An Efficient Contradiction Separation Based Automated Deduction Algorithm for Enhancing Reasoning Capability

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

Automated theorem prover for first-order logic, as a significant inference engine, is one of the hot research areas in the field of knowledge representation and automated reasoning. E prover, as one of the leading automated theorem provers, has made a significant contribution to the development of theorem provers for first-order logic, particularly equality handling, after more than two decades of development. However, there are still a large number of problems in the TPTP problem library, the benchmark problem library for automated theorem provers, that E has yet to solve. The standard contradiction separation rule is an inference method introduced recently that can handle multiple clauses in a synergized way and has a few distinctive features which complements to the calculus of E. Binary clauses, on the other hand, are widely utilized in the automated deduction process for first-order logic because they have a minimal number of literals (typically only two literals), few symbols, and high manipulability. As a result, it is feasible to improve a prover's deduction capability by reusing binary clause. In this paper, a binary clause reusing algorithm based on the standard contradiction separation rule is firstly proposed, which is then incorporated into E with the objective to enhance E's performance, resulting in an extended E prover. According to experimental findings, the performance of the extended E prover not only outperforms E itself in a variety of aspects, but also solves 18 problems with rating of 1 in the TPTP library, meaning that none of the existing automated theorem provers are able to resolve them.

Keywords: Knowledge representation; Automated reasoning; First-order logic; Automated theorem prover; Standard contradiction rule; E prover

1 Introduction

In the real world, knowledge plays a crucial role in intelligence as well as creating artificial intelligence (AI) [1]. For an AI system to behave accurately in response to some input, it must possess knowledge about the input. In other words, knowledge is necessary for intelligent behavior. Knowledge representation and reasoning (in short KR&R) [2] is a research focus in the field of artificial intelligence dedicated to expressing knowledge about world in a computer tractable form so that it can be used to enable AI systems to perform well [3]. Knowledge representation suggests an approach to understanding intelligent behavior that is radically different from other ways, such as psychology [4], neuroscience [5] and philosophy [6]. KR&R focuses on what humans know, i.e., knowledge, instead of studying humans very carefully, i.e., knower.

* Corresponding author.
E-mail addresses: swchen@swjtu.edu.cn (S. Chen), liupeiyao@my.swjtu.edu.cn (P. Liu)

In order to solve the complex problems encountered in AI, one generally needs a large amount of knowledge, and suitable mechanism for representing and manipulating all that knowledge. Knowledge is best represented through a knowledge representation language [7] which determine the computational objects, relations and inference available to a programmer. Apparently, natural language [8] is a knowledge representation language, in addition, programming languages [9], semantic networks [10] and logic [11] are also knowledge representation languages. Logic language represented by first-order logic (FOL) as a declarative language uses expression in formal logic to represent knowledge [12,13], and FOL calculus is one of the most important logical representation schemes for knowledge representation [14]. On the other hand, knowledge representation is inseparable from automated reasoning, which is to study how to infer conclusion based on given hypotheses, because one of the main purposes of knowledge representation is to be able to infer new knowledge [15]. Thus, automated reasoning plays a central supporting role in the field of knowledge representation. Almost all knowledge representation languages have a corresponding reasoning or inference engine. For example, the inference engine corresponding to FOL is called automated theorem prover (ATP) [16].

A knowledge-based system [17] is a system that uses AI techniques in problem-solving processes to support human decision-making, learning, and action, and it has two distinguishing central components: a knowledge base that represents facts about the world and an inference engine that is responsible for the application of knowledge base to the problem on hand. A good knowledge-based system needs an appropriate representation language. The trade-off between effectiveness and efficiency is the key to measuring whether a representation language is good or not. In terms of expressiveness and compactness, FOL has unparalleled advantages. In addition, FOL has several significant advantages. First of all, FOL is a formal method of reasoning and a mathematical representation formula, and it studies entailment relations, formal languages, truth conditions, semantics and inference. Many propositions can be translated into first-order logical symbolic representations, and computers can easily manipulate these symbolic formulas to infer various facts. After all, there is no more powerful formalism than mathematical formula used to define general propositions. Secondly, FOL guarantees the soundness and completeness of the corresponding reasoning methods. Therefore, FOL is an appropriate knowledge representation language for many AI problems and mathematical problems, and the corresponding ATP is an essential and powerful inference engine. Currently, ATPs are commonly applied in the field of knowledge representation [18-21], but also in other fields; for instance, program verification [22,23], the operating systems [24,25] and the design of compliers [26,27].

Both theoretical research and applications in the field of ATP has developed rapidly and achieved fruitful results since Robbinson proposed resolution principle [28] in 1965, which is still the mainstream inference method of theorem proving nowadays. It's the basic idea of resolution principle is to select two clauses from the given clause set, with one literal from each selected clause forming a complementary pair of literals, for resolution in each deduction step, and then a new clause (called resolvent) is obtained as the disjunction of the remaining literals of the two clauses after eliminating the complementary pair of literals. This deduction process continues until an empty clause is obtained. After superposition rule [29] was proposed, which is essentially a combination of resolution principle and paramodulation [30], most resolution-based provers adopt superposition rule for equality handling [31], e.g., Vampire [32], E [33], GKC [34] and Prover9 [35], etc.

As one of the representative ATPs, E is a state-of-the-art prover for FOL with equality entirely based on superposition rule [36], which has been developed for more than twenty years. E has participated on its own in CADE ATP System Competition (CASC) [37] every year since 1999, and has performed well in the full first-order, clause normal form, and unit equality proof categories, often coming in as one of the top three in FOF division of CASC [38].

The deduction framework of most current ATPs for FOL adopts saturation algorithm framework, typically OTTER [39] and DISCOUNT [40]. E introduces a modified version of the DISCOUNT algorithm, one of the variants of the given-clause algorithm [36]. The basic idea of the DISCOUNT algorithm is to divide the clause set into two disjoint subsets, the processed clause set P and the unprocessed clause set U. Initially, set P is empty and all clauses of the clause set are put into U. At each iteration of the main loop, the ATP selects a given clause from U according to the heuristic strategies and puts it into P, then performs all inference rules between this given clause and all clauses in P. The resulting new clauses are put into U. It can be found that the number of clauses of P grows rapidly when the number of iterations keeps increasing. Therefore, the ATPs adopting DISCOUNT algorithm rely on a lot of heuristic strategies such as term ordering [41-43], literal and clause selection [44-46], to reduce the search space. This is one crucial problem to be solved for the ATPs such as E adopting DISCOUNT algorithm. On the other hand, there are a number of problems which cannot be solved by E,

especially problems with rating of 1 meaning that none of the current ATPs are able to resolve them [47], in the latest released version (TPTP-v7.5.0) of the TPTP (Thousands of Problems for Theorem Provers) benchmark library [48]. Therefore, there is still space for improving the performance of E.

Aiming at enhancing the inference ability of resolution principle, Xu *et al*. [49] proposed a dynamic multi-clause synergized deduction theory, standard contradiction separation (S-CS) rule in 2018 motivated by the idea from multi-valued logic, which is perceived as a theoretical development of resolution principle. Specially, at each the deduction step of S-CS rule, multiple (two or more) clauses are selected as parent clauses of this deduction, and multiple (one or more) literals from each parent clause are selected to construct a contradiction, then a new clause formed as the disjunction of the non-selected literals of the parent clause is inferred [50]. The most obvious difference from resolution principle is that there are multiple (two or more) clauses participate the deduction at each deduction step of S-CS rule. In addition, S-CS rule has several abilities that resolution principle doesn't have, e.g., synergized ability, controllability and clause-reusing ability [51], where the clause-reusing ability, in particular, is the focus of this paper. Therefore, S-CS rule has the advantage of solving the crucial problem mentioned above and acts as a complement to E to a certain extent, while this part of content will be analyzed in detail in Section 3. At the same time, we have an idea of applying S-CS rule to E, expecting that S-CS rule can enhance the performance of E.

There are two motivations behind the research: 1) to build a novel deduction algorithm based on the capabilities of the S-CS rule; 2) to enhance E' performance by using the S-CS rule, especially the performance to solve a specific number of problems with a rating of 1. Consequently, we need to develop a more effective deduction algorithm based on S-CS rule, and a reasonable extended architecture of E for applying S-CS rule. In this paper, we design an effective implementation of S-CS rule, i.e., binary clause reusing algorithm, then this deduction algorithm is incorporated into E, to form an extended ATP of E, dubbed E_BCR. We design two groups of experiment~~s~~ to evaluate the performance of ~~this extended prover~~ E_BCR. According to experimental findings, the performance of E_BCR outperforms E itself in a variety of aspects, reflecting that binary clause reusing algorithm based on the S-CS rule is an effective deduction algorithm. Notably, reusing binary clause deduction algorithm has been initially implemented on the ATP CSE_E 1.3, which won the 3rd place in FOF division of CASC-28 (2021) [52]. Meanwhile, CSE_E 1.3 solved the same number of problems as iProver 3.5, that was the runner-up in FOF division of CASC-28 (2021). The contributions of the research work in this paper has three points: we have 1) studied S-CS rule and explore the distinctive features of S-CS rule in more depth; 2) further verified the implementation and effectiveness of S-CS rule; 3) proposed an innovative deduction algorithm, and significantly enhanced the ability of E.

The remainder of this paper is organized as follows. Section 2 introduces the preliminaries of FOL and S-CS rule. In Section 3, the binary clause reusing algorithm based on S-CS rule is proposed and its advantages are analyzed, then some heuristic strategies related to this algorithm are introduced. The extended architecture of E for applying S-CS rule is detailed in Section 4. The related experiments and the analysis of experimental results are studied in Section 5. In Section 6, we give conclusions of this paper and our future work plans.

We note that a short version of this paper appeared in the 2021 IEEE International Conference on Intelligent Systems and Knowledge Engineering (ISKE 2021) [53]. Our initial conference paper just introduced the basic idea of the binary clause reusing algorithm, while neither the detailed analysis of the proposed algorithm and related experimental results, nor the related heuristic strategies were provided. This manuscript addresses these issues and provides more extensive experimental studies and analyses to illustrate the performance of the extended ATP of E.

2 Preliminaries

In this section, we firstly provide some basic concepts of FOL, and the readers are referred to Ref. [54] for a detailed introduction. Secondly, we give the preliminaries of S-CS rule and the basic method of constructing contradiction.

2.1 Preliminaries of First-order Logic

We focus on conjunctive normal form (CNF) of first-order logic excluding quantifiers. A CNF formula is built from variables, function symbols, predicate symbols and conjunction symbols [54].

Definition 1 [54] (Term) A *term* is either a *variable* or an expression $f(t_1, t_2, \cdots, t_n)$, where f is a *function* symbol of arity n and $t_1, t_2, \cdots, t_n$ are terms.

$Term(F, V)$ is to denotes the set of terms over an enumerable set F of function symbols with associated arities and set V of variables. A 0-ary function is called *constant*. We use t (possibly primed or subscripted) to denote a term, e.g., t', t_1, t_2, x (possibly primed or subscripted) for a variable, e.g., x', x_{11}, x_{21}, f (possibly primed or subscripted) for a non-constant function symbol, e.g., f', f_1, f_2, and a for a constant, e.g., a', a_1, a_2. A term t is called *ground term* if it contains no variable.

Definition 2 [54] (Atom) An *atom* is an expression $P(t_1, t_2, \cdots, t_n)$, where P is a predicate symbol of arity n and $t_1, t_2, \cdots, t_n$ are terms.

We use P (possibly primed or subscripted) to denote a predicate symbol, e.g., P', P_1, P_2.

Definition 3 [54] (Literal) A *literal* is an expression A (a positive literal) or $\sim A$ (a negative literal), where A is an atom. Two literals A and $\sim A$ are said to be complementary.

We use l (possibly primed or subscripted) to denote a literal, e.g., l', l_1, l_2.

Definition 4 [33] (Clause) A clause is set of literals, sometimes written as $C = l_1 \vee l_2 \vee \cdots \vee l_n$ and interpreted as the disjunction of its literals.

Multiple occurrences of the same clause are usually considered as distinct clauses and we implicitly assume that any two clauses do not share the same variable. We use C (possibly primed or subscripted) to denote a clause, e.g., C', C_1, C_2. A clause C with *n* literals is called *n*-ary ($n \in \mathbb{N}$) clause. Specially, a 1-ary clause is called *unit clause*, 2-ary clause is called *binary clause*, and a 0-ary clause is *empty clause* denoted by $\emptyset$.

We usually use an uppercase bold letter (possibly primed or subscripted) to denote a clause set, e.g., $\boldsymbol{S}, \boldsymbol{T_1}, \boldsymbol{R}'$.

Definition 5 [55] (Substitution) A *substitution* is a function σ: $V \to Term(F, V)$ with property that $\{x|\sigma(x) \neq x\}$ is finite, where $x \in V$. A substitution σ also be written as $\sigma = \{t_1/x_1, t_1/x_1, \cdots, t_n/x_n\}$, where $t_i \in Term(F, V)$, $x_i \in V$ and $t_i \neq x_i$ $(i = 1,2, \cdots, n\})$. If t_i is a ground term, then σ is called a ground substitution.

In this paper, a substitution is denoted by a lowercase Greek letter (possibly primed or subscripted), e.g., σ, θ. The same clause or literal with different substitutions is regarded as different clauses or literals.

2.2 Preliminaries of S-CS Rule

Some concepts of S-CS rule are introduced as follows. We only recall some basic concepts, and readers are referred to Ref. [49] for a detailed introduction.

Definition 6 [49] (Contradiction) Let $\boldsymbol{S} = \{C_1, C_2, \cdots, C_m\}$ be a clause set in FOL 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 $\boldsymbol{S} = \bigwedge_{i=1}^{m} C_i$ is called a standard contradiction (in short, SC).

Definition 7 [49] Suppose a clause set $\boldsymbol{S} = \{C_1, C_2, \cdots, C_m\}$ in FOL. The following inference rule that produces a new clause from $\boldsymbol{S}$ 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).

The deduction sequence based on S-CS rule, the soundness and completeness of S-CS rule are described as follows.

Definition 8 [49] Suppose a clause set $\boldsymbol{S} = \{C_1, C_2, \cdots, C_m\}$ in FOL. $\Phi_1, \Phi_2, \cdots, \Phi_t$ is called a standard contradiction separation based dynamic deduction sequence (S-CS deduction) from $\boldsymbol{S}$ to a clause Φ_t, denoted as D^s, if

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

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

Theorem 1 [49] (Soundness) Suppose a clause set $\boldsymbol{S} = \{C_1, C_2, \cdots, C_m\}$ in FOL. $\Phi_1, \Phi_2, \cdots, \Phi_t$ is a S-CS based dynamic deduction from $\boldsymbol{S}$ to a clause Φ_t. If Φ_t is an empty clause, then $\boldsymbol{S}$ is unsatisfiable.

Theorem 2 [49] (Completeness) Suppose a clause set $\boldsymbol{S} = \{C_1, C_2, \cdots, C_m\}$ in FOL. If $\boldsymbol{S}$ is unsatisfiable, then exists an S-CS based dynamic deduction from $\boldsymbol{S}$ to an empty clause.

The crucial point of S-CS rule is the construction of the SC. During the process of S-CS deduction, each clause C participating the deduction is separated into the two parts, i.e., $C^{\sigma+}$ and $C^{\sigma-}$, under a certain substitution σ, where $C^{\sigma+}$ is the sub-clause of the CSC of the deduction and $C^{\sigma-}$ is put into the corresponding SC. In other words, a SC is constructed through a series of separating clauses. Therefore, the problem how to construct the SC is equivalent to the problem how to separate the clauses that participate the S-CS deduction.

Furthermore, there is one literal from each clause participating the S-CS deduction is called *decision literal* [56] that plays an important role on determining the SC, and thus the SC has a set of decision literals written as $\boldsymbol{D_l}$ in this paper. In fact, which clause is selected to construct the SC is determined by $\boldsymbol{D_l}$ during the process of constructing the SC. Therefore, we introduce the concept of *pairing condition*.

Definition 9 (Pairing condition) Suppose $\bigwedge_{i=1}^m C_i^{\sigma_i-}$ is a SC, and the set $\boldsymbol{D_l} = \{l_{d1}, l_{d2}, \cdots, l_{dm}\}$ 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. There exist a literal l_p in a clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$ and a substitution θ, and the literal l_p^θ can be put into the $C^{\theta-}$ when the clause C participates the S-CS deduction. If l_p satisfies the following 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 substitution θ, then this condition is called a *pairing condition*.

In order to make the deduction more efficient, we stipulate that any two literals in $\boldsymbol{D_l}$ cannot form a complementary pair. Pairing condition is considered the most fundamental condition for a clause to participate the S-CS deduction, namely a clause must satisfy at least pairing condition to be eligible to participate the S-CS deduction. If a clause does not have a literal that satisfies pairing condition, this clause cannot participate the S-CS deduction. Consequently, *clause separation method* of S-CS rule 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 $\boldsymbol{D_l} = \{l_{d1}, l_{d1}, \cdots, l_{dm}\}$ 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 an upcoming clause that will participates the S-CS deduction, then the process of separating clause C^θ into two parts $C^{\theta-}$ and $C^{\theta+}$ after a substitution θ is shown as follows.

Step 1: If the literal $l_j^{\theta_j}$ $(j = 1,2, \cdots, n)$ that 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 user-defined deduction conditions 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 $C^{\theta-}$.

The SC $\bigwedge_{i=1}^m C_i^{\sigma_i-}$ will 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 the 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.

The following example illustrates the process of S-CS deduction, especially the separation of each clause.

Example 1 Let $\boldsymbol{S} = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$ 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.

Applying S-CS rule to the clause set $\boldsymbol{S}$, and using clause separation method to $C_7, C_8, C_6, C_1, C_3, C_4$ in sequence. We obtain a CSC involving 6 clauses: $C_9 = \sim P_2(a_2, a_1)$, and the corresponding SC is shown in Table 1. The corresponding substitution and decision literal of each clause is shown in Table 2.

Table 1 Separation the $C_i^{\sigma_i}$ for Example 1

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	$P_1(a_2, a_2, f_1(a_2))$	$\sim P_2(a_2, a_1)$

Table 2 Corresponding substitution σ_i and decision literal of clause C_i for Table 1

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	$\{a_2/x_{41}\}$	$\emptyset$

3 Binary Clause Reusing Algorithm Based on S-CS Rule

This section discusses the theory of binary clause reusing and proposes the deduction algorithm based on binary clause reusing, and introduces the heuristic strategies associated with this deduction algorithm.

3.1 Binary Clause Reusing Algorithm

From Section 2, the core of S-CS rule is constructing the SC and thus obtaining the CSC. According to Ref. [51], we know that S-CS rule has at least ten featured characteristics, such as guidance and clause-reusing. Ref. [57] proposed a clause reusing deduction framework based on S-CS rule, which can take better advantage of guidance ability. In this paper, based on this framework, we then propose a binary clause reusing algorithm based on S-CS rule which fully takes advantage of deduction ability of binary clause.

First of all, we introduce the concept of clause reusing and the advantages of clause reusing in S-CS deduction.

According to Definition 9, we know that the set of decision literals $\boldsymbol{D_l}$ is crucial for the S-CS deduction. Intuitively, the more literals in $\boldsymbol{D_l}$, the more likely an upcoming clause that will participate in the S-CS deduction meets paring condition. On the other hand, if a literal l in an upcoming clause C that will participate in the S-CS deduction can form a complementary pair with a literal of $\boldsymbol{D_l}$ after a certain substitution, then the literal l will be put into the contradiction. Therefore, the more literals in $\boldsymbol{D_l}$, the more likely the $C^{\sigma+}$ after a clause C participating in the S-CS deduction under a

substitution σ is an empty clause. The number of literals in $\boldsymbol{D_l}$ has positive impact on improving the performance of S-CS rule.

At the same time, a literal in the upcoming clause that expects the S-CS deduction may form different complementary pairs with multiple literals of $\boldsymbol{D_l}$ respectively, so this upcoming clause may participate the S-CS deduction multiple times, i.e., clause reusing. During the construction of a SC, each deduction of this upcoming clause will generate a new decision literal, so clause reusing will increase a number of decision literals.

We've then come to two conclusions: 1) it is able to effectively generate different decision literals when reusing clauses in one SC creation process; 2) it is able to generate an empty clause in the development of this SC when there are more different decision literals in the SC.

In theory, a clause is reused until there is no literal in it that can form a complementary pair with a decision literal of $\boldsymbol{D_l}$. However, it is not achievable for each clause in a clause set to be fully reused, since resources available to a prover in practice, e.g., the running time and memory resources, is limited. Therefore, we introduce the concept of *well separation* to characterize a "good" separation of a clause in the S-CS deduction.

Definition 10 (Well separation) There is a constructing SC. A *n*-ary clause C as an upcoming clause that is going to participate the S-CS deduction. The separation that divides the clause C into $C^{\theta-}$ and $C^{\theta+}$ by applying clause separation method under a substitution θ is called *well separation*, if $C^{\theta+}$ is an empty clause.

This paper proposes *binary clause reusing algorithm* (in short BCR algorithm) based on S-CS rule, the main reasons why binary clauses are reused are based on the following analysis.

1. Empirically, both binary resolution and S-CS rule prefer short clauses, especially unit clause or binary clause, during the deduction process. Meanwhile, unit clauses and binary clauses usually occupy a significant number of clauses in a clause set. Therefore, it is reasonable to highlight the reusing of binary clause.

2. S-CS rule has the feature of reusing clause, that is, the deduction based on S-CS rule is a clause-reusing deduction. During the process of constructing a SC, one clause can participate in the S-CS deduction multiple time, i.e., reusing the clause, such that this clause may generate multiple different decision literals. The more and different decision literals, the better constraints on the CSC (fewer literals or symbols) and the higher synergy of the deduction. Therefore, reusing binary clause can effectively generate abundant decision literals.

3. According to Definition 10, the separation of a binary clause C must be well separation as long as a literal in C can form a complementary pair with a certain decision literal under the substitution σ, i.e., $C^{\sigma+} = \emptyset$. If the unit clauses and binary clauses are selected to participate in S-CS deduction in priority, the CSCs are all unit clauses during the period. These unit CSCs make the deduction path more optimal in the subsequent deduction, and some clauses in the clause set can be removed if they are subsumed by the unit CSCs which alleviates the search space expansion.

4. Compared to *n*-ary ($n > 2$) clause, binary clause is more manipulative, in other word, using binary clause could reduce the complexity of the deduction algorithm.

Therefore, reusing binary clause is a reasonable method for the S-CS deduction, which does not contribute any literal to the CSC. BCR algorithm based on S-CS rule is introduced as follows.

BCR algorithm is introduced in detail in the following steps, and the corresponding pseudo-code of BCR algorithm is provided in Algorithm 1.

Step 1: $\boldsymbol{S}$ is a given clause set. Before applying S-CS rule, $\boldsymbol{S}$ is divided into three subsets, i.e., $\boldsymbol{S_u}, \boldsymbol{S_b}, \boldsymbol{S_e}$, where $\boldsymbol{S_u}$ stores unit clauses, $\boldsymbol{S_b}$ holds binary clauses and the remaining clauses are stored in $\boldsymbol{S_e}$. Then according to the heuristic strategies, sort the clauses of $\boldsymbol{S_u}$.

Step 2: The processed clause participated the S-CS deduction after a certain substitution is stored in the clause set $\boldsymbol{P}$, the set $\boldsymbol{D_l}$ stores the decision literals during the S-CS deduction. The corresponding literal of each clause of $\boldsymbol{S_u}$ is put into $\boldsymbol{D_l}$ in sequence, i.e., the literal of each unit clause is used to initialize $\boldsymbol{D_l}$. Then each clause of $\boldsymbol{S_u}$ is put into $\boldsymbol{P}$.

Step 3: Traverse each clause C_b of the set $\boldsymbol{S_b}$ sequentially. If the traversal is completed, go to Step 6.

Step 4: Check whether there is a literal of the clause C_b that satisfies paring condition.[a] If not, go to Step 3.

[a] To avoid endless loop, BCR algorithm stipulates that a complementary pair that is formed with a literal l of an upcoming clause and a literal of $\boldsymbol{D_l}$ is only allowed appeared once with respect to the literal l during the S-CS deduction.

Algorithm 1 BCR algorithm

Input:
a given clause set $\boldsymbol{S}$
a set of unit clauses $\boldsymbol{S_u}$ (initially empty)
a set of binary clauses $\boldsymbol{S_b}$ (initially empty)
a set of *n*-ary ($n \geq 3$) clauses $\boldsymbol{S_e}$ (initially empty)
a set of processed clauses $\boldsymbol{P}$ (initially empty)
a set of decision literals $\boldsymbol{D_l}$ (initially empty)
a threshold Num
Output:
a set of new generated CSCs $\boldsymbol{R}$

1: *classifySet*($\boldsymbol{S}$, $\boldsymbol{S_u}$, $\boldsymbol{S_b}$, $\boldsymbol{S_e}$);
2: *sortSet*($\boldsymbol{S_u}$);
3: *initialDecisionSet*($\boldsymbol{S_u}$, $\boldsymbol{D_l}$);
4: $\boldsymbol{P} = \boldsymbol{P} \cup \boldsymbol{S_u}$;
5: **for each** $C_b \in \boldsymbol{S_b}$
6: if FALSE == *paringCondition*(C_b, $\boldsymbol{D_l}$) goto 5;
7: C_r = *separateCla*(C_b, $\boldsymbol{P}$, $\boldsymbol{D_l}$);
8: if $C_r == \emptyset$
9: return **UNSAT**;
10: *updateDecisionSet*(C_b, $\boldsymbol{D_l}$);
11: $\boldsymbol{R} = \boldsymbol{R} \cup \{C_r\}$;
12: $\boldsymbol{P} = \boldsymbol{P} \cup \{C_b\}$;
13: goto 6;
14: **end for**
15: $k = 0$;
16: **while** (C_g = *selectGivenCla*($\boldsymbol{S_e}$)) ≠ null begin
17: if FALSE == *paringCondition*(C_g, $\boldsymbol{D_l}$) goto 16;
18: C_r' = *separateCla*(C_g, $\boldsymbol{P}$, $\boldsymbol{D_l}$);
19: if $C_r' == \emptyset$
20: return **UNSAT**;
21: if TRUE == *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$)
22: *rollbackPath*(C_r', C_g, $\boldsymbol{P}$, $\boldsymbol{D_l}$);
23: goto 15;
24: if TRUE == *stopDeduction*(C_r') goto **end while**;
25: *updateDecisionSet*(C_g, $\boldsymbol{D_l}$);
26: $\boldsymbol{R} = \boldsymbol{R} \cup \{C_r'\}$;
27: $\boldsymbol{P} = \boldsymbol{P} \cup \{C_g\}$;
28: if $(k = k + 1) < Num$ goto 16;
29: else goto 5;
30: **end while**
31: **for each** $C_r'' \in \boldsymbol{R}$
32: *backwardSubsumed*($\boldsymbol{S}$, C_r'');
33: **end for**
34: $\boldsymbol{S} = \boldsymbol{S} \cup \boldsymbol{R}$;
35: output $\boldsymbol{R}$;

Step 5: Apply clause separation method of S-CS rule to the clause C_b and the two sets $\boldsymbol{P}$ and $\boldsymbol{D_l}$, separate the clause C_b, and generate a CSC C_r. If C_r is an empty clause, then output **UNSAT**, exit! If not, select a literal from the clause C_b as

a new decision literal, and put it into $\boldsymbol{D_l}$. C_r is put into the clause set $\boldsymbol{R}$ that is used to store new generated CSCs during the S-SC deduction. The clause C_b is put into $\boldsymbol{P}$. Go to Step 4.

Step 6: Select a clause from the clause set S_e as the next clause C_g to participate in the deduction according to the heuristic strategies. If there is no clause in S_e qualifying as the next clause C_g, go to Step 10.

Step 7: Check whether there is a literal of the clause C_b that satisfies paring condition. If not, go to Step 6.

Step 8: Apply clause separation method of S-CS rule to the clause C_g and the two sets $\boldsymbol{P}$ and $\boldsymbol{D_l}$, separate the clause C_g, and generate a CSC C_r'. If C_r' is an empty clause, then output **UNSAT**, exit! If not, check the clause C_r' to determine whether this deduction step is valid, where the determination method is introduced in the following. If this deduction step is invalid, perform the procedure of deduction path rollback, then go to Step 6.

Step 9: Check the exit condition of the deduction. If the exit condition is satisfied, go to Step 10. Otherwise, the new generated clause C_r' is put into the clause set R that is used to store new generated CSCs during the S-SC deduction. The clause C_g is put into P. When every Num (a predefined threshold and set as 10 by default according to our empirical experience) clauses of the set S_e have participated in the S-CS deduction, go to Step 3. Otherwise, go to Step 6.

Step 10: Traverse each clause C_r'' of the clause set $\boldsymbol{R}$. Then check each clause of the clause set $\boldsymbol{S}$ whether it is subsumed by the clause C_r''. If yes, delete the subsumed clauses from each clause set.

Step 11: Each clause of $\boldsymbol{R}$ is put into $\boldsymbol{S}$, then output $\boldsymbol{R}$.

The functions of the pseudo-code are described as follows.

- *classifySet*($\boldsymbol{S}$, $\boldsymbol{S_u}$, $\boldsymbol{S_b}$, $\boldsymbol{S_e}$): This function divides the clause set $\boldsymbol{S}$ into three subsets, i.e., $\boldsymbol{S_u}, \boldsymbol{S_b}, \boldsymbol{S_e}$, where $\boldsymbol{S_u}$ stores unit clauses, $\boldsymbol{S_b}$ holds binary clauses and the remaining clauses are stored in $\boldsymbol{S_n}$. Suppose that there are m clauses in the set $\boldsymbol{S}$, the time complexity of this function is $O(m)$.
- *sortSet*($\boldsymbol{S_u}$): It sorts the clauses of the clause set $\boldsymbol{S_u}$ according to the heuristic strategies. Suppose that there are m_u clauses in the set $\boldsymbol{S_u}$, the time complexity of this function is $O(m_u \log m_u)$.
- *initialDecisionSet*($\boldsymbol{S_u}$, $\boldsymbol{D_l}$): It puts the corresponding literal of each clause of the unit clause set $\boldsymbol{S_u}$ into $\boldsymbol{D_l}$ in sequence, i.e., the literal of each unit clause is used to initialize $\boldsymbol{D_l}$. Suppose that there are m_u clauses in the set $\boldsymbol{S_u}$, the time complexity of this function is $O(m_u)$.
- *paringCondition*(C_b, $\boldsymbol{D_l}$): Return TRUE if and only if there is a literal l of the clause C_b can form a complementary pair with a literal in the set $\boldsymbol{D_l}$, and this complementary pair is the first occurrence with respect to the literal l during the S-CS deduction. Suppose the clause C_b has n literals and the set $\boldsymbol{D_l}$ has x literals, the time complexity of this function is $T(n, x) = n * x = O(nx)$.
- *separateCla*(C_b, $\boldsymbol{P}$, $\boldsymbol{D_l}$): It applies clause separation method of S-CS rule with the clause C_b and the two set $\boldsymbol{P}$, $\boldsymbol{D_l}$, separate the clause C_b into $C_b^{\sigma+}$ and $C_b^{\sigma-}$ under a substitution σ, and generate a CSC C_r. Then $C_b^{\sigma-}$ is used to extend the SC. Suppose the clause C_b has n literals and the set $\boldsymbol{D_l}$ has x literals, the time complexity of this function is $T(n, x) = nx = O(nx)$.
- *updateDecisionSet*(C_b, $\boldsymbol{D_l}$): According to the heuristic strategies, this function selects a literal from the clause C_b after the separation as a new decision literal, then put the new decision literal into the literal set $\boldsymbol{D_l}$. Suppose the clause C_b has n literals and only one literal in C_b is separated into C_b^-, the time complexity of this function is $T(n) = n - 1 = O(n)$.
- *selectGivenCla*($\boldsymbol{S_e}$): According to the heuristic strategies, it selects a clause from the clause set $\boldsymbol{S_e}$ as the next clause to participate in the deduction. The time complexity of this function is $O(1)$.
- *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$): Return TRUE if and only if the deduction step generated the clause C_r' satisfies any one of the five conditions of invalid separation. (1) The clause C_r' is subsumed by one clause from the clause set $\boldsymbol{S}$ or the clause set $\boldsymbol{R}$; (2) The clause C_r' is tautology; (3) The maximum term depth of clause of the clause C_r' exceeds the set threshold; (4) The number of literals of the C_g^+ part of C_g after the separation exceeds the set threshold; (5) The maximum term depth of literal of each literal in the C_g^+ part of C_g after the separation exceeds the set threshold. Suppose the clause C_r' has r literals, the clause C_g has n literals and the set $\boldsymbol{S}$ or $\boldsymbol{R}$ has m clauses, the time complexity of this function is $T(r, n, m) = rm + r(r-1) + r + (n-1) + (n-1) = O(rm + r^2)$.
- *rollbackPath*(C_r', C_g, $\boldsymbol{P}$, $\boldsymbol{D_l}$): Clear and restore the corresponding information of the deduction. (1) Clear the substitutions corresponding to the deduction step generated the clause C_r'; (2) Clear the deduction identifiers of the clause participated in the deduction step that generated the clause C_r'; (3) Delete the generated clause C_r'; (4) Remove

the C_g^- part of C_g from the corresponding CSC and the C_g^+ part of C_g from the corresponding SC. Suppose the clause C_r' has r literals, the clause C_g has n literals and the set $\boldsymbol{S}$ or $\boldsymbol{R}$ has m clauses, the time complexity of this function is $T(r,n,m) = r + 1 + r + n = O(r+n)$.

- *stopDeduction*(C_r'): Return TRUE if and only if the generated clause C_r' during the deduction step satisfies the five exit conditions. (1) The clause C_r' is an empty clause, i.e., **UNSAT** found; (2) The number of literals in the clause C_r' reaches the set threshold; (3) The maximum term depth of clause in the clause C_r' reaches the set threshold; (4) The running time reaches the set threshold; (5) The consumed memory reaches the set threshold. Suppose the clause C_r' has r literals, the time complexity of this function is $T(r,n,m) = 1 + 1 + r + 1 + 1 = O(r)$.
- *backwardSubsumed*($\boldsymbol{S}$, C_r''): Remove the clause that is subsumed by the clause C_r'' from the clause set $\boldsymbol{S}$. Suppose the clause C_r'' has r literals and the set S has m clauses, the time complexity of this function is $T(r,n,m) = rm = O(rm)$.

Suppose the clause set $\boldsymbol{S}$, $\boldsymbol{S_u}$, $\boldsymbol{S_b}$, $\boldsymbol{S_e}$ have at most m, m_u, m_b, m_e clauses, respectively, in the deduction process, and the decision literal set $\boldsymbol{D_l}$ has at most x literals in the deduction process. Combining the above analysis of the time complexity of each function, we do a further time complexity analysis of Algorithm 1.

(1) Line 5 – 14: The set $\boldsymbol{S_b}$ holds binary clauses, then the number of literals in any clause C_b is 2 and the time complexity of the function *updateDecisionSet*(C_b, $\boldsymbol{D_l}$) is $O(1)$. Therefore, the time complexity of line 5 – 14 is a function of the variables m_b, x, and the corresponding time complexity is shown in formula (1).

$$T(m_b, x) = m_b + m_b 2x + m_b 2x = O(m_b x) \tag{1}$$

(2) Line 16 – 30: Suppose the longest clause in the set $\boldsymbol{S_e}$ has n literals, the clause C_r' has at most r literals. Therefore, the time complexity of lint 16 – 30 is a function of the variables m_e, m_b, n, x, r, and the corresponding time complexity is shown in formula (2).

$$\begin{aligned} T(m_b, m_e, n, x, r) &= m_e + m_e(nx + rm + r^2 + 2r + 2n + r + m_b x) \\ &= O(m_e nx + m_e rn + m_e r^2 + m_e m_b x) \end{aligned} \tag{2}$$

(3) Line 30 – 33: Suppose the set $\boldsymbol{R}$ has m_r clauses and the longest clause in the set $\boldsymbol{R}$ has r literals. Therefore, the time complexity of lint 30 – 33 is a function of the variables m, m_r, r, and the corresponding time complexity is shown in formula (3).

$$T(m, m_r, r) = m_r(rm) = O(m m_r r) \tag{3}$$

From the above analysis, the time complexity of BCR algorithm is a function of the variables m, m_u, m_b, m_e, m_r, n, x, r. The sets $\boldsymbol{S_u}$, $\boldsymbol{S_b}$, $\boldsymbol{S_e}$, $\boldsymbol{R}$ are actually subsets of the set $\boldsymbol{S}$ in the deduction process, so it is feasible for the variable m to replace the four variables m_u, m_b, m_e, m_r. Therefore, the time complexity of BCR algorithm is shown in formula (4).

$$\begin{aligned} &T(m, m_u, m_b, m_e, m_r, n, x, r) = T(m, n, x, r) \\ &= m + m_u \log m_u + m_u + m_b x + m_e nx + m_e rn + m_e r^2 + m_e m_b x + m m_r r \\ &= O(m^2 x + m^2 r + m \log m + m r^2 + mnx + mrn) \end{aligned} \tag{4}$$

Different from the DISCOUNT algorithm, BCR algorithm has the multiple advantages introduced as follows.

(1) *Multi-clause and dynamic deduction*. In the function *separateCla*(C_b, $\boldsymbol{P}$, $\boldsymbol{D_l}$), there are more than two clauses participated in the deduction, including the clause C_b, and the clauses whose decision literal forms the complementary pair with the literal of C_b. At the same time, at least one literal is eliminated in each clause participated in the deduction. The number of clauses and literals at each deduction step is determined according to how many decision literals of $\boldsymbol{D_l}$ form complementary pairs with the literals in the clause C_b. Therefore, each deduction of BCR algorithm is multi-clause and dynamic.

(2) *Controllable deduction*. BCR algorithm can control the deduction process and deduction results flexibly through the implementation of the two functions, *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$) and *stopDeduction*(C_r'). In the two functions, we set some conditions and thresholds to get the clauses that satisfy the set requirements. For example, if the threshold

mentioned in condition (5) of *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$) is set as 1, then each deduction generates unit clause. Of course, we can add or remove the condition according to some specific requirements in the two functions. For example, if the deduction needs the clauses without equality literal, then we add a condition that the clause C_r' has at least one equality literal in the function *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$).

(3) *Binary clause reusing deduction*. In Step 5 of BCR algorithm, the deduction sequence goes to Step 4 when the deduction on a binary clause C_b has completed and the generated CSC is not empty, then the clause C_b will be checked again whether there is a literal of C_b that satisfies paring condition. If yes, C_b will be reused. This iterative process will continue, i.e., reuse the clause C_b continuously until there is no literal in C_b satisfying paring condition. On the other hand, this iterative process will generate a lot of unit clauses, due to reusing binary clauses. Accordingly, some new decision literals will be generated during this iterative process.

(4) *Guided deduction*. The function *updateDecisionSet*(C_b, $\boldsymbol{D_l}$) can select a literal from the clause C_b after the separation as a new decision literal according to the heuristic strategies. Significantly, these decision literals can synergistically determine the selection of the clause participating in the deduction and the selection of eliminated literals. That is to say, the deduction path can be guided by the decision literals. We can set specify heuristic strategies to optimize the deduction path.

(5) *No need for a lot of heuristic strategies*. Because BCR algorithm has the above advantages, it can allow multiple clauses to participate in the deduction while eliminating more literals at each deduction step. As a result, the algorithm does not require a lot of heuristics to select clauses or literals to generated qualified clause, e.g., the unit clause with equality literal. At the same time, the algorithm can generate many short (unit or binary) clauses, thus it can reduce some simplification strategies.

3.2 Related heuristic strategies

From the description of BCR algorithm, we can see that there are some heuristic strategies which are divided into three categories such as global threshold strategy, clause selection strategy, decision literal selection strategy.

3.2.1 Global threshold strategy

The global thresholds of BCR algorithm is set ahead according to the global threshold strategy. These global thresholds are mainly implemented in the stage of deduction evaluation, specifically the two functions *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$) and *stopDeduction*(C_r'), and they are introduced in the following.

(1) Global threshold of the maximum term depth. There is minor difference about the definition of the maximum term depth between clause and literal in BCR algorithm.

Suppose a *n*-ary clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$, the maximum term depth of the clause C is equal to the maximum value of the maximum term depth of its literals, and the corresponding calculation formula.

$$MTD(C) = max\big(MTD(l_i)\big), 1 \le i \le n \tag{5}$$

The maximum term depth of the literal l_i is equal to the sum of function nesting depths of all terms in l_i, and the corresponding calculation formula.

$$MTD(l_i) = \sum Layer(t_j) \tag{6}$$

According to our empirical experience, the global threshold of the maximum term depth of clause does not greater than $1.3 * mtd$, where mtd is the maximum term depth of clause in the original clause set, and the global threshold of the maximum term depth of literal is set to 6.

(2) Global threshold of the number of literals of the C_g^+ part of a clause C_g after the separation ($GLeftNum$). This threshold is dynamically adjusted as the deduction progresses. In order to preferentially generate short clauses, the threshold is initially set to 1 by default, then gradually increased by 1, but generally not greater than 4.

(3) Global threshold of the number of literals in CSC. This threshold does not greater than $1.2 * LitNum$ according to our empirical experience, where $LitNum$ is the maximum number of literals of a clause in the original clause set.

(4) Global threshold of the running time. This threshold is maximum time for BCR algorithm to prove a single problem, where the units are seconds.

(5) Global threshold of the consumed memory. This threshold is maximum consumed memory for BCR algorithm to prove a single problem, where the units are megabytes.

3.2.2 Clause selection strategy

Clause selection strategy is used to select a clause from a given clause set as the next clause to participate the deduction, realized by the function *selectGivenCla*($\boldsymbol{S_e}$). The selection principle is based on some features or assigned weights of clause. Therefore, we introduce the features or weights of clause.

(1) Symbol-counting of a clause. This feather refers to Ref. [55]. The three symbols, i.e., function, variable and constant, are respectively assigned three weights w_f, w_v, w_c. Then the calculation formula of symbol-counting of a clause C.

$$Sym(C) = num_f(C) * w_f + num_v(C) * w_v + num_c(C) * w_c \tag{7}$$

Where $num_f(C)$, $num_v(C)$, $num_c(C)$ are the number of function symbols, variable symbols, constant symbols respectively. According to specific requirements, the three weights w_f, w_v, w_c can be set to required values. For example, if the deduction needs a clause with more constants, then $w_f = 1$, $w_v = 1$, $w_c = 2$.

(2) Invalid weight of a clause. This weight considers three features of a clause C participated the deduction, the number of invalid separations, the maximum term depth of clause and the number of literals in C^+ part. The invalid separation is the deduction step that does not satisfy any one of the five conditions of *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$). The corresponding calculation formula is as follows.

$$IW(C) = Itimes(C) + \frac{TD(C)}{maxTD} + \frac{LN(C)}{maxLN} \tag{8}$$

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

(3) Valid weight of a clause. This weight is the opposite of invalid weight of a clause. This weight considers also three features of a clause C participated in the deduction, the number of valid separations, the maximum term depth of clause and the number of literals in C^+ part. The valid separation is the deduction step that does not satisfy the five conditions of *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$). The corresponding calculation formula is as follows.

$$VW(C) = Vtimes(C) + \frac{TD(C)}{maxTD} + \frac{LN(C)}{maxLN} \tag{9}$$

Where $Vtimes(C)$ is the number of valid separations in which clause C has participated (the separation of clause C turned out to be valid once, $Itimes$ of clause C will be increased by 1), $TD(C)$ is the max term depth of clause and $LN(C)$ is the number of literals in C^+ part of C after this valid separation.

(4) The number of literals in a clause. S-CS rule can eliminate multiple literals from a clause participated in the deduction at a S-CS deduction step. Therefore, if the clause with fewer literals preferentially participate the deduction, the generated CSC is more likely to be a clause with fewer literals.

3.2.3 Decision literal selection strategy

Decision literal selection strategy is used to select one literal from a clause to participate in the deduction as a new decision literal in the function *updateDecisionSet*(C_b, $\boldsymbol{D_l}$). Similar to clause selection strategy, decision literal selection strategy considers some assigned weight of literal as well.

(1) Symbol-counting of literal. Similar to symbol-counting of clause, the three symbols, i.e., function, variable and constant, are respectively assigned three weights w_f, w_v, w_c in literal. The calculating formula is the same as that of symbol-counting of clause, thus we do not describe here.

(2) The number of times a literal becoming decision literals ($NumD$). This count records how many times a literal works as a decision literal during the deduction process. In order to make the literals in the decision literal set $\boldsymbol{D_l}$ as different as possible or avoid falling into the local optimum, a literal with smaller $NumD$ is selected preferentially to become a new decision literal in a clause.

(3) The ratio of independent variables to shared variables (independent-shared ratio) in a literal. Precisely, this ratio is that the ratio of the number of independent variables to the number of shared variables in a literal. A variable is called *shared variable* if it exists in more than one literal in a clause, otherwise it is called *independent variable*. In the process of S-CS deduction, if an independent variable is changed under a certain substitution, this will not affect other literals in a clause. Therefore, a literal with larger independent-shared ratio is preferred in the S-CS deduction.

4 Extended ATP of E with Binary Clause Reusing Algorithm

In order to improve the performance of E, BCR algorithm is integrated into the architecture of E as an independent algorithm module so as to form an extended ATP of E, dubbed E_BCR. The purpose of adopting this extended mode is to avoid affecting other modules of E.

On the other hand, for better interaction between BCR algorithm and E, we have made some modifications to the source code of the output module of E so that E can output the intermediate clauses that generated during the deduction process. BCR algorithm can receive the clauses generated from previous procedure of E_BCR, then output the generated CSCs to subsequent procedure of E_BCR.

At the same time, in order to generate more effective clauses for E during the S-CS deduction, we add some conditions to the function *invalidSeparation*(C_r', C_g, $\boldsymbol{R}$, $\boldsymbol{S}$) in BCR algorithm. (1) If the clause C_r' contains no equality literal, this deduction is invalid. (2) If the clause C_r' is generated by the goal clauses of the original clause set, this deduction is invalid. (3) If the symbol-counting of the clause C_r' exceeds the maximum symbol count of the clause in the original clause set, this deduction is invalid. In addition to these three conditions, we fix some thresholds of heuristic strategy. (1) Global threshold of the maximum term depth of clause is set to the maximum term depth of clause in the original clause set. (2) Global threshold of the maximum term depth of literal is set to 6. (3) $GLeftNum$ is fixed to be not greater than 2. (4) Global threshold of the number of literals in CSC is equal to the maximum number of literals of the clause in the original clause set. (5) The three weights w_f, w_v, w_c in the calculation formula of symbol-counting of clause or literal are set to 1, 1, 2 respectively.

Specifically, suppose $\boldsymbol{S}$ is a given clause set, the workflow of the extended ATP E_BCR for proving $\boldsymbol{S}$ is described as follows.

Step 1: E is performed to find the proof search for the clause set $\boldsymbol{S}$. If the proof is not found successfully, E outputs the clauses generated during the deduction process, then these generated clauses are put into the clause set $\boldsymbol{S}$. Otherwise, exit!

Step 2: The procedure of BCR algorithm is executed to the clause set $\boldsymbol{S}$ for the S-CS deduction. If the deduction generates an empty clause, then exit! Otherwise, these generated CSCs are put into the clause set $\boldsymbol{S}$.

Step 3: E is performed again to find the proof search for the clause set $\boldsymbol{S}$ from Step 2. If the proof is found, output **UNSAT**. Otherwise, exit!

Step 3 is the key part for improving the performance of E, since BCR algorithm offers many effective clauses for E in Step 2, which is illustrated by the experimental results in Section 5. Fig. 1 shows the flow chart of the extended E_BCR.

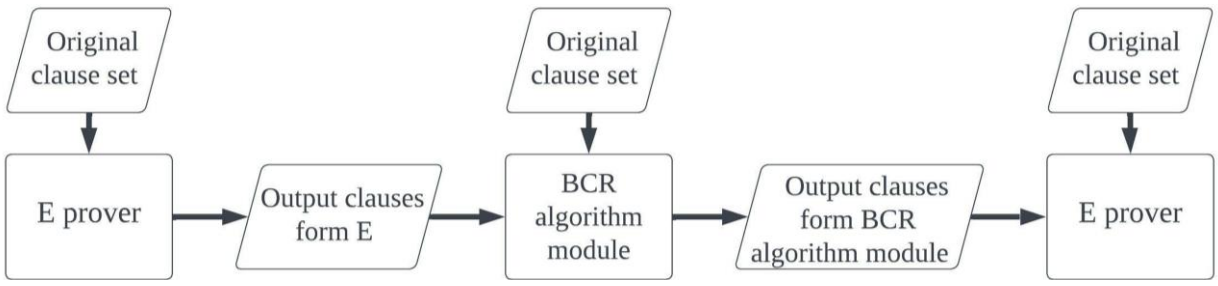

Fig. 1. The flow chart of the extended E_BCR

In order to facilitate BCR algorithm to accept the output clauses from E and perform operations on terms, BCR algorithm module uses a three-level structure of term storage.

1. Global shared level. Based on the idea of WAM-terms storage structure [58], a global table is used to store all the ground terms in the original clause set and deduction process, and another global table is used to store all symbols in the clause set, i.e., predicate symbols, function symbols and constant symbols.

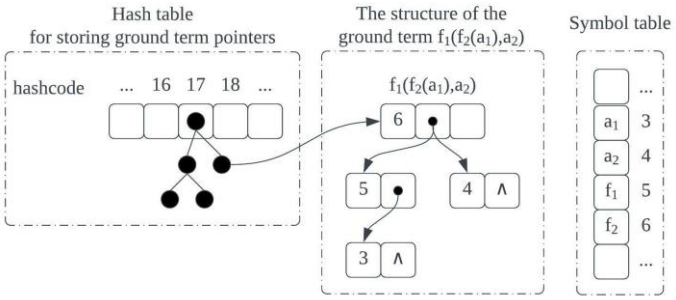

Fig. 2. The representation of the data structure of ground term

We use the data structure shown in Fig. 2 to store the ground terms. In Fig. 2, we use Hash-Consing [59] technology to quickly generate and store ground terms, and adopt a splay tree to address hash conflicts when then same hash code occurs; the structure of each ground term is a chain table, where each node is a ground term, including the index code in the symbol table and an array of sub-term pointers; the symbol table uses an array to store all symbols, including predicate symbols, functions symbols and constant symbols. Meanwhile, we also use a splay tree to store all symbols in order to find a given symbol quickly or reduce the time complexity of finding symbol (specifically, $T(n) = O(n)$ of the array and $T(n) = O(\log n)$ of the splay tree).

2. Literal shared level. BCR algorithm module places the composite terms and literals containing variables in literal shared level. In FOL, the set of variables between clauses does not intersect, that is, the variables in each clause are not the same. Therefore, the variables in a new inferred clause need to be renamed during the deduction process. For example, there are two clauses $C_1 = P_1(x_1) \vee P_3(a)$, $C_2 = P_2(x_1) \vee \sim P_3(a)$, the variable x_1 in C_1 and C_2 are essentially two different terms, although they have the same symbol, and therefore, the inferred clause $R(C_1, C_2) = P_1(x_1) \vee P_2(x_1)$ could be renamed to $P_1(x_{101}) \vee P_2(x_{102})$. For efficiency, each clause independently maintains a table of variables and a table of composite terms and literals containing variables. We also use hash table like Hash-Consing and splay tree to store composite terms and literals containing variables.

3. Clause shared level. Each clause has an independent array to store all variable terms in this clause. According to this array, the system can quickly identify independent variables and shared variables, as well as easily manipulate the substitution terms of the variables.

The advantages of this three-level structure are: (1) Avoiding frequent traversal and copy operations of composite terms due to variable renaming operations. (2) Ground terms are shared globally, and atom terms are shared in a clause. (3) Efficiently construct literals and find specified terms. (4) Efficiently delete specified clauses.

5 Experimental Studies

In order to evaluate the performance of E_BRC, we set two experimental groups: 1. The experiment of CASC competition problems and problems with status of theorem in the TPTP-v7.5.0; 2. The experiment of problems with rating of 1 in the TPTP-v7.5.0. Notably, TPTP is a standard library of test problems for ATP, which covers dozens of scientific research domains. The rating of a problem denotes the difficulty of this problem in TPTP benchmark library, which is a real number in the range 0.0 to 1.0, where the problem with rating of 0.0 means the easiest, and the problem with rating of 1.0 means that cannot be solved by all provers, i.e., the hardest problem [48].

The experiments are implemented on a PC with 3.6GHz Inter(R) Core (TM) i7-7700 processor and 16 GB memory, OS Ubuntu 20.04 64-bit. For one single problem, the CPU time limit is 300 seconds. In the experiments, the version of E to be extended is 2.6 that is latest version of E, thus the extended prover from E 2.6 is called E_BCR 2.6. In order to guarantee the correctness of proof search of BCR algorithm, the well-known prover, Prover9, is used to verify the deduction path.

The parameter settings of the heuristic strategy used for the experiment are described in the third paragraph of Section 4. In addition, for one single problem, the CPU time limit for Step 1 in the workflow of E_BCR is 25 seconds, and the CPU time limit for Step 2 in the workflow of E_BCR is 25 seconds, and the remaining of 250 seconds is the CPU time limit for Step 3 in the workflow of E_BCR.

5.1 The experiment on CASC competition problems and problems with status of theorem

Actually, this experimental group is divided into two experiments according to the type of problems. First experiment uses CASC FOF division problems (2016-2021) with a total number of 3000 problems (500 problems per year). Second experiment uses all problems with status of theorem in TPTP-v7.5.0, a total of 6537. Two experiments are comparison experiments between E_BCR 2.6 and E 2.6. Incidentally, the status of problem refers to the that every model of the axioms (and other non-conjecture formulae, e.g., hypotheses and lemmas), and there are some such models, is a model of all the conjectures [48].

We first present the results of the experiment on CASC competition problems below. Fig. 3 shows a comparison on performance of E 2.6 versus E_BCR 2.6 for data from 2016 to 2021, and it can be seen that E_BCR 2.6 outperforms E 2.6 both in terms of average run time and number of solved problems. From the 25 seconds point onwards, the trends of the two scatter lines in Fig. 3 begin to diverge, the scatter line representing E_BCR 2.6 is further down than the scatter line representing E 2.6. The trend of the two scatter lines also shows that E_BCR's performance outperforms E, while the BCR algorithm does improve the ability and efficiency of E after 25 seconds. Statistics of the experimental are shown in Table 3. The row "E 2.6" shows the number of problems solved by E 2.6; the row "E_BCR 2.6" shows the number of problems solved by E_BCR 2.6; the row "gap" shows the number of problems solved by E_BCR 2.6 more than E 2.6. We can see that the number of problems solved by E_BCR 2.6 is more than the number of problems solved by E 2.6 from the experimental results on each year CASC problems from 2016 to 2021. Notably, for the data in 2020, E_BCR 2.6 solved 417 problems with 28 more than E 2.6, and for the data in 2017, the number of problems solved by E_BCR 2.6 is 427, the maximum for data of six years. From the column "mean" of Table 3, the number of problems solved by E_BCR 2.6 is 420 for the data of six years, an average of 21 more than E 2.6 per year.

The results are further analyzed as follows. E 2.6 did not solve 602 problems out of 3000 problems, but some of 602 problems are the same. The number of unsolved problems by E 2.6, in fact, is 383, while among these problems, E_BCR 2.6 solved 106 problems accounting for 27.68% of these 383 problems.

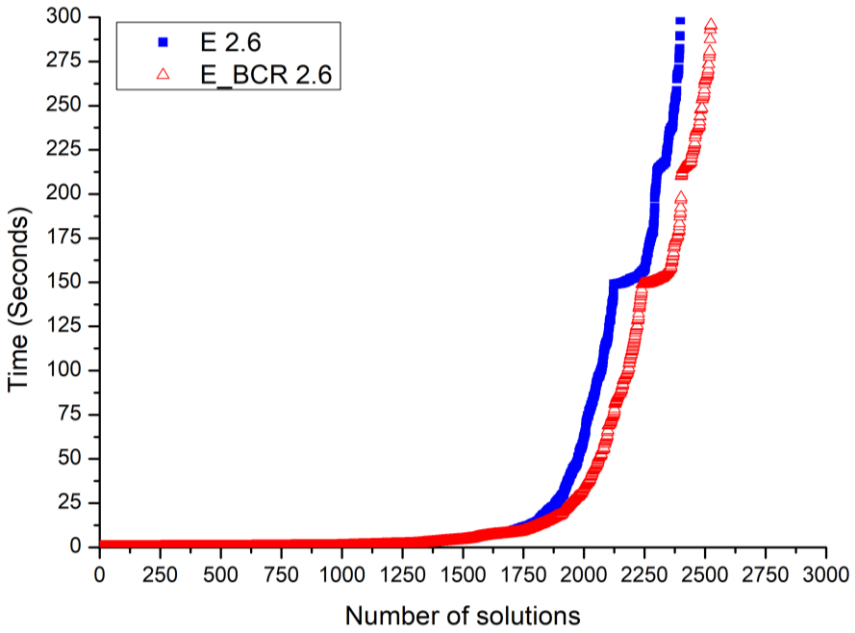

Fig. 3. Performance of E 2.6 versus E_BCR 2.6 over problems data 2016 to 2021

Table 3 Comparison of the number of problems solved by E 2.6 and E_BCR 2.6

	2016	2017	2018	2019	2020	2021	Mean
E 2.6	408	412	401	392	390	395	399
E_BCR 2.6	**425**	**427**	**419**	**418**	**418**	**418**	**420**
Gap	**17**	**15**	**18**	**26**	**28**	**23**	**21**

Table 4 The list of 106 problems solved by E_BCR 2.6 but not by E 2.6

No	Problem Name	Rating	No	Problem Name	Rating	No	Problem Name	Rating	No	Problem Name	Rating
1	AGT007+1	0.75	28	GEO450+1	0.67	55	LCL680+1.010	0.93	82	SWB068+1	0.72
2	AGT007+2	0.75	29	GEO502+1	0.61	56	NUM314+1	0.83	83	SWB081+1	0.89
3	AGT008+2	0.78	30	GEO506+1	0.75	57	PRO016+2	0.69	84	SWB082+1	0.89
4	AGT011+2	0.78	31	GEO512+1	0.69	58	REL016+2	0.87	85	SWB088+1	0.94
5	AGT012+2	0.64	32	GEO513+1	0.83	59	REL016+3	0.87	86	SWB093+1	0.89
6	AGT013+1	0.75	33	GRP620+2	0.89	60	REL017+4	0.87	87	SWB094+1	0.92
7	AGT013+2	0.81	34	GRP655+1	0.65	61	REL041+1	0.87	88	SWB095+1	0.94
8	AGT026+2	0.69	35	GRP720+1	0.83	63	SCT102+1	0.81	89	SWB098+1	0.89
9	BIO002+1	0.81	36	ITP003+1	0.97	63	SCT115+1	0.69	90	SWB102+1	0.97
10	BIO005+1	0.81	37	ITP004+4	0.83	64	SCT123+1	0.69	91	SWB107+1	0.92
11	COM148+1	0.81	38	ITP005+4	0.86	65	SCT169+3	0.97	92	SWB108+1	0.94
12	CSR179+1	0.81	39	ITP020+4	0.86	66	SCT170+3	0.94	93	SWV448+1	0.61
13	CSR191+1	0.97	40	KLE169+1	0.72	67	SCT170+6	0.94	94	SWV458+1	0.67
14	CSR215+1	0.89	41	KRS261+1	0.36	68	SET690+4	0.69	95	SWV459+1	0.61
15	CSR240+1	0.97	42	LAT286+2	0.78	69	SET796+4	0.56	96	SWV467+1	0.58
16	GEO273+1	0.64	43	LAT347+2	0.89	70	SET948+1	0.94	97	SWV472+1	0.61
17	GEO276+1	0.78	44	LAT349+1	0.97	71	SEU206+2	0.81	98	SWW096+1	0.75
18	GEO291+1	0.72	45	LCL557+1	0.92	72	SEU383+2	0.94	99	SWW218+1	0.67
19	GEO292+1	0.67	46	LCL564+1	0.86	73	SEU420+1	0.78	100	SWW470+5	0.64
20	GEO295+1	0.72	47	LCL570+1	0.94	74	SEU420+4	0.86	101	SWW470+7	0.72
21	GEO296+1	0.81	48	LCL642+1.015	0.36	75	SEU449+4	0.94	102	SWW474+7	0.78
22	GEO309+1	0.92	49	LCL652+1.015	0.64	76	SEV521+1	0.67	103	SYN076+1	0.75
23	GEO316+1	0.86	50	LCL652+1.020	0.79	77	SWB010+1	0.83	104	SYO604+1	0.21
24	GEO322+1	0.92	51	LCL660+1.005	0.36	78	SWB012+1	0.86	105	TOP025+3	0.81
25	GEO323+1	0.86	52	LCL660+1.010	0.43	79	SWB020+1	0.92	106	TOP035+2	0.89
26	GEO324+1	0.86	53	LCL660+1.015	0.43	80	SWB025+1	0.67			
27	GEO442+1	0.89	54	LCL660+1.020	0.57	81	SWB027+1	0.86			

Table 5 The experimental results of problems with status of theorem

	E 2.6	E_BCR 2.6	Advantage
[0.0, 0.2)	1982	1985	3
[0.2, 0.4)	1196	1215	19
[0.4, 0.6)	911	982	71
[0.6, 0.8)	690	863	173
[0.8, 1.0]	338	612	274
Total	5117	5657	540
Solving rate	78.28%	86.54%	8.26%

Table 6 The list of 18 problems with rating of 1 solved by E_BCR 2.6

No	Problem Name	Time(s)	No	Problem Name	Time(s)
1	ALG001-1	192.74	10	NUM657+4	238.93
2	CSR224+1	186.74	11	NUM658+4	241.96
3	GEO326+1	146.43	12	NUM659+4	241.92
4	ITP003+4	237.44	13	NUM726+4	284.17
5	LAT215-1	287.48	14	RNG027-10	180.22
6	LCL042-10	299.18	15	SET032-3	169.91
7	NUM428+1	247.68	16	SET033-3	152.82
8	NUM640+4	281.66	17	SET279-6	122.18
9	NUM642+4	284.23	18	SWB020+3	279.84

Table 4 lists the name and rating of these 106 problems, which shows that the average rating of these 106 problems is 0.78, and we can see that the problems that cannot be solved by E 2.6 but can be solved by E_BCR 2.6 are quite difficult. There are even 22 problems with rating greater than 0.9, and 58 problems with rating greater than 0.8 accounting for 54.72% of the 106 problems. On the other hand, these 106 problems cover 24 scientific domains (the domain corresponds to the first three letters of the problem name [48]), 44 of which respectively are GEO (representing Geometry), SWB (representing Semantic Web) and LCL (representing Logic Calculi, a branch of logic), accounting for majority of the 106 problems.

In the following, we present the results of the experiment problems with status of theorem. Table 5 shows the experimental results, and we divided the experimental results into 5 columns according to rating of problems. For these 6537 problems, E_BCR 2.6 solves 5657 problems with 540 more than E 2.6 which solves 5117 problems. When the experimental results converted to percentages, E_BCR 2.6 solves 86.54% of 6537 problems, while E solves 78.28% of 6537 problems. From this perspective, the BCR algorithm improves the performance of E 2.6 by 8.26%. Furthermore, from the statistics about the rating range, the number of problems solved by E_BCT 2.6 in the range of [0.6, 1.0] substantially exceeds that of E, especially for the more difficult problems, i.e., the range of [0.8, 1.0].

In general, the problems in CASC FOF division and the problems with status of theorem are plain problems, namely there are very few problems with rating of 1.0 or 0.0. Therefore, the experimental results illustrate the performance of the extended prover E_BCR is stronger than that of E for the plain problem solving, especially in the three domains problems of GEO, SWB and LCL, and shows BCR algorithm has strong universality.

5.2 Experiments on problems with rating of 1

From the analysis of Table 4 and Table 5, E_BCR initially shows the ability to solve the hard problems, in order to further evaluate this ability, we set up the experiment of problems with rating of 1. This experiment uses all problems with rating of 1 in TPTP-v7.5.0 that is the latest release of TPTP benchmark library, a total of 1584 problems. According to query of TSTP library (a library of solutions to problems for TPTP library), the problems with rating of 1 are not solved by any current theorem provers including E, and we also tested these 1584 problems using E in this experimental environment and found no problems solved by E. Therefore, only results of E_BCR 2.6 are listed as follows.

Table 6 lists the experimental results which shows that E_BCR 2.6 solved 18 problems. The average time for E_BCR 2.6 to solve each problem is 226.42 seconds. These 18 problems cover 10 scientific domains, 7 of which are NUM (representing number theory, a branch of mathematics), accounting for majority of the 18 problems. It's worth mentioning that the problem with rating of 1 is the most difficult problem in the TPTP benchmark library, which also means that no provers can solve it. Therefore, the ability of a prover to solve the problem with rating of 1 is a key indicator of the performance of the prover. The experimental results of E_BCR 2.6 solving 18 problems with rating of 1 are quite significant. The experimental results not only show that E_BCR can solve some problems that cannot be solved by all the other provers, but also illustrate that BCR algorithm can effectively improve the performance of E.

5.3 Analysis of experimental results

In conclusion, the experimental results shows that E_BCR is a powerful extended E prover. BCR algorithm not only improves the ability of E to solve plain problems, but also aids E in solving 18 of the hardest problems, i.e., problem with rating of 1. These fully illustrate that BCR algorithm is an effective deduction algorithm based on S-CS rule, and E_BCR is a successful extended E prover to improve the performance of E. BCR algorithm implements S-CS rule, so the multiple advantages of S-CS rule are inherited and implemented by BCR algorithm. The reasons for the increase of E_BCR are analyzed as follows.

1. In the BCR algorithm, the decision literals generated by the S-CS deduction can guide subsequent deduction paths. In one S-CS deduction, more different decision literals are generated, and these literals continuously optimize the subsequent deduction paths, i.e., more generated clauses with good characteristics, such as fewer literals, lower symbol-counting, fewer term depths.

2. When the S-CS deduction produces a certain number of decision literals, a binary clause reused in the deduction is capable of producing multiple (two or more) SCS, and these deductions involving the binary clause are well deductions. Meanwhile, the deduction process of reusing binary clause can generate a large number of unit clauses. The unit clauses are filtered by the heuristic strategy and then fed to E. In the calculus of E, the number of literals in the deduction step in which the unit clause is involved does not additionally increase the number of literals of generated clause, so these unit clauses can enhance the ability of E to generate empty clause.

3. The deduction in the BCR algorithm is a controllable deduction. According to set the heuristic strategy, BCR algorithm can control the deduction process and obtain the required decision literals and SCSs. Therefore, E can obtain eligible clauses from BCR algorithm, such as the clause with fewer literals and equality literals.

4. It is easier for BCR algorithm that is able to eliminate more than two literals from multiple clauses at each deduction and to generate the clause with fewer literals, especially, unit clauses. BCR algorithm can remove the clauses are subsumed by the generated unit clauses from the clause set and control the number of literals in the generated clauses, which can effectively alleviate the search space expansion to a certain extent.

The above analysis explains partly why BCR algorithm can effectively improve the performance of E.

6 Conclusions and Future Work

In the field of knowledge representation and reasoning, automated reasoning plays a central supporting role, and first-order logic as the basis of automated reasoning is an important knowledge representation language for many AI problems and mathematical problems. ATP is the essential and powerful inference engine corresponding to first-order logic. In the field of ATP, E is one of the most acclaimed ATPs, due to its excellent performance for the first-order logic problem. However, there are still a lot of unsolved problems by E in the latest release of TPTP benchmark library, especially the hard problems. S-CS rule is a novel inference method for automated reasoning, it has some abilities that E does not possess such as multi-clause, clause-reusing and controlled abilities. In the present work, we have proposed a binary clause reusing algorithm based on S-CS rule, dubbed BCR algorithm, along with some related heuristic strategies of BCR algorithm. This algorithm takes took better advantage of the abilities of the S-CS rule, especially clause-reusing ability. At the same time, BCR algorithm can obtain the preferred clauses according to its ability to control the features of generated clauses and eliminate more than two literals from multiple clauses at each the deduction step. We have proposed an extended ATP of E with BCR algorithm, dubbed E_BCR, to boost E's performance even further. In the architecture of E_BCR, BCR algorithm is integrated into E as an independent algorithm module. The CASC FOF division problems (2016-2021), problems with status of theorem and problems with rating of 1 in TPTP library have been used to evaluate the performance of E_BCR. The experimental results have shown that BCR algorithm can effectively improve the performance of E and can serve as an important complement to E. For the CASC FOF division problems, the number of problems solved by E_BCR is more than the number of problems solved by E 2.6 from the experimental results on each year CASC problems from 2016 to 2021. For the problems with status of theorem, the BCR algorithm has improved the performance of E 2.6 by 8.26%. Especially, E_BCR has solved 18 problems with rating of 1, which are unsolved by any current ATP. In particular, BCR algorithm can help E solve more problems in many scientific domains such as Geometry, Semantic Web and Logic Calculi.

BCR algorithm effectively improves the performance of E and has significant reasoning capability, while it still has some shortcomings and potential for improvement. For example, the number of heuristic strategies in the strategy library of BCR algorithm is much lower than that of other state-of-the-art ATPs, and BCR algorithm is weak for equality handling. For future work, we intend to further explore more superior inference mechanism based on S-CS rule and more related heuristic strategies from three aspects, so as to further improve the efficiency of provers. Firstly, we plan to further explore the mechanism of reusing clauses based on S-CS rule, and hope to extend this mechanism to non-binary clauses. Secondly, we will delve into the various types of strategies that can effectively enhance ability of S-CS deduction. Thirdly, we plan to investigate new integration modules for BCR algorithm with other leading ATPs, such as Vampire and E, to enable better interaction between BCR algorithm module and other modules of other leading ATPs. Finally, we will explore other applications of BCR algorithm in the field of knowledge representation. Specifically, our team has researched on group theory, therefore we plan to use knowledge-based reasoning in the SUMO (Suggested Upper Merged Ontology) to convert the axioms and theorems in group theory into TPTP format, and then use the ATP based BCR algorithm to prove related problems or theorem in our research.

Abbreviations

AI	Artificial intelligence
KR&R	Knowledge representation and reasoning
FOL	First-order logic
ATP	Automated theorem prover
CASC	CADE ATP System Competition
TPTP	Thousands of Problems for Theorem Provers
S-CS	Standard contradiction separation
CNF	Conjunctive normal form
CSC	Standard contradiction separation clause
SC	Separated standard contradiction
BCR	Binary clause reusing
SUMO	Suggested Upper Merged Ontology

CRediT authorship contribution statement

Peiyao Liu: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing – Original Draft, Writing – Review & Editing. **Shuwei Chen:** Conceptualization, Methodology, Visualization, Funding acquisition, Writing – Review & Editing. **Jun Liu:** Visualization, Writing – Review & Editing, Supervision. **Yang Xu:** Conceptualization, Resources, Supervision, Project administration. **Feng Cao:** Software, Validation, Data Curation, Funding acquisition. **Guanfeng Wu:** Data Curation, Funding acquisition.

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

References

[1] R. Davis, H. Shrobe, P. Szolovits, What is a knowledge representation? AI Mag. 14 (1) (1993) 17-33. https://doi.org/10.1609/aimag.v14i1.1029.
[2] J. Sowa, Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks Cole Pulishing Co., 1999.
[3] T.V. Avdeenko, E.S. Makarova, I.L. Klavsuts, Artificial intelligence support of knowledge transformation in knowledge management systems, in: 2016 13th International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE), 2016, pp. 195-201. https://doi.org/10.1109/APEIE.2016.7807053.
[4] A. Ruf, M. Berges, P. Hubwieser, Classification of programming tasks according to required skills and knowledge representation, in: International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2015), 2015, pp. 57-68. https://doi.org/10.1007/978-3-319-25396-1_6.
[5] P. Di Maio, System level knowledge representation for metacognition in neuroscience, in: International Conference on Brain Informatics (BI 2021), 2021, pp. 79-88. https://doi.org/10.1007/978-3-030-86993-9_8.
[6] W.T. Balke, K. Mainzer, Knowledge representation and the embodied mind: Towards a philosophy and technology of personalized informatics, in: Biennial Conference on Professional Knowledge Management/ Wissensmanagement (WM 2005), 2005, pp. 586–597. https://doi.org/10.1007/11590019_67.
[7] W. Yin, Standard model of knowledge representation, Front. Mech. Eng. 11 (2016) 275–288. https://doi.org/10.1007/s11465-016-0372-3.
[8] E. Cambria, B. White, Jumping NLP curves: A review of natural language processing research, IEEE Comput. Intell. Mag. 9 (2) (2014) 48-57. https://doi.org/10.1109/MCI.2014.2307227.
[9] M.M. Dastani, K.V. Hindriks, P. Novak, N.A.M. Tinnemeier, Combining multiple knowledge representation technologies into agent programming languages, in: International Workshop on Declarative Agent Languages and Technologies (DALT 2008), 2008, pp. 60-74. https://doi.org/10.1007/978-3-540-93920-7_5.
[10] W. Zheng, X. Liu, X. Ni, L. Yin, B. Yang, Improving visual reasoning through semantic representation, IEEE Access, 9 (2021) 91476-91486. https://doi.org/10.1109/ACCESS.2021.3074937.
[11] R. Peñaloza, A brief roadmap into uncertain knowledge representation via probabilistic description logics, Algorithms 14 (10) (2021). https://doi.org/ 10.3390/a14100280.
[12] S. Roychowdhury, M. Diligenti, M. Gori, Regularizing deep networks with prior knowledge: A constraint-based approach, Knowledge-Based Syst. 222 (2021) 106989. https://doi.org/10.1016/j.knosys.2021.106989.
[13] J. Schneider, D. Basin, S. Krstić, D. Traytel, A formally verified monitor for metric first-order temporal logic, in: International Conference on Runtime Verification (RV 2019), 2019, pp. 310-328. https://doi.org/10.1007/978-3-030-32079-9_18.
[14] K.H. Yang, D. Olson, J. Kim, Comparison of first order predicate logic, fuzzy logic and non-monotonic logic as knowledge representation methodology, Expert Syst. Appl. 27 (4) (2004) 501-519. https://doi.org/10.1016/j.eswa.2004.05.012.
[15] A. Bundy, The interaction of representation and reasoning, Proc. R. Soc. A-Math. Phys. Eng. Sci. 469 (2013) 20130194. https://doi.org/10.1098/rspa.2013.0194.
[16] D.W. Loveland, Automated theorem proving: a logical basis, North-Holland, Amsterdam, 1978. https://doi.org/10.1016/0378-4754(80)90081-6.
[17] M.L. Gueye, Modeling a knowledge-based system for cyber-physical systems: Applications in the context of learning analytics, in: International Conference on Computational Collective Intelligence (ICCCI 2019), 2019, pp. 568-580. https://doi.org/10.1007/978-3-030-28374-2_49.
[18] L. Bellomarini, D. Benedetto, G. Gottlob, E. Sallinger, Vadalog: A modern architecture for automated reasoning with large knowledge graphs, Inf. Syst. 105 (2022) 101528. https://doi.org/10.1016/j.is.2020.101528.
[19] L. Ramos, Semantic web for manufacturing, trends and open issues: Toward a state of the art, Comput. Ind. Eng. 90 (2015) 444–460. https://doi.org/10.1016/j.cie.2015.10.013.
[20] P. Quaresma, Automatic deduction in an AI geometry book, in: International Conference on Artificial Intelligence and Symbolic Computation (AISC 2018), 2018, pp. 221-226. https://doi.org/10.1007/978-3-319-99957-9_16.
[21] T. Weithöner, T. Liebig, M. Luther, S. Böhm, F. von Henke, O. Noppens, Real-world reasoning with OWL, in: European Semantic Web Conference (ESWC 2007), 2007, pp. 296-310. https://doi.org/10.1007/978-3-540-72667-8_22.
[22] S. Böhme, M. Moskal, Heaps and data structures: A challenge for automated provers, in: International Conference on Automated Deduction (CADE 2011), 2011, pp. 177-191. https://doi.org/10.1007/978-3-642-22438-6_15.
[23] L. Kovács, Symbolic computation and automated reasoning for program analysis. in: International Conference on Integrated Formal Methods (IFM 2016), 2016, pp. 20-27. https://doi.org/10.1007/978-3-319-33693-0_2.
[24] H. Tuch, G. Klein, A unified memory model for pointers, in: International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2005), 2005, pp. 474–488. https://doi.org/10.1007/11591191_33.
[25] G. Klein, Operating system verification—An overview, Sadhana 34 (2009) 27–69. https://doi.org/10.1007/s12046-009-0002-4.

[26] P. Curzon, A verified compiler for a structured assembly language, in: 1991 International Workshop on the HOL Theorem Proving System and Its Applications, 1991, pp. 253-262. https://doi.org/10.1109/HOL.1991.596292.
[27] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (2009) 107–115. https://doi.org/10.1145/1538788.1538814.
[28] J.A. Robinson, A machine-oriented logic based on the resolution principle, J. ACM. 12 (1965) 23–41. https://doi.org/10.1145/321250.321253.
[29] L. Bachmair, H. Ganzinger, Rewrite-based equational theorem proving with selection and simplification, J. Logic Comput. 4 (1994) 217–247. https://doi.org/10.1093/logcom/4.3.217.
[30] G. Robinson, L. Wos, Paramodulation and theorem-proving in first-order theories with equality, in: J.H. Siekmann, G. Wrightson (Eds.), Automation of Reasoning, Springer, Berlin, 1983, pp. 298-313. https://doi.org/10.1007/978-3-642-81955-1_19.
[31] J.A. Navarro Pérez, A. Rybalchenko, Separation logic + superposition calculus = heap theorem prover, ACM Sigplan Not. 46(6) (2011) 556–566. https://doi.org/10.1145/1993316.1993563.
[32] A. Riazanov, A. Voronkov, Vampire 1.1 (system description), in: International Joint Conference on Automated Reasoning (IJCAR 2001), 2001, pp. 376-380. https://doi.org/10.1007/3-540-45744-5_29.
[33] S. Schulz, E - a brainiac theorem prover, AI Commun. 15(2-3) (2002) 111–126. https://dl.acm.org/doi/10.5555/1218615.1218621.
[34] T. Tammet, GKC: A reasoning system for large knowledge bases, in: International Conference on Automated Deduction (CADE 2019), 2019, pp. 538-549. https://doi.org/10.1007/978-3-030-29436-6_32.
[35] W. McCune, Prover9 manual. https://www.cs.unm.edu/~mccune/prover9/manual/2009-02A/, 2020 (accessed 8 April 2022).
[36] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: International Conference on Automated Deduction (CADE 2019), 2019, pp. 495-507. https://doi.org/10.1007/978-3-030-29436-6_29.
[37] G. Sutcliffe, The CADE ATP system competition — CASC. AI Mag. 37 (2016) 99–101. https://doi.org/10.1609/aimag.v37i2.2620.
[38] S. Schulz, Awards. https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/Awards.html, 2022 (accessed 8 April 2022).
[39] W. McCune, L. Wos, Otter - the CADE-13 competition incarnations, J. Autom. Reasoning 18 (1997) 211–220. https://doi.org/10.1023/A:1005843632307.
[40] J. Denzinger, M. Kronenburg, S. Schulz, DISCOUNT - a distributed and learning equational prover, J. Autom. Reasoning 18(2) 1997 189–198. https://doi.org/10.1023/A:1005879229581.
[41] B. Löchner, Things to know when implementing lpo, Int. J. Artif. Intell. Tools 15(1) (2006) 53–79. https://doi.org/10.1142/S0218213006002564.
[42] B. Löchner, Things to know when implementing KBO, J. Autom. Reasoning 36 (2006) 289–310. https://doi.org/10.1007/s10817-006-9031-4.
[43] S. Schulz, Fingerprint indexing for paramodulation and rewriting, in: International Joint Conference on Automated Reasoning (IJCAR 2012), 2012, pp. 477-483. https://doi.org/10.1007/978-3-642-31365-3_37.
[44] S. Schulz, M. Möhrmann, Performance of clause selection heuristics for saturation-based theorem proving, in: International Joint Conference on Automated Reasoning (IJCAR 2016), 2016, pp. 330-345. https://doi.org/10.1007/978-3-319-40229-1_23.
[45] K. Hoder, G. Reger, M. Suda, A. Voronkov, Selecting the selection, in: International Joint Conference on Automated Reasoning (IJCAR 2016), 2016, pp. 313--329. https://doi.org/10.1007/978-3-319-40229-1_22.
[46] B. Gleiss, M. Suda, Layered clause selection for saturation-based theorem proving, in: Proceeding of Practical Aspects of Automated Reasoning and Satisfiability Checking and Symbolic Computation Workshop, 2020, pp. 34–52.
[47] G. Sutcliffe, The TPTP world – infrastructure for automated reasoning, in: International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2010), 2010, pp. 1–12. https://doi.org/10.1007/978-3-642-17511-4_1.
[48] G. Sutcliffe, The TPTP problem library and associated infrastructure. J. Autom. Reasoning 59 (2017), 483–502. https://doi.org/10.1007/s10817-017-9407-7.
[49] Y. Xu, J. Liu, S. Chen, X. Zhong, X. He, Contradiction separation based dynamic multi-clause synergized automated deduction, Inf. Sci. 462 (2008) 93–113. https://doi.org/10.1016/j.ins.2018.04.086.
[50] F. Cao, Y. Xu, J. Liu, S. Chen, J. Yi, A multi-clause dynamic deduction algorithm based on standard contradiction separation rule, Inf. Sci. 566 (2021) 281–299. https://doi.org/10.1016/j.ins.2021.03.015.
[51] 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 (FLINS 2018), 2018, pp. 725-732. https://doi.org/10.1142/9789813273238_0092.
[52] G. Sutcliffe, The CADE ATP system competition (CASC-28). https://tptp.org/CASC/28/, 2022 (accessed 8 April 2022).
[53] P. Liu, G. Wu, Y. Xu, F. Cao, Extending E prover with fully use binary clauses algorithm based on standard contradiction separation rule, in: 2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE), 2021, pp. 11-14. https://doi.org/10.1109/ISKE54062.2021.9755439.
[54] L. Bachmair, H. Ganzinger, D. McAllester, C. Lynch, Chapter 2 - Resolution theorem proving, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, North-Holland, 2001, pp. 19–99. https://doi.org/10.1016/B978-044450813-3/50004-7.
[55] S. Schulz, Learning search control knowledge for equational deduction, Dissertation, Technische Universität München (2000).
[56] 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.org10.1109/ISKE.2017.8258741.

[57] S. Chen, Y. Xu, J. Liu, F. Cao, Y. Jiang, Clause reusing framework for contradiction separation based automated deduction, in: Proceedings of the 13th International FLINS Conference (FLINS 2020), 2020, pp. 284–291. https://doi.org/10.1142/9789811223334_0035.

[58] T. P. Dobry, An abstract prolog machine, in: A High Performance Architecture for Prolog, Springer, Boston, 1990, pp.23-59. https://doi.org/10.1007/978-1-4613-1529-2_2.

[59] T. Braibant, J.H. Jourdan, D. Monniaux, Implementing and reasoning about hash-consed data structures in Coq. J. Autom. Reasoning 53 (2014) 271–304. https://doi.org/10.1007/s10817-014-9306-0.