# Chapter 4

# CLAUSE LEARNING AS A PROOF SYSTEM

We now move on to satisfiability algorithms and present in this chapter a new (and first-ever) proof theoretic framework for formally analyzing the core of the numerous practical implementations of such algorithms being developed today.

As discussed in Chapter 1, in recent years the task of deciding whether or not a given CNF formula is satisfiable has gone from a problem of theoretical interest to a practical approach for solving real-world problems. SAT procedures are now a standard tool for tasks such as hardware and software verification, circuit diagnosis, experiment design, planning, scheduling, etc.

The most surprising aspect of such relatively recent practical progress is that the best complete satisfiability testing algorithms remain variants of the DPLL procedure for backtrack search in the space of partial truth assignments (cf. Section 2.3). The key idea behind its efficacy is the pruning of the search space based on falsified clauses. Since its introduction in the early 1960's, the main improvements to DPLL have been smart branch selection heuristics such as by Li and Anbulagan [80], extensions like randomized restarts by Gomes et al. [58] and clause learning (cf. Section 2.3.2), and well-crafted data structures such as watched literals for fast unit propagation by Moskewicz et al. [88]. One can argue that of these, clause learning has been the most significant in scaling DPLL to realistic problems. This chapter attempts to understand the potential of clause learning and leads on to the next chapter which suggests practical ways of harnessing its power.

Clause learning grew out of work in artificial intelligence on explanation-based learning (EBL), which sought to improve the performance of backtrack search algorithms by generating explanations for failure (backtrack) points, and then adding the explanations as new constraints on the original problem. The results of de Kleer and Williams [46], Stallman and Sussman [103], Genesereth [55], and Davis [45] proved this approach to be quite effective. For general constraint satisfaction problems the explanations are called "conflicts" or "no goods"; in the case of Boolean CNF satisfiability, the technique becomes clause learning – the reason for failure is learned in the form of a "conflict clause" which is added to the set of given clauses. Through a series of papers and accompanying solvers, Bayardo Jr. and Schrag [13], Marques-Silva and Sakallah [84], Zhang [113], Moskewicz et al. [88], and Zhang et al. [115] showed that clause learning can be efficiently implemented and used to solve hard problems that cannot be approached by any other technique.

Despite its importance there has been little work on formal properties of clause

learning, with the goal of understanding its fundamental strengths and limitations. A likely reason for such inattention is that clause learning is a rather complex rule of inference – in fact, as we describe below, a complex family of rules of inference. A contribution of this work is a precise mathematical specification of various concepts used in describing clause learning.

Another problem in characterizing clause learning is defining a formal notion of the strength or power of a reasoning method. We address this issue by defining a new proof system called CL that captures the complexity of a clause learning algorithm on various classes of formulas. From the basic proof complexity point of view, only families of unsatisfiable formulas are of interest because only proofs of unsatisfiability can be large; minimum proofs of satisfiability are linear in the number of variables of the formula. In practice, however, many interesting formulas are satisfiable. To justify our approach of using a proof system CL, we refer to the work of Achlioptas, Beame, and Molloy [1] who have shown how negative proof complexity results for unsatisfiable formulas can be used to derive time lower bounds for specific inference algorithms, especially DPLL, running on satisfiable formulas as well. The key observation in their work is that before hitting a satisfying assignment, an algorithm is very likely to explore a large unsatisfiable part of the search space that corresponds to the first bad variable assignment.

Proof complexity does not capture everything we intuitively mean by the power of a reasoning system because it says nothing about how difficult it is to *find* shortest proofs. However, it is a good notion with which to begin our analysis because the size of proofs provides a lower bound on the running time of any implementation of the system. In the systems we consider, a branching function, which determines which variable to split upon or which pair of clauses to resolve, guides the search. A negative proof complexity result for a system tells us that a family of formulas is intractable even with a perfect branching function; likewise, a positive result gives us hope of finding a good branching function.

Recall from Chapter 2 that general resolution or RES is exponentially stronger than the DPLL procedure, the latter being exactly as powerful as tree-like resolution. Although RES can yield shorter proofs, in practice DPLL is better because it provides a more efficient way to search for proofs. The weakness of the tree-like proofs that DPLL generates is that they do not reuse derived clauses. The conflict clauses found when DPLL is augmented by clause learning correspond to reuse of derived clauses in the associated resolution proofs and thus to more general forms of resolution proofs. As a theoretical upper bound, all DPLL based approaches, including those involving clause learning, are captured by RES. An intuition behind the results we present is that the addition of clause learning moves DPLL closer to RES while retaining its practical efficiency.

It has been previously observed by Lynce and Marques-Silva [82] that clause learning can be viewed as adding resolvents to a tree-like resolution proof. However, we

provide the first mathematical proof that clause learning, viewed as a propositional proof system CL, is exponentially stronger than tree-like resolution. This explains, formally, the performance gains observed empirically when clause learning is added to DPLL based solvers. Further, we describe a generic way of extending families of formulas to obtain ones that exponentially separate CL from many refinements of resolution (see Section 2.2.2) known to be intermediate in strength between RES and tree-like resolution. These include regular and ordered resolution, and any other proper refinement of RES that behaves naturally under restrictions of variables, *i.e.*, for any formula $F$ and restriction $\rho$ on its variables, the shortest proof of $F|_\rho$ in the system is not any larger than a proof of $F$ itself.

The argument used in our result above involves a new clause learning scheme called FirstNewCut that we introduce specifically for this purpose. Our second technical result shows that combining a slight variant of CL, denoted CL--, with unlimited restarts results in a proof system as strong as RES itself. This intuitively explains the speed-ups obtained empirically when randomized restarts are added to DPLL based solvers, with or without clause learning.

**Remark 4.1.** MacKenzie [83] has recently used arguments similar to those of Beame et al. [15] to prove that a variant of clause learning can simulate all of regular resolution.

### *4.1 Natural Proper Refinements of a Proof System*

We discussed various refinements of resolution in Section 2.2.2. The concept of refinement applies to proof systems in general. We formalize below what it means for a refinement of a proof system to be natural and proper. Recall that the complexity $\mathcal{C}_S(F)$ of a formula $F$ under a proof system $S$ is the length of the shortest refutation of $F$ in $S$.

**Definition 4.1.** For proof systems $S$ and $T$, and a function $f : \mathbb{N} \to [1, \infty)$,

- $S$ is *natural* if for any formula $F$ and restriction $\rho$ on its variables, $\mathcal{C}_S(F|_\rho) \leq \mathcal{C}_S(F)$.
- $S$ is a *refinement* of $T$ if proofs in $S$ are also (restricted) proofs in $T$.
- A refinement $S$ of $T$ is $f(n)$*-proper* if there exists a witnessing family $\{F_n\}$ of formulas such that $\mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_T(F_n)$. The refinement is *exponentially-proper* if $f(n) = 2^{n^{\Omega(1)}}$ and *super-polynomially-proper* if $f(n) = n^{\omega(1)}$.

**Proposition 4.1.** *Tree-like, regular, linear, positive, negative, semantic, and ordered resolution are natural refinements of* RES.

The following proposition follows from the separation results of Bonet et al. [27] and Alekhnovich et al. [3].

**Proposition 4.2 ([27, 3]).** *Tree-like, regular, and ordered resolution are exponentially-proper natural refinements of* RES.

## *4.2 A Formal Framework for Studying Clause Learning*

Although many SAT solvers based on clause learning have been proposed and demonstrated to be empirically successful, a theoretical discussion of the underlying concepts and structures needed for our analysis is lacking. This section focuses on this formal framework.

For concreteness, we will use Algorithm 2.2 on page 17 as the basic clause learning algorithm. We state it again below for ease of reference. Recall that `DecideNextBranch` implements the variable selection process, `Deduce` apply unit propagation, `AnalyzeConflict` does clause learning upon reaching a conflict, and `Backtrack` unassigns variables up to the appropriate decision level computed during conflict analysis.

```
Input  : A CNF formula
Output : UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ← Deduce
            if status = CONFLICT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backtrack(blevel)
            else if status = SAT then
                Output current assignment stack
                return SAT
            else break
end
```

**Algorithm 4.1**: `DPLL-ClauseLearning`

### *4.2.1 Decision Levels and Implications*

Although we have already defined concepts such as decision level and implied variable in the context of the DPLL procedure, we did so with the simpler-to-understand re-

cursive version of the algorithm in mind. We re-define these concepts for the iterative version with clause learning given above.

Variables assigned values through the actual branching process are called *decision* variables and those assigned values as a result of unit propagation are called *implied* variables. *Decision* and *implied literals* are analogously defined. Upon backtracking, the last decision variable no longer remains a decision variable and might instead become an implied variable depending on the clauses learned so far. The *decision level of a decision variable* $x$ is one more than the number of current decision variables at the time of branching on $x$. The *decision level of an implied variable* is the maximum of the decision levels of decision variables used to imply it. The *decision level* at any step of the underlying DPLL procedure is the maximum of the decision levels of all current decision variables. Thus, for instance, if the clause learning algorithm starts off by branching on $x$, the decision level of $x$ is 1 and the algorithm at this stage is at decision level 1.

A clause learning algorithm stops and declares the given formula to be UNSAT whenever unit propagation leads to a conflict at decision level zero, i.e., when no variable is currently branched upon. This condition will be referred to as a *conflict at decision level zero*.

### *4.2.2 Branching Sequence*

We use the notion of branching sequence to prove an exponential separation between DPLL and clause learning. It generalizes the idea of a static *variable order* by letting the order differ from branch to branch in the underlying DPLL procedure. In addition, it also specifies which branch (TRUE or FALSE) to explore first. This can clearly be useful for satisfiable formulas, and can also help on unsatisfiable ones by making the algorithm learn useful clauses earlier in the process.

**Definition 4.2.** A *branching sequence* for a CNF formula $F$ is a sequence $\sigma = (l_1, l_2, \ldots, l_k)$ of literals of $F$, possibly with repetitions. A DPLL based algorithm $\mathcal{A}$ on $F$ *branches according to* $\sigma$ if it always selects the next variable $v$ to branch on in the literal order given by $\sigma$, skips $v$ if $v$ is currently assigned a value, and otherwise branches further by setting the chosen literal to FALSE and deleting it from $\sigma$. When $\sigma$ becomes empty, $\mathcal{A}$ reverts back to its default branching scheme.

**Definition 4.3.** A branching sequence $\sigma$ is *complete* for a formula $F$ under a DPLL based algorithm $\mathcal{A}$ if $\mathcal{A}$ branching according to $\sigma$ terminates before or as soon as $\sigma$ becomes empty. Otherwise it is *incomplete* or *approximate*.

Clearly, how well a branching sequence works for a formula depends on the specifics of the clause learning algorithm used, such as its learning scheme and backtracking process. One needs to keep these in mind when generating the sequence. It is also important to note that while the size of a variable order is always the same as the

number of variables in the formula, that of an effective branching sequence is typically much more. In fact, the size of a branching sequence complete for an unsatisfiable formula $F$ is equal to the size of an unsatisfiability proof of $F$, and when $F$ is satisfiable, it is proportional to the time needed to find a satisfying assignment.

### *4.2.3 Implication Graph and Conflicts*

Unit propagation can be naturally associated with an *implication graph* that captures all possible ways of deriving all implied literals from decision literals.

**Definition 4.4.** The *implication graph* $G$ at a given stage of DPLL is a directed acyclic graph with edges labeled with sets of clauses. It is constructed as follows:

Step 1: Create a node for each decision literal, labeled with that literal. These will be the indegree zero source nodes of $G$.

Step 2: While there exists a known clause $C = (l_1 \vee \ldots l_k \vee l)$ such that $\neg l_1, \ldots, \neg l_k$ label nodes in $G$,

i. Add a node labeled $l$ if not already present in $G$.

ii. Add edges $(l_i, l), 1 \leq i \leq k$, if not already present.

iii. Add $C$ to the label set of these edges. These edges are thought of as grouped together and associated with clause $C$.

Step 3: Add to $G$ a special "conflict" node $\bar{\Lambda}$. For any variable $x$ that occurs both positively and negatively in $G$, add directed edges from $x$ and $\neg x$ to $\bar{\Lambda}$.

Since all node labels in $G$ are distinct, we identify nodes with the literals labeling them. Any variable $x$ occurring both positively and negatively in $G$ is a *conflict variable*, and $x$ as well as $\neg x$ are *conflict literals*. $G$ contains a *conflict* if it has at least one conflict variable. DPLL at a given stage has a *conflict* if the implication graph at that stage contains a conflict. A conflict can equivalently be thought of as occurring when the residual formula contains the empty clause $\Lambda$.

By definition, an implication graph may not contain a conflict at all, or it may contain many conflict variables and several ways of deriving any single literal. To better understand and analyze a conflict when it occurs, we work with a subgraph of an implication graph, called the *conflict graph* (see Figure 4.1), that captures only one among possibly many ways of reaching a conflict from the decision variables using unit propagation.

**Definition 4.5.** A *conflict graph* $H$ is any subgraph of an implication graph with the following properties:

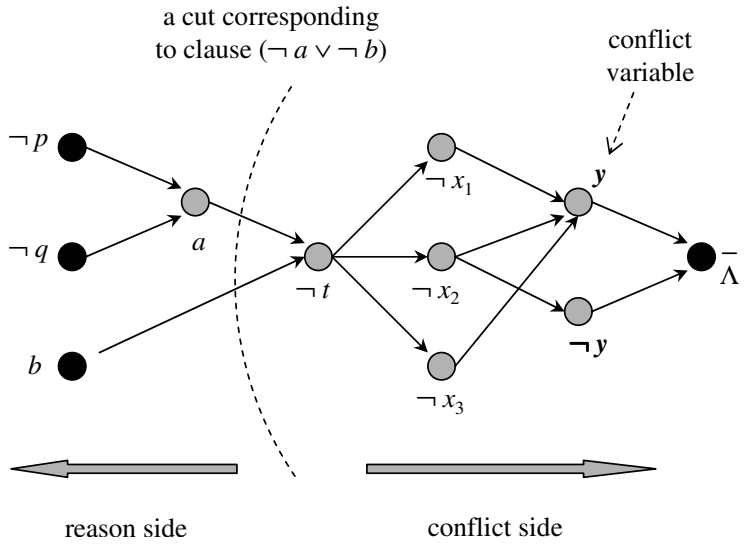


Figure 4.1: A conflict graph

(a) $H$ contains $\bar{\Lambda}$ and exactly one conflict variable.

(b) All nodes in $H$ have a path to $\bar{\Lambda}$.

(c) Every node $l$ in $H$ other than $\bar{\Lambda}$ either corresponds to a decision literal or has precisely the nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$ as predecessors where $(l_1 \vee l_2 \vee \ldots \vee l_k \vee l)$ is a known clause.

While an implication graph may or may not contain conflicts, a conflict graph always contains exactly one. The choice of the conflict graph is part of the strategy of the solver. A typical strategy will maintain one subgraph of an implication graph that has properties (b) and (c) from Definition 4.5, but not property (a). This can be thought of as a *unique inference* subgraph of the implication graph. When a conflict is reached, this unique inference subgraph is extended to satisfy property (a) as well, resulting in a conflict graph, which is then used to analyze the conflict.

*Conflict clauses*

Recall that for a subset $U$ of the vertices of a graph, the *edge-cut* (henceforth called a cut) corresponding to $U$ is the set of all edges going from vertices in $U$ to vertices not in $U$.

Consider the implication graph at a stage where there is a conflict and fix a conflict graph contained in that implication graph. Choose any cut in the conflict graph that has all decision variables on one side, called the *reason side*, and $\bar{\Lambda}$ as well as at least

one conflict literal on the other side, called the *conflict side*. All nodes on the reason side that have at least one edge going to the conflict side form a *cause* of the conflict. The negations of the corresponding literals forms the *conflict clause* associated with this cut.

### *4.2.4 Trivial Resolution and Learned Clauses*

**Definition 4.6.** A resolution derivation $(C_1, C_2, \ldots, C_k)$ is *trivial* iff all variables resolved upon are distinct and each $C_i, i \geq 3$, is either an initial clause or is derived by resolving $C_{i-1}$ with an initial clause.

A trivial derivation is tree-like, regular, linear, as well as ordered. As the following propositions show, trivial derivations correspond to conflicts in clause learning algorithms.

**Proposition 4.3.** *Let $F$ be a CNF formula. If there is a trivial resolution derivation of a clause $C \notin F$ from $F$ then setting all literals of $C$ to* FALSE *leads to a conflict by unit propagation.*

*Proof.* Let $\pi = (C_1, C_2, \ldots, C_k = C)$ be a trivial resolution derivation of $C$ from $F$. Let $C_k = (l_1 \vee l_2 \vee \ldots \vee l_q)$ and $\rho$ be the partial assignment that sets all $l_i, 1 \leq i \leq q$, to FALSE. Assume without loss of generality that clauses in $\pi$ are ordered so that all initial clauses precede any derived clause. We give a proof by induction on the number of derived clauses in $\pi$.

For the base case, $\pi$ has only one derived clause, $C = C_k$. Assume without loss of generality that $C_k = (A \vee B)$ and $C_k$ is derived by resolving two initial clauses $(A \vee x)$ and $(B \vee \neg x)$ on variable $x$. Since $\rho$ falsifies $C_k$, it falsifies all literals of $A$, implying $x =$ TRUE by unit propagation. Similarly, $\rho$ falsifies $B$, implying $x =$ FALSE and resulting in a conflict.

When $\pi$ has at least two derived clauses, $C_k$, by triviality of $\pi$, must be derived by resolving $C_{k-1} \notin F$ with a clause in $F$. Assume without loss of generality that $C_{k-1} = (A \vee x)$ and the clause from $F$ used in this resolution step is $(B \vee \neg x)$, where $C_k = (A \vee B)$. Since $\rho$ falsifies $C = C_k$, it falsifies all literals of $B$, implying $x =$ FALSE by unit propagation. This in turn results in falsifying all literals of $C_{k-1}$ because all literals of $A$ are also set to FALSE by $\rho$. Now $(C_1, \ldots, C_{k-1})$ is a trivial resolution derivation of $C_{k-1} \notin F$ from $F$ with one less derived clause than $\pi$, and all literals of $C_{k-1}$ are falsified. By induction, this must lead to a conflict by unit propagation. □

**Proposition 4.4.** *Any conflict clause can be derived from initial and previously derived clauses using a trivial resolution derivation.*

*Proof.* Let $\sigma$ be the cut in a fixed conflict graph associated with the given conflict clause. Let $V_{conflict}(\sigma)$ denote the set of variables on the conflict side of $\sigma$, but including the conflict variable only if it occurs both positively and negatively on the

conflict side. We will prove by induction on $|V_{conflict}(\sigma)|$ the stronger statement that the conflict clause associated with a cut $\sigma$ has a trivial derivation from known (i.e. initial or previously derived) clauses resolving precisely on the variables in $V_{conflict}(\sigma)$.

For the base case, $V_{conflict}(\sigma) = \phi$ and the conflict side contains only $\bar{\Lambda}$ and a conflict literal, say $x$. Informally, this cut corresponds to the immediate cause of the conflict, namely, the single unit propagation step that led to the derivation of $\neg x$. More concretely, the clause associated with this cut consists of the node $\neg x$ which has an edge to $\bar{\Lambda}$, and nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$, corresponding to a known clause $C_x = (l_1 \vee l_2 \vee \ldots \vee l_k \vee x)$, that each have an edge to $x$. The conflict clause for this cut is simply the known clause $C_x$ itself, having a length zero trivial derivation.

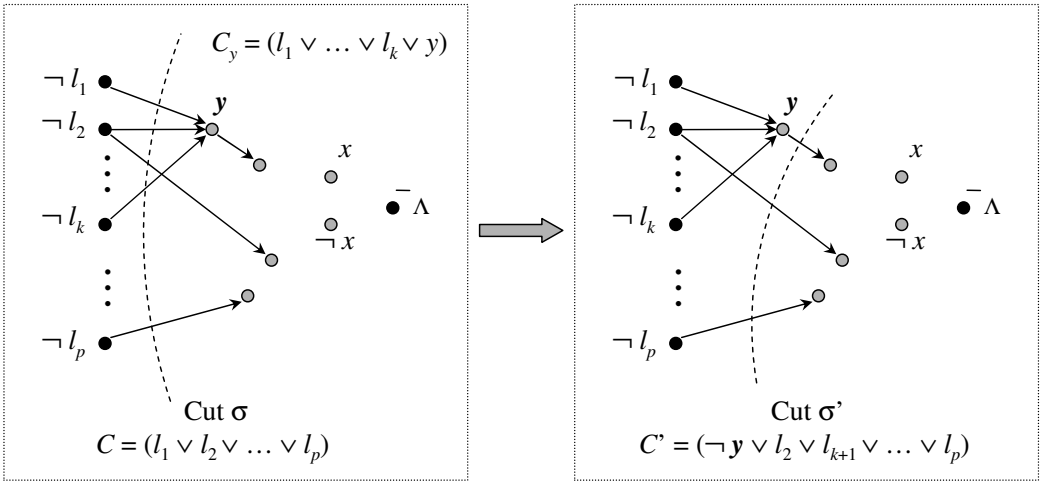


Figure 4.2: Deriving a conflict clause using trivial resolution. Resolving $C'$ with $C_y$ on variable $y$ gives the conflict clause $C$.

When $V_{conflict}(\sigma) \neq \phi$, choose a node $y$ on the conflict side all whose predecessors are on the reason side (see Figure. 4.2). Let the conflict clause be $C = (l_1 \vee l_2 \vee \ldots \vee l_p)$ and assume without loss of generality that the predecessors of $y$ are $\neg l_1, \neg l_2, \ldots, \neg l_k$ for some $k \leq p$. By definition of unit propagation, $C_y = (l_1 \vee l_2 \vee \ldots \vee l_k \vee y)$ must be a known clause. Obtain a new cut $\sigma'$ from $\sigma$ by moving node $y$ from the conflict side to the reason side. The new associated conflict clause must be of the form $C' = (\neg y \vee D)$, where $D$ is a subclause of $C$. Now $V_{conflict}(\sigma') \subset V_{conflict}(\sigma)$. Consequently, by induction, $C'$ must have a trivial resolution derivation from known clauses resolving precisely upon the variables in $V_{conflict}(\sigma')$. Recall that no variable occurs twice in a conflict graph except the conflict variable. Hence $V_{conflict}(\sigma')$ has precisely the variables of $V_{conflict}(\sigma)$ except $y$. Using this trivial derivation of $C'$ and finally resolving $C'$ with the known clause $C_y$ on variable $y$ gives us a trivial derivation of $C$ from known clauses. This completes the inductive step. □

### *4.2.5 Learning Schemes*

The essence of clause learning is captured by the *learning scheme* used to analyze and learn the "cause" of a failure. More concretely, different cuts in a conflict graph separating decision variables from a set of nodes containing $\bar{\Lambda}$ and a conflict literal correspond to different learning schemes (see Figure 4.3). One may also define learning schemes based on cuts not involving conflict literals at all such as a scheme suggested by Zhang et al. [115], but the effectiveness of such schemes is not clear. These will not be considered here.

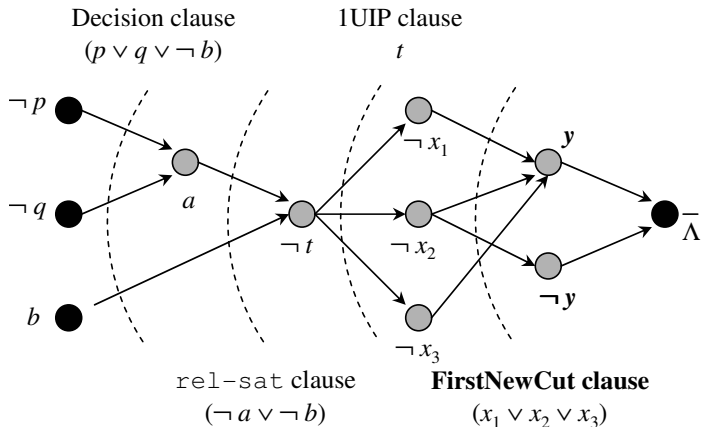


Figure 4.3: Various learning schemes

It is insightful to think of the *nondeterministic* scheme as the most general learning scheme. Here we select the cut nondeterministically, choosing, whenever possible, one whose associated clause is not already known. Since we can repeatedly branch on the same last variable, nondeterministic learning subsumes learning multiple clauses from a single conflict as long as the sets of nodes on the reason side of the corresponding cuts form a (set-wise) decreasing sequence. For simplicity, we will assume that only one clause is learned from any conflict.

In practice, however, we employ deterministic schemes. The *decision* scheme [115], for example, uses the cut whose reason side comprises all decision variables. rel-sat [13] uses the cut whose conflict side consists of all implied variables at the current decision level. This scheme allows the conflict clause to have exactly one variable from the current decision level, causing an automatic flip in its assignment upon backtracking.

This nice flipping property holds in general for all *unique implication points* (UIPs) [84]. A UIP of an implication graph is a node at the current decision level $d$ such that any path from the decision variable at level $d$ to the conflict variable as well as its negation must go through it. Intuitively, it is a *single* reason at level $d$ that causes the conflict. Whereas rel-sat uses the decision variable as the obvious UIP, Grasp [84]

and zChaff [88] use *FirstUIP*, the one that is "closest" to the conflict variable. Grasp also learns multiple clauses when faced with a conflict. This makes it typically require fewer branching steps but possibly slower because of the time lost in learning and unit propagation.

The concept of UIP can be generalized to decision levels other than the current one. The *1UIP scheme* corresponds to learning the FirstUIP clause of the current decision level, the *2UIP scheme* to learning the FirstUIP clauses of both the current level and the one before, and so on. Zhang et al. [115] present a comparison of all these and other learning schemes and conclude that 1UIP is quite robust and outperforms all other schemes they consider on most of the benchmarks.

*The FirstNewCut Scheme*

We propose a new learning scheme called *FirstNewCut* whose ease of analysis helps us demonstrate the power of clause learning. We would like to point out that we use this scheme here only to prove our theoretical bounds using specific formulas. Its effectiveness on other formulas has not been studied yet. We would also like to point out that the experimental results that we present are for the 1UIP learning scheme, but can also be extended to certain other schemes, including FirstNewCut.

The key idea behind FirstNewCut is to make the conflict clause as relevant to the current conflict as possible by choosing a cut close to the conflict literals. This is what the FirstUIP scheme also tries to achieve in a slightly different manner. For the following definitions, fix a cut in a conflict graph and let $S$ be the set of nodes on the reason side that have an edge to some node on the conflict side. $S$ is the reason side *frontier* of the cut. Let $C_S$ be the conflict clause associated with this cut.

**Definition 4.7.** *Minimization* of conflict clause $C_S$ is the following process: while there exists a node $v \in S$ all of whose predecessors are also in $S$, move $v$ to the conflict side, remove it from $S$, and repeat.

**Definition 4.8.** *FirstNewCut scheme*: Start with a cut whose conflict side consists of $\bar{\Lambda}$ and a conflict literal. If necessary, repeat the following until the associated conflict clause, after minimization, is not already known: choose a node on the conflict side, and move all its predecessors that lie on the reason side, other than those that correspond to decision variables, to the conflict side. Finally, learn the resulting new minimized conflict clause.

This scheme starts with the cut that is closest to the conflict literals and iteratively moves it back toward the decision variables until a new associated conflict clause is found. This backward search always halts because the cut with all decision variables on the reason side is certainly a new cut. Note that there are potentially several ways of choosing a literal to move the cut back, leading to different conflict clauses. The FirstNewCut scheme, by definition, always learns a clause not already known. This motivates the following:

**Definition 4.9.** A clause learning scheme is *non-redundant* if on a conflict, it always learns a clause not already known.

### *4.2.6 Clause Learning Proofs*

The notion of clause learning proofs connects clause learning with resolution and provides the basis for our complexity bounds. If a given CNF formula $F$ is unsatisfiable, clause learning terminates with a conflict at decision level zero. Since all clauses used in this final conflict themselves follow directly or indirectly from $F$, this failure of clause learning in finding a satisfying assignment constitutes a logical proof of unsatisfiability of $F$. We denote by CL the proof system consisting of all such proofs. Our bounds compare the sizes of proofs in CL with the sizes of (possibly restricted) resolution proofs. Recall that clause learning algorithms can use one of many learning schemes, resulting in different proofs.

**Definition 4.10.** A *clause learning (CL) proof* $\pi$ of an unsatisfiable CNF formula $F$ under learning scheme $\mathcal{S}$ and induced by branching sequence $\sigma$ is the result of applying DPLL with unit propagation on $F$, branching according to $\sigma$, and using scheme $\mathcal{S}$ to learn conflict clauses such that at the end of this process, there is a conflict at decision level zero. The *size* of the proof, $size(\pi)$, is $|\sigma|$.

### *4.2.7 Fast Backtracking and Restarts*

Most clause learning algorithms use *fast backtracking* or *conflict-directed backjumping* introduced by Stallman and Sussman [103], where one uses the conflict graph to undo not only the last branching decision but also all other recent decisions that did not contribute to the current conflict. In particular, the SAT solver zChaff that we will use for our experiments in Chapters 5 and 6 backtracks to decision level zero when it learns a unit clause. This property influences the structure of a branching sequence generation algorithm we will present in Section 5.2.1.

More precisely, the level that a clause learning algorithm employing this technique backtracks to is one less than the maximum of the decision levels of all decision variables (i.e. the *sources* of the conflict) present in the underlying conflict graph. Note that the current conflict might use clauses learned earlier as a result of branching on the apparently redundant variables. This implies that fast backtracking in general cannot be replaced by a "good" branching sequence that does not produce redundant branches. For the same reason, fast backtracking cannot either be replaced by simply learning the decision scheme clause. However, the results we present here are independent of whether or not fast backtracking is used.

*Restarts*, introduced by Gomes et al. [58] and further developed by Baptista and Marques-Silva [12], allow clause learning algorithms to arbitrarily restart their branching process from decision level zero. All clauses learned so far are retained and now

treated as additional initial clauses. As we will show, unlimited restarts, performed at the correct step, can make clause learning very powerful. In practice, this requires extending the strategy employed by the solver to include when and how often to restart. Unless otherwise stated, however, clause learning proofs in the rest of this chapter will be assumed to allow no restarts.

## *4.3 Clause Learning and Proper Natural Refinements of* RES

We prove that the proof system CL, even without restarts, is stronger than *all* proper natural refinements of RES. We do this by first introducing a way of extending any CNF formula based on a given RES proof of it. We then show that if a formula $F$ $f(n)$-separates RES from a natural refinement $S$, its extension $f(n)$-separates CL from $S$. The existence of such an $F$ is guaranteed for all $f(n)$-proper natural refinements by definition.

### *4.3.1 The Proof Trace Extension*

**Definition 4.11.** Let $F$ be a CNF formula and $\pi$ be a RES refutation of it. Let the last step of $\pi$ resolve $v$ with $\neg v$. Let $S = \pi \setminus (F \cup \{\neg v, \Lambda\})$. The *proof trace extension* $PT(F, \pi)$ of $F$ is a CNF formula over variables of $F$ and new trace variables $t_C$ for clauses $C \in S$. The clauses of $PT(F, \pi)$ are all initial clauses of $F$ together with a trace clause $(\neg x \vee t_C)$ for each clause $C \in S$ and each literal $x \in C$.

We first show that if a formula has a short RES refutation, then the corresponding proof trace extension has a short CL proof. Intuitively, the new trace variables allow us to simulate every resolution step of the original proof individually, without worrying about extra branches left over after learning a derived clause.

**Lemma 4.1.** *Suppose a formula $F$ has a* RES *refutation $\pi$. Let $F' = PT(F, \pi)$. Then $\mathcal{C}_{\mathsf{CL}}(F') < size(\pi)$ when* CL *uses the FirstNewCut scheme and no restarts.*

*Proof.* Suppose $\pi$ contains a derived clause $C_i$ whose strict subclause $C_i'$ can be derived by resolving two previously occurring clauses. We can replace $C_i$ with $C_i'$, do trivial simplifications on further derivations that used $C_i$ and obtain a simpler proof $\pi'$ of $F$. Doing this repeatedly will remove all such redundant clauses and leave us with a simplified proof no larger in size. Hence we will assume without loss of generality that $\pi$ has no such clause.

Viewing $\pi$ as a sequence of clauses, its last two elements must be a literal, say $v$, and $\Lambda$. Let $S = \pi \setminus (F \cup \{v, \Lambda\})$. Let $(C_1, C_2, \ldots, C_k)$ be the subsequence of $\pi$ that has precisely the clauses in $S$. Note that $C_i = \neg v$ for some $i, 1 \leq i \leq k$. We claim that the branching sequence $\sigma = (t_{C_1}, t_{C_2}, \ldots, t_{C_k})$ induces a CL proof of $F$ of size $k$ using the FirstNewCut scheme. To prove this, we show by induction that after $i$ branching steps, the clause learning procedure branching according to $\sigma$ has

learned clauses $C_1, C_2, \ldots, C_i$, has trace variables $t_{C_1}, t_{C_2}, \ldots, t_{C_i}$ set to TRUE, and is at decision level $i$.

The base case for induction, $i = 0$, is trivial. The clause learning procedure is at decision level zero and no clauses have been learned. Suppose the inductive claim holds after branching step $i-1$. Let $C_i = (x_1 \vee x_2 \vee \ldots \vee x_l)$. $C_i$ must have been derived in $\pi$ by resolving two clauses $(A \vee y)$ and $(B \vee \neg y)$ coming from $F \cup \{C_1, C_2, \ldots, C_{i-1}\}$, where $C_i = (A \vee B)$. The $i^{th}$ branching step sets $t_{C_i} =$ FALSE. Unit propagation using trace clauses $(\neg x_j \vee t_{C_i})$, $1 \leq j \leq l$, sets each $x_j$ to FALSE, thereby falsifying all literals of $A$ and $B$. Further unit propagation using $(A \vee y)$ and $(B \vee \neg y)$ implies $y$ as well as $\neg y$, leading to a conflict. The cut in the conflict graph containing $y$ and $\neg y$ on the conflict side and everything else on the reason side yields $C_i$ as the FirstNewCut clause, which is learned from this conflict. The process now backtracks and flips the branch on $t_{C_i}$ by setting it to TRUE. At this stage, the clause learning procedure has learned clauses $C_1, C_2, \ldots, C_i$, has trace variables $t_{C_1}, t_{C_2}, \ldots, t_{C_i}$ set to TRUE, and is at decision level $i$. This completes the inductive step.

The inductive proof above shows that when the clause learning procedure has finished branching on all $k$ literals in $\sigma$, it will have learned all clauses in $S$. Adding to this the initial clauses $F$ that are already known, the procedure will have as known clauses $\neg v$ as well as the two unit or binary clauses used to derive $v$ in $\pi$. These immediately generate $\Lambda$ in the residual formula by unit propagation using variable $v$, leading to a conflict at decision level $k$. Since this conflict does not use any decision variable, fast backtracking retracts all $k$ branches. The conflict, however, still exists at decision level zero, thereby concluding the clause learning procedure and finishing the CL proof. □

**Lemma 4.2.** *Let $S$ be an $f(n)$-proper natural refinement of* RES *whose weakness is witnessed by a family $\{F_n\}$ of formulas. Let $\{\pi_n\}$ be the family of shortest* RES *proofs of $\{F_n\}$. Let $\{F'_n\} = \{PT(F_n, \pi_n)\}$. For* CL *using the FirstNewCut scheme and no restarts, $\mathcal{C}_S(F'_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F'_n)$.*

*Proof.* Let $\rho_n$ the restriction that sets every trace variable of $F'_n$ to TRUE. We claim that $\mathcal{C}_S(F'_n) \geq \mathcal{C}_S(F'_n|_{\rho_n}) = \mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{RES}}(F_n) > f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F'_n)$. The first inequality holds because $S$ is a natural proof system. The following equality holds because $\rho_n$ keeps the original clauses of $F_n$ intact and trivially satisfies all trace clauses, thereby reducing the initial clauses of $F'_n$ to precisely $F_n$. The next inequality holds because $S$ is an $f(n)$-proper refinement of RES. The final inequality follows from Lemma 4.1. □

This gives our first main result and its corollaries using Proposition 4.2:

**Theorem 4.1.** *For any $f(n)$-proper natural refinement $S$ of* RES *and for* CL *using the FirstNewCut scheme and no restarts, there exist formulas $\{F_n\}$ such that $\mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F_n)$.*

**Corollary 4.1.** CL *can provide exponentially shorter proofs than tree-like, regular, and ordered resolution.*

**Corollary 4.2.** *Either* CL *is not a natural proof system or it is equivalent in strength to* RES*.*

*Proof.* As clause learning yields resolution proofs of unsatisfiable formulas, CL is a refinement of RES. Assume without loss of generality that it is an $f(n)$-proper refinement for some function $f$; this is true for instance when $f(n) = 1$ for all $n$. If CL is a natural proof system, Theorem 4.1 implies that there exists a family $\{F_n\}$ of formulas such that $\mathcal{C}_{\mathsf{CL}}(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F_n)$. Since $f : \mathbb{N} \to [1, \infty)$ by the definition of $f(n)$-proper, $f(n)$ must be 1 for all $n$, proving the result. □

### *4.4 Clause Learning and General Resolution*

We begin this section by showing that CL proofs, irrespective of the learning scheme, branching strategy, or restarts used, can be efficiently simulated by RES. In the reverse direction, we show that CL, with a slight variation and with unlimited restarts, can efficiently simulate RES in its full generality. The variant relates to the variables one is allowed to branch upon.

**Lemma 4.3.** *For any formula $F$ over $n$ variables and* CL *using any learning scheme and any number of restarts, $\mathcal{C}_{\mathsf{RES}}(F) \leq n \cdot \mathcal{C}_{\mathsf{CL}}(F)$.*

*Proof.* Given a CL proof $\pi$ of $F$, a RES proof can be constructed by sequentially deriving all clauses that $\pi$ learns, which includes the empty clause $\Lambda$. From Proposition 4.4, all these derivations are trivial and hence require at most $n$ steps each. Consequently, the size of the resulting RES proof is at most $n \cdot size(\pi)$. Note that since we derive clauses of $\pi$ individually, restarts in $\pi$ do not affect the construction. □

**Definition 4.12.** Let CL-- denote the variant of CL where one is allowed to branch on a literal whose value is already set explicitly or because of unit propagation.

Of course, such a relaxation is useless in ordinary DPLL; there is no benefit in branching on a variable that doesn't even appear in the residual formula. However, with clause learning, such a branch can lead to an immediate conflict and allow one to learn a key conflict clause that would otherwise have not been learned. We will use this property to show that RES can be efficiently simulated by CL-- with enough restarts.

We first state a generalization of Lemma 4.3. CL-- can, by definition, do all that usual CL can, and is potentially stronger. The simulation of CL by RES can in fact be extended to CL-- as well. The proof goes exactly as the proof of Lemma 4.3 and uses the easy fact that Proposition 4.4 doesn't change even when one is allowed to branch on variables that are already set. This gives us:

**Proposition 4.5.** *For any formula $F$ over $n$ variables and* CL-- *using any learning scheme and any number of restarts, $\mathcal{C}_{\mathsf{RES}}(F) \leq n \cdot \mathcal{C}_{\mathsf{CL}\text{--}}(F)$.*

**Lemma 4.4.** *For any formula $F$ over $n$ variables and* CL *using any non-redundant scheme and at most $\mathcal{C}_{\mathsf{RES}}(F)$ restarts, $\mathcal{C}_{\mathsf{CL}\text{--}}(F) \leq n \cdot \mathcal{C}_{\mathsf{RES}}(F)$.*

*Proof.* Let $\pi$ be a RES proof of $F$ of size $s$. Assume without loss of generality as in the proof of Lemma 4.1 that $\pi$ does not contain a derived clause $C_i$ whose strict subclause $C_i'$ can be derived by resolving two clauses occurring previously in $\pi$. The proof of this Lemma is very similar to that of Lemma 4.1. However, since we do not have trace variables to allow us to simulate each resolution step individually and independently, we use explicit restarts.

Viewing $\pi$ as a sequence of clauses, its last two elements must be a literal, say $v$, and $\Lambda$. Let $S = \pi \setminus (F \cup \{v, \Lambda\})$. Let $(C_1, C_2, \ldots, C_k)$ be the subsequence of $\pi$ that has precisely the clauses in $S$. Note that $C_i = \neg v$ for some $i, 1 \leq i \leq k$. For convenience, define an *extended branching sequence* to be a branching sequence in which certain places, instead of being literals, can be marked as restart points. Let $\sigma$ be the extended branching sequence consisting of all literals of $C_1$, followed by a restart point, followed by all literals of $C_2$, followed by a second restart point, and so on up to $C_k$. We claim that $\sigma$ induces a CL-- proof of $F$ using any non-redundant learning scheme. To prove this, we show by induction that after the $i^{th}$ restart point in $\sigma$, the CL-- procedure has learned clauses $C_1, C_2, \ldots, C_i$ and is at decision level zero.

The base case for induction, $i = 0$, is trivial. No clauses have been learned and the clause learning procedure is at decision level zero. Suppose the inductive claim holds after the $(i-1)^{st}$ restart point in $\sigma$. Let $C_i = (x_1 \vee x_2 \vee \ldots \vee x_l)$. $C_i$ must have been derived in $\pi$ by resolving two clauses $(A \vee y)$ and $(B \vee \neg y)$ coming from $F \cup \{C_1, C_2, \ldots, C_{i-1}\}$, where $C_i = (A \vee B)$. Continuing to branch according to $\sigma$ till before the $i^{th}$ restart point makes the CL-- procedure set all if $x_1, x_2, \ldots, x_l$ to FALSE. Note that when all literals appearing in $A$ and $B$ are distinct, the last branch on $x_l$ here is on a variable that is already set because of unit propagation. CL--, however, allows this. At this stage, unit propagation using $(A \vee y)$ and $(B \vee \neg y)$ implies $y$ as well as $\neg y$, leading to a conflict. The conflict graph consists of $\neg x_j$'s, $1 \leq j \leq l$, as the decision literals, $y$ and $\neg y$ as implied literals, and $\bar{\Lambda}$. The only new conflict clause that can learned from this very simple conflict graph is $C_i$. Thus, $C_i$ is learned using any non-redundant learning scheme and the $i^{th}$ restart executed, as dictated by $\sigma$. At this stage, the CL-- procedure has learned clauses $C_1, C_2, \ldots, C_i$, and is at decision level zero. This completes the inductive step.

The inductive proof above shows that when the CL-- procedure has finished with the $k^{th}$ restart in $\sigma$, it will have learned all clauses in $S$. Adding to this the initial clauses $F$ that are already known, the procedure will have as known clauses $\neg v$ as well as the two unit or binary clauses used to derive $v$ in $\pi$. These immediately generate $\Lambda$

in the residual formula by unit propagation using variable $v$, leading to a conflict at decision level zero, thereby concluding the clause learning procedure and finishing the CL-- proof. The bounds on the size of this proof and the number of restarts needed immediately follow from the definition of $\sigma$. □

Combining Lemma 4.4 with Proposition 4.5, we get

**Theorem 4.2.** CL-- *with any non-redundant scheme and unlimited restarts is polynomially equivalent to* RES.

**Remark 4.2.** Baptista and Marques-Silva [12] showed that by choosing the restart points in a smart way, CL together with restarts can be converted into a *complete* algorithm for satisfiability testing, i.e., for all unsatisfiable formulas given as input, it will halt and provide a proof of unsatisfiability. Our theorem makes a much stronger claim about a slight variant of CL, namely, with enough restarts, this variant can always find proofs of unsatisfiability that are as short as those of RES.

### *4.5 Discussion*

In this chapter, we developed a mathematical framework for studying the most widely used class of complete SAT solvers, namely the one based on DPLL and clause learning. We studied clause learning from a proof complexity perspective and obtained two significant results for the proof system CL summarized in Figure 4.4. The first of these is that CL can provide exponentially smaller proofs than any proper natural refinement of RES. We derived from this as a corollary that CL is either not natural or is as powerful as RES itself. This is an interesting and somewhat surprising statement. The second noteworthy result is that a variant of clause learning with unrestricted restarts has exactly the same strength as RES.

Our argument used the notion of a proof trace extension of a formula which allowed one to convert a formula that is easy for RES to an extended formula that is easy for CL, at the same time retaining the hardness with respect to any natural refinement of RES. We also defined and made use of a new learning scheme, FirstNewCut.

Understanding where clause learning stands in relation to well studied proof systems should lead to better insights on why it works well on certain domains and fails on others. For instance, we will see in Chapter 5 an example of a domain (pebbling problems) where our results say that learning is necessary and sufficient, given a good branching order, to obtain sub-exponential solutions using clause learning based methods.

On the other hand, the connection with resolution also implies that any problem that contains as a sub-problem a formula that is inherently hard even for RES, such as the pigeonhole principle to be described in detail in Chapter 6, must be hard for any variant of clause learning. For such domains, theoretical results suggest practical extensions such as symmetry breaking and counting techniques for obtaining efficient

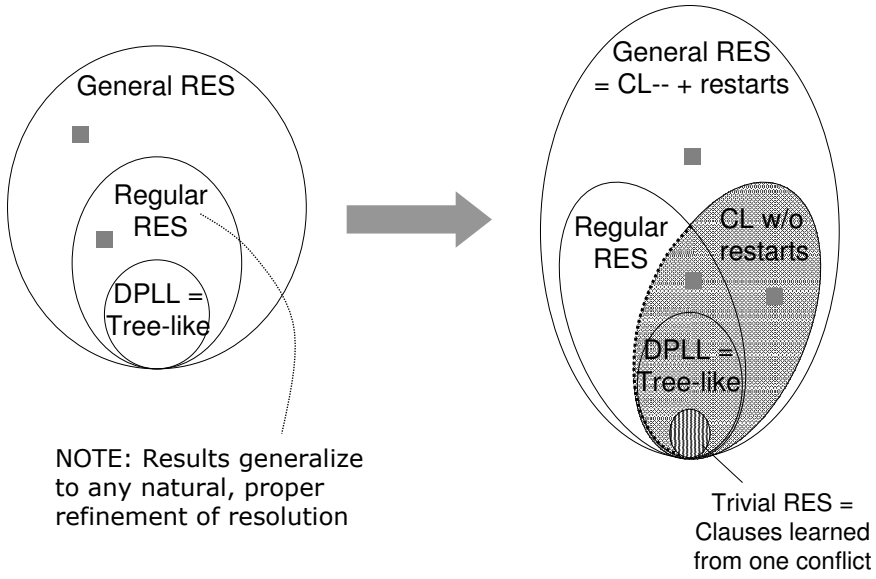


Figure 4.4: Results: Clause learning in relation to resolution

solutions. The first of these serves as a motivation for the work we will present in Chapter 6.