Fully reusing clause deduction algorithm based on standard contradiction separation rule

Liu, P., Xu, Y., Liu, J., Chen, S., Cao, F., & Wu, G. (2023). Fully reusing clause deduction algorithm based on standard contradiction separation rule. *Information Sciences*, *622*, 337-356. Advance online publication. https://doi.org/10.1016/j.ins.2022.11.128

Link to publication record in Ulster University Research Portal

Published in:
Information Sciences

Publication Status:
Published (in print/issue): 30/04/2023

DOI:
10.1016/j.ins.2022.11.128

Document Version
Author Accepted version

General rights
Copyright for the publications made accessible via Ulster University's Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Ulster University's institutional repository that provides access to Ulster's research outputs. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact pure-support@ulster.ac.uk.

Download date: 06/12/2023

Fully Reusing Clause Deduction Algorithm Based on Standard Contradiction Separation Rule

Peiyao Liu [a,d], Yang Xu [a,d], Jun Liu [b,d], Shuwei Chen [a,d,*], Feng Cao [c,d], Guanfeng Wu [a,d]

[a] *School of Mathematics, Southwest Jiaotong University, Chengdu 611756, China*

[b] *School of Computing, Ulster University, Belfast BT15 1ED, Northern Ireland, UK*

[c] *School of Information Engineering, Jiangxi University of Science and Technology, Ganzhou 341000, China*

[d] *National-Local Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University, Chengdu 611756, China*

Abstract: An automated theorem proving (ATP) system's capacity for reasoning is significantly influenced by the inference rules it uses. The recently introduced standard contradiction separation (S-CS) inference rule extends binary resolution to a multi-clause, dynamic, contradiction separation inference mechanism. The S-CS rule is used in the present work to provide a framework for fully clause reusing deductions. Accordingly, a fully reusing clause deduction algorithm (called the FRC algorithm) is built. The FRC algorithm is then incorporated as an algorithm module into the architecture of a top ATP, Vampire, creating a single integrated ATP system dubbed V_FRC. The objective of this integration is to enhance Vampire's performance while assessing the FRC algorithm's capacity for reasoning. According to experimental findings, V_FRC not only outperforms Vampire in a variety of aspects, but also solves 46 problems in the TPTP benchmark database that have a rating of 1, meaning that none of the existing ATP systems are able to resolve them.

Keywords: Theorem proving; ATP system; Inference rule; Deduction algorithm; Standard contradiction; S-CS rule; Vampire

1. Introduction

Automated reasoning is a technical means of using computers to automatically verify mathematical theorems or computer software and hardware systems, etc. in the form of theorem proving [1], and it is an important part of the field of artificial intelligence. Automated theorem proving (ATP), as the core research area of automated reasoning, has achieved fruitful results during the years of development. The refutation inference of an ATP system for first-order logic (FOL) is

* Corresponding author at: School of Mathematics, Southwest Jiaotong University, Chengdu, China.
E-mail address: swchen@swjtu.edu.cn (S. Chen).

a process that eventually generates empty clause from an unsatisfiable clause set for FOL and its derivation clauses [2]. ATP systems have been applied in a wide range of fields, initially to prove mathematical problems, and with the proposal of new problems, techniques and ideas, they have then been applied to many other fields where the problems could be converted into logical form for ATP systems to process, such as program verification [3, 4] and knowledge representation [5, 6]. Inference rules have a big impact on efficiency and performance during the inference process. An inference rule, we might even say, determines the reasoning capability of an ATP system to a great extent. At present, ATP systems for FOL can be divided into three categories according to the inference rules they are based on: 1) ATP systems [7, 8] based on tableaux calculus [9]; 2) ATP systems [10-12] based on saturation algorithm [13] of binary resolution [14]; 3) ATP systems [15, 16] based on SAT/SMT theories [17, 18]. Binary resolution proposed by Robinson has become the most well-known inference method due to its simplicity, reliability and efficiency, and research on the ATP systems based on binary resolution is still the mainstream research trend in this field. During the past few decades, many scholars have proposed a mass of different variants of binary resolution [19-21], e.g., linear resolution [22], locking resolution [23], sematic resolution [24], hyper resolution [25], and heuristic strategies based on binary resolution [26-28].

The essential feature of inference methods based on binary resolution is that only two clauses involved in each deduction step, and only one complementary pair from parent clauses is eliminated [3]. This feature also implies that binary resolution is a static and binary inference method, and a mass of redundant clauses are generated during the inference process, which may cause the search space explosion problem [29]. Although the appearance of redundancy elimination techniques [30-32] reduces the search space to a certain extent, this leads to additional deduction overhead. In 2018, multi-clause standard contradiction separation (S-CS) rule for FOL was proposed [33], which theoretically broke through the limitation of binary resolution. S-CS rule offers multiple characteristics, such as multi-clause, dynamic and guidance, etc. [34]. The basic idea is that S-CS rule takes multiple clauses (two or more) as parent clauses, and selects multiple literals (one or more) from each parent clause to construct a contradiction, then a clause is inferred by taking the disjunction of the non-selected literals of the parent clauses [35]. A derivation clause from one deduction step of S-CS rule may require multiple deduction steps for binary resolution to obtain. Binary resolution is a special case of S-CS rule.

The use of the S-CS inference rule allows automated deduction to take “large steps” in the search space, and the sound and complete multi-clause dynamic automated S-CS deduction theory for first-order logic was introduced in [33]. However, the necessary proof search algorithms and strategies are still required to support this theory and enable automation implementation. Different deduction techniques and implementations were developed based on the various distinctive

properties of S-CS deduction theory. For example, Ref. [35] introduced a S-CS dynamic deduction algorithm (called SDDA). In order to further take advantage of the abilities of S-CS rule, especially guidance and synergy, this paper proposes a novel effective deduction framework for fully clause reusing based on the S-CS rule, then designs and implements a novel multi-clause dynamic deduction algorithm based on this deduction framework, which is one research objective of this present work.

Binary resolution and its variants are used by many ATP systems [10-12], including the top system, Vampire [36]. Vampire has been the champion of CADE ATP System Competition (CASC) [37] for nearly two decades. In the annual CASC, Vampire can solve the vast majority of 500 problems in FOF division. Vampire has so powerful performance, but there are still a lot of problems in the latest released version (TPTP-v7.5.0) of the TPTP (Thousands of Problems for Theorem Provers) [38] benchmark library of ATP systems, especially many hard problems with the rating of 1 that cannot be solved by Vampire [39]. Therefore, it is a very meaningful and challenging work to further improve the performance of Vampire, which is another research objective of this present work.

In order to achieve the two research objectives, this paper divides the research content into two parts. The first part in-deep analyzes the S-CS rule, then proposes a fully clause reusing deduction framework based on the S-CS rule and designs and implements a multi-clause dynamic deduction algorithm based on this deduction framework (in short, FRC algorithm). The second part integrates the FRC algorithm (as an independent algorithm module) into Vampire to generate a single integrated ATP system, denoted as V_FRC, to improve the performance of Vampire. The reasoning capability of V_FRC is evaluated through two experimental groups: 1) 2016-2021 CASC (FOF division) problems; 2) all problems with rating of 1 in TPTP-v7.5.0. Both the experimental results show that FRC algorithm effectively improves the deduction capability of Vampire, which also suggests that a deduction framework with full clause reuse is more suited for creating contradictions.

We note that a shorter conference version of this paper has been presented in FLINS 2022 [40]. Our initial conference paper neither introduced the algorithmic steps and pseudocode of the FRC algorithm, nor had sufficient experimental data to show the effectiveness of the FRC algorithm. This manuscript not only addresses these issues and provides a more adequate analysis on fully reusing clause deduction algorithm and the S-CS rule, but also provides more experiments and offers sufficient verification and analysis of the experiments results. The key contributions of this manuscript include: 1) a feasible implementation method for S-CS rule and an efficient multi-clause dynamic synergized deduction algorithm, i.e., FRC algorithm, are proposed; 2) FRC algorithm is applied to improve the performance of Vampire.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the

related terms of FOL and the S-CS rule. Section 3 deeply analyzes key characteristics of the S-CS rule, then introduces a fully clause reusing deduction framework based on the S-CS rule. In Section 4, a multi-clause dynamic deduction algorithm based on fully reusing clause framework, called FRC algorithm, is proposed and detailed. The heuristic strategies involved in the FRC algorithm are introduced in Section 5. Section 6 introduces the integrated architecture of incorporating the FRC algorithm to Vampire. Detailed experiments and the analysis of experimental results are provided in Section 7. Finally, Section 8 summarizes the research content of this paper and introduces future research work.

2. Preliminaries

This section first introduces the related concepts of first-order logic, then introduces multi-clause standard contradiction separation (S-CS) rule. We assume that the reader is already familiar with related concepts of FOL. This section only recalls some basic concepts, and the readers are referred to [33] for a detailed introduction.

FOL is a rich language with complex, hierarchical formulae, a large set of operators, and the use of quantifiers. In order to obtain more proof procedures, we restrict our discussion to conjunctive normal form (CNF) of FOL, a subset of first-order predicate that eliminates quantifiers and allows only conjunctions of clauses (which are disjunctions of elementary literals) as formulae. A literal is either an atom or a negated form, where an atom is an n-ary predicate (denoted P) with n terms. A term (denoted by t is either a variable (denoted x), a constant (denoted a), or an n-ary function (denoted f) with n terms. By the way, we use T to denote the set of all terms and use V to denote the set of all variables. A term is called a ground term if it contains no variables. A clause (denoted by C) is a disjunction of a finite set of literals (denoted by l). The empty clause with no literal is denoted by $\emptyset$. If clause C has only one literal, the clause is called a unit clause. A formula (denoted by S) is a conjunction of a finite clauses. A substitution (denoted by σ) is a mapping from V to T with the property that $\{x|\sigma(x) \neq x\}$ is finite.

Definition 1. [33] Let $S = \{C_1, C_2, \cdots, C_m\}$ be a clause set. The Cartesian product of $C_1, C_2, \cdots, C_m$, denoted as $\prod_{i=1}^{m} C_i$, is the set of all ordered tuples $(l_1, l_2, \cdots, l_m)$ such that $l_i \in C_i$ $(i = 1,2, \cdots, m)$, where l_i is a literal, and C_i is also regarded as a set of literals $(i = 1,2, \cdots, m)$.

Definition 2. (Contradiction) [33] Let $S = \{C_1, C_2, \cdots, C_m\}$ be a clause set. If $\forall(l_1, l_2, \cdots, l_m) \in \prod_{i=1}^{m} C_i$, there exists at least one complementary pair among $\{l_1, l_2, \cdots, l_m\}$, then $S = \bigwedge_{i=1}^{m} C_i$ is called a standard contradiction (in short, SC).

Example 1. Let $S = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ be a clause set, where $C_1 = \sim P_1(x_1) \vee P_3\big(x_1, f_1(x_1)\big)$, $C_2 = \sim P_1(x_2) \vee P_2(x_2) \vee P_4\big(f_1(x_2)\big)$, $C_3 = P_1(a)$, $C_4 = \sim P_3(a, x_3) \vee P_5(x_3)$, $C_5 = \sim P_5(x_4) \vee \sim P_2(x_4)$, $C_6 = \sim P_5(x_5) \vee \sim P_4(x_5)$. Here a is constant, f_1 is function symbol, x_1, x_2, x_3, x_4, x_5 are variables, P_1, P_2, P_3, P_4, P_5 are predicate symbols.

According to Definitions 1 and 2, $S^\sigma = C_1^\sigma \wedge C_2^\sigma \wedge C_3^\sigma \wedge C_4^\sigma \wedge C_5^\sigma \wedge C_6^\sigma$ is a standard contradiction (see Table 1), where $\sigma = \{a/x_1, a/x_2, f_1(a)/x_3, a/x_4, f_1(a)/x_5\}$.

Table 1

Standard contradiction in Example 1.

C_1^σ	C_2^σ	C_3^σ	C_4^σ	C_5^σ	C_6^σ
$\sim P_1(a)$ $\vee P_3(a, f_1(a))$	$\sim P_1(a) \vee P_2(a)$ $\vee P_4(f_1(a))$	$P_1(a)$	$\sim P_3(a, f_1(a))$ $\vee P_5(f_1(a))$	$\sim P_5(a) \vee \sim P_2(a)$	$\sim P_5(f_1(a))$ $\vee \sim P_4(f_1(a))$

Definition 3. [33] Suppose a clause set $S = \{C_1, C_2, \cdots, C_m\}$ in FOL. The following inference rule that produces a new clause from S is called a standard contradiction separation rule, in short, an S-CS rule:

For each C_i $(i = 1,2, \cdots, m)$, firstly apply a substitution σ_i to C_i (σ_i could be an empty substitution but not necessary the most general unifier), denoted as $C_i^{\sigma_i}$; then separate $C_i^{\sigma_i}$ into two sub-clauses $C_i^{\sigma_i -}$ and $C_i^{\sigma_i +}$ such that

1) $C_i^{\sigma_i} = C_i^{\sigma_i -} \vee C_i^{\sigma_i +}$, where $C_i^{\sigma_i -}$ and $C_i^{\sigma_i +}$ have no common literals;

2) $C_i^{\sigma_i +}$ can be an empty clause itself, but $C_i^{\sigma_i -}$ cannot be an empty clause;

3) $\bigwedge_{i=1}^m C_i^{\sigma_i -}$ is a standard contradiction, that is $\forall (x_1, x_2, \cdots, x_m) \in \prod_{i=1}^m C_i^{\sigma_i -}$, there exists at least one complementary pair among $\{x_1, x_2, \cdots, x_m\}$.

The resulting clause $\bigvee_{i=1}^m C_i^{\sigma_i +}$, denoted as $\mathbb{C}_m^{s\sigma}(C_1, \cdots, C_m)$, is called a standard contradiction separation clause (CSC) of $C_1, \cdots, C_m$, and $\bigwedge_{i=1}^m C_i^{\sigma_i -}$ is called a separated standard contradiction (SC).

Example 2. Let $S = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$ be a clause set, where $C_1 = \sim P_1(x_{11}, x_{12}, x_{13}) \vee \sim P_2(x_{11}, x_{13})$, $C_2 = P_1(x_{22}, x_{21}, x_{23}) \vee \sim P_1(x_{21}, x_{22}, x_{23})$, $C_3 = P_2(x_{31}, x_{34}) \vee \sim P_3(x_{31}) \vee \sim P_1(x_{32}, x_{33}, x_{34}) \vee \sim P_2(x_{31}, x_{32}) \vee \sim P_2(x_{31}, x_{33})$, $C_4 = P_1(x_{41}, x_{41}, f_1(x_{41}))$, $C_5 = P_1(a_1, f_1(a_1), f_1(a_2))$, $C_6 = P_3(a_1)$, $C_7 = P_2(a_1, a_2)$. Here a_i $(i = 1,2)$ is constant, f_1 is function symbol, x_i $(i = 11, \cdots, 41)$ is variable, P_i $(i = 1,2,3)$ is predicate symbol.

The clause set S applies the S-CS rule to separate the contradiction. Table 2 shows the result of clause separation, and Table 3 shows the corresponding substitution of the clause.

The SC is $C_1^{\sigma_1 -} \wedge C_2^{\sigma_2 -} \wedge C_3^{\sigma_3 -} \wedge C_4^{\sigma_4 -} \wedge C_5^{\sigma_5 -} \wedge C_6^{\sigma_6 -} \wedge C_7^{\sigma_7 -}$, and the CSC is $C_8 = \mathbb{C}_m^{s\sigma}(C_1, C_2, C_3, C_4, C_5, C_6, C_7) = P_1\big(f_1(a_1), a_1, f_1(a_2)\big)$, where $\sigma = \bigcup_{i=1}^7 \sigma_i$.

Table 2

Partition the $C_i^{\sigma_i}$ for Example 2

i	$C_i^{\sigma_i-}$	$C_i^{\sigma_i+}$
1	$\sim P_1(a_1, f_1(a_1), f_1(a_2)) \vee \sim P_2(a_1, f_1(a_2))$	$\emptyset$
2	$\sim P_1(a_1, f_1(a_1), f_1(a_2))$	$P_1(f_1(a_1), a_1, f_1(a_2))$
3	$\sim P_3(a_1) \vee \sim P_1(a_2, a_2, f_1(a_2)) \vee \sim P_2(a_1, a_2) \vee \sim P_2(a_1, a_2) \vee P_2(a_1, f_1(a_2))$	$\emptyset$
4	$P_1(a_2, a_2, f_1(a_2))$	$\emptyset$
5	$P_1(a_1, f_1(a_1), f_1(a_2))$	$\emptyset$
6	$P_3(a_1)$	$\emptyset$
7	$P_2(a_1, a_2)$	$\emptyset$

Table 3

Corresponding substitution σ_i of clause C_i for Example 2

i	σ_i
1	$\{a_1/x_{11}, f_1(a_1)/x_{12}, f_1(a_2)/x_{13}\}$
2	$\{a_1/x_{21}, f_1(a_1)/x_{22}, f_1(a_2)/x_{23}\}$
3	$\{a_1/x_{31}, a_2/x_{32}, a_2/x_{33}, f_1(a_2)/x_{34}\}$
4	$\{a_2/x_{41}\}$
5	$\emptyset$
6	$\emptyset$
7	$\emptyset$

The final deduction result (usually an empty clause $\emptyset$) is derived from a series of deduction steps. Definition 4 describes deduction sequence based on the S-CS rule.

Definition 4. [33] Suppose a clause set $S = \{C_1, C_2, \cdots, C_m\}$ in FOL. $\Phi_1, \Phi_2, \cdots, \Phi_t$ is called a standard contradiction separation based dynamic deduction sequence (S-CS deduction) from S to a clause Φ_t, denoted as D^S, if

1) $\Phi_i \in S$, $i \in \{1,2,\cdots,t\}$; or

2) there exist $r_1, r_2, \cdots, r_{k_i} < i$, $\Phi_i = \mathbb{C}_{k_i}^S(\Phi_{r_2}, \Phi_{r_2}, \cdots, \Phi_{r_{k_i}})$.

The soundness and completeness of S-CS deduction are guaranteed by the following two theorems.

Theorem 1. (Soundness) [33] Suppose a clause set $S = \{C_1, C_2, \cdots, C_m\}$ in FOL. $\Phi_1, \Phi_2, \cdots, \Phi_t$ is a S-CS based dynamic deduction from S to a clause Φ_t. If Φ_t is an empty clause, then S is unsatisfiable.

Theorem 2. (Completeness) [33] Suppose a clause set $S = \{C_1, C_2, \cdots, C_m\}$ in FOL. If S is unsatisfiable, then exists an S-CS based dynamic deduction from S to an empty clause.

3. Fully Clause Reusing Deduction Framework Based on S-CS Rule

S-CS rule separates a clause set $S = \{C_1, C_2, \cdots, C_m\}$ into two parts during the process of contradiction separation: contradiction and contradiction separation clause. For each clause $C_i \in S$

$(i = 1,2,\cdots,m)$, C_i is separates into C_i^- and C_i^+ by the S-CS rule. When $\bigwedge_{i=1}^m C_i^-$ is unsatisfiable, a corresponding contradiction is formed, and $\bigvee_{i=1}^m C_i^+$ is the contradiction separation clause.

The crucial point of constructing SC is how to separate each clause C_i participating the S-CS deduction into two parts, i.e., which literals in C_i should be selected into the SC and other non-selected literals joining the CSC, after the corresponding substitutions. In order to sequentially construct a SC, we need some literals from these literals in SC to form the "frame structure" of the contradiction which is similar to the frame structure of a building. There is one literal from each clause participating the S-CS deduction is called *decision literal* [41] that plays an important role on constructing SC. After each separation of one clause in S-CS deduction, a literal from this clause needs to be selected as a decision literal. A set of decision literals in SC is like the "frame structure" of the contradiction. In the present work, we use symbol D_l to denote the set of decision literals in SC.

In Definition 2, contradiction has a typical characteristic that any Cartesian product of the clause set S has at least one complementary pair. Therefore, it is a key factor of constructing SC that continuously searching literal complementary pairs from a given clause set. For a subsequent clause that is about to participate the S-CS deduction, the decision literal set D_l will determine which literals from this subsequent clause are put into SC according to *pairing condition* in Definition 5.

Definition 5. (Pairing condition) Suppose $\bigwedge_{i=1}^m C_i^{\sigma_i-}$ is a SC, and the set $D_l = \{l_{d1}, l_{d2}, \cdots, l_{dm}\}$ is a decision literal set in $\bigwedge_{i=1}^m C_i^{\sigma_i-}$, where $l_{di} \in C_i^{\sigma_i-}$, $i = 1,2,\cdots,m$ and σ_i is a substitution corresponding to C_i. Assume that there exists a literal l_p in a clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$ and a substitution θ, the literal l_p^θ can be put into the $C^{\theta-}$ when the clause C participates in the S-CS deduction. The following condition that l_p satisfies is called a *pairing condition*:

There exists a literal $l_{di} \in D_l$ and a substitution θ, such that $l_p^\theta = \sim l_{di}^\theta$, i.e., l_{di} and l_p can form a complementary pair after a substitution θ.

Remark 1. Any two literals in the decision literal set D_l cannot form a complementary pair.

If a clause currently does not have a literal that satisfies pairing condition, this clause cannot participate the S-CS deduction. In summary, decision literals set D_l not only determine clauses participating S-CS deduction, but also determines the literals of the SC. Next, a clause separation method during the S-CS deduction is introduced as follows.

Suppose $\bigwedge_{i=1}^m C_i^{\sigma_i-}$ is a constructed SC and $\bigvee_{i=1}^m C_i^{\sigma_i+}$ is the corresponding CSC, and the set $D_l = \{l_{d1}, l_{d2}, \cdots, l_{dm}\}$ is a decision literal set in $\bigwedge_{i=1}^m C_i^{\sigma_i-}$, where $l_{di} \in C_i^{\sigma_i-}$, $i = 1,2,\cdots,m$ and σ_i is a substitution corresponding to C_i. A clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$ as a subsequent clause that is about to participate the S-CS deduction. Then the process of separating clause C^θ into two parts $C^{\theta-}$ and $C^{\theta+}$ after a substitution $\theta = \bigcup_{j=1}^n \theta_j$ is shown as follows.

Step 1. If the literal $l_j^{\theta_j}$ $(j = 1,2,\cdots,n)$ satisfies pairing condition after a substitution θ_j, then it is put into $C^{\theta-}$; otherwise, $l_j^{\theta_j}$ is added to $C^{\theta+}$.

Step 2. If $C^{\theta+}$ has no literal or $C^{\theta+}$ satisfies some predefined conditions[1], then end the separation of clause C; otherwise, go to Step 3.

Step 3. Select a literal l_d from $C^{\theta+}$ to put into D_l as a new decision literal, then remove l_d from $C^{\theta+}$ and put l_d into $C^{\theta-}$.

SC $\bigwedge_{i=1}^{m} C_i^{\sigma_i-}$ will then be updated to $\bigwedge_{i=1}^{m+1} C_i^{\sigma_i-}$ where $C_{m+1}^{\sigma_{m+1}-} = C^{\theta-}$, since $C^{\theta-}$ is put into the SC. And CSC $\bigvee_{i=1}^{m} C_i^{\sigma_i+}$ also will be updated to $\bigvee_{i=1}^{m+1} C_i^{\sigma_i+}$ where $C_{m+1}^{\sigma_{m+1}+} = C^{\theta+}$, since $C^{\theta+}$ is put into the CSC.

Remark 2. The same clause or literal with different substitutions will be regarded as different clause or literal.

In implementation of the S-CS rule, the clause uses clause separation method to participate in the S-CS deduction. The following example illustrates the process of S-CS deduction.

Example 3. Let $S = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$ be a clause set, where

$C_1 = \sim P_1(x_{11}, x_{12}, x_{13}) \vee \sim P_2(x_{11}, x_{13})$, $C_2 = P_1(x_{22}, x_{21}, x_{23}) \vee \sim P_1(x_{21}, x_{22}, x_{23})$,

$C_3 = P_2(x_{31}, x_{34}) \vee \sim P_3(x_{31}) \vee \sim P_1(x_{32}, x_{33}, x_{34}) \vee \sim P_2(x_{31}, x_{32}) \vee \sim P_2(x_{31}, x_{33})$,

$C_4 = P_1(x_{41}, x_{41}, f_1(x_{41})) \vee \sim P_2(x_{41}, a_1)$,

$C_5 = \sim P_1(f_1(x_{51}), x_{51}, x_{52}) \vee P_1(x_{51}, a_2, x_{53}) \vee P_2(x_{53}, x_{51}, x_{53})$,

$C_6 = P_1(a_1, f_1(a_1), f_1(a_2))$, $C_7 = P_3(a_1)$, $C_8 = P_2(a_1, a_2)$.

Here a_i $(i = 1,2)$ is constant, f_1 is function symbol, x_i $(i = 11,\cdots,53)$ is variable, P_i $(i = 1,2,3)$ is predicate symbol.

Using the S-CS rule for 6 clauses $C_1, C_3, C_4, C_6, C_7, C_8$, we obtain a CSC involving 6 clauses: $C_9 = \mathbb{C}_m^{s\sigma_9}(C_1, C_3, C_4, C_6, C_7, C_8) = \sim P_2(a_2, a_1)$, while the corresponding SC is $C_1^{\sigma_1-} \wedge C_3^{\sigma_2-} \wedge C_4^{\sigma_3-} \wedge C_6^{\sigma_4-} \wedge C_7^{\sigma_5-} \wedge C_8^{\sigma_6-}$ in Table 4 that shows the result of clause separation. Table 5 shows the corresponding substitution and decision literal of the clause.

Table 4

Partition the $C_i^{\sigma_i}$ for Example 3

i	$C_i^{\sigma_i-}$	$C_i^{\sigma_i+}$
7	$P_3(a_1)$	$\emptyset$
8	$P_2(a_1, a_2)$	$\emptyset$
6	$P_1(a_1, f_1(a_1), f_1(a_2))$	$\emptyset$
1	$\sim P_1(a_1, f_1(a_1), f_1(a_2)) \vee \sim P_2(a_1, f_1(a_2))$	$\emptyset$
3	$P_2(a_1, f_1(a_2)) \vee \sim P_3(a_1) \vee \sim P_1(a_2, a_2, f_1(a_2)) \vee \sim P_2(a_1, a_2) \vee \sim P_2(a_1, a_2)$	$\emptyset$
4	$P_1(a_2, a_2, f_1(a_2))$	$\sim P_2(a_2, a_1)$

[1] The predefined conditions are introduced in Section 5.

Table 5

Corresponding substitution σ_i and decision literal of clause C_i for Table 4

i	σ_i	Decision literal
7	$\emptyset$	$P_3(a_1)$
8	$\emptyset$	$P_2(a_1, a_2)$
6	$\emptyset$	$P_1(a_1, f_1(a_1), f_1(a_2))$
1	$\{a_1/x_{11}, f_1(a_1)/x_{12}, f_1(a_2)/x_{13}\}$	$\sim P_2(a_1, f_1(a_2))$
3	$\{a_1/x_{31}, a_2/x_{32}, a_2/x_{33}, f_1(a_2)/x_{34}\}$	$\sim P_1(a_2, a_2, f_1(a_2))$
4	$\{a_2/x_{41}\}$	$\emptyset$

In addition to multi-clause and dynamic (the details are shown in [34]), there are several other advantages that can be summarized below:

1) *Guided deduction*. The clauses in SC can guide subsequent deduction paths, i.e., the clauses in SC can determine the selection of subsequent clauses of S-CS deduction. The clauses in SC actually rely on their decision literals to play their guiding role.

2) *Clause-reusing deduction*. In one S-CS deduction, i.e., the process of constructing a SC, one clause can be reused, such that this clause may generate multiple different decision literals.

3) *Controllable deduction*. In the process of a S-CS deduction, the number of clauses and the number of literals in SC, i.e., the size of the contradiction, can be controlled. On the other hand, the controllability of S-CS deduction is achieved through the number of literals or the features of literals in CSC. For example, if the deduction only requires the CSC with one literal, when the deduction generates the CSC with one literal, this S-CS deduction process will stop. Or if the deduction requires the CSC without equality, when the deduction generates the CSC with one equality, this S-CS deduction process will stop.

4) *Synergized deduction*. For an unsatisfiable problem in FOL, its unsatisfiable property is more difficult to judge by several resolution deduction steps than several multi-clause S-CS deduction steps. The generated CSC of each S-CS deduction step is the result of the participation of multiple clauses, and therefore S-CS deduction reflects the overall synergized logical relationship between the clauses.

Because of the aforesaid advantages, a single S-CS deduction eliminates more literals than a single resolution deduction, and the number of literals in the CSC can be limited to the number required for the deduction. The SC, on the other hand, is typically greater in size, but the CSC has fewer literals. The final goal of S-CS deduction for an unsatisfiable problem in FOL is to construct an empty clause, hence the fewer literals in the CSC, the easier it is to generate an empty clause.

As a result, the fact that extra literals from a following clause can be inserted into SC is useful for creating SC. We can extrapolate from the above analysis that the more different literals in D_l, the easier it is to produce the empty clause $\emptyset$. A clause should be utilized numerous times in a built SC to give D_l more distinct literals. Because a clause might have many literals, the decision literal must be one of them. In D_l, on the other hand, a single literal can form a complimentary pair with

several literals. This is the main motivation for reusing clauses.

We use Example 4 to illustrate the performance of reusing clauses.

Example 4. If reusing some clauses, then the clause set S of Example 3 can deduce an empty clause by constructing only one SC.

Using S-CS rule for 8 clauses $C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8$, we directly obtain the empty clause $\emptyset$. Clauses C_1 and C_2 are each reused twice continuously. Table 6 shows the result of clause separation. Table 7 shows the corresponding substitution and decision literal of the clause.

Table 6

Partition the $C_i^{\sigma_i}$ for Example 3.3

i	$C_i^{\sigma_i-}$	$C_i^{\sigma_i+}$
7	$P_3(a_1)$	$\emptyset$
8	$P_2(a_1, a_2)$	$\emptyset$
6	$P_1(a_1, f_1(a_1), f_1(a_2))$	$\emptyset$
1_1[2]	$\sim P_1(a_1, a_2, a_2) \vee \sim P_2(a_1, a_2)$	$\emptyset$
1_2	$\sim P_1(a_1, f_1(a_1), f_1(a_2)) \vee \sim P_2(a_1, f_1(a_2))$	$\emptyset$
2_1	$P_1(a_1, a_2, a_2) \vee \sim P_1(a_2, a_1, a_2)$	$\emptyset$
2_2	$\sim P_1(a_1, f_1(a_1), f_1(a_2)) \vee P_1(f_1(a_1), a_1, f_1(a_2))$	$\emptyset$
3	$P_2(a_1, f_1(a_2)) \vee \sim P_3(a_1) \vee \sim P_1(a_2, a_2, f_1(a_2)) \vee \sim P_2(a_1, a_2) \vee \sim P_2(a_1, a_2)$	$\emptyset$
4	$P_1(a_2, a_2, f_1(a_2)) \vee \sim P_2(a_2, a_1)$	$\emptyset$
5	$\sim P_2(a_2, a_1) \vee \sim P_1(a_1, a_2, a_2) \vee P_1(a_2, a_1, a_2) \vee \sim P_1(f_1(a_1), a_1, f_1(a_2))$	$\emptyset$

Table 7

Corresponding substitution σ_i and decision literal of clause C_i for Table 6

i	σ_i	Decision literal
7	$\emptyset$	$P_3(a_1)$
8	$\emptyset$	$P_2(a_1, a_2)$
6	$\emptyset$	$P_1(a_1, f_1(a_1), f_1(a_2))$
1_1	$\{a_1/x_{11}, a_2/x_{12}, a_2/x_{13}\}$	$\sim P_1(a_1, a_2, a_2)$
1_2	$\{a_1/x_{11}, f_1(a_1)/x_{12}, f_1(a_2)/x_{13}\}$	$\sim P_2(a_1, f_1(a_2))$
2_1	$\{a_2/x_{21}, a_1/x_{22}, a_2/x_{23}\}$	$\sim P_1(a_2, a_1, a_2)$
2_2	$\{a_1/x_{21}, f_1(a_1)/x_{22}, f_1(a_2)/x_{23}\}$	$P_1(f_1(a_1), a_1, f_1(a_2))$
3	$\{a_1/x_{31}, a_2/x_{32}, a_2/x_{33}, f_1(a_2)/x_{34}\}$	$\sim P_1(a_2, a_2, f_1(a_2))$
4	$\{a_2/x_{41}\}$	$\sim P_2(a_2, a_1)$
5	$\{a_1/x_{51}, f_1(a_2)/x_{52}, a_2/x_{53}\}$	$\emptyset$

In the S-CS deduction, how many times does a subsequent clause that is about to participate in the S-CS deduction need to be reused properly? The concept of *repetition value* is therefore introduced as follows.

Definition 6. Two literals l_1 with n-ary predicate and l_2 with n-ary predicate can form *predicate complementary*, if the predicate symbol of l_1 is negated form of the predicate symbol of l_2.

Definition 7. (Repetition value) Suppose $\bigwedge_{i=1}^{m} C_i^{\sigma_i-}$ is a SC, and the set $D_l = \{l_{d1}, l_{d2}, \cdots, l_{dm}\}$ is a

[2] Because the same clause can participate in the deduction multiple times under different substitutions during the deduction process, in order to distinguish the clauses, the manuscript uses the index such as "1_1". Specifically, the index "1_1" means that the clause C_1 participates in the deduction for the first time in this S-CS deduction step, the former "1" denotes the index of clause C_1, and the latter "1" denotes the times that clause C_1 participates in the deduction.

decision literal set in $\bigwedge_{i=1}^{m} C_i^{\sigma_i -}$, where $l_{di} \in C_i^{\sigma_i -}$, $i = 1,2,\cdots,m$ and σ_i is a substitution corresponding to C_i. There is a subsequent clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$ that is about to participate the S-CS deduction. *Literal repetition value* is defined as follows.

For any literal $l_p \in C$, if there are k literals in the set D_l that can respectively form a predicate complementary with literal l_p, then the literal repetition value of l_p is $LRv(l_p) = k$.

Clause repetition value of clause C is $CRv(C) = \sum_{i=1}^{m} k_i$, where $LRv(l_i) = k_i$ $(i = 1,2,\cdots,n)$.

The initial value of each clause and literal in a clause set is 0. For a subsequent clause that is about to participate in the S-CS deduction, if its $CRv(C) = u$, then this clause needs to be reused u times, which is the main idea of fully reusing clause. After a clause has been fully used, the SC will obtain multiple (one or more) decision literals. On the other hand, the number of decision literals from a clause is regarded as a measure of deductive performance of this clause. The question of how to achieve the maximum deductive performance of each clause can be transformed into how to fully reuse each clause. The idea of fully reusing clauses can be described as follows.

1) Each clause in the original clause set needs to be fully reused. It means that each clause in the original clause set should participate in the S-CS deduction at least once.

2) Each literal in a clause needs to be fully reused. For a clause participating the S-CS deduction, i.e., constructing the SC and the separation of the clause, there exists a literal from this clause whose literal repetition value $LRv = k$, then this literal needs to be reused k times where the SC could obtain k different decision literals. There are more decision literals in SC, the situation where all literals of a clause that is about to participate in the deduction satisfy pairing condition are more likely to occur.

Of course, since the running time and memory resources of an ATP system are limited, it is not feasible for all clauses in a clause set to be fully reused. Meanwhile, the process of reusing a clause may generate many redundant clauses or make the number of literals in the CSC grow too fast. Therefore, we selected some clauses from the clause set for fully reusing according to heuristic strategies that will be detailed in Section 5, these selected clauses are called *eligible-reuse clause*s. Unit clauses and binary clauses are easier to reuse than other clauses due to their nature (fewer number of literals and less complexity of deduction, etc.). A fully clause reusing deduction framework is described in the following steps for the implementation purpose.

Given a clause set $S = \{C_1, C_2, \cdots, C_m\}$ in FOL, D_l denotes the set of decision literal, G_2 denotes the set of all 2-ary clauses in S and G_3 denotes the set of all t-ary clauses $(t \geq 3)$ in S, whose initialization are empty.

Step 1. All unit clauses in S are put into D_l, all binary clause in S are put into G_2, and all non-unit and non-binary clauses in S are put into G_3.

Step 2. Traverse each clause C_2 of G_2. Count the clause repetition value $CRv(C_2)$ of C_2, and

C_2 is reused $CRv(C_2)$ times by applying clause separation method. Each separation of C_2 generates a new clause C_R. If C_R is not $\emptyset$, then it is added to S; otherwise, output **UNSAT**, and Exit!

Step 3. End traverse of G_2. Mark the eligible-reuse clauses in G_3 according to the heuristic strategy. Traverse each clause C_3 of G_3. If C_3 is eligible-reuse clause, then count the clause repetition value $Rv(C_3)$ of C_3, and C_3 is reused $CRv(C_2)$ times by applying clause separation method; otherwise, C_3 is used only once by applying clause separation method. Each clause separation generates a new clause C_R. If C_R is not $\emptyset$, then it is added to S; otherwise, output **UNSAT**, and Exit!

Step 4. End traverse of G_3. Exit!

The heuristic strategies mentioned above will be introduced in Section 5.

4. A Novel Deduction Algorithm Based on Fully Clause Reusing Deduction Framework

To obtain an effective implementation of this deduction framework, we need to develop a feasible deduction algorithm according to this framework. Therefore, we propose a novel deduction algorithm based on fully clause reusing deduction framework (in short FRC algorithm). FRC algorithm is not only a deduction algorithm based on the S-CS rule, but also an algorithm for constructing the contradiction. The overall flow of the FRC algorithm is shown in Fig. 1.

In Fig. 1, the left swim lane denotes the main routine of the FRC algorithm, the middle swim lane denotes a subroutine of the FRC algorithm, i.e., fully_reusing_set, and the right swim lane denotes a subroutine of fully_reusing_set, i.e., fully_reusing_single_clause. Specifically, the subset $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, $\boldsymbol{G_3}$ of the clause set $\boldsymbol{P}$ denotes unit clause subset, binary clause subset and other clause subset respectively. More specific details of the FRC algorithm are introduced as follows.

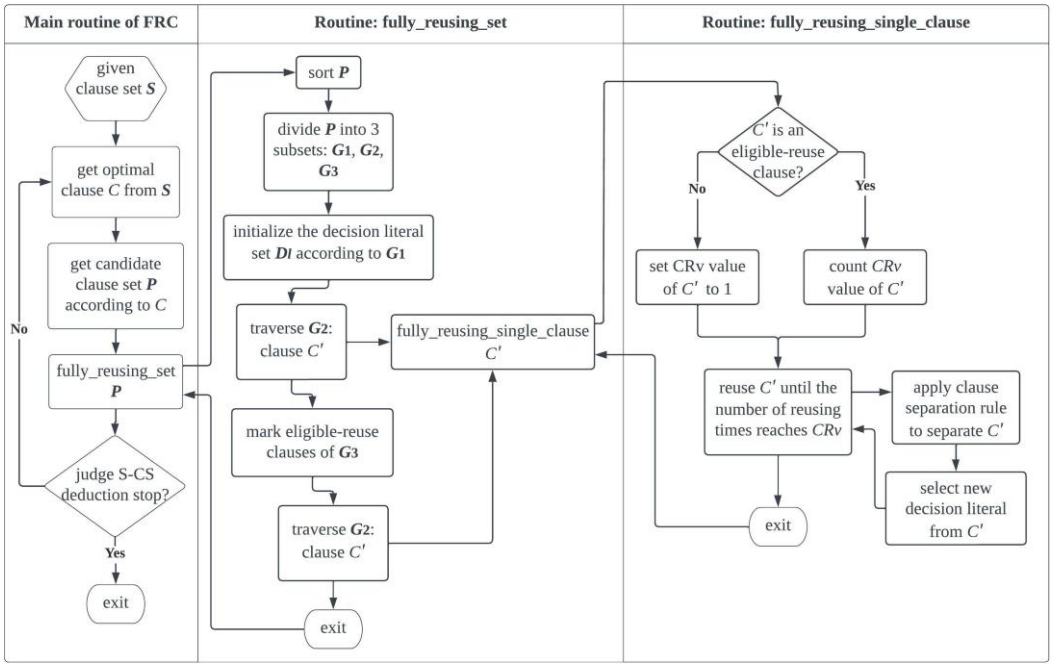

Fig. 1. The overall flow of FRC algorithm

Firstly, the main routine of the FRC algorithm is described in the following steps:

Step 1. Given a clause set $\boldsymbol{S}$, select a clause C from $\boldsymbol{S}$, which will be the start clause of the S-CS deduction.

Step 2. Select all clauses from $\boldsymbol{S}$ whose literals can form complementary pairs with the literals of clause C into the clause set $\boldsymbol{P}$.

Step 3. Enter the iterative process of S-CS deduction.

Step 4. Execute the subroutine fully_reusing_set, i.e., the clause set $\boldsymbol{P}$ and the clause C are fully reused to generate CSCs and construct the contradiction, and the CSCs are temporary stored in the clause set $\boldsymbol{R}$.

Step 5. Check whether $\boldsymbol{R}$ is an empty set. If $\boldsymbol{R}$ is an empty set, the proof found (UNSAT) and S-CS deduction stop. Exit!

Step 6. Check whether current deduction satisfies other stop conditions of S-CS deduction. If a deduction stop condition is met, go to Step 8.

Step 7: Empty the clause set $\boldsymbol{P}$. Select a clause C from clause set $\boldsymbol{S}$. Select all clauses from $\boldsymbol{S}$ whose literals can form complementary pairs with the literals of clause C into the clause set $\boldsymbol{P}$, then go to Step 3.

Step 8. Exit the iterative process of S-SC deduction.

Step 9. Traverse each clause C' of $\boldsymbol{R}$, remove the clauses from $\boldsymbol{S}$ which are subsumed by clause C'.

Step 10. All clauses of clause set $\boldsymbol{R}$ are put into clause set $\boldsymbol{S}$. Then output clause set $\boldsymbol{S}$. Exit!

The pseudo-code for the main routine of the FRC algorithm is shown in Algorithm 1, and the explanations of the subroutines are detailed in Table 8.

Algorithm 1. The pseudo-code for the main routine of FRC algorithm

Input

$\boldsymbol{S}$: set of given clauses
$\boldsymbol{P}$: temporary store for pending clauses (the initial is empty)
$\boldsymbol{R}$: temporary store for newly generated clauses (the initial is empty)
$\boldsymbol{D_l}$: set of decision literals (the initial is empty)
C: the optimal clause with respect to some heuristic strategies (the initial is null)
C': temporary handle clause (the initial is null)

Output

clause set $\boldsymbol{S}$ containing newly generated clauses

```
1: C = select_optimal(S);
2: P = select_set(C, S);
3: while P ≠ ∅ begin
4:   R = fully_reusing_set(C, P, D_l);
5:   if R == ∅
6:     proof found; exit
7:   if TRUE == judge_break(R)
8:     goto 13
9:   P = ∅;
10:  C = select_optimal(S);
11:  P = select_set(C, S);
12:  P = P ∪ {C};
13: end while
14: foreach C' ∈ R
15:  S = S \ backward_subsumption(C', S);
16: S = S ∪ R;
```

Table 8

Subroutines for the pseudo-code in Algorithm 1

Subroutine name	Function
select_optimal($\boldsymbol{S}$)	Return the optimal (with respect to some heuristic strategies) clause from $\boldsymbol{S}$.
select_set(C, $\boldsymbol{S}$)	Return the clause set that all clauses from $\boldsymbol{S}$ whose literals can form literal complementary pairs with the literals of clause C.
fully_reusing_set(C, $\boldsymbol{P}$, $\boldsymbol{D_l}$)	The clause set $\boldsymbol{P}$ and the clause C are fully used to generate CSCs and construct the contradiction based on fully reusing clause principle, then return the clause set consisting of newly generated clauses (CSCs).
judge_break($\boldsymbol{R}$)	Return true if and only if the clause set $\boldsymbol{R}$ satisfies the deduction exit condition (with respect to some heuristic strategies).
backward_subsumption(C', $\boldsymbol{S}$)	Return the clauses from $\boldsymbol{S}$ which are subsumed by clause C'.

The subroutine fully_reusing_set(C, $\boldsymbol{P}$, $\boldsymbol{D_l}$) can be further described with more details as follows.

Step 1. Sort clauses in clause set $\boldsymbol{P}$ according to the heuristic strategy and divide $\boldsymbol{P}$ into three subsets $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, $\boldsymbol{G_3}$, where G_1 stores 1-ary clauses, $\boldsymbol{G_2}$ stores 2-ary clauses and $\boldsymbol{G_3}$ stores t-ary clauses ($t \geq 3$) clauses.

Step 2. Put all literals of each clause in the set $\boldsymbol{G_1}$, i.e., unit clause set, into the decision literal set $\boldsymbol{D_l}$ as decision literals.

Step 3. If the decision literal set $\boldsymbol{D_l}$ is an empty set, then select one literal (with respect to the

Algorithm 2. The pseudo-code for the subroutine fully_reusing_set(C, $\boldsymbol{P}$, $\boldsymbol{D_l}$)

Input

$\boldsymbol{P}$: store for pending clauses
$\boldsymbol{G_1}$: store for 1-ary clauses (the initial is empty)
$\boldsymbol{G_2}$: store for 2-ary clauses (the initial is empty)
$\boldsymbol{G_3}$: store for t-ary clauses ($t \geq 3$) clauses (the initial is empty)
$\boldsymbol{R}$: store for newly generated clauses (the initial is empty)
$\boldsymbol{R'}$: temporary store for newly generated clauses (the initial is empty)
$\boldsymbol{D_l}$: set of decision literals (the initial is empty)
C: begin clause
C': temporary handle literal (the initial is null)

Output

newly generated clause set $\boldsymbol{R}$

```
1: sort_set(P);
2: classify_set(P, G1, G2, G3);
3: initialize_decisionSet(G1, Dl);
4: if Dl == ∅
5:   select_decision(C, Dl);
6: else
7:   R' = fully_reusing_single_clause(C, Dl);
8:   if R' == ∅ goto 23;
9:   R = R ∪ R';
10: foreach C' ∈ G2
11:   R' = ∅;
12:   R' = fully_reusing_single_clause(C', Dl);
13:   if R' == ∅ goto 23;
14:   R = R ∪ R';
15: end for
16: mark_eligible-reuse(G3);
17: foreach C' ∈ G3
18:   R' = ∅;
19:   R' = fully_reusing_single_clause(C', Dl);
20:   if R' == ∅ goto 23;
21:   R = R ∪ R';
22: end for
23: return R;
```

heuristic strategy) from the clause c as a new decision literal and put the new decision literal into $\boldsymbol{D_l}$ and the contradiction, go to Step 5; otherwise, go to Step 4.

Step 4. Execute the subroutine fully_reusing_single_clause, i.e., C is fully reused to participate in the S-CS deduction, then obtain the newly generated clauses. If there is an empty clause in these newly generated clauses, then go to Step 12; otherwise, put these newly generated clauses into the clause set $\boldsymbol{R}$ which stores newly generated clauses.

Step 5. Traverse each clause C' in the clause $\boldsymbol{G_2}$.

Step 6. Execute the subroutine fully_reusing_single_clause, i.e., C' is fully reused to participate in the S-CS deduction, then obtain the newly generated clauses. If there is an empty clause in these newly generated clauses, then go to Step 12; otherwise, put these newly generated clauses into the clause set $\boldsymbol{R}$.

Step 7. End traverse of $\boldsymbol{G_2}$.

Step 8. Mark the eligible-reuse clause in $\boldsymbol{G_3}$ with respect to some heuristic strategies.

Step 9. Traverse each clause C' in the clause $\boldsymbol{G_3}$.

Step 10. Execute the subroutine fully_reusing_single_clause, i.e., C' is fully reused to participate in the S-CS deduction, then obtain the newly generated clauses. If there is an empty clause in these newly generated clauses, then go to Step 12; otherwise, put these newly generated clauses into the clause set $\boldsymbol{R}$.

Step 11. End traverse of $\boldsymbol{G_3}$.

Step 12. Output the clause set R.

The pseudo-code for the subroutine fully_reusing_set(C, $\boldsymbol{P}$, $\boldsymbol{D_l}$) is shown in Algorithm 2, and the explanations of the subroutines of the pseudo-code can be found as in Table 9.

Table 9

Subroutines for the pseudo-code in Algorithm 2

Subroutine name	Function
sort_set($\boldsymbol{P}$)	Sort clauses (with respect to some heuristic strategies) in clause set $\boldsymbol{P}$.
classify_set $\boldsymbol{P}$, $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, $\boldsymbol{G_3}$)	$\boldsymbol{P}$ is divided into three subsets $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, $\boldsymbol{G_3}$, where $\boldsymbol{G_1}$ stores 1-ary clauses, $\boldsymbol{G_2}$ stores 2-ary clauses and $\boldsymbol{G_3}$ stores t-ary clauses ($t \geq 3$) clauses.
initialize_decisionSet($\boldsymbol{G_1}$, $\boldsymbol{D_l}$)	All literals of $\boldsymbol{G_1}$ are put into $\boldsymbol{D_l}$ as decision literals.
select_decision(C, $\boldsymbol{D_l}$)	Select one literal (with respect to some heuristic strategies) from the clause C as a new decision literal and put the new decision literal into $\boldsymbol{D_l}$ and the contradiction, then non-selected literals are put into the CSC.
fully_reusing_single_clause(C, $\boldsymbol{D_l}$)	Clause C is fully reused to participate in the S-CS deduction, return a newly generated clause set.
mark_eligible-reuse($\boldsymbol{G_3}$)	Mark the eligible-reuse clause (with respect to some heuristic strategies) in $\boldsymbol{G_3}$.

The subroutine fully_reusing_single_clause(C, $\boldsymbol{D_l}$) is detailed in the following steps.

Step 1. If the given clause C is an eligible-reuse clause, go to Step 2; otherwise, go to Step 5.

Step 2. Count the repetition value Rv of the clause C. Reuse the clause C until the number of uses reaches Rv.

Step 3. During each reusing process of the clause C, it participates in the S-CS deduction by applying clause separation method, then a new generated clause C' is generated.

Step 4. Check whether the clause C' is subsumed by the clause in the clause set $\boldsymbol{S}$ (given in Algorithm 1), or clause C' is a tautology, or the deduction is noneffective. If one of the two conditions is satisfied, the backtracking operation is performed (please see Table 10 for the explanation of backtracking); Otherwise, the clause C' is put into the clause set $\boldsymbol{R}$ where newly generated clauses store, and select one literal (with respect to the heuristic strategy) from the clause C as a new decision literal, that is then put into $\boldsymbol{D_l}$ and the contradiction. Go to Step 7.

Step 5. The clause C participates in the S-CS deduction by applying clause separation method, then a new generated clause C' is generated.

Step 6. Check whether the clause C' is subsumed by the clause in the clause set $\boldsymbol{S}$ (given in Algorithm 1), or clause C' is a tautology, or the deduction is noneffective. If one of the two conditions is satisfied, the backtracking operation is performed (please see Table 10 for the explanation of backtracking); Otherwise, the clause C' is put into the clause set $\boldsymbol{R}$ that stores newly

generated clauses, and select one literal (with respect to the heuristic strategy) from the clause C as a new decision literal, that is then put into $\boldsymbol{D_l}$ and the contradiction.

Step 7. Output the clause set R.

Algorithm 3 shows the pseudo-code for the subroutine fully_reusing_single_clause(C, $\boldsymbol{D_l}$), and Table 10 explains each subroutines of the pseudo-code.

Algorithm 3. The pseudo-code for the subroutine fully_reusing_single_clause(C, $\boldsymbol{D_l}$)

Input

C: the given clause
$\boldsymbol{D_l}$: set of decision literals
$\boldsymbol{S}$: the clause set in Algorithm 1
$\boldsymbol{R}$: store for newly generated clauses (the initial is empty)
C': newly generated clause (the initial is null)

Output

newly generated clause set $\boldsymbol{R}$

```
1: if TRUE == judge_eliglible-reuse(C)
2:   Rv = count_repetitionValue(C, D_l);
3:   while Rv ≠ 0 begin
4:     C' = separate_clause(C, D_l);
5:     if forward_subsumption(C', S) or tautology(C', S) or evaluate_deduction(C')
6:       backtracking(C');
7:     else
8:       R = R ∪ {C'};
9:       select_decision(C, D_l);
10:    Rv = Rv − 1;
11:  end while
12: else
13:  C' = separate_clause(C, D_l);
14:  if forward_subsumption(C', S) or tautology(C', S) or evaluate_deduction(C')
15:    backtracking(C');
16:  else
17:    R = R ∪ {C'};
18:    select_decision(C, D_l);
19: return R;
```

Table 10

Subroutines for the pseudo-code in Algorithm 3

Subroutine name	Function
judge_eliglible-reuse(C)	Return true if and only if the clause C is an eliglible-reuse clause.
count_repetitionValue(C, $\boldsymbol{D_l}$)	Count the clause repetition value Rv of the clause C, then return Rv.
separate_clause(C, $\boldsymbol{D_l}$)	The clause C participates in the S-CS deduction by applying clause separation rule, then obtain a newly generated clause C'. Return the clause C'.
forward_subsumption(C', $\boldsymbol{S}$)	Return true if and only if the clause C' is subsumed by one clause from the clause set $\boldsymbol{S}$.
tautology(C', $\boldsymbol{S}$)	Return true if and only if the clause C' is a tautology clause.
evaluate_deduction(C')	Return true if and only if the deduction of the clause C' participation is effective (with respect to some heuristic strategies)
backtracking(C')	Clear the record of proof search by the clause C', and remove literals of the clause c' in the CSC and the contradiction, and clear substitutions which are caused by the clause C'.
select_decision(C, $\boldsymbol{D_l}$)	Shown in Table 9.

The heuristic strategies involved in the FRC algorithm will be introduced in Section 5.

5. Heuristic Strategies of FRC Deduction Algorithm

For FOL, ATP systems search for proof in an infinite search space. This search is typically guided by heuristic strategies that select the most promising one in deduction paths. The proof search efficiency is a crucial factor that a deduction algorithm must consider. Up to now, hundreds of effective heuristic strategies [43-45], including those based on machine learning methods [46, 47], have been proposed by researchers. The strategy of each ATP system is developed based on the inference rule of the ATP system and requires extensive experiments to fix. In the subsequent section, we introduce the heuristic strategies involved in the FRC deduction algorithm. According to Algorithm 1, Algorithm 2 and Algorithm 3, the heuristic strategies involved in the FRC algorithm mainly are applied in clause selection, literal selection, and deduction evaluation respectively.

5.1. Related Thresholds and Measures

Before describing related heuristic strategies, we introduce several thresholds and measures of clause or literal, which can provide some reference for setting of heuristic strategies.

A. Related thresholds

(1) The pre-defined threshold $maxTD$, i.e., the maximum term depth, generally is set $1.7 * t$ by default according to our empirical experience, where t is the maximum term depth in the original clause set.

(2) The pre-defined threshold $maxLN$, i.e., the maximum number of literals in S-CSC, generally is set $1.5 * n$ by default according to our empirical experience, where n is the maximum number of literals of a clause in the original clause set.

(3) The threshold $leftLN$, i.e., the maximum number of literals in part C^+ of a clause C after participating the S-CS deduction. To make the deduction more flexible, this threshold varies during the deduction process. According to our empirical experience, $leftLN$ is set to 1 at the beginning of the deduction, then gradually increased by 1, but generally is not greater than 4.

B. Related measures

(1) Clause weight

This weight has three arguments: a weight for function symbol w_f, a weight for variable symbol w_v, and a weight for constant symbol w_c. It returns the sum of the term weight in all literals of a clause. In general, $w_f = w_v = w_c = 1$, and these three arguments also vary with deduction process. For example, $w_f = w_v = 1, w_c = 2$ if the ground clause needs to be selected; $w_f = 2, w_v = w_c = 1$ if the clause with less function symbols is wanted.

(2) Literal weight

Like clause weight, literal weight also has three arguments: a weight for function symbol w_f,

a weight for variable symbol w_v, and a weight for constant symbol w_c. It returns the sum of the term weight in a literal.

(3) Efficient-reusing weight

Definition 8. The *efficient-reusing weight* of a clause C under substitution σ is defined as

$$ErW(C,\sigma) = \frac{1}{N_R} + \frac{NC_A - NC_B}{NC_B} - \frac{Nf_A - Nf_B}{Nf_B}. \quad (1)$$

where N_R is the number of literals in clause C separated into the CSC after substitution, NC_A and NC_B are the number of constants in clause C after and before substitution respectively, Nf_A and Nf_B are the number of functions in clause C after and before substitution respectively.

The efficient-reusing weight of a literal under substitution σ is defined similarly, so we do not add details here. The efficient-reusing weight provides a way to order clauses when they are reused. In general, the efficient-reusing weight queue is sorted in a descending order, i.e., the clause with larger efficient-reusing weight is preferentially reused. Therefore, larger efficient-reusing weight allows a literal with lower function nesting layer as new decision literal.

(4) Invalid weight and valid weight

In addition to efficient-reusing weight, we consider the concept of *invalid separation*.

Definition 9. The separation of a clause is called an *invalid separation* if this clause after the separation does not satisfy two pre-defined thresholds, $maxTD$ (the maximum term depth) or $maxLN$ (the maximum number of literals in CSC). Accordingly, *invalid weight* of clause C is defined as follow.

$$IW(C) = Itimes(C) + \frac{TD(C)}{maxTD} + \frac{LN(C)}{maxLN}. \quad (2)$$

where $Itimes(C)$ is the number of invalid separations in which clause C has participated (the separation of clause C is an invalid separation once, $Itimes$ of clause C is increased by 1), $TD(C)$ is the max term depth in part C^+ of C after this invalid separation, and $LN(C)$ is the number of literals in part C^+ of C after this invalid separation.

Accordingly, the opposite concept of invalid weight is *valid weight*.

Definition 10. The separation of a clause is called a *valid separation* if this clause after the separation satisfies two pre-defined thresholds, $maxTD$ (the maximum term depth) or $maxLN$ (the maximum number of literals in CSC). Accordingly, *valid weight* of clause C is defined as follow.

$$VW(C) = Vtimes(C) + \frac{maxTD}{TD(C)} + \frac{maxLN}{LN(C)}. \quad (3)$$

where $Vtimes(C)$ is the number of valid separations in which clause C has participated (the

separation of clause C is a valid separation once, $Vtimes$ of clause C is increased by 1), $TD(C)$ is the max term depth in part C^+ of C after this valid separation, and $LN(C)$ is the number of literals in part C^+ of C after this valid separation.

(5) Goal distance

A clause set for FOL consists of multiple premise clauses and a negated-conjecture clause, a negated-conjecture clause, or a clause a goal whose literals are all negative literals is called a goal clause. Therefore, the deduction path can be set to be goal oriented. The literals in the goal clause are called *goal literal*s. Therefore, two concepts, such as *goal distance tree* and *goal distance* emerge which measure the distance from a literal or a clause to the goal clause.

Definition 11. (Goal distance tree) A goal distance tree is a rooted tree with the following properties:

1) Each node is a literal;

2) The root node is the goal literal;

3) Each node can form a complementary pair with its child node.

Definition 12. (Goal distance) In a goal distance tree, the level of tree is called the *goal literal distance* (denoted by $dist_L$) of the literal in this level. If a literal has multiple distinct goal literal distance values $d_1, d_2 \cdots, d_n$, then $dist_L = \min\{d_1, d_2 \cdots, d_n\}$. The *goal clause distance* (denoted by $dist_C$) of a clause $C = l_1 \vee l_2 \vee \cdots \vee l_m$ is defined as $dist_C = \min\{dist_L(l_i) | i = 1,2, \cdots, m\}$, where $dist_L(l_i)$ is the goal literal distance of literal l_i.

In general, a literal or a clause with smaller goal distance also is selected or reused preferentially during the S-CS deduction. Because the efficiency of an ATP system is closely related to the goal clauses during the deduction process [48].

5.2. Clause Selection

The clause selection strategy is used to determine which clause is the eligible-reuse clause and select a clause to participate in the S-CS deduction or clause sorting. The clause selection strategy is a weighted round-robin scheme with several queues, where the order within each queue is determined by a sort function and a weight function. The queue of a class of clauses is specialized by the sort function, the weight function representing a measure of the availability of a clause within the queue assigns a numerical value to a clause.

FRC algorithm implements **four sort functions**. The function of the sort functions is to add the clauses that satisfies certain condition into the corresponding queue. In other words, a sort function corresponds to a queue. Four sort functions are detailed below:

1) **PreferGoal** function. This function corresponds to a queue consisting of goal clauses.

2) **PreferUnit** function. This function always prefers unit clause, which corresponds to a queue consisting of unit clauses; **PreferNegUnit** function. This function is more specific than **PreferUnit**, the corresponding queue collects the negative unit clauses.

3) **PreferBinary** function. This function prefers binary clause, which corresponds to a queue consisting of binary clauses.

4) **PreferNew** function. This function corresponds to a queue consisting of the clauses generated by the deduction.

As the most important segment of a heuristic strategy, weight function is based on properties or features of the clause, some of which we have introduced above. Correspondingly, we set **five weight functions**.

1) **InValidWeight** and **ValidWeight**. This function considers invalid weight and valid weight of each clause.

2) **ClauseWeight**. This function considers the clause weight of each clause. In general, the clause with the smaller clause weight is selected or reused preferentially during the S-CS deduction.

3) **LitNumWeight**. The **LitNumWeight** of a clause is equal to the number of literals in this clause.

4) **GoalDistWeight**. This function considers the goal clause distance of each clause, i.e., The **GoalDistWeight** of a clause is equal to its goal clause distance.

5) **EReusingWeight**. This function considers efficient-reusing weight of each clause, i.e., the **EReusingWeight** of a clause is equal to its efficient-reusing weight.

The complete clause selection heuristic strategies are defined by a set of sort functions and weight functions. Each sort function combines a weight function to form an easy clause selection function. A list of weight functions and a sort function can form a composite clause selection strategy.

Example 5. A clause selection strategy is specified as

PreferNew[**ValidWeight**(2***ClauseWeight**(**GoalDistWeight**, 1, 1.5, 2),

1***ClauseWeight**(**LitNumWeight**, 1, 2, 2)].

In the queue consisting of clauses generated by the deduction, the first half of this strategy represents that this queue is first sorted by **GoalDistWeight**, then sorted by **ClauseWeight** where $w_f = 1, w_v = 1.5, w_c = 2$, and finally two clauses are chosen from this queue. The last half represents that this queue is first sorted by **LitNumWeight**, then sorted by **ClauseWeight** where $w_f = 1, w_v = 2, w_c = 2$, and finally one clause is chosen from this queue. This strategy finally chooses one of the three clauses based on **ValidWeight**.

5.3. Literal Selection

The literal selection strategy is implemented based on several weight functions, and some of weights have been introduced above. Literal selection strategy is mainly used to select the decision literal from a clause. Therefore, a clause has only one queue for sorting its literals. We describe four weight functions as follows.

1) **LiteralWight**. This function considers the literal weight of each literal. In general, the literal with the smaller literal weight is selected or reused preferentially during the S-CS deduction.

2) **GoalDistWeight**. Like **GoalDistWeight** in the clause, this function considers the goal literal distance of each literal.

3) **VarNumWeight**. This measure is equal to the number of variables in a literal.

4) **EReusingWeight**. Like **EReusingWeight** of a clause, the **EReusingWeight** of a literal is equal to its efficient-reusing weight.

Like clause selection strategy, a weight function can form a literal selection strategy, and a strategy generally is composed of multiple weight functions.

5.4. Deduction Evaluation

Each S-CS deduction step is evaluated by the deduction evaluation strategy. Specifically, whether each S-CS deduction step is effective or noneffective according to three thresholds and the CSC. Therefore, the deduction evaluation strategy evaluates the CSC and the latest clause to participate in the deduction. Deduction strategy is implemented based on three thresholds, such as $maxTD$, $maxLN$ and $leftLN$, and the three thresholds are set through abundant experimental data. After a clause C participated S-CS deduction, two attributes of CSC, such as the maximum term depth and the maximum number of literals, and one attribute of the clause C, i.e., the maximum number of literals in part C^+, need to be respectively evaluated, and if one of the following three conditions is not satisfied, the deduction is noneffective.

1) The maximum term depth of the generated CSC exceeds $maxTD$.

2) The maximum number of literals of the generated CSC exceeds $maxLN$.

3) The maximum number of literals of the generated CSC exceeds $leftLN$.

If a S-CS deduction is noneffective, then this S-CS deduction path needs to backtrack to the previous deduction step.

6. Integrating FRC into the Leading ATP System

Vampire is one of the most successful ATP systems. Through more than two decades of

research, Vampire has fully developed in terms of pre-processing technique [42], heuristic strategy [43,44], equality handle [36] and inference rule [36].

However, it can be found that there are still a lot of problems in TPTP library that cannot be solved by Vampire. In order to further improve the performance of Vampire, and also to evaluate the capability of the FRC algorithm, we integrate the FRC algorithm into Vampire to form a single integrated prover, named V_FRC. In V_FRC, FRC algorithm plays a positive role as an algorithm module of Vampire so that other procedure modules of Vampire, such as pre-processing and simplification procedures, won't be affected. In the integrated architecture of V_FRC, the clauses generated by the FRC algorithm are provided to Vampire as the lemmas. The workflow of V_FRC is as follows:

Step 1. Vampire is firstly applied to the initial clause set S for proof search, and a deduction result is obtained.

Step 2. If Vampire finds a proof in Step 1, then exit. Otherwise, FRC algorithm is performed to initial clause set S for deduction and generate some clauses which is put into a clause set R.

Step 3. If the clause set R has the empty clause, then the proof is found and output **UNSAT**. Exit!

Step 4. Filter some clauses from R to form a clause set R'.

Step 5. Let $S' = S \cup R'$. Input S' into Vampire for proof search and obtain the final result. Exit!

Some of the lemma clauses generated by the FRC algorithm may be redundant or useless for Vampire, which need to be deleted in Step 4, since the FRC algorithm could generate a large number of clauses. We set several conditions to filter clauses.

1) The number of literals in the clause should be less than the pre-defined threshold k. k usually set to 1 or 2, because a unit clause can easily be deduced to generate an empty clause, and the deduction involving binary clauses usually yield clauses with the smaller number of literals.

2) The maximum term depth of the clause should be less than the pre-defined threshold d, where d equals to the maximum term depth in the original clause set.

3) The goal clause distance of the clause should be less than the threshold $dist$. According to our empirical experience, the default setting of $dist$ is 6.

4) The clause weight of the clause should be less than the pre-defined threshold cw. cw equals to the maximum clause weight in the original clause set.

7. Experiments and Results Analysis

7.1. Experimental Setup

In order to evaluate the performance of V_FRC, we designed two experimental groups

(Experiment 1 and Experiment 2). The test problems of the two experiments come from TPTP benchmark library that is an international standard problem library for ATP systems and covers 49 scientific domains, where the domain corresponds to the first three letters of the problem name. The test problems used in Experiment 1 are CASC FOF division (2016-2021) problems with a total of 3000 problems (500 problems per year). The test problems used in Experiment 2 are all the problems with a rating of 1 in the latest released version of TPTP library, TPTP-v7.5.0, a total of 1584 problems. The rating denotes the difficulty of the problem, and it is a real number in the range 0.0 to 1.0, where 0.0 means that all ATP systems can solve the problem (i.e., the problem is easy), and 1.0 means no current ATP system can solve the problem (i.e., the problem is hard) [38]. The ratings of almost all CASC FOF division problems are less than 1.0 (there are just several problems with a rating of 1). Therefore, Experiment 1 is to evaluate the performance of V_FRC to solve general problems, and Experiment 2 is to evaluate the performance of V_FRC to solve hard problems.

The experimental environment is a PC with 3.6GHz Inter(R) Core (TM) i7-7700 processor and 16 GB memory, OS Ubuntu 20.04 64-bit. The test time for a single problem is 300 seconds (CPU time) which is the standard test time for TPTP library). In the experiments, the version of Vampire is 4.5.1. Finally, in order to verify the correctness of proof procedure by FRC algorithm, we use the well-known ATP system Prover9 [11] to verify each deduction path.

7.2. Experimental Results and Analysis

7.2.1. Results and Analysis of Experiment 1

First of all, we introduce the comparison experiment between V_FRC and Vampire 4.5.1, where V_FRC is an integrated system formed by integrating FRC algorithm into Vampire 4.5.1. Experiment 1 uses the problems which are CASC FOF division (2016-2021) problems with a total number of 3000 problems (500 problems per year).

Table 11 shows the experimental results of CASC FOF division problems (2016-2021) for each year. In each year from 2016 to 2021, the number of problems solved by V_FRC is more than the number of problems solved by Vampire 4.5.1. For example, in the “2017” column, V_FRC has solved 476 problems with 14 (denote by “(+14)”) more than Vampire 4.5.1, which has solved 462 problems. It isn’t difficult to calculate that in these six years, the number of problems solved by V_FRC is 11 more than that solved by Vampire 4.5.1 on average.

In addition, there are 272 problems of 3000 problems that Vampire has not solved, but some of 272 problems are the same. In fact, there are 182 problems that Vampire has not solved, among which V_FRC solved 50 problems accounting for 27.47% of these 182 problems. The corresponding name and rating of these 50 problems are shown in Table 12. The average rating of 50 problems listed in Table 12 is 0.84. There are 6 problems with rating of 1, 21 problems with

rating greater than 0.9 accounting for 42%, and 33 problems with rating greater than 0.8 accounting for 66% of the 50 problems. Particularly, these 50 problems involve 12 scientific domains, 26 of which are respectively from Logic Calculi (denoted by LCL) and Number Theory (denoted by NUM or NUN), accounting for half of the 50 problems.

The experimental results show that FRC algorithm can significantly enhance the performance of Vampire. FRC algorithm provides effective lemmas to assist Vampire to solve some problems that Vampire cannot solve by itself, especially some hard problems. On the other hand, V_FRC has strong generality, and FRC algorithm significantly improves the performance of Vampire on solving general problems.

Table 11

Comparison on solved problems by V_FRC and Vampire 4.5.1

	2016	2017	2018	2019	2020	2021
V_FRC	467 (+10)	476 (+14)	473 (+13)	458 (+12)	462 (+12)	459 (+6)
Vampire 4.5.1	457	462	460	446	450	453

Table 12

The list of 50 problems solved by V_FRC but not by Vampire 4.5.1

No	Problem	Rating	No	Problem	Rating
1	BOO109+1	0.71	26	LCL682+1.020	0.86
2	CAT025+4	0.92	27	LCL888+1	0.78
3	CAT032+2	0.89	28	LCL898+1	0.78
4	CSR036+3	0.5	29	NUM669+4	0.97
5	CSR049+6	0.97	30	NUM671+4	1.00
6	CSR064+6	0.86	31	NUM694+4	0.86
7	GEO316+1	0.86	32	NUM695+4	0.94
8	GEO495+1	0.69	33	NUM697+4	0.97
9	GEO506+1	0.75	34	NUM701+4	0.97
10	GRP745+1	0.65	35	NUM736+4	0.94
11	GRP780+1	0.78	36	NUM781+4	0.94
12	HWV108+1	0.97	37	NUM782+4	1.00
13	KLE016+2	0.75	38	NUN055+1	0.64
14	LCL468+1	0.89	39	NUN056+1	0.86
15	LCL572+1	1.00	40	REL024+2	0.35
16	LCL575+1	1.00	41	SEU357+2	0.89
17	LCL646+1.010	1.00	42	SEU383+2	0.94
18	LCL648+1.010	0.79	43	SEU410+1	0.86
19	LCL650+1.010	0.64	44	SWB028+1	0.89
20	LCL650+1.015	0.71	45	SWB070+1	0.92
21	LCL650+1.020	0.86	46	SWB092+1	1.00
22	LCL664+1.020	0.93	47	SWB103+1	0.97
23	LCL666+1.010	0.79	48	SWB107+1	0.92
24	LCL668+1.005	0.43	49	SWC125+1	0.78
25	LCL678+1.005	0.93	50	SWW228+1	0.89

7.2.2. Results and Analysis of Experiment 2

Experiment 2 uses all problems with rating of 1 in TPTPv7.5.0, a total of 1584 problems. The experimental result that V_FRC is shown in Table 13.

Table 13 shows that V_FRC solved 46 problems with a rating of 1. The average time for V_FRC to solve each problem is 196.04 s, which illustrates that FRC respectively enhances the

efficiency of Vampire in addition to improve the reasoning capability. Considering that these problems with a rating of 1 are the most difficult problems in TPTP library, where no current state-of-the-art ATP system can solve these problems, the experimental results are quite promising. Compared with general problems, the proof of the problem with a rating of 1 can better reflect the capability of an ATP system. V_FRC is able to solve 46 problems with a rating of 1, which shows that FRC plays a crucial role in the integrated system. Meanwhile, these 46 problems come from 13 scientific domains, and most problems (accounting for 27) are distributed in three domains: Logic Calculi (denoted by LCL), Number Theory (denoted by NUM or NUN) and Set Theory (denoted by SET or SEU).

Table 13

The list of 46 problems with rating of 1 solved by V_FRC

No	Problem	Time(s)	No	Problem	Time(s)
1	ALG001-1	171.08	24	LCL646+1.010	263.57
2	COL107-1	242.53	25	NUM656+4	220.61
3	CSR039+6	173.16	26	NUM657+4	220.31
4	CSR040+6	181.90	27	NUM658+4	232.97
5	CSR168+1	141.08	28	NUM659+4	234.95
6	FLD044-3	281.99	29	NUM671+4	115.79
7	FLD049-1	134.11	30	NUM726+4	253.19
8	FLD050-1	136.93	31	NUM782+4	115.93
9	FLD086-3	252.63	32	NUM791+4	268.88
10	FLD089-3	249.44	33	SCT160+1	213.53
11	GRP622+3	139.01	34	SET032-3	147.39
12	ITP008+4	210.57	35	SET033-3	140.39
13	ITP009+4	210.08	36	SET035-3	140.60
14	LAT355+2	267.93	37	SET279-6	136.74
15	LAT360+2	127.97	38	SET350-6	124.55
16	LAT380+4	181.06	39	SET368-6	172.27
17	LCL231-10	205.91	40	SET967+1	285.26
18	LCL478+1	164.84	41	SEU227+2	227.35
19	LCL479+1	165.47	42	SEU288+1	278.67
20	LCL532+1	227.13	43	SEU370+1	226.06
21	LCL572+1	215.06	44	SEU442+1	230.36
22	LCL574+1	164.69	45	SWB092+1	216.71
23	LCL575+1	166.20	46	SYO602-1	141.05

The experimental analysis of the two experimental groups shows that The FRC algorithm can significantly enhance Vampire's capacity for reasoning and efficiency. Since Vampire applies some deduction rule based on binary resolution, and FRC algorithm, as a multi-clause dynamic deduction algorithm, is an effective complementation to binary resolution, the performance of V_FRC is significantly improved. In addition to this reason, we consider several other reasons.

1) In contrast to binary resolution, S-CS rule is able to process multiple (two or more) clauses in one deduction step. The theoretical basis of FRC algorithm is S-CS rule, and FRC algorithm can quickly process multiple clauses to construct SC and the corresponding CSC.

2) In each clause separation, multiple (one or more) literals of a clause involving S-CS deduction can be eliminated, and the specific number of eliminated literals changes dynamically

based on the decision literals in the SC. Furthermore, the number of eliminated literals can be controlled by the heuristic strategy, so some required clauses (for example, unit clause or clause with equality literal) could be guided to generate and be provided as lemmas to Vampire.

3) Compared with saturation under heuristic strategy of binary resolution, the FRC algorithm continuously reuses a clause, and so its deduction potential will be exhausted as much as possible by the FRC algorithm in a smaller search space. This greatly improves the efficiency and capability of the FRC algorithm.

4) Based on pre-defined thresholds of heuristic strategy, the FRC algorithm is able to generate a large number of characterized clauses, such as unit clause, clause with less term depth or clause with less weight. These characterized clauses are generated by just a few deduction steps in the S-CS deduction, but several times more deduction steps are usually necessary for these clauses to be generated in the calculus of Vampire. Therefore, the FRC algorithm improves the inference efficiency of Vampire. Meanwhile, these characterized clauses involved in the inference process are more likely to infer the empty clause, so most of these clauses can shorten the inference path of generating the empty clause and thus further improve the performance of Vampire.

5) In the FRC algorithm, a clause is fully reused to generate multiple different decision literals during a SC construction process. In other words, literals in a clause may turn to be multiple decision literals because the clause is reused, which is an expression of the deductive potential of a clause being fully utilized. Furthermore, these decision literals guide subsequent deduction paths by finding complementary literals synergistically, i.e., selecting the subsequent clauses participated the deduction.

8. Conclusions and Future Works

For FOL, the Standard Contradiction Separation (S-CS) rule is a multi-clause dynamic deduction method that differs from binary resolution. In this paper, we proposed a novel way for implementing the S-CS rule (clause separation method), as well as a detailed analysis of some noteworthy benefits of the S-CS rule. The purpose of fully reusing clause was discussed first, and then we presented a fully clause reuse deduction framework based on the S-CS rule to take better benefit its abilities, which is one research goal of this study. Consequently, we developed a fully clause reusing deduction algorithm, called FRC algorithm, which is based on the fully clause reusing deduction framework. Subsequently, each algorithm procedure and pseudo-code of FRC algorithm were introduced. Notice that there are still many problems in the latest released version (TPTP-v7.5.0) of TPTP that have not been solved by those state-of-the-art ATP systems, such as Vampire, especially the problem with a rating of 1. Another research objective of this paper is improving the performance of Vampire to solve more problems in TPTP library. To achieve this objective, FRC

algorithm is integrated into Vampire as an algorithm module to form an integrated ATP system, called V_FRC.

The experimental results have shown that the performance of V_FRC outperforms that of Vampire. Especially, V_FRC solved 46 problems with a rating of 1, which were unsolved by any other ATP system. These experimental results demonstrated not only that the FRC algorithm effectively enhanced Vampire's performance and that the FRC method is an effective deduction process for theorem proving, but also that the two research objectives of this study have been achieved.

Although the FRC algorithm possesses effective reasoning capability, there is also much potential for improvement. For example, FRC algorithm is weak on equality handling, and input format of FRC algorithm needs to be optimized. In the future, we will optimize the FRC algorithm from three aspects. Firstly, we will design more heuristic strategies which optimize the proof path of S-CS rule and guide the selection of clauses or literals, and more clause ordering mechanisms. We are currently exploring unit clause ordering mechanism and more effective clause or literal features. Secondly, we will continue to study the reasoning mechanism of S-CS rule based on fully reusing clause principle. In the FRC algorithm, all the SCs generated during the deduction process are discarded and only the corresponding CSCs are retained, while these discarded SCs preserve many valid deduction information. To make better use of this information, we plan to develop a new deduction mechanism that can combine the useful SCs for further deduction. Finally, we will continue to improve the deduction efficiency of the FRC algorithm and incorporate some equality handling methods into it. In addition, the integration architecture of FRC algorithm into other leading ATP systems still has some shortcomings that needs to be improved, such as the interaction between FRC algorithm and other modules of the ATP system can be made more sufficient, and the lemmas provided by FRC algorithm may be further filtered.

Abbreviations

ATP	Automated theorem proving
S-CS	Standard contradiction separation
FRC algorithm	Fully reusing clause deduction algorithm
SDDA	S-CS dynamic deduction algorithm
CADE	Conference on Automated Deduction
CASC	CADE ATP System Competition
FOF	First-order form
TPTP	Thousands of Problems for Theorem Provers
CNF	Conjunctive normal form
SC	Standard contradiction
CSC	Contradiction separation clause

CRediT authorship contribution statement

Peiyao Liu: Methodology, Software, Validation, Investigation, Writing – original draft. **Yang Xu:** Supervision, Conceptualization, Methodology. **Jun Liu:** Conceptualization, Writing – Review & Editing. **Shuwei Chen:** Conceptualization, Writing – Review & Editing, Project administration. **Feng Cao:** Software, Project administration. **Guanfeng Wu:** Data curation, Project administration.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This paper is supported by the Natural Science Foundation of China (Grant Nos. 61976130, 62106206), the General Research Project of Jiangxi Education Department (Grant No. GJJ200818).

References

[1] N. Jiang, Q. Li, L. Wang, X. Zhang, Y. He, Overview on mechanized theorem proving, Journal of Software 31 (1) (2020) 82-112, https://doi.org/10.13328/j.cnki.jos.005870.

[2] J.A. Robinson, Theorem-proving on the computer, Journal of the ACM 10 (2) (1963) 163–174, https://doi.org/10.1145/321160.321166.

[3] P.W. O'Hearn, Incorrectness logic, in: Proceedings of the ACM on Programming Languages, 2020, pp. 1-32, https://doi.org/10.1145/3371078.

[4] G. Reger, A. Voronkov, Induction in saturation-based proof search, in: Automated Deduction – CADE 27, 2019, pp. 477-494, https://doi.org/10.1007/978-3-030-29436-6_28.

[5] L. Bellomarini, D. Benedetto, G. Gottlob, E. Sallinger, Vadalog: A modern architecture for automated reasoning with large knowledge graphs, Information Systems 105 (2022) 101528, https://doi.org/10.1016/j.is.2020.101528.

[6] P. Quaresma, Automatic deduction in an AI geometry book, in: Artificial Intelligence and Symbolic Computation, AISC 2018, 2018, pp. 221-226, https://doi.org/10.1007/978-3-319-99957-9_16.

[7] J. Otten, nanoCoP: A non-clausal connection prover, in: Automated Reasoning, IJCAR 2016, 2016, pp. 302-312, https://doi.org/10.1007/978-3-319-40229-1_21.

[8] J. Otten, Restricting backtracking in connection calculi, AI Communications 23 (2010) 159–182, https://doi.org/10.5555/1735921.1735931.

[9] R. Letz, G. Stenz, Proof and model generation with disconnection tableaux, in: Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2001, 2001, pp. 142-156, https://doi.org/10.1007/3-540-45653-8_10.

[10] W. McCune, OTTER 3.3 reference manual, 2003, https://doi.org/10.2172/822573.
[11] W. McCune, Release of Prover9, in: Proceeding of Mile High Conference on Quasigroups, Loops and Nonassociative Systems, 2005.
[12] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: Automated Deduction – CADE 27, 2019, pp. 495-507, https://doi.org/10.1007/978-3-030-29436-6_29.
[13] S. Schulz, M. Möhrmann, Performance of clause selection heuristics for saturation-based theorem proving, in: Automated Reasoning, IJCAR 2016, 2016, pp. 330–345, https://doi.org/10.1007/978-3-319-40229-1_23.
[14] J.A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM 12 (1) (1965) 23–41, https://doi.org/10.1145/321250.321253.
[15] K. Korovin, iProver – an instantiation-based theorem prover for first-order logic (system description), in: Automated Reasoning, IJCAR 2008, 2008, pp. 292-298, https://doi.org/10.1007/978-3-540-71070-7_24.
[16] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, CVC4, in: Computer Aided Verification, CAV 2011, 2011, pp. 171-177, https://doi.org/10.1007/978-3-642-22110-1_14.
[17] S. Alouneh, S. Abed, M.H. Al Shayeji, R. Mesleh, A comprehensive study and analysis on SAT-solvers: advances, usages and achievements, Artificial Intelligence Review 52 (2019) 2575–2601, https://doi.org/10.1007/s10462-018-9628-0.
[18] J. Chen, F. He, Leveraging control flow knowledge in SMT solving of program verification, ACM Transaction on Software Engineering and Methodology 30 (41) (2021) 1-26, https://doi.org/10.1145/3446211.
[19] L. Bachmair, H. Ganzinger, D. McAllester, C. Lynch, Chapter 2 - resolution theorem proving, in: Handbook of Automated Reasoning, 2001, pp. 19-99, https://doi.org/10.1016/B978-044450813-3/50004-7.
[20] J. Harrison, Handbook of practical logic and automated reasoning, Cambridge University Press, Cambridge, 2009, https://doi.org/10.1017/CBO9780511576430.
[21] D.A. Plaisted, History and prospects for first-order automated deduction, in: Automated Deduction – CADE-25, 2015, pp. 3-28, https://doi.org/10.1007/978-3-319-21401-6_1.
[22] R. Kowalski, D. Kuehner, Linear resolution with selection function, Artificial Intelligence 2 (3) (1971) 227–260, https://doi.org/10.1016/0004-3702(71)90012-9.
[23] R.S. Boyer, Locking: a restriction of resolution, Ph. D. Thesis, University of Texas at Austin, 1971.
[24] J.R. Slagle, Automatic theorem proving with Renamable and semantic resolution, Journal of the ACM 14 (4) (1967) 687–697, https://doi.org/10.1145/321420.321428.
[25] L. Wos, J. A. Robinson. Automatic deduction with hyper-resolution. International journal of computer mathematics, vol. 1 no. 3 (1965), 99. 227-234, Journal of Symbolic Logic 39 (1) (1974) 189–190, https://doi.org/10.2307/2272384.
[26] J. Jakubův, K. Chvalovský, M. Olšák, B. Piotrowski, M. Suda, J. Urban, ENIGMA Anonymous: symbol-independent inference guiding machine (system description), in: Automated Reasoning, IJCAR 2020, 2020, pp. 448-463, https://doi.org/10.1007/978-3-030-51054-1_29.
[27] M. Rawson, G. Reger, Old or heavy? Decaying gracefully with age/weight shapes, in: Automated Deduction – CADE 27, 2019, pp. 462–476, https://doi.org/10.1007/978-3-030-29436-6_27.

[28] S. Schäfer, S. Schulz, Breeding theorem proving heuristics with genetic algorithms, in: Global Conference on Artificial Intelligence, GCAI 2015, 2015, pp. 263–274, https://doi.org/10.29007/gms9.

[29] S. Schulz, Learning search control knowledge for equational deduction, Ph. D. Thesis, Technische Universität München, 2000.

[30] B. Löchner, A redundancy criterion based on ground reducibility by ordered rewriting, in: Automated Reasoning, IJCAR 2004, 2004, pp. 45-59, https://doi.org/10.1007/978-3-540-25984-8_2.

[31] B. Kiesl, M. Suda, A unifying principle for clause elimination in first-order logic, in: Automated Deduction – CADE 26, 2017, pp. 274–290, https://doi.org/10.1007/978-3-319-63046-5_17.

[32] U. Furbach, T. Krämer, C. Schon, Names are not just sound and smoke: Word embeddings for axiom selection, in: Automated Deduction – CADE 27, 2019, pp.250-268, https://doi.org/10.1007/978-3-030-29436-6_15.

[33] Y. Xu, J. Liu, S. Chen, X. Zhong, X. He, Contradiction separation based dynamic multi-clause synergized automated deduction, Information Sciences 462 (2018) 93–113, https://doi.org/10.1016/j.ins.2018.04.086.

[34] Y. Xu, S. Chen, J. Liu, X. Zhong, X. He, Distinctive features of the contradiction separation based dynamic automated deduction, in: Proceedings of the 13th International FLINS Conference, 2018, pp. 725–732, https://doi.org/10.1142/9789813273238_0092.

[35] F. Cao, Y. Xu, J. Liu, S. Chen, J. Yi, A multi-clause dynamic deduction algorithm based on standard contradiction separation rule, Information Sciences 566 (2021) 281–299, https://doi.org/10.1016/j.ins.2021.03.015.

[36] L. Kovács, A. Voronkov, First-order theorem proving and vampire, in: Computer Aided Verification, CAV 2013, 2013, pp. 1-35, https://doi.org/10.1007/978-3-642-39799-8_1.

[37] G. Sutcliffe, The CADE ATP system competition — CASC, AI Magazine 37 (2016), 99–101. https://doi.org/10.1609/aimag.v37i2.2620.

[38] G. Sutcliffe, The TPTP problem library and associated infrastructure. Journal of Automated Reasoning 59 (2017), 483–502. https://doi.org/10.1007/s10817-017-9407-7.

[39] G. Sutcliffe, The TPTP world – infrastructure for automated reasoning, in: International Conference on Logic for Programming Artificial Intelligence and Reasoning, 2010, pp.1–12, https://doi.org/10.1007/978-3-642-17511-4_1.

[40] P. Liu, Y. Xu, S. Chen, F. Cao, Fully reusing clause method based standard contradiction separation rule, in: The 15th of International FLINS Conference, In press.

[41] S. Chen, Y. Xu, Y. Jiang, J. Liu, X. He, Some synergized clause selection strategies for contradiction separation based automated deduction, in: 2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE), 2017, pp. 1–6, https://doi.org/10.1109/ISKE.2017.8258741.

[42] G. Reger, M. Suda, A. Voronkov, New techniques in clausal form generation, in: 2nd Global Conference on Artificial Intelligence, 2016, pp. 11–23, https://doi.org/10.29007/dzfz.

[43] B. Gleiss, M. Suda, Layered clause selection for theory reasoning, in: Automated Reasoning, IJCAR 2020, 2020, pp. 402–409, https://doi.org/10.1007/978-3-030-51074-9_23.

[44] G. Reger, J. Schoisswohl, A. Voronkov, Making theory reasoning simpler, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021, 2021, pp. 164–180, https://doi.org/10.1007/978-3-030-72013-1_9.

[45] T. Tammet, GKC: a reasoning system for large knowledge bases, in: Automated Deduction – CADE 27, 2019, pp. 528-549, https://doi.org/10.1007/978-3-030-29436-6_32.

[46] I. Abdelaziz, M. Crouse, B. Makni, V. Austel, C. Cornelio, S. Ikbal, P. Kapanipathi, N. Makondo, K. Srinivas, M. Witbrock, A. Fokoue, Learning to guide a saturation-based theorem prover, IEEE Transactions on Pattern Analysis and Machine Intelligence (2022) 1–1. https://doi.org/10.1109/TPAMI.2022.3140382.

[47] E.K. Holden, K. Korovin, Heterogeneous heuristic optimisation and scheduling for first-order theorem proving, in: International Conference on Intelligent Computer Mathematics, 2021, pp.107–123. https://doi.org/10.1007/978-3-030-81097-9_8.

[48] D.W. Loveland, D.W. Reed, D.S. Wilson, SATCHMORE: SATCHMO with Relevancy, Journal of Automated Reasoning 14 (1995) 325–351, https://doi.org/10.1007/BF00881861.