January 16, 2001

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

Splitting Without Backtracking

Alexandre Riazanov and Andrei Voronkov

Department of Computer Science
University of Manchester
Preprint Series
CSPP-10

Splitting Without Backtracking

Alexandre Riazanov* Andrei Voronkov†

January 16, 2001

Abstract

Integrating the splitting rule into a saturation-based theorem prover may be highly beneficial for solving certain classes of first-order problems. The use of splitting in the context of saturation-based theorem proving based on explicit case analysis (as implemented in SPASS) employs backtracking which is difficult to implement as it affects design of the whole system. Here we present a "cheap" and efficient technique for implementing splitting that does not use backtracking.

Copyright © 2001, University of Manchester. All rights reserved. Reproduction (electronically or by other means) of all or part of this work is permitted for educational or research purposes only, on condition that no commercial gain is involved.

Recent preprints issued by the Department of Computer Science, Manchester University, are available on WWW via URL `http://www.cs.man.ac.uk/preprints/index.html` or by ftp from `ftp.cs.man.ac.uk` in the directory `pub/preprints`.

*Department of Computer Science, University of Manchester. Partially supported by grants from EPSRC, the Faculty of Science and Technology and an ORS award.

†Department of Computer Science, University of Manchester. Partially supported by grants from EPSRC and the Faculty of Science and Technology.

Contents

---

1 Introduction

Case analysis in the form of the *β-rule* is the core of tableau-based theorem proving methods. If our aim is to refute the set of clauses $S \cup \{\phi \vee \psi\}$, we can reduce this task to refuting two simpler sets: $S \cup \{\phi\}$ and $S \cup \{\psi\}$. This is justified by the following argument. To show that S is false in all the models of the formula $\phi \vee \psi$, one can separately consider two cases: all the models of ϕ and the ones not necessarily satisfying ϕ but satisfying ψ. The saturation based theorem proving has adopted the case analysis principle in the form of splitting that can be formulated in the propositional case as the following inference rule:

$$\begin{tikzcd} & S \cup \{C_1 \vee C_2\} \arrow[dl] \arrow[dr] & \\ S \cup \{C_1\} & & S \cup \{C_2\} \end{tikzcd}$$

where S is a set of clauses, C_1 and C_2 are clauses. Unlike most of the other inference rules in saturation based theorem proving that only modify a set of clauses by adding and removing

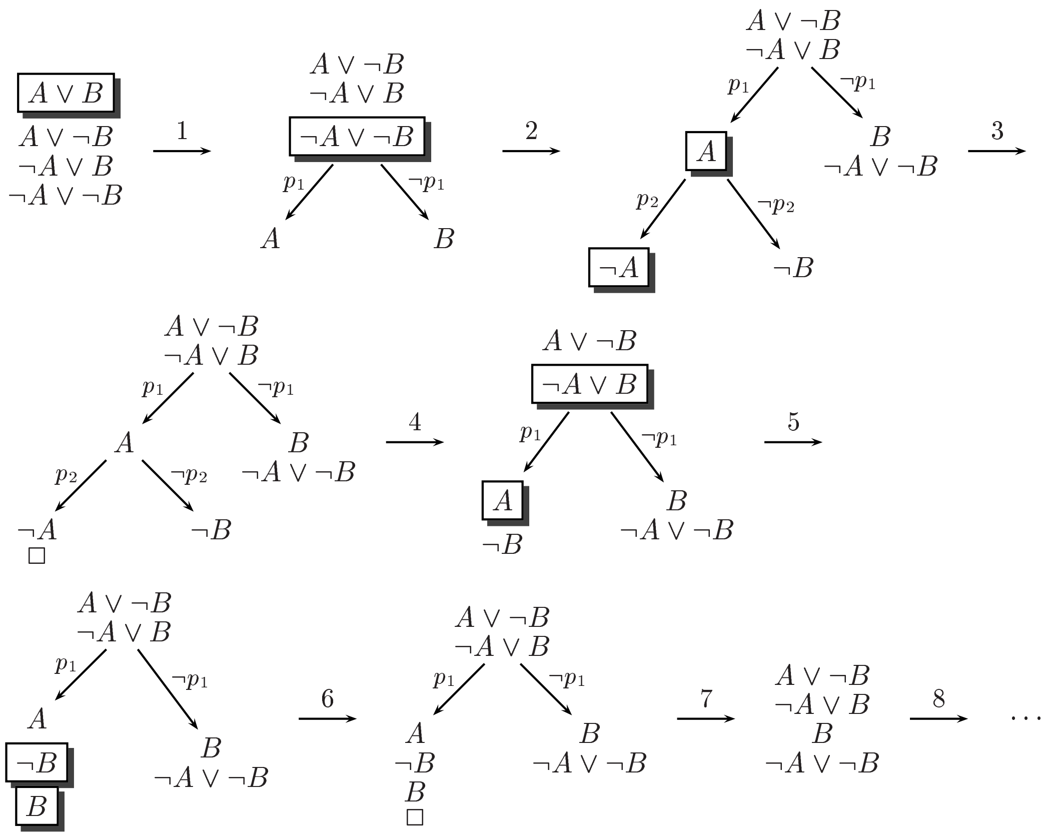

Figure 1: A derivation with splitting and unit resolution

some clauses, this rule makes two sets of clauses out of one. Now, to refute the original set $S \cup \{C_1 \vee C_2\}$, one must separately refute the two new sets: $S \cup \{C_1\}$ and $S \cup \{C_2\}$.

In the propositional case, this rule together with unit resolution gives a complete inference system. Finding refutation in this system can be organized, for example, using depth-first exploration of branches. After finding refutation on one of the splitting branches, the procedure backtracks and proceeds with the second branch. In what follows, we say that such procedures use *explicit case analysis*. A simple example of a propositional derivation which combines the splitting rule with resolution on branches is shown in Figure 1. The current search state is depicted as a tree. The nodes contain clauses common for all the branches below them. The clause or clauses to which an inference rule has been applied is put in a shaded box. For a moment, ignore labels of the edges, for example p_1.

In the first-order case the splitting rule can be formulated in the same way as in the propositional case with an additional restriction: the clauses C_1 and C_2 cannot have common variables (a generalization of splitting where a clause can be split even in presence of common variables has recently been considered by Baumgartner [2000]). This rule can be combined with inference rules like resolution and paramodulation. For example, the SPASS prover [Weidenbach, Afshordel, Brahm, Cohrs, Engel, Keen, Theobalt and Topic 1999] implements an explicit case analysis procedure.

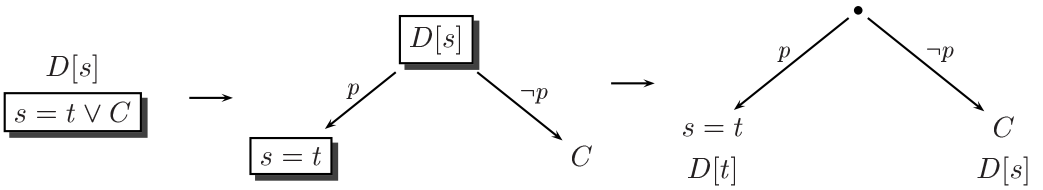

Figure 2: A split followed by a simplification

The cost of naive backtracking is very high, since the clause set S can contain tens of thousands of clauses and several indexes on terms, literals, and clauses, so after several splits one needs a lot of memory to store backtrack points and a lot of time to restore the clause sets and indexes upon backtracking. Therefore, in practice explicit case analysis is implemented by adding to each clause its *split history*: the set of splits used to obtain this clause, represented by an array of bits. The split history also helps to implement *intelligent backtracking*: by analyzing the history list one can check which splits have actually been used to obtain a contradiction on the current branch.

Clauses which do not belong to the currently exploited branch are put in a special waiting list, and popped back from the waiting list upon backtracking. There are complications caused by simplification by unit equalities belonging to a branch which can be illustrated by the following example.

EXAMPLE 1.1 Consider the split on the left-hand side of Figure 2, where C and $s = t$ have no common variables, $s \succ t$ in the simplification order used in the proof-search, and $D[s]$ is a clause with an occurrence of s. After the split the left branch contains two clauses $s = t$ and $D[s]$, so $D[s]$ can be simplified and replaced by $D[t]$, but only on the left branch, since there is no clause $s = t$ on the right branch. This means that $D[s]$ should still be kept on the right branch, so the simplification by $s = t$ should be implemented according to the right-hand side of Figure 2.

In general, backtracking in resolution-based theorem provers requires a nontrivial implementation. Among the modern saturation-based theorem provers for first-order logic, splitting with backtracking is implemented only in SPASS [Weidenbach et al. 1999].

In this report we discuss an alternative to explicit case analysis which can be implemented in resolution-based theorem provers without a significant overhead. We call it *splitting without backtracking*. The idea of splitting without backtracking can be formulated as follows. Consider again the split of Example 1.1, but let us mark the branches of the search tree by literals as follows. We take a new propositional symbol p and mark the left branch with p and the right branch with $\neg p$ as shown in Figure 2.

If there are several splits, we introduce a new propositional symbol for every split. Now, instead of dealing with branches, we simply add to each clause on a branch all labels used on that branch. Then the split can be described as the following inference rule which replaces a clause by two new clauses:

$$S \cup \{D[s], s = t \vee C\} \rightarrow S \cup \{D[s], s = t \vee p, C \vee \neg p\}.$$

It is not hard to argue that such a splitting preserves the unsatisfiability of the set of clauses (we will prove it formally below). The effect of splitting is very similar to the effect of explicit case analysis, but branches are no more needed, since we are still dealing with just one set of clauses. The following simplification of $D[s]$ by $s = t$ on the left branch can now be described in at least two possible ways:

$$S \cup \{D[s], s = t \vee p, C \vee \neg p\} \rightarrow S \cup \{D[s], s = t \vee p, C \vee \neg p, D[t] \vee p\}$$

or

$$S \cup \{D[s], s = t \vee p, C \vee \neg p\} \rightarrow S \cup \{s = t \vee p, C \vee \neg p, D[t] \vee p, D[s] \vee \neg p\}.$$

The second way corresponds to the simplification of Example 1.1, but we prefer the first way because it gives simpler clauses and it does not require any special implementation, since it can be obtained by the ordinary superposition rule.

We will show that splitting without backtracking can simulate, in some sense, explicit case analysis. Moreover, by using various selection functions one can simulate a parallel version of case analysis, in which branches are exploited simultaneously. The corresponding version of splitting without backtracking has several advantages over the binary version: firstly, we obtain the effect of intelligent backtracking for free, and secondly, some simplification rules can move clauses between the branches thus avoiding repeated work on different branches.

This article is organized as follows. In Section 2 we define some fundamental notions of this paper, for example those of *component* and *split*. We also recall some notions of the theory of resolution. In Section 3 we discuss several versions of the splitting rule, namely *binary* and *hyper* splitting, with or without *naming*. In Section 4 we show how one simulate different strategies of splitting by using appropriate literal selection functions. In particular, we show how one can simulate explicit case analysis by using so-called *blocking extension* of selection functions. In Section 5 we introduce an efficient algorithm for finding the maximal split of a clause into components. In Section 6 we discuss the *clause variance problem*: checking if a clause is a variant of another clause. We show that the clause variant problem is polynomial-time equivalent to the graph isomorphism problem. Finally, in Section 7 we discuss experiments carried out with VAMPIRE over a large collection of problems.

2 Preliminaries

We assume a fixed signature Σ. By $\mathbb{P}$ we denote a countable set of predicate symbols of arity 0 disjoint from Σ. Elements of $\mathbb{P}$ will be denoted by $p_1, p_2, \ldots$ and used as new names for clause components, or alternatively, as identifiers for branches.

We call a *clause* a set of literals. Usually clauses will be written as disjunction of their literals. We denote by $var(E)$ the set of all variables occurring in an expression E (e.g. clause). For a clause C, the *universal closure* of C, denoted by $\forall C$, is the formula $\forall x_1 \ldots \forall x_n C$, where $x_1, \ldots, x_n$ are all variables of C.

2.1 Components and splits

The clause C is called a *component* of a clause $C \vee D$ if C is nonempty, $var(C) \cap var(D) = \emptyset$ and C contains no predicate symbols in $\mathbb{P}$. The component C is called *minimal* if no proper subset of C is a component in $C \vee D$. The component C is called *trivial* if D is a clause of the signature $\mathbb{P}$. A *split* of a clause D is any representation of C as $C_1 \vee \ldots \vee C_n \vee D'$, where $C_1, \ldots, C_n$ are different components of C. The split is *trivial* if $n = 1$ and C_1 is the trivial component of D. The split is *maximal* if $C_1, \ldots, C_n$ are all the minimal components of D. Note that in the maximal split the subclause D' only consists of the literals of the signature $\mathbb{P}$. Also, it is obvious that the maximal split of a clause is unique.

2.2 Resolution

Our technique for implementing splitting works in the context of a saturation procedure for the calculus of ordered resolution. For the purposes of this paper it is enough to recall only the nonground version of the ordered resolution rule itself, for other rule and calculi see e.g. [Bachmair and Ganzinger 2001]. Let $\succ$ be a simplification order on terms, extended in the usual way on literals. We assume that a *selection function* selects in every nonempty clause either a negative literal, or a subset of positive literals, and if any positive literal is selected, then all maximal w.r.t. $\succ$ literals are selected too. Then *ordered resolution* is the following rule:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta},$$

where θ is a most general unifier of A and B, the atom A is selected in the clause $A \vee C$, the literal $\neg B$ is selected in the clause $\neg B \vee D$, and there is no literal in $D\theta$ greater than $A\theta$ in the used simplification ordering.

3 Splitting without backtracking

In this section we formulate several versions of splitting without backtracking and prove their soundness.

3.1 Binary splitting

DEFINITION 3.1 (binary splitting rule) Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$, $C \vee D$ be a clause with a nontrivial component C, and $p \in \mathbb{P}$ be an atom not occurring in $S \cup \{C \vee D\}$. Then the *binary splitting rule* is the following inference rule:

$$S \cup \{C \vee D\} \to S \cup \{C \vee p, D \vee \neg p\}. \tag{1}$$

We also say that $S \cup \{C \vee p, D \vee \neg p\}$ is obtained from $S \cup \{C \vee D\}$ by *binary splitting*.

THEOREM 3.2 *The binary splitting rule preserves satisfiability.*

PROOF. Any model for $S \cup \{C \vee p, D \vee \neg p\}$ is also a model for $S \cup \{C \vee D\}$ since $C \vee D$ is a logical consequence of $C \vee p$ and $D \vee \neg p$. In the reverse direction, a model M' for $S \cup \{C \vee p, D \vee \neg p\}$ can be obtained from a model M for $S \cup \{C \vee D\}$ by redefining p so that $M' \models p \leftrightarrow \neg\forall C$.

□

3.2 Binary splitting with naming

It follows from the proof of Theorem 3.2 that the new predicate symbol p introduced by an application of the binary splitting rule (see (1)) can be regarded as a new name of the formula $\neg\forall C$. This implies an optimization of binary splitting which allows one to reuse the new names. This optimization can be formalized as follows.

DEFINITION 3.3 (naming function) A *naming function* is any function N from the set of non-empty clauses of the signature Σ to the set $\mathbb{P}$ with the following property: $N(C_1) = N(C_2)$ if and only if C_1 is a variant of C_2.

DEFINITION 3.4 (binary splitting rule with naming) Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$ and $C \vee D$ be a clause with a nontrivial component C. Let also N be a naming function. Then the *binary splitting rule with naming* is the following inference rule:

$$S \cup \{C \vee D\} \to S \cup \{C \vee N(C), D \vee \neg N(C)\}. \tag{2}$$

One can also consider a modification of the binary splitting rule with naming where only the first application of the splitting introduces the clause $C \vee N(C)$, and all further applications of splitting with the same component C simply replace $C \vee D$ by *one* clause $D \vee \neg N(C)$.

Let us give an example that illustrates the difference between the binary splittings with and without naming. Suppose that we have a set of clauses $S \cup \{C \vee D_1, C \vee D_2\}$ such that C is a nontrivial component in both $C \vee D_1$ and $C \vee D_2$. Using binary splitting, we can perform the following derivation:

$$\begin{array}{l} S \cup \{C \vee D_1, C \vee D_2\} \to \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee D_2\} \to \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee p_2, D_2 \vee \neg p_2\}. \end{array}$$

Using binary splitting with naming, we can perform a different derivation (assuming $N(C) = p_1$).

$$\begin{array}{l} S \cup \{C \vee D_1, C \vee D_2\} \to \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee D_2\} \to \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, D_2 \vee \neg p_1\}. \end{array}$$

Preservation of satisfiability in the case of binary splitting with naming is more difficult to formulate. We can only guarantee preservation of satisfiability under some natural conditions.

THEOREM 3.5 *Let I be an inference system on sets of clauses with the following properties: every inference preserves the set of models, i.e. for every inference $S \to S'$ in this system, S and S' have the same models. Let I' be the extension of I by binary splitting with naming. Then for every derivation*

$$S_0 \to S_1 \to \dots$$

in I' such that S_0 is a set of clauses of the signature Σ and every $i \geq 0$, the set S_i is satisfiable if and only if so is S_{i+1}.

PROOF. The proof essentially repeats the proof of Theorem 3.2. Take any model M of S_0 of the signature Σ and define its extension M' to the signature $\Sigma \cup \mathbb{P}$ as follows: define in M' each new predicate symbol $N(C) \in \mathbb{P}$ be equivalent to $\neg \forall C$. Then arguing as in the proof of Theorem 3.2 one can show that M' is a model of each S_i.

In the converse direction, note that every S_i is a logical consequence of S_{i+1}. Indeed, when S_{i+1} is obtained from S_i by applying an inference of the original system I, this holds by our assumption. When S_{i+1} is obtained from S_i by applying binary splitting with naming (2), this is also obvious, since S_{i+1} contains the clause $C \vee N(C)$. $\square$

Note that this proof can be easily adapted to the modified version of splitting. For the modified version in general it is not true any more that S_{i+1} contains the clause $C \vee N(C)$, but one can show that $C \vee N(C)$ is a logical consequence of S_{i+1}.

The conditions on the inference system I are natural and not restrictive. For example, the standard proofs of completeness in the theory of resolution [Bachmair and Ganzinger 2001] use inference systems with two kinds of rule: addition of a clause implied by other clauses, and removal of a clause implied by other (smaller) clauses. It is easy to see that every such inference system preserves models.

3.3 Hyper splitting rule

Similar to the binary splitting, we can define hyper splitting, in which we split more than one component off a clause.

Definition 3.6 (hyper splitting rule) Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$ and $C_1 \vee \ldots \vee C_n \vee D$ be a clause with nontrivial components $C_1, \ldots, C_n$. Let $p_1, \ldots, p_n$ be predicate symbols in $\mathbb{P}$ occurring in neither this clause nor S. Then the *hyper splitting rule* is the following inference rule:

$$S \cup \{C_1 \vee \ldots \vee C_n \vee D\} \to S \cup \{C_1 \vee p_1, \ldots, C_n \vee p_n, D \vee \neg p_1 \vee \ldots \vee \neg p_n\}. \tag{3}$$

Let also N be a naming function. Then the *hyper splitting rule with naming* is the following inference rule:

$$S \cup \{C_1 \vee \ldots \vee C_n \vee D\} \to S \cup \{C_1 \vee N(C_1), \ldots, C_n \vee N(C_n), D \vee \neg p_1 \vee \ldots \vee \neg p_n\}. \tag{4}$$

Hyper splitting can be interpreted as repeatedly applied binary splitting. One can prove that hyper splitting (with or without naming) preserves satisfiability in exactly the same way as for binary splitting.

4 Literal selection and simulation of explicit case analysis

So far we only considered splitting as an inference rule. Here we will consider splitting as part of an inference system. As the inference system we consider binary resolution with superposition and negative selection. We assume a simplification ordering in which every literal of the signature Σ is greater than any atom in the signature $\mathbb{P}$. In order to define the inference system we have to define a selection function on clauses. We will define two kinds of selection functions: blocking and parallel. Both will be obtained by extending an arbitrary selection function on the clauses of the signature Σ to the clauses of the extended signature $\Sigma \cup \mathbb{P}$.

4.1 Blocking extension of a selection function

Definition 4.1 (blocking extension) Let s be any selection function on the clauses of the signature Σ. The *blocking extension* of s is the selection function on the clauses of the signature $\Sigma \cup \mathbb{P}$ defined as follows.

1. Let D be a clause containing a negative literal $\neg p$ of the signature $\mathbb{P}$, then the maximal one among such literal is selected in D.

2. Let D be a clause of the form $D' \vee P$, where D is a nonempty clause of the signature Σ and P is a disjunction of literals of the signature $\mathbb{P}$. Then the literals selected in D are exactly those selected in D'.

3. Let a clause P consists of positive literals of the signature $\mathbb{P}$. Then the maximal literal is selected in P.

It is not difficult to prove that we indeed obtain a selection function, since all literals of the signature $\mathbb{P}$ are less than all literals of the signature Σ. Since we will usually assume some selection function on the clauses of Σ, we will simply speak about the *blocking selection function* instead of the blocking extension of the selection function.

Now assume that we are using a blocking selection function. Let us also assume that each time we apply the splitting rule, the newly introduced atom p is greater than any atom previously introduced by splitting. Let us show that the resulting inference system simulates, to some extent, explicit case analysis.

Consider a sequence of splits and applications of nonsimplifying inference rules. Let us call the *label* of a branch the set of all positive literals that mark edges of this branch. We transform clauses used in a derivation with splitting as follows. To each clause C that appears on a branch we add the label P of this branch, obtaining the clause $C \vee P$. We can show that each nonsimplifying inference that can be performed on a branch, can also be performed on the transformed clauses. For simplicity, we only consider the resolution rule.

Let $A_1 \vee D_1$ and $\neg A_2 \vee D_2$ be two clauses in which A_1 and $\neg A_2$ are selected, θ be a most general unifier of A_1 and A_2, and no literal in $D_1\theta$ is greater than $A_1\theta$. Then we can apply a resolution inference as follows.

$$\frac{A_1 \vee D_1 \quad \neg A_2 \vee D_2}{D_1\theta \vee D_2\theta} \, .$$

Let P_1 and P_2 be the labels of the branches of $A_1 \vee D_1$ and $\neg A_2 \vee D_2$ respectively. Then the corresponding transformed clauses will be $A_1 \vee D_1 \vee P_1$ and $\neg A_2 \vee D_2 \vee P_2$. By our definition of a blocking selection function and the order $\succ$ it is not hard to argue that A_1 and $\neg A_2$ will be selected in the transformed clauses and that no literal in $D_1\theta \vee P_1\theta$ is greater than $A_1\theta$. Therefore, we can apply the rule

$$\frac{A_1 \vee D_1 \vee P_1 \quad \neg A_2 \vee D_2 \vee P_2}{D_1\theta \vee D_2\theta \vee P_1 \vee P_2} \, .$$

It is not hard to argue that $P_1 \vee P_2$ is the label of the common branch for $A_1 \vee D_1$ and $\neg A_2 \vee D_2$.

It is interesting to observe what happens to the literals which do *not* belong to the currently investigated branch. Suppose, for, simplicity, that we are working with the leftmost branch labelled by $p_1 \vee \ldots \vee p_n$. Then all clauses which are not on the current branch contain a negative literal of the signature $\mathbb{P}$, which by our choice of the selection function, will be selected. Take any such clause, for example, $\neg p_n \vee D$. The only inference rule we can apply to this clause is resolution with a clause $p_n \vee P$, where p_n is greater than any literal of P. But we can obtain such a clause only when the empty clause is obtained on the leftmost branch labelled by $p_n \vee P$, so $\neg p_n \vee D$ will be blocked for inferences until a contradiction on the leftmost branch is obtained. As soon as such a contradiction is obtained, we can "unblock" the clause $\neg p_n \vee D$ by resolving it against $p_n \vee P$ and obtaining $D \vee P$.

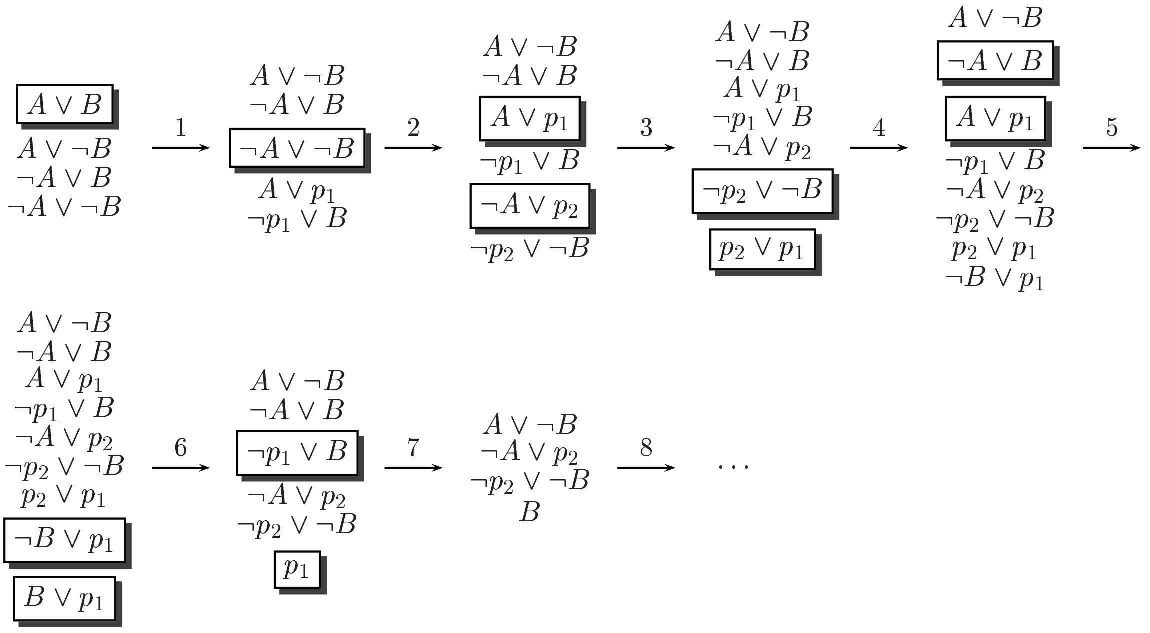

Figure 3: A derivation using binary splitting and blocking selection

Thus, we obtain a nearly complete correspondence between search states of explicit case analysis and the system with binary splitting without backtracking. It is not an exact correspondence for several reasons. Firstly, simplification rules behave in a slightly different way. Secondly, splits done by splitting without backtracking often do not belong to particular branches, but are "shared" across different branches. To explain this effect, we show in Figure 3 a derivation using binary splitting and a blocking selection function which corresponds to the derivation of Figure 1. We preserve the same enumeration of inferences as in that figure. If a literal in the new signature $\mathbb{P}$ is selected, we put it in front of the clause, for example $\neg p_1 \vee B$ means that $\neg p_1$ is selected. For simplicity, we remove subsumed clauses and pure literals.

After inference 7 of the derivation using explicit case analysis, we have a copy of the clause $\neg A \vee \neg B$ which was put on the right branch, after we performed a split on this clause on the left branch. In the derivation using binary splitting, the split of $\neg A \vee \neg B$ into $\neg A \vee p_2$ and $\neg p_2 \vee \neg B$ remains even when we finish working on the left branch.

4.2 Parallel extension of a selection function

Definition 4.2 (parallel extension) Let s be any selection function on the clauses of the signature Σ. The *parallel extension* of s is a selection function s' on the clauses of the signature $\Sigma \cup \mathbb{P}$ defined as follows. Consider a clause of the form $D' \vee P$, where D' is a clause of the signature Σ and P is a clause of the signature $\mathbb{P}$. If D' is not empty, then s' selects in $D' \vee P$ exactly the same literals as s selects in D'. When D' is empty, the maximal literal from P is selected.

Again, it is not difficult to prove that we indeed obtain a selection function, since all literals of the signature $\mathbb{P}$ are less than all literals of the signature Σ. We will simply speak about a *parallel selection function* instead of the parallel extension of a selection function.

A parallel selection function ignores the splitting history of clause completely, and thus gives the effect of parallel search on all branches.

4.3 Subsumption resolution and splitting

One of the serious advantages of splitting without backtracking is that it can be productively combined with the simplification rule called *subsumption resolution* [Bachmair and Ganzinger 2001]. Essentially, subsumption resolution is a special case of binary resolution that guarantees that the resolvent subsumes one of the parents. We say that a clause D_1 *subsumes* a clause D_2 if there exists a substitution σ such that $D_1\sigma \subseteq D_2$. It is known that clauses subsumed by other clauses can be removed from the search space.

Subsumption resolution is the following inference rule on sets of clauses:

$$S \cup \{A_1 \vee D_1, \neg A_2 \vee D_2\} \to S \cup \{A_1 \vee D_1, D_2\}$$
$$S \cup \{\neg A_1 \vee D_1, A_2 \vee D_2\} \to S \cup \{\neg A_1 \vee D_1, D_2\}$$

where there exists a substitution σ such that $A_1\sigma = A_2$ and $D_1\sigma \subseteq D_2$. The rule is called subsumption resolution because of its resemblance to subsumption: this rule is applicable if and only if $A_1 \vee D_1$ subsumes $A_2 \vee D_2$. Thus subsumption algorithms can be modified for finding pairs of clauses to which subsumption resolution is applicable.

Since subsumption resolution is a simplification rule (the clause $\neg A_2 \vee D_2$ or $A_2 \vee D_2$ is replaced by a smaller clause D_2), it can be applied independently of the selection function or order on literals.

Subsumption resolution combined with splitting can give the following effect: literals can be moved across different branches. For example, consider two clauses $C \vee p$ and $D \vee \neg p$ such that C subsumes D. In explicit case analysis, these clauses would belong to two different branches, because $C \vee p$ should be on the left branch of splitting on p, while $D \vee \neg p$ on the right branch. If we use subsumption resolution, we can replace $D \vee \neg p$ by the clause D, which corresponds to moving B from the right branch to the top.

The effect of subsumption resolution is even stronger when C is a variant of D or coincides with D. Then we obtain two clauses $C \vee p$ and $C \vee \neg p$, which by subsumption resolution followed by subsumption will be replaced by C. This has the effect of noting that C belongs to two different branches, and putting it on top and can save a lot of resources compared to explicit case analysis. Splitting with backtracking would repeat inferences with C on different branches.

Let us show that subsumption resolution and splitting can, in some cases, give the effect of condensing a clause. Condensing is defined in [Joyner, Jr. 1976]. A clause D' is said to be obtained from a clause D by *condensing* if D' is a proper subset of D and D subsumes D'.

EXAMPLE 4.3 Let $C[\bar{x}]$ be a clause with variables in $\bar{x}$, $\bar{y}$ be a sequence of variables disjoint from $\bar{x}$. Then $C[\bar{y}]$ is a variant of $C[\bar{x}]$. Consider the clause $C[\bar{x}] \vee C[\bar{y}]$. This clause can be condensed into $C[\bar{x}]$. Binary splitting will replace this clause by two clauses

$$C[\bar{x}] \vee p \quad \text{and} \quad C[\bar{y}] \vee \neg p.$$

Application of subsumption resolution to these clauses yields the clause $C[\bar{x}]$ which subsumes each of these clauses. Thus, binary splitting and subsumption resolution in this example give the effect of condensing.

```
procedure DivideIntoComponents(set of literals {L_1, ..., L_n})
begin
  let components[1, ..., n] be an array of integers
  let x_1, ..., x_m be all variables that can occur in L_1, ..., L_n
  let literals[1, ..., m] be an array of integers
  registered := ∅ /* registered variables */
  for lit = 1, ..., n /* current literal number */
    components[lit] := lit
    for x_k ∈ vars(L_lit)
      if x_k ∈ registered /* x_k occurred earlier */
        then
          first := literals[k]
          while components[first] ≠ first
            comp := components[first]
            components[first] := lit
            first := comp
          end while
          components[first] := lit
        else
          /* first occurrence of the variable */
          literals[k] := lit
          registered := registered ∪ {x_k}
    end for
  end for
  for lit = n, ..., 1 /* moving backwards */
    if components[lit] = lit
      then report lit identifies a new component
      else components[lit] := components[components[lit]]
    report L_lit belongs to the component components[lit]
  end for
end
```

First phase

linking loop

Second phase

Figure 4: Algorithm for finding the maximal split of a clause

5 Efficient algorithm for finding the maximal split

To implement the splitting rule efficiently, one needs to check whether a given clause admits a nontrivial splitting. More generally, an algorithm is required to find the maximal split of a clause. In this section we show an efficient algorithm for finding the maximal split. For simplicity we assume that all input literals are nonground. The algorithm is shown in Figure 4.

During the first phase we iterate over the literals of the clause $\{L_1, \ldots, L_n\}$, and for every literal L_i we process its variables $vars(L_i)$ one by one. When we process a variable, register it by adding it to a set *registered*, which is initially empty.

The algorithm makes use of two integer arrays, one is $literals[1, \ldots, m]$, where $x_1, \ldots, x_m$ are all variables that can occur in the input clause. At any stage, $literals[k]$ is the number i of the literal L_i where the variable x_k occurred for the first time, if x_k has been registered.

Another array used by the algorithm is $components[1, \ldots, n]$. The integer $components[i]$ contains a number $j \geq i$ such that L_i and L_j currently belong to the same component, i.e.

are linked by some registered variables. In particular, when L_j has the maximal number in its component, $components[j] = j$.

In order to achieve this property of *components*, in the *linking loop* we examine the chain of literals *reachable from literals*$[k]$, i.e. literals with numbers $literals[k] = j_1 < \cdots < j_s$, $components[j_r] = j_{r+1}, components[j_s] = j_s$. We link these literals to the currently processed L_{lit} by assigning *lit* to $components[j_r]$.

THEOREM 5.1 *The algorithm is correct.*

PROOF. (Sketch). We will show some invariants of the algorithm which will immediately imply its correctness. At every step s of the algorithm, let us denote by D_s the clause obtained from the input clause by replacing all yet unprocessed occurrences of variables by constants.

We argue that at every iteration of the algorithm D_s is already divided into components in the following sense: some literals L_i and L_j belong to one component if and only if the two chains of literals reachable from i and j have the same end which is exactly the literal with the greatest number in this component. This property is trivially guaranteed by the linking loop.

Note that at the end of phase 1 D_s is the input clause D, so by the above arguments D is already divided into component.

Let us represent the intuition behind the second phase and then argue its correctness. The second phase is intended to represent the result of linking in a usable form. To relate literals to the corresponding components, we iterate over $\{L_1, \ldots, L_n\}$ again, but in the reverse order. When some $\{L_i, \ldots, L_n\}$ have been processed, for any $j \in \{i, \ldots, n\}$ $components[j]$ is the identifier of the component containing L_j. To achieve this, the operation $components[lit] := components[components[lit]]$ is performed for every current number of literal *lit* which is not a component identifier.

It can be shown by a simple inductive argument that for any such *lit* the number $components[components[lit]]$ contains the identifier for the corresponding component. □

Notice that when there is only one component in the clause, the second phase is redundant. To make use of this observation, the algorithm can be slightly modified to count the number of different components during the first phase.

6 Complexity of splitting with naming

To implement splitting with naming, it is required to check whether a component is a variant of another component. In this section we prove that the complexity of checking whether one clause is a variant of another one is polynomial-time equivalent to the graph isomorphism problem.

DEFINITION 6.1 (clause variance problem) *Clause variance* is the following decision problem. An instance of this problem is a pair of clauses (C_1, C_2). The answer is "yes" if C_1 is a variant of C_2.

THEOREM 6.2 *The clause variance problem is polynomial-time equivalent to the graph isomorphism problem.*

PROOF. Let us show a polynomial-time reduction of the graph isomorphism problem to the clause variance problem. We only consider the graph isomorphism problem for pairs of graphs

G_1, G_2 such that G_1 and G_2 have the same number of nodes. The graph isomorphism problem for such graphs is evidently equivalent to the general graph isomorphism problem.

Let G be a graph with the nodes $n_1, \ldots, n_k$. Define a clause $C(G)$ in the signature consisting of a binary function symbol r and variables in a set $\{x_1, \ldots, x_k\}$ as follows.

$$C(G) = \{r(x_i, x_j) \mid (x_i, x_j) \in G\}.$$

It is not hard to argue that the following statement holds: G_1 is isomorphic to G_2 if and only if $C(G_1)$ is a variant of $C(G_2)$. Since $C(G)$ can obviously be built from G in polynomial time, the function $(G_1, G_2) \mapsto (C(G_1), C(G_2))$ gives us a polynomial-time reduction of the graph isomorphism problem to the clause variance problem.

The converse reduction is slightly more involved. We have to show how to represent clauses by graphs. Without loss of generality we can restrict ourselves to strictly positive clauses. We also assume that the signature consists of function symbols $f_1, \ldots, f_s$ and predicate symbols $f_{s+1}, \ldots, f_m$.

Let C be a strictly positive clause in such a signature and let $x_1, \ldots, x_k$ be all variables of C.

Let us fix k nodes $n_1, \ldots, n_k$. We will now show how to represent every term t and clause C with variables among $x_1, \ldots, x_k$ by a graph $G(t)$ and $G(C)$ respectively containing the nodes $n_1, \ldots, n_k$. Every graph $G(t)$ will contain a distinguished node, called the *input node* of $G(t)$.

1. For every variable x_i the graph $G(x_i)$ is the graph consisting of the node n_i. This node is also the input node of $G(x_i)$.

2. Consider a compound term or atom $f_i(t_1, \ldots, t_u)$ and let us define the graph $G(f_i(t_1, \ldots, t_u))$. Let $v_1, \ldots, v_{l+1}$ be a sequence of different nodes. We call a *line based on* $v_1, \ldots, v_{l+1}$ the graph whose nodes are $v_1, \ldots, v_{l+1}$ and whose arcs are the pairs (v_i, v_{i+1}), for $i \in \{1, \ldots, l\}$. To construct $G(f_i(t_1, \ldots, t_u))$ we will put together several graphs:

   (a) A $u + 2$-clique H with two distinguished different nodes I and O.

   (b) The graphs $G(t_1), \ldots, G(t_u)$.

   (c) Let $\{v_{i,j} \mid 1 \leq i \leq u, 1 \leq j \leq i + 1\}$ be pairwise different nodes. Take the u lines based on $v_{i,1}, \ldots, v_{i,i+1}$ for all $i \in \{1, \ldots, u\}$.

   (d) Add arcs connecting O with each of the nodes $v_{i,1}$, for $i \in \{1, \ldots, l\}$.

   (e) Add the arc connecting the node $v_{i,i+1}$ with the input node of $G(t_i)$, for all $i \in \{1, \ldots, l\}$.

   In the resulting graph $G(t_1, \ldots, t_n)$ we declare I to be the input node.

3. The graph for a clause $G(A_1 \vee \ldots \vee A_n)$ is obtained by putting together the graphs $G(A_1), \ldots, G(A_n)$ using the common nodes $n_1, \ldots, n_k$.

To illustrate this construction, let us consider the clause $f_3(f_1(x_1), x_2, x_1) \vee f_4(f_1(x_2))$. The graph for this clause is shown in Figure 5.

It is not hard to argue that $G(C)$ can be constructed from C in polynomial time. There is small complication caused by the fact that to represent f_i one needs a graph of the size $O(i^2)$ but not $O(\log i)$, but one can get around this problem by renaming function symbols. It

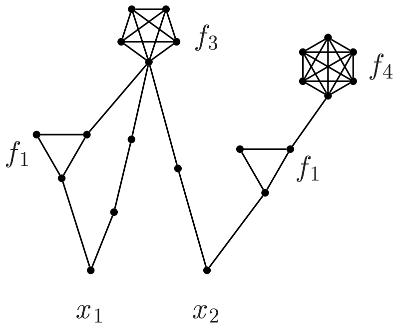

Figure 5: The graph encoding the clause $f_3(f_1(x_1), x_2, x_1) \vee f_4(f_1(x_2))$

is not hard to argue that the graph $G(C)$ represents the clause C up to renaming variables. Indeed, the predicate symbols $f_{s+1}, \ldots, f_m$ can be uniquely identified in the graph as cliques of the size at least $s + 3$, while function symbols correspond to cliques of smaller sizes. After a predicate symbol has been identified, one can uniquely identify its arguments by checking the length of the lines coming out of the predicate symbol's output node. Then one can proceed from the arguments of the predicate symbols to their arguments in a similar way. Finally, when all variables in the graph have been identified, one can note that the graph automorphisms correspond to the renamings of variables. $\square$

More generally, in practice we have to check whether a given component C is a variant a component in a set of components S. In Vampire we implement this *many-to-one clause variance* check using the code tree indexing described in [Voronkov 1995, Riazanov and Voronkov 2000*b*]. Code trees are used in the implementation of many-to-one forward subsumption, it is not hard to modify them to implement the many-to-one clause variance.

7 Experimental results

We implemented several versions splitting with backtracking in the new version of Vampire [Riazanov and Voronkov 1999]. In this section we present experimental results over 4658 nonunit clause form problems. Of these problems, 2820 problems come from the TPTP library [Sutcliffe and Suttner 1998] and 1838 problems from the experiments in list software reuse described in [Schumann and Fischer 1997]. Of the 4658 problems 3300 contain equality. For problems with equality, in addition to splitting without backtracking we implemented the *branch rewriting* rule which replaces a clause $C[s\sigma] \vee P$ by a clause $C[t\sigma] \vee P$, if a clause $s = t \vee P'$ with $P' \subseteq P$ is present. The effect of this simplification rule is similar to the effect of simplification by unit equalities on a splitting branch in an explicit case analysis procedure. To support our viewpoint that we describe a "cheap" implementation of rewriting, we note that branch rewriting was implemented in less than 3 hours.

We present the results for problems with and without equality separately. For all problems we used the same algorithm based on the limited resource strategy [Riazanov and Voronkov 2000*a*] and the same selection function. This algorithm and selection function were chosen because they experimentally proved to be superior to other algorithms and selection functions when splitting was not used. When running Vampire with splitting, we used the following

no	166:11	220:7	227:12	256:6	280:4	300:1	359:3	388:3
PNR		94:36	75:15	125:30	130:9	153:9	215:14	241:11
NR			71:69	43:6	121:58	127:41	147:4	175:3
PN				95:60	111:50	104:20	180:39	198:28
N					98:72	101:52	115:9	143:8
PR						68:45	118:38	144:35
P							110:53	130:44
R								37:8
	PNR	NR	PN	N	PR	P	R	

Figure 6: Results on problems with equality

PN	65:3	74:11	76:0	80:1
N		14:13	16:2	18:1
no			19:6	21:5
P				10:7
	N	no	P	

Figure 7: Results on problems without equality

parameters:

1. Hyper splitting with a Parallel selection function (P) vs. binary splitting with blocking selection.

2. Naming (N) vs. no naming.

3. Branch Rewriting (R) vs. no branch rewriting.

We will use combinations of the three letters P, N, R, to denote different strategies used in the experiments. For example, PR means that the Parallel version with branch Rewriting but no naming was used. We use "no" to denote the strategy that uses no splitting at all, and "−" to denote the strategy that uses splitting without P,R, and N, i.e. simple binary splitting with a blocking selection function.

We used a Linux-running PC with 256M RAM and a 400MHz Intel III processor. For all problems we used the time limit of 1 minute. The chosen strategy without splitting solves 3091 problems. The total number of problems solved with or without splitting is 3189. So with splitting, VAMPIRE could solve 98 problems that could not be solved without splitting. Of these 98 problems, 77 are without equality and 21 with equality. Figures 6 and 7 show the comparative behavior of various strategies depending on the settings of the parameters P, N, R on problems with and without equality, respectively. The pair $n_1 : n_2$ in a row labelled by a strategy S_1 and the column labelled by a strategy S_2 means that there were n_1 problems solved by S_1 but not solved by S_2 and n_2 problems solved by S_2 but not solved by S_1. For example NR (Naming + branch Rewriting) could solve 71 problems with equality not solved by PN (Parallel selection + Naming), while PN could solve 69 problems with equality not solved by NR.

The following conclusion can be derived from these results:

1. On problems with equality, VAMPIRE with any version of splitting is on the average weaker than VAMPIRE without splitting, while on problems without equality it is on the average stronger.

2. Each of the following settings: naming, parallel splitting, and branch rewriting improve the results, by far the strongest one is naming.

Both for logic with and without equality, there have been problems that could be solved only with the help of splitting. Some problems not solved without splitting could be solved with splitting in less than one second.

We believe that the performance of splitting without backtracking can be considerably improved, but optimized implementation, more experiments, and insights in the behavior of splitting are required.

References

BACHMAIR L. AND GANZINGER H. [2001], Resolution theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Elsevier Science and MIT Press. To appear.

BAUMGARTNER P. [2000], FDPLL — a first-order Davis-Putnam-Logemann-Loveland procedure, *in* D. McAllester, ed., '17th International Conference on Automated Deduction (CADE-17)', Vol. 1831 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Pittsburgh, pp. 200–219.

JOYNER, JR. W. H. [1976], 'Resolution strategies as decision procedures', *Journal of the ACM* **23**(3), 398–417.

RIAZANOV A. AND VORONKOV A. [1999], Vampire, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, pp. 292–296.

RIAZANOV A. AND VORONKOV A. [2000*a*], Limited resource strategy in resolution theorem proving, Preprint CSPP-7, Department of Computer Science, University of Manchester.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

RIAZANOV A. AND VORONKOV A. [2000*b*], Partially adaptive code trees, *in* M. Ojeda-Aciego, I. de Guzmán, G. Brewka and L. Pereira, eds, 'Logics in Artificial Intelligence. European Workshop, JELIA 2000', Vol. 1919 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Málaga, Spain, pp. 209–223.

SCHUMANN J. AND FISCHER B. [1997], NORA/HAMMR: Making deduction-based software component retrieval practical, *in* 'Proc. Automated Software Engineering (ASE-97)', IEEE Computer Society Press, Lake Tahoe, pp. 246–254.

SUTCLIFFE G. AND SUTTNER C. [1998], 'The TPTP problem library — CNF release v. 1.2.1', *Journal of Automated Reasoning* **21**(2).

VORONKOV A. [1995], 'The anatomy of Vampire: Implementing bottom-up procedures with code trees', *Journal of Automated Reasoning* **15**(2), 237–265.

WEIDENBACH C., AFSHORDEL B., BRAHM U., COHRS C., ENGEL T., KEEN E., THEOBALT C. AND TOPIC D. [1999], System description: SPASS version 1.0.0, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, pp. 378–382.

Index