RePro assuming the same properties of the underlying resolution calculus, however completeness here means something different from completeness in the theory of resolution. The reason for this difference is that RePro contains essentially no rules for dealing with the

propositional part of clauses. In the completeness theorem below, we assume knowledge of the theory of resolution [1,5]. Also, we do not specify the underlying calculus, for example, the calculus used in Vampire can be used.

Theorem 1 (Completeness). *Let $S_0, S_1, S_2, \ldots$ be a fair sequence of sets of pro-clauses such that S_0 is unsatisfiable. Then there exists $i \geq 0$ such that the set of propositional pro-clauses in S_i is unsatisfiable too.*

The proof is omitted here. Note that this theorem implies that the proof-search in RePro can be carried out by using any standard fair saturation algorithm to perform RePro inferences corresponding to the rules of the underlying calculus plus unsatisfiability checking for the propositional part. This is how it is implemented in Vampire.

To implement such an algorithm for RePro on top of a standard implementation of the resolution calculus one needs to address the following questions:

(q1) representation of the propositional part of pro-clauses;
(q2) representation of propositional pro-clauses (which can be different from the representation of the propositional part of pro-clauses);
(q3) unsatisfiability checking for sets of propositional pro-clauses;
(q4) efficient simplification rules for pro-clauses.

There are some other implementation details to be addressed. For example, the inference selection process in saturation algorithms usually depends on the weights of clauses (which is usually their size measured in the number of symbols). One can use different size measures for pro-clauses, especially when their propositional parts are not necessarily clauses. This adds one more question:

(q5) pro-clause selection.

Before discussing possible answers we will introduce some other rules that can be used in RePro.
Propositional tautology deletion is a deletion rule of RePro formulated as follows:

$$\frac{D \mid P}{}\,,$$

where P is a tautology.
The merge rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad C \mid P_2}{C \mid (P_1 \wedge P_2)}$$

Note that so far this is the only rule that introduces propositional formulas other than clauses.
The merge subsumption rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad D \mid P_2}{D \mid (P_1 \wedge P_2)}\,,$$

where C subsumes D. This rule can also introduce propositional formulas that are not clauses.

The Calculus ReProR One can also define simplifying rules on pro-clauses in an alternative way. Namely, the modification is as follows. Consider a simplifying rule (1) of the underlying resolution calculus. Then the following can be considered as a simplifying inference rule:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \cancel{D \mid P}}{C \mid (P_1 \vee \ldots \vee P_n) \quad D \mid (P_1 \vee \ldots \vee P_n \to P)}\ .$$

One can see that the previously defined simplifying rule (2) is a special case of this one, since, if $P_1 \vee \ldots \vee P_n \to P$ is a tautology, the second inferred clause can be removed. One can also reformulate the deletion rules in the same way. We will denote the resulting calculus ReProR (the refined RePro). Note that the simplification rules of the refined calculus introduce non-clauses in the propositional part.

The advantage of the alternative formulation of simplification and deletion rules is that one clause can be simplified away into a tautology using a sequence of simplifying or deletion rules impossible in the standard formulation of RePro. For example, a clause $A \vee B \mid (p \wedge q)$ is redundant in the presence of $A \mid p$ and $B \mid q$ using the following sequence of subsumption deletion rules:

$$\dfrac{B \mid q \quad \dfrac{A \mid p \quad \cancel{A \vee B \mid (p \wedge q)}}{\cancel{A \vee B \mid (p \to (p \wedge q))}}}{A \vee B \mid (q \to (p \to (p \wedge q)))}$$

whose conclusion is a tautology.

3 Splitting

In very simple terms, splitting is based on the following idea. Suppose that S is a set of clauses and $C_1 \vee C_2$ a clause such that the variables of C_1 and C_2 are disjoint. Then the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. There is more than one way to implement splitting. Before discussing them let us introduce some definitions.

Recall that a clause is a disjunction $L_1 \vee \ldots \vee L_n$ of *literals*, where a literal is an atomic formula or a negation of an atomic formula. A literal or clause is *ground* if it contains no occurrences of variables. In the context of splitting we consider a clause as a set of its literals. In other words, we assume that clauses do not contain multiple occurrences of the same literal and clauses equal up to permutation of literals are considered equal. Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. Then we say that the clause $C \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ is *splittable* into *components* $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a *splitting* of C. For example, every ground multi-literal clause is splittable. There may be more than one way to split a clause, however there is always a unique splitting such that each component C_i is non-splittable: we call this splitting *maximal*. It is easy to see that a maximal splitting has the largest number of components and every splitting with the largest number of components is the maximal one. There is a simple algorithm for finding the maximal splitting of a clause [6], which is, essentially, the union-find algorithm.

Splittable clauses appear especially often when theorem provers are used for software verification and static analysis. Problems used in these applications usually have a large number of ground clauses (coming from program analysis) and a small number of non-ground clauses (for example, an axiomatisation of memory or objects).

There are essentially two ways of using splitting in a first-order resolution theorem prover. One is splitting with backtracking as implemented in Spass [14] and another *splitting without backtracking* [6]. Each of them is described in the next subsections, where we also point out potential efficiency problems associated with each kind of splitting.

When we discuss the use of splitting in resolution theorem provers, it is very important to understand how the use of splitting affects other components of such provers. The efficiency and power of modern resolution theorem provers comes from two techniques: *redundancy elimination* (see [1] for the theory and [8] for the implementation aspects) and *term indexing* [10]. Another component important for understanding efficiency is the saturation algorithm and especially the clause selection algorithm used to implement this algorithm.

Redundancy Elimination. Unlike backtracking algorithms used in DPLL, saturation algorithms are backtracking-free. When clauses are simplified or deleted, these simplifications and deletions do not have to be undone. On the contrary, some forms of splitting may require backtracking.

Term Indexing. Even when simplifications are used, the search space can quickly grow to hundreds of thousands of clauses. To perform inferences on such a large search space efficiently, theorem provers maintain several indexes storing information about terms and clauses. These indexes make it easier to find candidates for inferences. In some cases inferences can be performed only by using the relevant index, without retrieving clauses used for these inferences. The number of different indexes in theorem provers varies and can be as many as about 10. Frequent insertion and deletion in an index can affect performance of a theorem prover. A typical example is when a theorem prover generates an equality $a = b$ between two constants and uses it to rewrite a into b. For nearly all indexing techniques used in the resolution theorem provers, every term and clause containing a must be removed from all indexes and a new term containing b inserted in them again. Doing this single simplification step on an indexes set with 100,000 clauses can take a very long time.

Clause Selection. Selection of generating inferences in resolution theorem provers is implemented using *clause selection*. For selection, clauses are put in one or more priority queues and selected based on their priorities. Normally, the majority of selected clauses are taken from the available clauses of the smallest weight.

The use of splitting may heavily affect all these parts of the saturation algorithm implementation: redundancy elimination, term indexing and clause selection. Let us discuss this in more detail in the rest of this section.

Splitting without Backtracking. Splitting without backtracking [6] can be implemented using a naming technique. Suppose we have a splittable clause $C_1 \vee \ldots \vee C_n$ with components $C_1, \ldots, C_n$. We introduce new propositional variables $p_1, \ldots, p_{n-1}$ to "name" the first $n - 1$ components. That is, we introduce them together with definitions $p_i \leftrightarrow C_i$. Then we use the rule

$$\frac{\cancel{C_1 \vee \ldots \vee C_n}}{C_1 \vee \neg p_1 \qquad \ldots \qquad C_{n-1} \vee \neg p_{n-1} \qquad C_n \vee p_1 \vee \ldots p_{n-1}}$$

If the same components appear more than once in a splittable clause, their names can be reused. In fact, they should be reused, as shown in [6].

The advantage of this approach is that we do not need to perform any backtracking, which spares us the costs of inserting and deleting clauses from indexes. The only additional cost to the implementation of saturation is an index of components required to reuse names. Such an index is used for all kinds of splitting in Vampire. Checking whether one component is a variant of another is equivalent to the graph isomorphism problem, see [6], yet it practice maintaining and using this index requires the time negligible compared to the overall running time.

Splitting without backtracking is very efficient on some problems but may also be very inefficient, since it can introduce thousands of propositional variables and long clauses containing these variables.

Another drawback of this method is that simplifying inferences are not being performed "across branches." For example, when we split clause $f(a) = a \vee q(a)$, we obtain $f(a) = a \vee p$, so we cannot use the equality $f(a) = a$ to simplify expressions such as $q(f(a))$ by demodulation. In the case of splitting with backtracking, we would obtain the unit clause $f(a) = a$, and all the demodulation simplifications would be performed (though at the cost of having to backtrack them later).

Splitting with Backtracking. Splitting with backtracking is based on the idea of the DPLL splitting. It uses the labelled clause calculus introduced in [3]. We will first describe the use of labels and then show how it can be captured by a variation of the RePro calculus.

When we have a splittable clause $C_1 \vee C$, where C_1 is a minimal component, it is first replaced by C_1, and when we derive a contradiction that follows from C_1, we (well, almost) backtrack to the point of the split and introduce the rest of the clause C. If C_1 is ground (and therefore a literal), we may also add, in the spirit of DPLL, $\neg C_1$ at this point. (Whether we do so is controlled by a Vampire option.)

To implement this technique, we assign a label to every split that we perform, and augment each clause C by a set of split labels on which it depends. Each newly derived clause depends on a set of labels that is the union of the sets belonging to its parents. When a clause is deleted, we need to examine the labels of the clauses which justified the deletion. If the deletion was justified by some labels on which the deleted clause itself does not depend, we do not delete the clause, but rather keep it aside to be restored if we backtrack beyond the label that justified its deletion.

Our implementation of the backtracking splitting can be captured by the RePro calculus, if we restrict the inferences that can be performed at any given point, and introduce a different treatment for simplification inferences.

We do not use any of the RePro rules that would introduce non-clauses into the propositional part. The propositional parts of pro-clauses are therefore clauses, and their literals correspond to the labels that the clause has assigned in the labelled calculus.

We are maintaining a partial model M which is initially empty and to which we add propositional literals corresponding to active splits. At each point we restrict inferences of the RePro calculus to the pro-clauses whose propositional part is not satisfied by the model M.

The split labels are seen as fresh propositional variables and the label introduction at splitting $C_1 \vee C$ can be viewed as naming C_1 with a propositional variable. More precisely, when we split $C_1 \vee C \mid P$ and use the name p_1 for C_1, we add pro-clauses $C_1 \mid (\neg p_1 \vee P)$ and $C \mid (p_1 \vee P)$. We also make a note that p_1 *depends* on every propositional variable in P and add p_1 into the partial model M.

At this point, having p_1 in M keeps the clause $C \mid (p_1 \vee P)$ from participating in any inferences for now. Also, the original clause $C_1 \vee C \mid P$ is subsumed by the newly introduced $C_1 \mid (\neg p_1 \vee P)$ modulo the partial model M.

Clause simplification and restoring upon backtracking is the part that does not fit well into the RePro calculus and for which we need to introduce a special treatment. Among the pro-clauses with propositional parts not satisfied by the partial model M we perform simplifications as we would have done it in the base calculus. Except that there may come a point when we will restore the simplified clause back into the search space: When a clause $C \mid P$ is simplified with $C_1 \mid P_1, \ldots, C_n \mid P_n$ as premises, the restoration of the simplified clause in the labelled calculus corresponds to the point when the formula $F \equiv (P_1 \vee \ldots \vee P_n) \rightarrow P$ becomes no longer satisfied by the partial model M. As a matter of fact, F is actually the propositional formula in one of the conclusions of simplifying inferences in the ReProR calculus. One would be therefore tempted to use the ReProR calculus to capture the splitting with backtracking. However, the problem is that the formula F is not a clause, and the labelled calculus we use to implement backtracking splitting does not easily capture general formulas using the clause labels.

We therefore rather check with each change of the model M whether the condition for restoring any of the simplified clauses does occur, and if so, we put the clause back into the saturation algorithm.

When we derive a propositional pro-clause $\Box \mid Q$, we select an atom p in the clause $Q \equiv Q' \vee \neg p$ so that no atom in Q' *depends* on it (if there is more than one such atom, we choose arbitrarily). Let us remind that there is only one pro-clause with a positive occurrence of p — the clause $C \mid (p \vee P)$ which we introduced after splitting $C_1 \vee C \mid P$. This clause became inactive as we added p into the partial model M, so it could not spread the literal p any further. Now we resolve the pro-clauses $C \mid (p \vee P)$ and $\Box \mid Q' \vee \neg p$ on the atom p to obtain $C \mid (Q' \vee P)$, delete the clause $C \mid (p \vee P)$ and replace p in M with $\neg p$.

Note that we have removed p from M which means that the originally split clause $C_1 \vee C \mid P$ is restored, even though just to become subsumed again by $C \mid (Q' \vee P)$. Now there are two possibilities. If Q' is an empty clause, $C_1 \vee C \mid P$ will never be restored as it is subsumed by $C \mid P$ which has the same propositional part. This corresponds to the case when we have refuted the first branch of the split without any additional assumptions. If Q' is a non-empty clause, the original split clause may be restored if some of the assumptions on which we refuted the split is backtracked.

It should be noted that while we can change the polarity of a propositional variable in the model M from positive to negative, we never change it from negative to positive. Therefore, once we assign false to a propositional variable, all pro-clauses that contain it in a negative literal may be deleted.

Drawbacks. The disadvantage of this kind of splitting is that, upon backtracking, we sometimes have to delete and restore many clauses. This leads to costly index maintenance operations, and also a lot of work can be wasted.

For example, suppose we split a clause $a = b \vee C$ where the symbol b is the smallest in the simplification ordering and does not appear anywhere else in the problem, while a has many occurrences. Splitting this clause will introduce a unit clause $a = b$ and the backward demodulation will replace every occurrence of a by b, resulting in massive updates in all indexes. Since b does not appear anywhere else, the equality will not be helpful in any way, but all the rewritten clauses will depend on this split. Once we reach a refutation using the rewritten clauses, we will have to restore all the clases containing a, once again resulting in massive updates in all indexes. Also note that we may end up doing a lot of repeated work as the proof search on the branch using $a \neq b$ will be likely similar to the one on the $a = b$ branch.

4 Implementation and Parameters

In this section we describe various parameters implemented in Vampire and related to splitting and/or use of propositional variables. We also discuss these parameters in the context of the questions (q1)–(q5). These parameters and their values are summarised in Table 1.

Splitting. The main parameter controlling splitting is `splitting`. It has three values: `backtracking`, `nobacktracking` and `off`. All other options have two values: `on` and `off`.

Clauses may be split either eagerly, before they enter the passive clause container, or the splitting can be postponed until a clause is selected for activation. This is controlled by the option `split_at_activation`.

The set of split clauses can be restricted in several ways. Option `split_goal_only` restricts splitting only to goal clauses and clauses that are derived from them. Enabling `split_input_only` excludes derived clauses from splitting, allowing splits only on the clauses which were initially passed to the saturation algorithm.

A different kind of restriction is to add a requirement that both split components contain fewer positive literals than the original clause. Such splitting will lead to clauses that are closer to Horn form which allows at most one positive literal per clause. This setting is enabled by the `split_positive` option.

Splitting with Backtracking. The implementation is based on [14]. We extended it by use of time stamps and reference counters on clauses. This allows us to implement the structure for restoring clauses upon backtracking more efficiently — upon backtracking we only traverse the list of clauses that is to be restored, and let the time-stamping ensure that we never restore the same clause twice.

If `split_add_ground_negation` is enabled, upon backtracking caused by splitting a ground literal L, we add its negation $\neg L$ as a new clause. The option

Table 1. Option names, short names and values

option	short	values
general		
splitting	spl	backtracking, nobacktracking, off
split_add_ground_negation	sagn	on, off
what and when to split		
split_goal_only	sgo	on, off
split_input_only	sio	on, off
split_positive	spo	on, off
split_at_activation	sac	on, off
propositional pro-clauses		
propositional_to_bdd	ptb	on, off
sat_solver_for_empty_clause	ssec	on, off
sat_solver_with_naming	sswn	on, off
simplifications		
sat_solver_with_subsumption_resolution	sswsr	on, off
empty_clause_subsumption	ecs	on, off
bdd_marking_subsumption	bms	on, off
clause and literal selection		
nonliterals_in_clause_weight	nicw	on, off
splitting_with_blocking	swb	on, off

`nonliterals_in_clause_weight` means that the weight of each clause will be increased by the number of splits on which it depends.

Propositional Parts of Pro-Clauses. There are several possible implementations of pro-clauses with a clausal propositional part. However, variants of RePro using non-clausal propositional parts quickly lead to very complex formulas for which the only suitable data structure we could think of was ordered binary decision diagrams [2], or simply BDDs. Thus, we extended the clause objects in Vampire by a reference to the BDD representing the propositional part of a pro-clause.

Since the refined calculus ReProR requires the use of arbitrary formulas, we also used BDDs to implement this calculus. We hoped that it will be very efficient for some problems since many more clauses would be simplified away. In reality ReProR turned out to be almost dysfunctional. The refined simplification rules created ever more complex propositional formulas with very large BDDs. In many cases these BDDs quickly used all the available memory. It was also common that nearly all runtime of Vampire was consumed by BDD operations. Therefore, we decided not to use ReProR and report no results on it in this paper.

The option `propositional_to_bdd` (`q1`) chooses whether BDDs are used to store propositional parts of pro-clauses. If BDDs are not in use, we treat propositional literals in the same way as all other literals. It is a separate issue how to deal with purely propositional clauses. One can also treat them as ordinary clauses. However, one can choose to pass them to a SAT solver instead. Since every propositional clause

can be considered as a pro-clause $\Box \mid P$ with the empty non-propositional part, options for dealing with such clauses use `empty_clause` in their names. In the option `sat_solver_for_empty_clause` (q2,q3) is on, such clauses are passed to a SAT solver.

When we use BDDs for pro-clauses but not for propositional clauses, whenever we obtain a BDD for a propositional clause, we must convert this BDD to a set of clauses. The number of propositional clauses obtained from a BDD can be exponential. To cope with this problem, we added an option `sat_solver_with_naming` (q2) that would make conversion of BDDs to clauses almost linear time by introducing new propositional variables. An alternative to `sat_solver_with_naming` is the option `sat_solver_with_subsumption_resolution` which uses subsumption resolution to shorten the long clauses generated when converting BDDs to CNF without the introduction of new propositional variables.

If we decide to represent the propositional pro-clauses as BDDs, the transition from a first-order pro-clause into propositional is straightforward. We keep at most one propositional pro-clause by eagerly applying a propositional merging rule

$$\frac{\cancel{\Box \mid P} \quad \cancel{\Box \mid P'}}{\Box \mid (P \wedge P')}$$

whenever obtain a new propositional pro-clause $\Box \mid P'$. This way we know that if the set of propositional clauses becomes unsatisfiable, we will derive a propositional clause with associated $\bot$ BDD node.

For first-order pro-clauses we may decide to reflect the complexity of the propositional part in the clause selection process. To this end, enabling the option `nonliterals_in_clause_weight` in presence of BDDs increases the size of clauses by the depths of the BDD of their propositional parts.[2]

When we derive a propositional pro-clause $\Box \mid P$, clauses $C \mid P'$ such that $P \to P'$ become redundant. This follows from the RePro version of the subsumption rule as an empty clause subsumes any other clause:

$$\Box \mid P \quad \cancel{C \mid P'} \quad \text{if } P \to P'$$

It would not be feasible to make an implication check between P and the propositional part of every pro-clause present in the saturation algorithm. We have implemented two incomplete checks for subsumption by propositional pro-clauses.

The first one focuses on the premises of the derived propositional pro-clause, as there is a good chance that some of the ancestors will also have P as its propositional part. If we succeed with some of the premises, we carry on the check with their premises and further on in the derivation graph, as long as we are succeeding. This check is controlled by the option `empty_clause_subsumption`.

[2] The dag size of BDDs would probably better reflect the complexity of propositional formulas, but computing this measure is not a "local" operation on BDDs — one would need to traverse the whole BDD subgraph to count the distinct nodes. The depth of a BDD can, however, be computed by using just the depths of immediate successors. The tree size of a BDD can be computed locally as well, however it can grow exponentially with the size of the BDD.

The second of the checks uses the shared structure of the BDDs. When we derive a propositional pro-clause $\Box \mid P$, we set a *subsumed* flag in the BDD node corresponding to P. Whenever we see a first-order pro-clause to have a BDD node with the *subsumed* flag, we know it is redundant and can be deleted. Moreover, our BDD implementation is aware of this flag and attempts to "spread" the mark while performing other BDD operations. For example, when performing a disjunction operation, if one of the operands has the flag set, it will be set also for the node representing the disjunction of the operands. This subsumption algorithm is controlled by the option `bdd_marking_subsumption`.

5 Evaluation

There are all together 481 different combinations of values for the Vampire parameters related to splitting and propositional variables, so analysing the results was far from trivial. For simplicity, we will call them *splitting parameters,* though this name is a bit misleading since some of them are actually related to dealing with propositional variables.

For benchmarking we used unsatisfiable TPTP problems having non-unit clauses and rating greater than 0.2 and less than 1. Essentially, the rating is the percentage of existing provers that cannot solve a problem. For example, rating greater than 0.2 means that less than 80% of existing theorem provers can solve the problem. Likewise, rating 1 means that the problem cannot solved by the existing provers. However, the rating evaluation uses a single mode of every prover, so it is possible that the same prover can solve a problem of rating 1 using a different mode. For this reason, we also added problems of rating 1 and solvable by Vampire.[3] We excluded very large problems since for them it was preprocessing, but not other options, that affect results the most. This resulted in 4,869 TPTP problems.

To conduct the experiments, we took a Vampire strategy that is believed to be nearly the best in the overall number of solved problems, and generated the 481 variations of this strategy obtained by setting the splitting parameters to all possible values. For each of these variations, we ran it on the selected problems with a 30 seconds time limit. This resulted in 2,341,989 runs, which roughly correspond to 1.5 years of run time on a single core.

We evaluated the experiments in two different ways. First, we looked at the best overall strategies for the backtracking and non-backtracking splitting, and how many problems they solve. However, the number of solved problems for a single (even the best) setting of parameters is not the main criterion of importance for splitting parameters.

The reason for this is that it is known that problems are normally best solved by attempting them with a cocktail of strategies. The CASC [11] version of Vampire uses a sequence of strategies to solve a problem, and using such a sequence is also a recommended mode for the users. Therefore in the second part of evaluation we looked at the numbers of problems solvable only by particular settings of the splitting parameters.

[3] Solvable by Vampire means solvable with at least one of the 481 different strategies.

Table 2. Problems solved by each setting of the splitting strategy

splitting	strategies	worst	average	best
off	25	2708	2720	2737
backtracking	64	1825	2710	3143
non-backtracking	416	1756	2608	2929

Table 3. Best and worst strategies

	worst	best
`splitting`	`nobacktracking`	`backtracking`
`propositional_to_bdd`	`on`	
`split_at_activation`	`off`	`on`
`split_goal_only`	`off`	`off`
`split_input_only`	`off`	`off`
`split_positive`	`off`	`off`
`nonliterals_in_clause_weight`	`off`	`off`
`bdd_marking_subsumption`	`off`	
`empty_clause_subsumption`	`on`	
`sat_solver_for_empty_clause`	`off`	
`split_add_ground_negation`		`on`

The Best and the Worst Strategies. Only 3,598 (about 74% of all problems) were solved by at least one splitting strategy. The top-level results are summarised in Table 2. The best and the worst strategies are shown in Table 3. Some of the option values in the table are left out because not all combinations of parameters make sense. For example, for backtracking splitting we use labeled clauses, not BDDs, so all BDD related options are left out. As one can see, without splitting all strategies behave very similar, which is expected, since problems normally contain few propositional symbols. However, the use of splitting makes a very substantial difference, especially for the best strategies. For example, the best strategy using splitting solved 3143 problems versus 2737 problems solved without splitting. Another interesting point is a huge gap between the performance of the worst and the best strategies using the same kind of splitting. However, the biggest surprise for us was the fact that the best strategies used splitting with backtracking.

Importance of Particular Parameters. To determine the importance of various splitting options, we put the numbers of problems that can be solved only with a particular value of an option into Table 4. Under (a) we show the number of problems that can be solved either only by backtracking or non-backtracking splitting. The number of problems solvable only without any splitting at all is zero. This perhaps surprising result is due to the fact that splitting can be restricted using the options `split_input_only`, `split_goal_only` and `split_positive` to the extent that almost no splits are actually performed.

The cases (b)–(m) show the numbers of problems requiring a particular value of an option for some of the following cases: `off`, `backtracking`, `nobacktracking` or

Table 4. Problems solved only by a single value of an option

a) splitting

off	0
noback	128
back	198

b) split_at_activation

	on	off
back	147	73
noback	91	93
all	145	113

c) split_goal_only

	on	off
back	31	155
noback	21	207
all	17	159

d) split_input_only

	on	off
back	43	414
noback	67	302
all	33	384

e) split_positive

	on	off
back	37	262
noback	28	146
all	35	181

f) propositional_to_bdd

	on	off
off	62	45
noback	227	107
all	226	106

g) nonliterals_in_clause_weight

	on	off
off	17	11
back	55	45
noback	23	62
all	33	91

h) splitting_with_blocking

	on	off
noback	20	290

i) sat_solver_for_empty_clause

	on	off
off	8	5
noback	34	21
all	34	21

j) sat_solver_with_naming

	on	off
off	2	0
noback	22	0
all	22	0

k) sat_solver_with_subsumption_resolution

	on	off
off	2	1
noback	1	2
all	2	2

l) bdd_marking_subsumption

	on	off
off	62	45
noback	227	107
all	226	106

m) empty_clause_subsumption

	on	off
off	5	7
noback	18	46
all	18	46

n) split_add_ground_negation

	on	off
back	191	6

all. In the first three cases, the numbers for columns off and on stand for the number of problems which could be solved for the specified value of splitting only with the option enabled or disabled. More precisely, e.g. for the column off of option A we give the number of problems for which there existed values of other options so that problem was solved with option A disabled, but for all the combinations of parameters the problem was not solved when the option A was enabled. The row all gives numbers of problems where particular option value was required across all relevant splitting modes.

From the Table 4 (j) it can be seen that the use of naming in clausification of BDDs is always a good thing to do, as none of the problems required to have this option disabled. From case (n) it can be seen that it is very rarely the case that adding ground negations after refuting a splitting branch will harm, as only 6 problems are lost by enabling this setting, however 191 problems required to have this setting enabled. On the other hand, for many other options, having the possibility to enable or disable them is important, as either setting can solve problems which cannot be solved by the other.

6 Conclusion

We implemented two variants of clause splitting and many ways of implementing them in a first-order theorem prover, and through extensive experiments we have shown that the backtracking splitting in our setup gives the best performance. More importantly, we have also shown the importance of keeping a large portfolio of strategies, because a large group of problems can be solved only by a variety of different approaches, not by having only one strategy, even though performing well on average.

Aside of the extensive experimental evaluation, we also presented new families of calculi RePro and ReProR which separate propositional from first-order reasoning.

All the described parameters are supported by the current version of the Vampire theorem prover which is available for download at `http://www.vprover.org`.

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 2, vol. I, pp. 19–99. Elsevier Science (2001)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transaction on Computers 35(8), 677–691 (1986)
3. Fietzke, A., Weidenbach, C.: Labelled splitting. Ann. Math. Artif. Intell. 55(1-2), 3–34 (2009)
4. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
5. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier Science (2001)
6. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Nebel, B. (ed.) 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, vol. 1, pp. 611–617 (2001)
7. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Commun. 15(2,3), 91–110 (2002)
8. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computations 36(1-2), 101–115 (2003)
9. Schulz, S.: E – a brainiac theorem prover. Journal of AI Communications 15(2-3), 111–126 (2002)
10. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 26, vol. II, pp. 1853–1964. Elsevier Science (2001)
11. Sutcliffe, G.: The 4th ijcar automated theorem proving system competition - casc-j4. AI Communications 22(1), 59–72 (2009)
12. Sutcliffe, G.: The tptp problem library and associated infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
13. Voronkov, A.: The anatomy of Vampire: Implementing bottom-up procedures with code trees. Journal of Automated Reasoning 15(2), 237–265 (1995)
14. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 27, vol. II, pp. 1965–2013. Elsevier Science (2001)