INSTITUT FÜR INFORMATIK

der Ludwig-Maximilians-Universität München

Diploma Thesis

Resolution Proofs and DLL Algorithms with Clause Learning

Jan Hoffmann

Aufgabensteller:	Prof. Dr. Martin Hofmann
Betreuer:	Prof. Samuel R. Buss, Ph.D
	Dr. Jan Johannsen
Abgabetermin:	27. September 2007

Abstract

This thesis analyzes the connections between resolution proofs and satisfiability search procedures. It is well known that DLL search algorithms that do not use learning are equivalent to tree-like resolution in terms of proof complexity. To generalize this result to DLL algorithms that use learning, two natural generalizations of regular resolution that are based on resolution trees with lemmas (RTL) are introduced. It is shown that dag-like resolution is equivalent to these resolution refinements when there is no regularity condition. On the other hand an exponential separation between the regular versions (regular weak resolution trees with input lemmas and regular weak resolution trees with lemmas) and regular dag-like resolution is given.

It is proved that executions of DLL algorithms that use learning based on the conflict graph and unit propagation, like most of the current state of the art SAT-solvers, can be simulated by regular WRTL. Inspired by this simulation, a new generalization of learning in DLL algorithms, which is polynomially equivalent to regular WRTL, is presented. This algorithm can simulate general resolution without doing restarts.

Contents

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification.

München, September 25, 2007   Jan Hoffmann

Preface

Contents

The satisfiability problem of the propositional logic (SAT) is one of the most studied algorithmic problems in computer science. It has been the first problem proved to be NP-complete and up to now a polynomial time algorithm for SAT could not be found.

There exist SAT-solvers that can decide SAT on present-day computers for many formulas that are relevant in practice. Nearly all of the fastest deterministic SAT-solvers are based on a proof search algorithm schema that is known as DLL algorithm. The schema is called a proof search procedure because an execution of a DLL algorithm on an unsatisfiable CNF formula corresponds to a resolution refutation of that formula.

This thesis analyzes the properties of resolution refutations that correspond to different variants of the DLL algorithm with a view to understand and improve state-of-the-art SAT-solvers.

In *Chapter 1* propositional logic, the satisfiability problem and proof systems are introduced.

Chapter 2 presents three versions of the DLL algorithm schema. In Section 2.1 the classical DLL algorithm is defined through the algorithm schema DLL. Section 2.2 gives a recursive definition of the variation of the DLL algorithm that is used in modern SAT-solvers. This algorithm schema is called DLL-L-UP since its most important features are *learning* and *unit propagation*. In Section 2.3 the algorithm schema DLL-Learn is introduced which is a new natural generalization of both, DLL and DLL-L-UP.

Chapter 3 describes and analyzes several types of tree-like resolution proofs with respect to their relations to DLL algorithms. Section 3.2 proves the well-known fact that every execution of DLL with an unsatisfiable input formula can be seen as a *resolution tree* with a node for every recursive call of the execution and that every resolution tree can be used to define an execution of DLL that performs at most one recursive call for every node in the tree. In Section 3.4 an analogous statement is shown for the algorithm schema DLL-Learn and *regu-*

lar weak resolution trees with lemmas (regular weak RTL) that are introduced in Section 3.3. Finally, it is proved in Section 3.5 that an execution of DLL-L-UP can be transformed into a *regular weak resolution tree with input lemmas (regular weak RTI)* which size is quadratic in the number of the recursive calls of the execution. Therefrom it follows that DLL-L-UP can be polynomially simulated by DLL-LEARN.

In *Chapter 4* the introduced tree-like resolution refinements are compared to (regular) dag-like resolution proofs and known lower bounds and separations are applied to the DLL algorithms. The main results are the polynomial equivalence of dag-like resolution, RTL, RTI, weak RTL and weak RTI (Section 4.1); an exponential lower bound on the running times of all algorithms that are based on DLL, DLL-L-UP or DLL-LEARN; and an exponential separation of DLL and DLL-L-UP (Section 4.2). Furthermore it is shown in Section 4.1 that regular dag-like resolution proofs can be simulated by regular RTI and in Section 4.3 an exponential separation of regular RTI from regular dag-like resolution is given. Another major result that is presented in Section 4.3 is the simulation of general resolution by the algorithm DLL-LEARN.

Chapter 5 contains some notes on the representation of resolution proofs. In Section 5.1 sequence-like resolution proofs are introduced and in Section 5.2 it is shown that it is NP-complete to decide for a given sequence of clauses whether it is a regular resolution poof.

Acknowledgments

Chapter 1

Propositional Proof Complexity

1.1 Propositional Logic

A *logic* is a formal system that consists of symbols, formulas and models. The *formulas* are composed of the symbols according to given rules and a *model* of a formula is a interpretation of the symbols such that the formula evaluates to a specific value. Even though there are logics (so called multi-valued and fuzzy logics) in which formulas can evaluate to as far as infinite many values, this value is most often defined to be either *true* or *false*.

The *propositional logic* or *Boolean logic* was already studied by George Boole in the mid 19th century and its roots go back to Aristoteles. It is maybe the simplest and most natural logic. The formulas of propositional logic consist of *propositional variables* that are adjunct with logical connectives *and*, *or* and *not*. The propositional models are *variable assignments* that define the propositional variables to be either true or false. If a total variable assignment is applied to a formula then the formula evaluates to true or false.

Definition (Propositional Formulas) The set of propositional variables $\mathcal{V}$ is an infinite set of variables with $\mathcal{V} \cap \{\wedge, \vee, \neg\} = \varnothing$.

The *set of formulas* (over $\mathcal{V}$) is the smallest set $\mathcal{F}$ such that

1) $\mathcal{V} \subseteq \mathcal{F}$.

2) If $F \in \mathcal{F}$ then $\neg F \in \mathcal{F}$.

3) If $F_0 \in \mathcal{F}$ and $F_1 \in \mathcal{F}$ then $(F_0 \wedge F_1) \in \mathcal{F}$ and $(F_0 \vee F_1) \in \mathcal{F}$.

Variables, i.e., elements of $\mathcal{V}$, are often named $x, y, z, x_1, x_2, \ldots$ in this paper.

Sometimes the braces of a formula are left out. For example $F_0 \land F_1 \land F_3$ is written instead of $((F_0 \land F_1) \land F_3)$.

Definition Let F be a formula. $Var(F) \subseteq \mathcal{V}$ is defined to be set of variables that occur as symbols in F.

The *size* $|F|$ of F is the number of symbols in F other then parentheses, i.e.,

$$|x| = 1 \qquad\qquad |\neg F| = 1 + |F|$$
$$|F_0 \land F_1| = |F_0| + |F_1| + 1 \qquad |F_0 \lor F_1| = |F_0| + |F_1| + 1$$

Definition A *variable assignment* α is a mapping $\alpha : dom(\alpha) \to \{0,1\}$ with $dom(\alpha) \subseteq \mathcal{V}$. α is identified with the set $\{ (x, \alpha(x)) \mid x \in dom(\alpha) \}$.

An assignment α is called *total* for the formula F if $Var(F) \subseteq dom(\alpha)$.

To point out that a given assignment α is potentially not total for a formula F, α is called *partial* for F, i.e., every assignment for a formula is a partial assignment. In this case α is also called a *restriction*.

If $\alpha(x) = 1$ for a variable x then x is set to true by α. If $\alpha(x) = 0$ then x is set to false. Otherwise x is unset by α.

A given total assignment for a formula defines it either to be true or false according to the following definition.

Definition Let α be an assignment and let $\mathcal{F}_\alpha = \{F \in \mathcal{F} \mid \alpha \text{ is total for } F\}$. Then α can be extended recursively to a function $\alpha : \mathcal{F}_\alpha \to \{0,1\}$ as follows.

$$\alpha(\neg F) = 1 - \alpha(F)$$
$$\alpha(F_0 \land F_1) = \alpha(F_0) \cdot \alpha(F_1)$$
$$\alpha(F_0 \lor F_1) = 1 - ((1 - \alpha(F_0)) \cdot (1 - \alpha(F_1)))$$

If $\alpha(F) = 1$ for a formula F then F is *satisfied* by α. In that case one also writes $\alpha \vDash F$ and calls α a *satisfying assignment* of F.

F is *satisfiable* if it has a satisfying assignment. Otherwise F is *unsatisfiable*. F is called a *tautology* if $\alpha \vDash F$ for every total assignment α of F. One also writes $\vDash F$ to state that F is a tautology.

Definition Let F and G be formulas. F and G are equivalent, in signs $F \equiv G$, iff $\alpha(F) = \alpha(G)$ for every assignment α that is total for F and G.

The usual formula abbreviations are defined as follows.

Definition Let x_0 be a variable. Define 0 to be the formula $x_0 \land \neg x_0$ and define 1 to be the formula $x_0 \lor \neg x_0$. Sometimes one writes $\square$ instead of 0.

Let F_0 and F_1 be formulas. Then $(F_0 \to F_1)$ is the formula $\neg F_0 \lor F_1$.

It is convenient to put $Var(0) = Var(1) = \varnothing$.

There are other ways to define the formulas 0 and 1. They should only satisfy Proposition 1.1.1.

Proposition 1.1.1 The formula 0 is unsatisfiable and the formula 1 is a tautology.

Since there are infinitely many variables, it is assumed w.l.o.g that 0 and 1 are the only formulas that contain the variable x_0.

Tautologies are related to unsatisfiable formulas in the following way.

Proposition 1.1.2 Let F be a formula. F is a tautology if and only if $\neg F$ is unsatisfiable.

PROOF F is a tautology   iff $\alpha(F) = 1$ for every total assignment α iff $\alpha(\neg F) = 0$ for every total assignment α iff $\neg F$ is unsatisfiable.   ∎

The next proposition is helpful in the following chapters.

Proposition 1.1.3 Let F be a formula. Then $\vDash F \rightarrow 0$ if and only if F is unsatisfiable.

PROOF By definition $F \rightarrow 0$ is the formula $G = 0 \vee \neg F$. Thus $\vDash G$ iff $\alpha(\neg F) = 1$ for every total assignment α for G iff $\neg F$ is a tautology   iff F is unsatisfiable. Thereby the last implication follows from Proposition 1.1.2. ∎

Definition (SAT) The *Boolean satisfiability problem* (SAT) is the algorithmic problem to decide for a given propositional formula whether it is satisfiable.

coSAT is defined to be the complementary problem to SAT, i.e., to decide for a given formula whether it is unsatisfiable.

In terms of deterministic algorithms the problems SAT and coSAT are identical. But for non-deterministic algorithms SAT and coSAT differ since a non-deterministic algorithm for SAT can not be used directly to solve coSAT. In fact there is a non-deterministic polynomial time algorithms for SAT but there is no such algorithm for coSAT unless NP=coNP, which is widely believed to be false.

In general an algorithmic problem is called *decidable* if there exists an algorithm that solves the problem. For logics with quantifiers, the problem whether a given formula is satisfiable is often undecidable. But there is a trivial algorithm for SAT.

Theorem 1.1.4 SAT is decidable.

PROOF The following algorithm decides SAT. Let F be a formula. Then the number of variables $|Var(F)| = n$ is finite. Thus there are 2^n different total assignments for F. These assignments are picked systematically one after another and it is checked whether F is satisfied by the picked assignment. For a given total assignment α the value $\alpha(F)$ can be computed in linear time by simply following the recursive definition from above. Therefore the described algorithm runs in time $O(|F|2^n)$.   ∎

The above algorithm is called the *trivial algorithm* for SAT. Every deterministic algorithm for SAT can also be used to decide whether a formula is a tautology in the same running time.

Definition (TAUT) The *Boolean tautology problem* (TAUT) the algorithmic problem to decide for a given propositional formula whether it is a tautology.

Corollary 1.1.5 TAUT is decidable.

PROOF Let F be a formula. By Proposition 1.1.2 is F tautological iff $\neg F$ is unsatisfiable. But the latter can be decided by the trivial algorithm for SAT.∎

1.2   The Complexity of the SAT-Problem

Apart from the philosophic attraction of the analysis of the propositional logic, propositional formulas have also important applications in the field of computer science.

On the one hand they are related to Boolean circuits and an understanding of the propositional logic conveys to an understanding of the capabilities and limits in the design of digital circuits, which are the fundamental unit of computer hardware.

On the other hand many important algorithmic problems can be expressed efficiently in terms of propositional formulas. The maybe most important result in that area is the theorem of Cook [13] which states that SAT is decidable deterministically in polynomial time if and only if P = NP, i.e., iff every algorithmic problem that can be solved by a non-deterministic algorithm in polynomial time can also be solved deterministically in polynomial time.

Theorem 1.2.1 (Cook 1971) SAT is NP-complete.

The proof (see [13]) of Cook's theorem is constructive and therefore algorithms with a running time $O(f(n))$ for SAT can be used to solve a problem in NP in time $O(f(p(n)))$ where p is a polynomial if f is a non-decreasing function.

It is generally assumed in this paper that every time bound function f as above is non-decreasing.

Even though the question is still unsolved, it is widely believed that P $\neq$ NP. In that case it would be impossible to find a polynomial time algorithm for SAT.

By definition coSAT is coNP-complete and therefore it follows from Proposition 1.1.2 that TAUT is coNP-complete, too.

Corollary 1.2.2 coSAT and TAUT are coNP-complete.

It is shown next that a restriction of the syntax of the propositional formulas is already enough to study the computational complexity of SAT. This restricted syntax is called conjunctive normal form (CNF) and every formula F can be transformed in polynomial time into a CNF formula F' such that F is satisfiable if and only if F' is satisfiable.

Definition (Conjuctive Normal Form) A formula l is a *literal* if there is a variable $x \in \mathcal{V}$ with $l = x$ or $l = \neg x$.

A formula C is called a *clause* if $C = \square$ or if there exist an integer $k \geq 1$ and literals $l_1, \ldots, l_k$ with $C = l_1 \vee \ldots \vee l_k$. If $k = 1$ then C is called *unit clause*. If $C = \square$ then C is called the *empty clause*. Identify $l_1 \vee \ldots \vee l_k$ with the set $\{l_1, \ldots, l_k\}$ and identify $\square$ with the empty set $\varnothing$.

A clause C is called *tautological* iff there is a variable x with $\{x, \neg x\} \subseteq C$.

Let F be a formula. F is in *conjunctive normal form (CNF)* if there are clauses $C_1, \ldots, C_m$ such that $F = C_1 \wedge \ldots \wedge C_m$ $(m > 0)$. In that case F is identified with the set $\{\{l_1, \ldots, l_k\} \mid l_1 \vee \ldots \vee l_k = C_i$ for a $i \in \{1, \ldots, m\}\}$.

F is in k-CNF iff every clause in F has at most k literals.

Note that by definition the formulas 0 and 1 are CNF formulas.

It is convenient to introduce the following notation.

Definition Let C be a clause, x a variable and l a literal with $Var(l) = \{x\}$. Let $\epsilon \in \{0, 1\}$. Define

$$x^\epsilon = \begin{cases} x & \text{if } \epsilon = 1 \\ \neg x & \text{if } \epsilon = 0 \end{cases}, \qquad \bar{l} = x^{1-\epsilon} \text{ if } l = x^\epsilon \qquad \text{and} \qquad \bar{C} = \{\, \bar{l} \mid l \in C \,\}$$

The proof of the following theorem shows how to transform a formula into a CNF formula. For the proof, the notion of a subformula is needed.

Definition (Subformula) Let F be a formula. The set $Sub(F)$ of *subformulas* of F is defined recursively as follows.

$$Sub(x) = \{x\} \qquad\qquad Sub(F_0 \vee F_1) = \{(F_0 \vee F_1)\} \cup Sub(F_0) \cup Sub(F_1)$$
$$Sub(\neg F) = \{\neg F\} \cup Sub(F) \quad Sub(F_0 \wedge F_1) = \{(F_0 \wedge F_1)\} \cup Sub(F_0) \cup Sub(F_1)$$

Theorem 1.2.3 For every formula F exists a CNF formula $CNF(F)$ with $|CNF(F)| = O(|F|)$ such that $CNF(F)$ is satisfiable if and only if $CNF(F)$ is satisfiable. $CNF(F)$ is computable in polynomial time.

PROOF For every subformula G of F let v_G be the variable x if $G = x$ and let v_G otherwise be a new variable such that $v_G \neq v_{G'}$ for $G \neq G'$ with $G, G' \in Sub(F)$. For every $G \in Sub(F)$ the set of clauses $\mathcal{C}_G$ states that v_G is true if and only if G is true. $\mathcal{C}_G$ is defined as follows.

If $G = x$ then $\mathcal{C}_x = \varnothing$
If $G = \neg H$ then $\mathcal{C}_G = \{\{v_G \vee v_H\}, \{\neg v_G \vee \neg v_H\}\}$
If $G = (H_0 \wedge H_1)$ then $\mathcal{C}_G = \{\{\neg v_{H_0} \vee \neg v_{H_1} \vee v_G\}, \{\neg v_G \vee v_{H_0}\}, \{\neg v_G \vee v_{H_1}\}\}$
If $G = (H_0 \vee H_1)$ then $\mathcal{C}_G = \{\{\neg v_G \vee v_{H_0} \vee v_{H_1}\}, \{\neg v_{H_0} \vee v_G\}, \{\neg v_{H_1} \vee v_G\}\}$

Let $CNF(F) = \bigcup \{ \mathcal{C}_G \mid G \in Sub(F) \} \cup \{\{v_F\}\}$ be the union of $\{\{v_F\}\}$ with all clause sets $\mathcal{C}_G$. Then F is satisfiable if and only if $CNF(F)$ is satisfiable.

For the "if part", let $\tilde{\alpha}$ be an assignment with $\tilde{\alpha} \vDash CNF(F)$. Then it follows by induction on G that for every $G \in Sub(F)$, $\alpha(G) = \alpha(v_G)$. But since $\{v_F\} \in CNF(F)$, $\alpha(v_F) = 1$ and thus $\alpha(F) = 1$.

For the "only if part", let α be an assignment with $\alpha \vDash F$. Define the total assignment $\tilde{\alpha}$ for $CNF(F)$ by $\tilde{\alpha}(v_G) = \alpha(G)$. Then it follows directly from the definition of $CNF(F)$ that $\tilde{\alpha} \vDash CNF(F)$. ∎

Note that there are formulas $F_1, F_2, \ldots$ with $|F_k| = O(k)$ such that $|G_k| \geq k2^k$ for every CNF formula G_k with $G_k \equiv F_k$. Thus it is generally impossible to transform a formula in a equivalent CNF formula with polynomial blow-up.

Definition The algorithmic problem CNF-SAT is to decide for a CNF formula F whether F is satisfiable.

Let $k > 1$. The problem k-SAT is to decide for a k-CNF formula whether it is satisfiable.

Since SAT is in NP, CNF-SAT is in NP and because Theorem 1.2.3 is a reduction of SAT to 3-SAT, 3-SAT and CNF-SAT are NP-complete.

Corollary 1.2.4 3-SAT and CNF-SAT are NP-complete.

The following theorem is given without a proof and not used during this thesis. For a proof see [13] and [16].

Theorem 1.2.5 2-SAT is decidable in polynomial time.

Example (The Ordering Principle) The *ordering principle* for $n > 1$ is the fact that there is a minimal element in every total order $\prec$ of the set

$\{1, 2, \ldots, n\}$. A generalization of the ordering principle, namely the *well-ordering priciple*, is one of most fundamental theorems of mathematics and equivalent to the *axiom of choice* in the Zermelo-Fraenkel set theory.

The unsatisfiable CNF formulas OP_n $(n > 1)$ are a formalization of the negation of the ordering principle in terms of CNF-SAT, i.e., OP_n is satisfiable if there is a total order of $\{1, 2, \ldots, n\}$ that has no minimal element. Thereby $Var(OP_n) = \{ x_{ij} \mid i, j \in \{1, \ldots, n\}, i \neq j \}$ and a total assignment α of OP_n corresponds to the order $\prec_\alpha$ with $i \prec_\alpha j$ iff $\alpha(x_{ij}) = 1$. OP_n consists of the following clauses.

$$(\bar{x}_{ij} \vee \bar{x}_{ji}) \wedge (x_{ij} \vee x_{ji}) \quad \text{for } 1 \leq i < j \leq n \qquad \text{(antisymmetric)}$$

$$\bar{x}_{i_1 i_2} \vee \bar{x}_{i_2 i_3} \vee \bar{x}_{i_3 i_1} \quad \text{for any distinct } i_1, i_2, i_3 \in \{1, \ldots, n\} \qquad \text{(transitive)}$$

$$\bigvee_{1 \leq k \leq n, k \neq j} x_{kj} \quad \text{for } j \in \{1, \ldots, n\} \qquad \text{(no mininmal)}$$

The transitivity axioms are written differently from the usual form $x_{i_1 i_2} \wedge x_{i_2 i_3} \rightarrow x_{i_1 i_3}$. The symmetric form used here is more convenient for the following examples and equivalent to the usual form because of the antisymmetry clauses. Note that there are exactly two such transitivity axioms for any set of three distinct $i_1, i_2, i_3 \in \{1, \ldots, n\}$.

$$(x_{12} \vee x_{21}), \ (\bar{x}_{12} \vee \bar{x}_{21}), \quad (x_{13} \vee x_{31}), \ (\bar{x}_{13} \vee \bar{x}_{31}), \quad (x_{14} \vee x_{41}), \ (\bar{x}_{14} \vee \bar{x}_{41})$$
$$(x_{23} \vee x_{32}), \ (\bar{x}_{23} \vee \bar{x}_{32}), \quad (x_{24} \vee x_{42}), \ (\bar{x}_{24} \vee \bar{x}_{42}), \quad (x_{34} \vee x_{43}), \ (\bar{x}_{34} \vee \bar{x}_{43})$$

$$(\bar{x}_{12} \vee \bar{x}_{23} \vee \bar{x}_{31}), \quad (\bar{x}_{13} \vee \bar{x}_{32} \vee \bar{x}_{21}), \quad (\bar{x}_{12} \vee \bar{x}_{24} \vee \bar{x}_{42}), \quad (\bar{x}_{14} \vee \bar{x}_{42} \vee \bar{x}_{21})$$
$$(\bar{x}_{13} \vee \bar{x}_{34} \vee \bar{x}_{41}), \quad (\bar{x}_{14} \vee \bar{x}_{43} \vee \bar{x}_{31}), \quad (\bar{x}_{23} \vee \bar{x}_{34} \vee \bar{x}_{42}), \quad (\bar{x}_{24} \vee \bar{x}_{43} \vee \bar{x}_{32})$$

$$(x_{21} \vee x_{31} \vee x_{41}), \quad (x_{12} \vee x_{32} \vee x_{42}), \quad (x_{13} \vee x_{23} \vee x_{43}), \quad (x_{14} \vee x_{24} \vee x_{34})$$

Figure 1.1: OP_4, the ordering principle for $n = 4$

The unsatisfiable CNF formula OP_4 (see Figure 1.1) is used as a running example to present definitions and concepts in this thesis. A proof of its unsatisfiability can be found in Section 3.1.

If a given assignment α is partial for a formula F then one can apply α to F anyway to obtain the restricted formula $F|_\alpha$. The idea is to look at $F|_\alpha$ with regard of the total assignments β for F with $\beta \supseteq \alpha$ in order to determine for example if already $\beta(F) = 0$ or $\beta(F) = 1$ for all of these β.

Although it is possible to define restrictions for arbitrary formulas, it is done here only for CNF formulas since it is easier and sufficient for this paper.

Definition (Restriction) Let C be a clause, F a CNF formula and α an assignment. The *restriction* of C under α is the formula

$$C|_\alpha = \begin{cases} 1 & \text{if there is a } l \in C \text{ with } \alpha(l) = 1 \\ \square & \text{if } \alpha(l) = 0 \text{ for every } l \in C \\ \{\, l \in C \mid l \notin dom(\alpha) \,\} & \text{otherwise} \end{cases}$$

The *restricted CNF formula* or the *restriction* $F|_\alpha$ of F under α is defined as follows.

$$F|_\alpha = \begin{cases} 0 & \text{if there is a } C \in F \text{ with } C|_\alpha = 0 \\ 1 & \text{if } C|_\alpha = 1 \text{ for every } C \in F \\ \{ \, C|_\alpha \mid C \in F \, \} - \{1\} & \text{otherwise} \end{cases}$$

If $F|_\alpha = 1$ then α is called a partial satisfying assignment for F.

Note that 0 and 1 in the above definition are not numbers but formulas. The next lemma states that the restriction of a formula has the mentioned properties.

Lemma 1.2.7 Let F be a CNF formula, α an assignment and x a variable with $x \in Var(F) - dom(\alpha)$. Let β be a total assignment for F with $\alpha \subseteq \beta$. Then

  (a) $\beta \vDash F$ if and only if $\beta \vDash F|_\alpha$

  (b) $F|_\alpha$ is satisfiable if and only if $F|_{\alpha \cup \{(x,0)\}}$ or $F|_{\alpha \cup \{(x,1)\}}$ is satisfiable.

PROOF It is shown first that $\beta \vDash F$ iff $\beta \vDash F|_\alpha$. Therefore it suffices to show that $\beta(C|_\alpha) = \beta(C)$ for all $C \in F$. But $\beta(C) = 1$ iff there is an $l \in C$ with $\beta(l) = 1$ iff there is an $l \in C$ with $\alpha(l) = 1$ or $(\beta - \alpha)(l) = 1$ iff $\beta(C|_\alpha) = 1$.

To show the second part of the lemma let at first β be an assignment with $\beta \vDash F|_\alpha$ and $\alpha \subseteq \beta$. Then by (a) $\beta \vDash F$ and since $\beta \supseteq \alpha \cup \{(x, \epsilon)\}$ for an $\epsilon \in \{0,1\}$ it follows by (a) $\beta \vDash F|_{\alpha \cup \{(x,0)\}}$ or $\beta \vDash F|_{\alpha \cup \{(x,1)\}}$.

Let on the other hand $\alpha' = \alpha \cup \{(x, \epsilon)\}$ and let $\beta \vDash F|_{\alpha'}$ for an $\epsilon \in \{0,1\}$. Assume again w.l.o.g that $\alpha \subseteq \beta$. Let $\beta' = (\beta - \{(x, \beta(x))\}) \cup \{(x, \epsilon)\}$. Then $\alpha \subseteq \alpha' \subseteq \beta'$ and $\beta' \vDash F|_{\alpha'}$ because $x \notin Var(F|_{\alpha'})$. Thus one can apply (a) twice to get first $\beta' \vDash F$ and then $\beta' \vDash F|_\alpha$.   ∎

1.3 Proof Systems

Proofs in a logic are exactly defined mathematical objects that show that formulas are tautological. Although proofs are maybe the most important concept in mathematics, an attempt of an exact definition has been done first by Gottlob Frege in the late 19th century ([20]).

Historically a proof for a tautology is something that can be obtained from a set of *axioms* by applying *inference rules*. A common inference rule is for example $x, (x \to y) \vdash y$ which is called *modus ponens*. Besides the correctness, the only demand on the inference rules and axioms is, that it should be easy to check whether the rule can be applied or whether a formula is an axiom,

respectively. A formal definition of *easy* in the above definition is at least computable in polynomial time. Such a collection of axioms and inference rules is called a *calculus* for the propositional logic. Rule based proof systems that are *implicationally complete* and that have axioms and rules of inference that are closed and substitution are called *Frege systems*.

This common notion of a proof system has been generalized 1979 by Cook and Reckhow in [14]. In their definition a proof does not necessary have to be obtained from inference rules but only needs to be a string for which it is computable in polynomial time whether it is a valid proof for a given formula or not. Frege systems match this definition since the inference rules are decidable in polynomial time and it is decidable in polynomial time whether a formula is an axiom.

Definition (Proof System) A *proof system* is a binary relation $R \subseteq \mathcal{F} \times \Sigma^*$ between formulas and strings over an alphabet Σ such that R is decidable in polynomial time and a formula F is a tautology if and only if there exists a $\omega \in \Sigma^*$ with $R(F, \omega)$.

If $R(F, \omega)$ then ω is called a R-proof of F.

The subject of *propositional proof complexity* is the analysis of the size of the proofs in different proof systems. A proof system is considered to be *efficient* or *strong* if tautologies have short proofs in that system. The following definition is essential for a analysis of the strength of proof systems.

Definition Let $R \subseteq \mathcal{F} \times \Sigma^*$ be a proof system, F a tautology and $\omega \in \Sigma^*$.

The *size* $|\omega|$ of ω is the number of symbols in ω, i.e., $|\omega| = k$ iff $\omega \in \Sigma^k$.

The *complexity* $\mathcal{C}_R(F)$ of F in R is the size of the smallest R-proof for F, i.e., $\mathcal{C}_R(F) = \min\{ |\omega| \mid R(F, \omega) \}$.

R is *polynomially bounded* iff there is a polynomial p such that $\mathcal{C}_R(F) \leq p(|F|)$ for every tautology F.

One of the results that has been proved in [14] is Theorem 1.3.1. It states that it would follow from the existence of a polynomially bounded proof system that NP = coNP and this is believed to be false for reasons that are similar to those in the case P vs NP.

Theorem 1.3.1 There exists a polynomially bounded proof system if and only if NP = coNP.

PROOF It is known from complexity theory (see for example [29]) that there exists a non-deterministic, polynomial time Turing machine for a problem L if and only if there is a polynomially decidable relation R and a polynomial p such that $x \in L$ iff $R(x, \omega)$ for a ω with $|\omega| \leq p(|x|)$.

Therefrom the theorem follows immediately: If R is a polynomial bounded proof system then R is decidable in polynomial time and there is a polynomial p such that for every tautology F there is a ω with $|\omega| \leq p(|F|)$ and $R(F, \omega)$. By the above this is the case if and only if TAUT is in NP. But since TAUT is coNP-complete by Corollary 1.2.2, the latter is true if and only if NP = coNP.∎

Since deterministic complexity classes are closed under complement, P $\neq$ NP is a consequence of NP $\neq$ coNP and the next corollary follows immediately from Theorem 1.3.1.

Corollary 1.3.2 If there is no polynomially bounded proof system then P $\neq$ NP.

To show that a proof system R is not polynomially bounded one has to find a sequence of formulas $F_1, F_2, F_3, \ldots$ such that $|F_k| \leq p(k)$ for a polynomial p but $\mathcal{C}_R(F_k) \geq f(k)$ for a superpolynomial function f. Such an f is called a *superpolynomial lower bound* for R.

Up to the present, there are a couple of proof systems which are already known to have superpolynomial lower bounds (see [4] for a survey) and there are attempts to prove lower bounds for stronger and stronger proof systems until enough techniques are developed to prove a superpolynomial lower bound for all proof systems and thus NP $\neq$ coNP. This research enterprise is often called *Cook's program*.

In this paper lower bounds for the proof systems resolution and regular resolution are presented in Chapter 2.

To compare different proof systems in terms of there efficiency, the following notions are helpful.

Definition Let R and Q be proof systems.
If there is a polynomial p such that $\mathcal{C}_R(F) \leq p(\mathcal{C}_Q(F))$ for each tautology $F \in \mathcal{F}$ then R *simulates* Q and one writes $Q \leq R$. Otherwise Q is *separated* from R.
If $R \leq Q$ and $Q \leq R$ then one writes $Q \equiv R$ and calls Q and R *equivalent*.
If $R \leq Q$ and Q is separated from R then $R < Q$ and Q is called *stronger* than R.
If neither $R \leq Q$ nor $Q \leq R$ then Q and R are *incomparable*.

For example it has been shown in [14] that all Frege systems are equivalent.

The subject of this thesis is the connection between proof systems and algorithms for SAT. A run of a SAT algorithm on a unsatisfiable formula F can be seen as a proof for the tautology $\neg F$ and by this means the algorithm defines a proof system.

Thanks to Theorem 1.2.3 the statement holds also for CNF-SAT algorithms.

Proposition 1.3.3 Let A be an algorithm for SAT or CNF-SAT which runs in time $f(|F|)$. Then there is a proof system R_A such that $\mathcal{C}_{R_A}(F) \leq f(c|F|)$ for every tautology F and a constant c.

PROOF Let A be an algorithm for SAT or CNF-SAT. In both cases the alphabet Σ of the proofs consists only of one symbol 0, i.e., $\Sigma = \{0\}$. Note that f is assumed to be a non-decreasing function.

If A is a SAT algorithm then $R_A \subseteq \mathcal{F} \times \Sigma$ is defined through $R_A(F, \omega)$ iff $A(\neg F)$ terminates after at most $|\omega|$ steps and returns that $\neg F$ is unsatisfiable. The latter can be obviously decided in time $O(|\omega|)$ and F is a tautology if and only if $R_A(F, 0^{f(|F|)})$. Thus $\mathcal{C}_{R_A}(F) \leq f(|F|)$ and R_A is as wanted.

If A is a CNF-SAT algorithm then define $R_A(F, \omega)$ iff $A(CNF(\neg F))$ terminates in time $|\omega|$ and returns unsatisfiable whereby $CNF(\neg F)$ is the CNF formula from Theorem 1.2.3 for $\neg F$.

By Theorem 1.2.3 there is a constant c such that $|CNF(G)| \leq c|G|$ for every formula $CNF(G)$. Thus F is a tautology if and only if $R_A(F, 0^{f(c|F|)})$ and thus $\mathcal{C}_{R_A}(F) \leq f(c|F|)$. Since R_A is decidable in linear time it is a proof system as wanted. ∎

By applying Proposition 1.3.3 to the trivial algorithm one gets the following corollary.

Corollary 1.3.4 There is a proof system R such that $\mathcal{C}_R(F) = O(|F|2^n)$ for every tautology F.

Despite of this this trivial relation of SAT algorithms and proof systems, there are surprisingly strong connections between SAT algorithms and natural, rule based proof systems. For one thing lower bounds of the natural systems can be used to derive lower bounds for whole families of algorithms and for another thing the relationship of different families of algorithms can be analyzed by comparing the corresponding proof systems.

Both is accomplished for variations of DLL algorithms and resolution in the next chapters.

Chapter 2

DLL Algorithms

2.1 The Basic DLL Algorithm

A trivial way to decide whether a given Boolean formula F over n variables is satisfiable, is to go through all 2^n possible assignments α and determine whether $\alpha \vDash F$. Since the latter can be decided in linear time, this algorithm runs in time $O(|F|2^n)$.

A significant improvement of the trivial search algorithm is to set the n variables of an assignment one after another and test for each partial assignment α whether the given formula F is already satisfied or falsified by α, i.e., whether $F|_\alpha = 1$ or $F|_\alpha = 0$. In the first case every total assignment $\beta \supseteq \alpha$ satisfies F and in the second case it is possible to skip the testing of all assignments that set the first variables to same values as α (that are $2^{n-|\alpha|}$ many).

This improved search algorithm is called *DLL algorithm* after the authors Davis, Logeman and Loveland, who first considered it in [15]. Below is a recursive definition of the basic DLL algorithm in pseudo-code. The input is a CNF F and an assignment α with $dom(\alpha) \subseteq Var(F)$. The top level call $\mathrm{DLL}(F, \varnothing)$ returns UNSAT if F is unsatisfiable or else a satisfying assignment for F.

Algorithm 2.1.1 (Basic DLL Algorithm)

```
DLL(F, α)
1      if  F|α = 0 then
2            return UNSAT
3      if  F|α = 1 then
4            return α
5      choose  x ∈ Var(F|α)
6      β ←DLL(F, α ∪ {(x,0)})
7      if  β ≠ UNSAT then
8            return β
9      else
10           return DLL(F, α ∪ {(x,1)})
```

A variable that is chosen in line 5 of DLL is called a *branching variable.*

Note that this definition does not describe a single deterministic algorithm because it depends on the choices of the variables in line 5. It would therefore be more exact to call it either a non-deterministic algorithm or an algorithm schema for deterministic and randomized algorithms. Because only the deterministic versions of DLL algorithms are of interest in this paper, the latter point of view is taken here.

Good choices of branching variables in line 5 improve the experimental running time of the algorithm enormously. It can be, for example, helpful to first select variables that occur in short clauses of the current restricted formula. It can also be faster to choose variables that occur in many clauses before variables that occur rarely. For an overview on variable branching heuristics see for example [19] or [28].

For satisfiable formulas it often makes a big running time difference if the branch with the current variable x set to 1 or the branch with x set to 0 is searched first. Some heuristics of selecting the first value can be found in [19] and [28], too. But since unsatisfiable formulas are the ones of more interest in this paper and the choice of the first value of the branching variable causes no running time differences for these, it is assumed that 0 is always the first value that is assigned to the branching variable.

The next theorem states that every choice of the branching variables leads to correct and complete algorithms for SAT.

Theorem 2.1.2 (Correctness of DLL) Let F be a formula. Then every execution of $\mathrm{DLL}(F, \varnothing)$ terminates and returns a partial satisfying assignment if F is satisfiable and UNSAT otherwise.

PROOF Let F be a formula, α a partial assignment and $n = |Var(F|_\alpha)|$.

It is shown by induction on n that $\mathrm{DLL}(F, \alpha)$ terminates and that it returns UNSAT if $F|_\alpha$ is unsatisfiable or a partial satisfying assignment $\alpha' \supseteq \alpha$ for F if $F|_\alpha$ is satisfiable.

Induction Basis: If there are no variables in $F|_\alpha$ then either $F|_\alpha = 0$ or $F|_\alpha = 1$ and the algorithm terminates within the first 4 lines. In the former case $F|_\alpha$ is unsatisfiable and the algorithm returns UNSAT. In the latter case $F|_\alpha$ is satisfiable and the partial satisfying assignment α is returned.

Induction Step: Let $n > 0$, $x \in Var(F|_\alpha)$ and $\alpha_i = \alpha \cup \{(x, i)\}$ for $i = 0, 1$. Then $F|_{\alpha_i}$ contains at most $n - 1$ variables. If $F|_\alpha$ is unsatisfiable then $F|_{\alpha_i}$ is unsatisfiable for $i = 0$ and $i = 1$ by Lemma 1.2.7. Therefore $\mathrm{DLL}(F, \alpha_i)$ terminates by induction for $i = 0, 1$ and in both cases UNSAT is returned. Thus $\mathrm{DLL}(F, \alpha)$ also terminates and returns UNSAT by definition.

Assume now that $F|_\alpha$ is satisfiable. Then it follows by Lemma 1.2.7 that $F|_{\alpha_i}$ is satisfiable for $i = 0$ or $i = 1$. If $F|_{\alpha_0}$ is satisfiable then by induction the

execution of $\mathrm{DLL}(F, \alpha_0)$ terminates and returns a partial satisfying assignment and thus so does $\mathrm{DLL}(F, \alpha)$. If $F|_{\alpha_0}$ is unsatisfiable then the execution of $\mathrm{DLL}(F, \alpha_0)$ terminates and returns UNSAT by and furthermore $F|_{\alpha_1}$ must be satisfiable. Therefore by induction the execution of $\mathrm{DLL}(F, \alpha_1)$ returns a partial satisfying assignment and thus, so does $\mathrm{DLL}(F, \alpha)$. ∎

The following lemma is needed for a later proof. It follows immediately from the correctness theorem.

Lemma 2.1.3 Let F be an unsatisfiable formula. Then an execution of $\mathrm{DLL}(F, \alpha)$ performs an even number of recursive calls

PROOF By induction on the number of recursive calls. If there are no recursive calls then the statement holds since 0 is even. Now assume the $\mathrm{DLL}(F, \alpha)$ performs $r > 0$ recursive calls. From Theorem 2.1.2 it follows that DLL returns UNSAT. Thus by definition there are calls of DLL in line 6 and in line 10 that performs r_0 and r_1 recursive calls, respectively. Then $r_0 + r_1 + 2 = r$. But by induction r_0 and r_1 are even. ∎

As mentioned above, good choices of the branching variables are important to obtain fast real world DLL algorithms. The next proposition describes a simple choice of the branching variables which leads already to a k-SAT algorithm that has a better upper bound than the trivial search algorithm. The idea is just to pick one clause after another and to branch first on every variable in the picked clause.

Proposition 2.1.4 Let F be a k-CNF formula over n variables. Then there is a branching heuristic such that $\mathrm{DLL}(F, \varnothing)$ runs in time $O(|F|(2^k - 1)^{\frac{n}{k}})$.

PROOF Let $F = \{C_1, \ldots, C_m\}$ be a k-CNF formula over n variables. Select the branching variables in line 5 of DLL in the order $\pi = x_{1,1}, \ldots, x_{1,k_1}, x_{2,1},$ $\ldots, x_{2,k_2}, x_{3,1}, \ldots, x_{m',1}, \ldots x_{m',k_{m'}}$ where $x_{i,1}, \ldots x_{i,k_i}$ are the variables of the clause C_i that do not occur in a clause C_j for a $j < i$. It is assumed w.l.o.g that all variables of F occur in the first m' clauses such that that every of these clauses contains at least one new variable.

To estimate the running time of $\mathrm{DLL}(F, \varnothing)$ with that heuristic, a bound on the size of the resulting recursion tree is given. By branching over the k_i variables of the clause C_i that do not accur in a clause C_j for a $j < i$, there are at most 2^{k_i} possible assignments of these variables and therefore at most 2^{k_i} recursive calls of DLL. But if the first k_1 branching variables are selected according to π then there is exactly one assignment that falsifies C_1 and hence one of the corresponding recursive calls of DLL terminates immediately and has no children in the recursion tree.

Equally in every group of the k_i unset variables of the clause C_i that are selected according to π there is at least one clause in F that is falsified by one of the 2^{k_i} possible assignments. Therefore there are at most $(2^{k_i} - 1)$ inner nodes in the recursion tree for every group of unset variables of a clause that are selected according to π. Since F has n variables, there are over all less then $(2^{k_1} - 1) \cdot \ldots \cdot (2^{k_{m'}} - 1)$ inner notes in the recursion tree for integers k_i with $k_i \leq k$ and $k_1 + \ldots + k_{m'} = n$. By the convexity of this function it follows that the algorithm runs in time $O(|F|(2^k - 1)^{\frac{n}{k}})$. ∎

One could think that there might be a great, undiscovered strategy in the choosing of the branching variables of the basic DLL algorithm that leads to a k-SAT algorithm with a sub exponential worst case running time. It is not obvious to see that all algorithms that can be derived from the basic DLL algorithm schema are cursed to have an exponential worst case running time. Actually it is shown in Section 4.2 that there are k-CNF formulas $F_1, F_2, F_3, \ldots$ and a constant $c > 1$ such that $\mathrm{DLL}(F_i, \varnothing)$ needs time $\Omega(c^i)$ for every possible choice of the branching variables.

Like basic DLL algorithms, all other known algorithms for SAT have a worst case running time that is exponential in the number of the variables of the input formula. Nevertheless there are different ways to implement and to improve the basic DLL algorithm in order to solve real-word SAT problems with today's computers (see [19], [32]).

Most of the fastest complete SAT solvers like Chaff ([27]) use these improved versions of the basic DLL algorithm. Thereby it is common practice to implement an imperative version of the algorithm as well as efficient data structures that allow a fast undo of variable assignments as described in [18].

2.2   DLL with Learning by Unit Propagation

Two of the most successful improvements of DLL that are used by most of the modern SAT solvers are unit propagation and learning.

Unit clause propagation was already studied by Davis and Putnam in [16] and is based on the following observation. Suppose there is a recursive call $\mathrm{DLL}(F, \alpha)$ of DLL. As long as there is a clause $C = \{l\}$ in $F|_\alpha$ that has only one unset literal, one can set the variable x of l so as to satisfy C since the other assignment would falsify F. Such a clause C is called a *unit clause*, and the assignment of a value to x to satisfy C is called a *unit clause assignment*. It is possible that a new unit clause appears after a unit clause assignment. In that case, or if there is another unit clause in $F|_\alpha$, the variable of this clause can be set to satisfy the unit clause. If the empty clause $\square$ occurs after a sequence of unit clause assignments then one speaks of a *conflict by unit propagation*. The term *unit*

propagation is used to name the process of assigning unit clauses until a the empty clause occurs or no unit clauses are left. Sometimes unit propagation is also called logical implication in the literature.

Note that the basic DLL algorithm schema given in Section 2.1 is able to simulate unit propagation by choosing the variables that appear in unit clauses as branching variables in line 5. Thus the lower bound for DLL algorithms that is given in Section 4.2 holds also for DLL algorithms that use unit propagation.

The concept of *learning in DLL algorithms* was first introduced by Silva and Sakallah [32]. It is based on the idea that if a restricted formula $F|_\alpha$ is unsatisfiable then this information can be coded in a clause C_α such that $F \cup \{C_\alpha\} \equiv F$ and it is possible to continue the search for truth assignments on the formula $F \cup \{C_\alpha\}$. The clause C_α is easy to find: it is the largest clause that is falsified by α.

The advantage of continuing the search on $F' = F \cup \{C_\alpha\}$ is that the algorithm is able to stop searching for satisfying assignments $\beta' \supset \beta$ each time it tries an assignment $\beta \supseteq \alpha$, since $F'|_\beta = 0$.

The following lemma shows that adding clauses C_α as described above does not affect the behavior of a formula F under assignments, i.e., that $F \equiv F \cup \{C_\alpha\}$. On the other hand every clause C with $F \equiv F \cup \{C\}$ corresponds to an assignment for which $F|_\alpha$ is unsatisfiable. That is why the only way to learn clauses is to use the information $F|_\alpha \to 0$ for an assignment α directly or indirectly.

Definition Let α be an assignment. Then C_α is defined as the the clause $\{\neg x \mid \alpha(x) = 1\} \cup \{x \mid \alpha(x) = 0\}$, i.e., C_α is the largest clause that is falsified by α.

Lemma 2.2.1 Let F be a formula and α an assignment. Then $F \equiv F \cup \{C_\alpha\}$ if and only if $F|_\alpha$ is unsatisfiable.

PROOF "$\Rightarrow$": By contradiction. Let $G = F|_\alpha$ be satisfiable and suppose that β is an assignment with $G|_\beta = 1$. Then it follows for $\gamma = \beta \cup \alpha$ that $F|_\gamma = 1$ but $C_\alpha|_\gamma = 0$ and therefore $F \not\equiv F \cup \{C_\alpha\}$.

"$\Leftarrow$": Let $G = F|_\alpha$ be unsatisfiable and β a total assignment. It suffices to show that if $F|_\beta = 1$ then $C_\alpha|_\beta = 1$. Therefore suppose that $F|_\beta = 1$. Then $\alpha \not\subseteq \beta$ because otherwise $\beta - \alpha$ would be a satisfying assignment for $F|_\alpha$. But since β is a total assignment and $Var(C_\alpha) \subseteq dom(\alpha)$ there exists a literal $a \in C_\alpha$ with $\beta(a) = 1$ and thus $C_\alpha|_\beta = 1$. ∎

Note that it is NP-hard to decide for a formula F and an arbitrary assignment α whether $F|_\alpha$ is satisfiable because the problem is a generalization of SAT. Therefore one has to find a way to provide the information $F|_\alpha \to 0$ during the run of a DLL algorithm efficiently in order to learn clauses.

Obviously $F|_\alpha \to 0$ follows every time if $\mathrm{DLL}(F, \alpha)$ returns `UNSAT`. But the clause C_α is worthless for the rest of the execution since no assignment $\beta \supseteq \alpha$ is tried anymore during a recursive execution of DLL and thus C_α is never falsified. For that reason one provides additional information during an execution of DLL to discover small assignments $\alpha' \subset \alpha$ with $F|_{\alpha'} \to 0$.

The first, and to the knowledge of the author so far only, method of finding these small assignments in DLL algorithms is *learning by unit propagation* as presented first in [32]. Almost all deterministic state of the art SAT solvers that are fast according to SAT competitions (see [7], [8] and [9]) like Chaff [27], Zchaff [26] or MiniSAT [17] use DLL algorithms with learning by unit propagation.

The idea is to keep track of the exact variable assignments which cause a variable to be set by unit propagation. If there appear two contradicting clauses $\{x\}$ and $\{\neg x\}$ in $F|_\alpha$ then $F|_\alpha$ is unsatisfiable and the clause that corresponds to variables that causes $x = 0$ and $x = 1$, respectively, is learned. One can think of the variable settings that are represented in this clause as the *reasons* for the *conflict*.

To provide an easy way to compute a set of variables that are responsible for a conflict, the reasons for every unit propagation are saved in a data structure that is called *UP-graph* in this paper.

Definition (UP-Graph) Let F be a CNF formula. A *UP-graph G* for F is a directed acyclic graph (dag) $G = (V, E)$ with $V \subseteq Var(F) \cup \{ \neg x \mid x \in Var(F) \}$ such that

-   $|V \cap \{x, \neg x\}| \leq 1$ for every $x \in Var(F)$

-   every $v \in V$ has in-degree 0 or there is a clause $C_v \in F$ with $C_v = \{v\} \cup \{ \bar{l} \mid (l, v) \in E \}$.

Define $\alpha_G = \{ (x, \epsilon) \mid x^\epsilon \in V \}$ to be the partial assignment induced by G.

A leaf (i.e., a vertex with in-degree 0) in a UP-graph represents a setting of a branching variable in the DLL algorithm and an inner node v represents an assignment that has been forced by unit propagation with respect to a unit clause $\{v\}$ that arises out of F by satisfying the predecessors of v.

If a conflict occurs in the formula F restricted by the induced assignment of a UP-graph G, i.e., $\{x\} \in F|_{\alpha_G}$ and $\{\neg x\} \in F|_{\alpha_G}$ for a variable x, then one speaks of a *conflict* and can extend G to a *conflict graph* as follows.

Definition (Conflict Graph) Let F be a CNF formula and $G = (V, E)$ a UP-graph for F. Let $C_1, C_2 \in F$ be clauses with $C_1|_{\alpha_G} = \{y\}$ and $C_2|_{\alpha_G} = \{\neg y\}$. Then the dag $G' = (V', E')$ with $V' = V \cup \{y, \neg y, \Box\}$ and

$E' = \{(y, \Box), (\neg y, \Box)\} \cup \{ (\bar{v}, y) \mid v \in C_1, v \neq y \} \cup \{ (\bar{v}, \neg y) \mid v \in C_2, v \neq \neg y \}$
is called a *conflict graph* for F and y is called the *conflict variable* of G'.

The *conflict clause* of G' is the clause $C_{G'} = \{ \bar{l} \mid l \in V$ has in-degree 0 and there is a path from l to $\Box$ in $G' \}$.
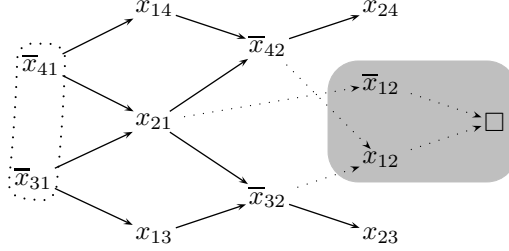

Figure 2.1: A UP-graph for OP$_4$ that is extended to a conflict graph with the nodes x_{12}, $\bar{x}_{12}$ and $\Box$. The conflict clause is ($x_{31} \vee x_{41}$).

The conflict clause C_G of a conflict graph G codes the fact that $F|_\beta$ is unsatisfiable for $\beta = \{ (x, \epsilon) \mid x^{1-\epsilon} \in C_G \}$. This follows from Lemma 2.2.1 and Lemma 2.2.2 which is proved below.

In order to prove Lemma 2.2.2 the following definitions are useful.

Definition Let $G = (V, E)$ be a dag and $u \in V$. Define

$$V_G^0 = \{v \in V \mid v \text{ has in-degree 0} \}$$

$$Depth_G(u) = \begin{cases} 0 & \text{if } u \in V_G^0 \\ 1 + \max\{ Depth_G(v) \mid (v, u) \in E \} & \text{otherwise} \end{cases}$$

$$V_{G\lceil u} = \{ v \in V \mid \text{there is a path from } v \text{ to } u \text{ in } G \}$$

$$E_{G\lceil u} = E \cap (V_{G\lceil u} \times V_{G\lceil u})$$

$$G\lceil u = (V_{G\lceil u}, E_{G\lceil u}), \text{ the } \textit{subgraph} \text{ of } u \text{ in } G$$

With the above notation follows for a conflict graph G that $C_G = \overline{V_{G\lceil \Box}^0}$.

Lemma 2.2.2 Let F be a CNF formula. If $G = (V, E)$ is a conflict graph for F then $F \equiv F \cup \{C_G\}$.

PROOF Let F and G be as above, put $\alpha = \{ (x, \epsilon) \mid x^{1-\epsilon} \in C_G \}$ and let x be the conflict variable of G.

Let β be an assignment with $\beta \supseteq \alpha$ and $\beta \vDash F$. It is shown by induction on $Depth_G(v)$ that $\beta(v) = 1$ for each $v \in V$, $v \neq \Box$. But since $x \in V$ and $\neg x \in V$ such an assignment β can not exist and thus $F|_\alpha$ is unsatisfiable.

For the induction basis let $v \in V$ be a vertex with $Depth_G(v) = 0$. Then $\beta(v) = \alpha(v) = 1$ by the definition of α_G.

For the induction step let $Depth_G(v) = n + 1$. By induction it holds $\beta(u) = 1$ for every predecessor u of v. By the definition of a conflict graph there is a clause $C_v = \{v\} \cup \{ \bar{l} \mid (l, v) \in E \} \in F$ and since $\beta \vDash F$, $\beta(v) = 1$.

That completes the induction and proves that $F|_\beta = 0$ for every assignment $\beta \supseteq \alpha$. From Lemma 1.2.7 it follows hence that $F|_\alpha$ is unsatisfiable. But $C_G = C_\alpha$ and by Lemma 2.2.1 $F \equiv F \cup \{C_G\}$.   ∎

Apart from the conflict clauses, there is another type of clauses C with $F \equiv F \cup \{C\}$ that can be learned from a conflict graph G for the CNF F: if an assignment satisfies all the leaves $l_1, \ldots, l_k$ from which an inner node v is reachable then v must be satisfied in order to satisfy F. This fact can be coded in the clause $C = \{v, \bar{l}_1, \ldots, \bar{l}_k\}$.

Definition Let $G = (V, E)$ be a conflict graph or a UP-graph and let $u \in V - (V_G^0 \cup \{\Box\})$. The *induced clause* of $G\lceil_u$ is $C_{G\lceil_u} = \overline{V_{G\lceil_u}^0} \cup \{u\}$.

Lemma 2.2.3 Let F be a CNF formula, $G = (V, E)$ a UP-graph for F and $u \in V - (V_G^0 \cup \{\Box\})$. Then $F \equiv F \cup \{C_{G\lceil_u}\}$.

PROOF Let $\alpha = \{ (x, \epsilon) \mid x^{1-\epsilon} \in C_{G\lceil_u} \}$. Since $C_{G\lceil_u} = C_\alpha$ and by Lemma 2.2.1, it suffices to show that $F|_\alpha$ is unsatisfiable.

Let $\{l_1, \ldots, l_k\}$ be the set of leaves of G from which u is reachable, $u = x^\epsilon$ and $\alpha' = \alpha - \{(x, 1 - \epsilon)\}$. Let $\beta \supseteq \alpha'$ be an assignment that satisfies F. Then it can be shown analogous to the proof of Lemma 2.2.2 that $\beta(v) = 1$ for every predecessor v of a l_i. But since $\alpha(u) = 0$, it follows from Lemma 1.2.7 that $F|_\alpha$ is unsatisfiable.   ∎

The following algorithm schema DLL-L-UP is a modification of the schema DLL. In addition to the input formula and the actual assignment it receives a UP-graph as third argument and in addition to a satisfying assignment or UNSAT, it returns a modified formula that might include learned clauses, i.e., if F is a CNF formula, G is a UP-graph for F and α is an assignment then DLL-L-UP(F, G, α) returns (F', α') for a formula $F' \supseteq F$ such that $F' \equiv F$ and α is an partial satisfying assignment for F or UNSAT.

If a unit clause $C|_\alpha = \{x^\epsilon\}$ occurs in the restricted formula $F|_\alpha$ during a call of DLL-L-UP(F, G, α) then no branching variable is chosen but x is set to ϵ in order to satisfy $\{x^\epsilon\}$ and this information is coded in the UP-graph G for future recursive calls by edges from vertices of the other literals in C to x^ϵ.

If there is no unit clause in $F|_\alpha$ then a branching variable x is chosen and set to a value ϵ. That is represented in the UP-graph by adding a new leaf x^ϵ.

Either DLL-L-UP finally finds a satisfying assignment on the search path or it runs into conflict, i.e., there are two conflicting unit clauses $\{x\}$ and $\{\neg x\}$ in the restricted formula $F|_\alpha$. In that case the UP-graph G is expanded to a conflict graph G' and a set of clauses $\mathcal{C}$ is learned from G' and added to F.

To derive the results of Section 3.5, the choice of $\mathcal{C}$ is restricted to a set of clauses that corresponds to a *compatible set of subgraphs* of G. It is possible that there are generalizations of this restriction that lead also to the results in Section 3.5 and it is planned to study these generalizations in a future paper with Sam Buss. Nevertheless, the notion of compatible subgraphs includes nearly all learning strategies that are used in algorithms like GRASP.

A *proper sub conflict graph* of G' is a subgraph $H = (V, E)$ of G' that is a conflict graph such that there is no path from a $u \in V - V_H^0$ to a $v \in V_H^0$ in G'.

Definition (Proper Sub Conflict Graph) Let F be a CNF formula, $G = (V, E)$ a conflict graph for F and $H = (V', E')$ a subgraph of G.

H is called *sub conflict graph* of G if H is a conflict graph for F.

H is *proper* iff $V' \cap V_{G\lceil_u} \subseteq V_H^0$ for every $u \in V_H^0$.

If H is a proper sub conflict graph of G then V_H^0 is called a *proper cut* in G.

The definition of a proper sub conflict graph generalizes the concept of a *cut* in a conflict graph (see Figure 2.2) that was introduced in [32] and that is used in all well known SAT solvers to select sub conflict graphs.
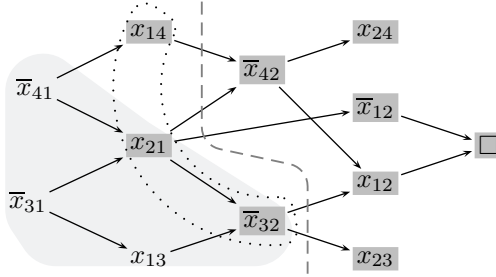

Figure 2.2: The conflict graph G for OP_4 with a proper sub conflict graph and the corresponding cut. The conflict clause of the subgraph is $C = (\bar{x}_{14} \lor \bar{x}_{21} \lor x_{32})$. It is possible to include G and the UP-graph $G\lceil_{\bar{x}_{32}}$ in a compatible set of subgraphs and to learn the clauses C and $\{\bar{x}_{32}, x_{41}, x_{31}\}$.

Definition (Compatible Subgraphs) Let G be a conflict graph. A set $\mathcal{C}$ of subgraphs of G is *compatible* if $\mathcal{C} = \varnothing$ or if $\mathcal{C} = \{H_1, \dots H_s\} \cup \{G\lceil_{v_1}, \dots, G\lceil_{v_t}\}$ and

- H_s is a sub conflict graph of G

- H_i is a proper sub conflict graph of H_{i+1} for every $1 \leq i < s$

- $|V_{H_1 \lceil_\square}| > 3$

- $v_j \in V_{H_s}^0 - V_G^0$ for every $1 \leq j \leq t$

The *conflict clauses* of $\mathcal{C}$ are $CC(\mathcal{C}) = \{ C_H \mid H \in \mathcal{C} \}$.

The clauses $C_{G\lceil_u}$ that are learned from the sub UP-graphs $G\lceil_u$ can be seen as a generalization of the concept of *unique implication points* that has been introduced in [32].

Algorithm 2.2.4 (DLL with Learning by Unit Propagation)

```
DLL-L-UP(F, (V, E), α)
1       if F|α = 1 then return (F, α)
2       if there is a y ∈ Var(F) with {y}, {¬y} ∈ F|α then
3           V ← V ∪ {y, ¬y, □}
4           choose C₁, C₂ ∈ F with C₁|α = {y} and C₂|α = {¬y}
5           N ← { (v̄, y) | v ∈ C₁, v ≠ y } ∪ { (v̄, ¬y) | v ∈ C₂, v ≠ ¬y }
6           E ← E ∪ {(y, □), (¬y, □)}) ∪ N
7           if N ≠ ∅ then choose a compatible set C
8                   of subgraphs of (V, E)
9                       F ← F ∪ CC(C)                         -- learn clauses
10          return (F, UNSAT)
11      if there is a unit clause {xᵋ} ∈ F|α then
12          choose D ∈ F with D|α = {xᵋ}
13          V ← V ∪ {xᵋ}
14          E ← E ∪ { (v̄, xᵋ) | v ∈ D, v ≠ xᵋ }
15          return DLL-L-UP(F, (V, E), α ∪ {(x, ε)})
16      choose x ∈ Var(F|α) and ε ∈ {0, 1}
17      (G, β) ← DLL-L-UP(F, (V ∪ {xᵋ}, E), α ∪ {(x, ε)})
18      if β ≠ UNSAT then return (G, β)
19      if G|α = 0 then return (G, UNSAT)              -- fast backtracking
20      else return DLL-L-UP(G, (V ∪ {x^{1-ε}}, E), α ∪ {(x, 1 - ε)})
```

Note that this algorithm schema is not a generalization of schema DLL because DLL-L-UP is not able to branch on arbitrary variables since unit propagation always takes priority.

Algorithms that match the schema DLL-L-UP can differ in the choice of the branching variable, the choice of the unit clauses in line 4 and line 12 and the selection of the compatible subgraphs in line 7. The running time of the algorithms depend highly on these choices and there are some examples given below of how well known SAT solvers like GRASP, zChaff and MiniSAT take these decisions.

The next branching variable can be a variable that occurs in a short clause or a variable that occurs in many clauses. Some SAT solvers make the choice with

respect to the number of conflicts in which the variable has occurred so far. For an overview on branching heuristics see for example [28].

It appears that SAT solvers like Zchaff that use learning by unit propagation select the unit clauses in lines 3 and 12 in an arbitrary way that depends on the data structures.

A choice of the compatible subgraphs $\mathcal{C}$ in line 8 could be for example just $\{G\}$. That strategy is sometimes called *decision* since the corresponding conflict clause contains only variables that have been chosen as branching variables (or decision variables) in the DLL algorithm. But often it is more convenient to select subgraphs that lead to smaller conflict clauses. Most of the strategies that try to achieve that goal, such as those used in Zchaff and GRASP, compute so called unique implication points, which are cutting vertices in the conflict graph, to reduce the size of the learned clauses. On the other hand it has also turned out in practice that it is not the best strategy to learn only small conflict clauses. An overview on learning heuristics for DLL algorithms is given by Zhang et. al. in [34].

For the decision whether to keep a learned clause it seems to be the a good strategy to include mostly short clauses and to delete clauses that have not contributed to conflicts for a longer time. A similar strategy is for example used by Zchaff.

The last goal of this section is to show that algorithms that match DLL-L-UP are complete and correct algorithms for SAT. The next lemma states that $F|_\alpha$ is unsatisfiable if DLL-L-UP(F, H, α) returns $(G, \texttt{UNSAT})$. It follows easily from results in Chapter 3 and is hence proved there.

Lemma 2.2.5 Let F be a CNF formula, H a UP-graph for F and α an assignment. If DLL-L-UP(F, H, α_H) returns $(F', \texttt{UNSAT})$ then $F|_{\alpha_H}$ is unsatisfiable.

PROOF See Section 3.5. ∎

With the help of Lemma 2.2.5 it is easy to proof the following theorem.

Theorem 2.2.6 (Correctness of DLL-L-UP) Let F be a CNF formula. Then every execution of DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ terminates. It returns (F', α) for a partial satisfying assignment α and a formula $F' \supseteq F$ if F is satisfiable and $(F', \texttt{UNSAT})$ otherwise.

PROOF It is evident that every execution of DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ terminates and by Lemma 2.2.5 it follows that F is unsatisfiable if the algorithm returns $(F', \texttt{UNSAT})$.

Thus it suffices to show that that $F|_{\alpha'} = 1$ if DLL-L-UP(F, H, α) returns (F', α') for an assignment α.

It is shown to this end by induction on the number of recursive calls s that $F' \equiv F$ if DLL-L-UP(F, H, α) returns (F', α') and $F|_{\alpha'} = 1$ if $\alpha' \neq$ UNSAT.

Induction basis: If there are no recursive calls and $\alpha' \neq$ UNSAT then $F' = F$, $\alpha = \alpha'$ and $F|_\alpha = 1$.

If there are no recursive calls and $\alpha' =$ UNSAT then $F' = F \cup CC(\mathcal{C})$ for a compatible set of subgraphs $\mathcal{C}$ of the confilct graph (V, E) for F. Then it follows $F \equiv F \cup C_{H'}$ for every $H' \in \mathcal{C}$ from Lemma 2.2.2 and Lemma 2.2.3. Thus $F \equiv F'$.

Induction step: Let DLL-L-UP(F, H, α) perform $s > 0$ recursive calls. If (F', α') is the return value of a recursive call in line 15 or in line 17 then the statement follows by induction since these executions perform less recursive calls.

Otherwise DLL-L-UP(F, H, α) performs two recursive calls. It follows by induction that $F \equiv G$ for the return value (G, β) of the first recursive call in line 17 and the return value (F', α') of the second recursive call is returned by DLL-L-UP(F, H, α). By induction $G \equiv F'$ and $G|_{\alpha'} = 1$ if $\alpha' \neq$ UNSAT. But since $G \equiv F$, also $F' \equiv F$ and $F|_{\alpha'} = 1$ if $\alpha' \neq$ UNSAT. ∎

SAT competitions and other experimental results (see for example [32] and [9]) have shown that DLL algorithms that use learning by unit propagation outperform basic DLL algorithms widely. In the following chapters a theoretical approval for that observation is given. It is proved in Section 3.5 that executions of DLL-L-UP can be transformed to proofs of a proof system that is exponentially stronger than the basic DLL algorithm. In [2] it was shown that there exist formulas F_i for $i \in \mathbb{N}$ such that every basic DLL algorithm needs exponential time and in contrast there is an unit propagation learning heuristic such that DLL-LEARN(F_i) runs in polynomial time. Section 4.2 presents a similar result.

Learning by unit propagation can be extended by the possibility of doing so-called *restarts* during an execution of DLL-L-UP. This is based on the observation that the opportunity to learn a clause depends on two factors: on the clauses that are learned so far and on the current assignment that falsifies the input formula. Thus different choices of the branching variables result in different opportunities to learn clauses.

To enhance the capabilities to learn clauses it is therefore helpful to run DLL-L-UP several times partially with different branching variable orders and save the learned clauses after each call.

It has been shown by experiments (see [24] and references therein) that restarts can highly improve the running time of SAT algorithms. Hence restarts are used by the fastest complete SAT solvers that are based on DLL. Zchaff for example restarts constantly after some interval.

To date, no examples have been found that demonstrate the superiority of DLL with learning and restarts, i.e., there are no formulas known on which every algorithm that matches the schema DLL-L-UP needs super-polynomial time while algorithms that use restarts are able to decide the satisfiability of those formulas in polynomial time.

It has be shown in [2] that DLL with learning by unit propagation and restarts can simulate the proof system dag-like resolution that is introduced in Chapter 4.

2.3 A Generalization of Learning by Unit Propagation

In this section a new natural generalization of DLL with learning by unit propagation is presented by means of the algorithm schema DLL-LEARN.

There are two main extensions with respect to learning by unit propagation. On the one hand the possibility of learning is separated from the need to do unit propagation, which means that the algorithm can do unit propagation as part of its branching heuristic or not, while being able to learn clauses in both cases.

On the other hand DLL-LEARN uses more of the information that is provided during an execution of a DLL algorithm. If $\mathrm{DLL}(F, \alpha)$ branches on a variable x and performs two recursive calls that both return UNSAT then it follows that $F|_\alpha$ is unsatisfiable. This information is not used when doing learning by unit propagation.

The idea of DLL-LEARN is to extend DLL to provide a small clause $C_{DLL(F,\alpha)}$ with $C_{DLL(F,\alpha)}|_\alpha = 0$ and $F \equiv F \cup \{C\}$ for every call $\mathrm{DLL}(F, \alpha)$.

If $F|_\alpha = 0$ then $C_{DLL(F,\alpha)}$ is simply a clause $C \in F$ with $C|_\alpha = 0$. If $F|_\alpha \neq 0$ and $F|_\alpha \neq 1$ then a branching variable x is chosen. If there is no fast backtracking then the $\mathrm{DLL\text{-}LEARN}(F, \alpha)$ performs two recursive calls and the clause $C_{DLL(F,\alpha)} = (C_0 - \{x^{1-\epsilon}\}) \cup (C_1 - \{x^\epsilon\})$ is learned whereby $C_0 \subseteq C_{\alpha \cup \{(x,0)\}}$ and $C_1 \subseteq C_{\alpha \cup \{(x,1)\}}$ are the clauses that are provided by the recursive calls.

Since x does not have to occur in $Var(C_0)$ or in $Var(C_1)$, C arises out of C_0 and C_1 by means of a *weak resolution rule*.

A feature of the algorithm that the reader might find strange is that it can continue to branch on variables even if the formula is already unsatisfied. This feature is called *continued learning* and is needed for a direct proof of the results in Section 3.4.

Besides it could be also helpful in an implementation of the algorithm: Think of a call of $\mathrm{DLL}(F, \alpha)$ such that $F|_\alpha = 0$ and suppose that all of the falsified clauses $C \in F$ contain nearly all variables that are set by α and recall that one

wants to provide a small clause $C_{DLL(F,\alpha)}$. It might, for example, be the case that $F|_\alpha$ contains two conflicting unit clauses $C_0|_\alpha = \{x\}$ and $C_1|_\alpha = \{\neg x\}$ such that C_0 and C_1 are small. In that case, and in similar cases, it could be better to set the variable x and start the learning process with the clauses C_0 and C_1. Therefore DLL-LEARN has the possibility to continue to chose branching variables even if the formula is already falsified by the current assignment. For the same reasons it is also possible to skip fast backtracking.

Algorithm 2.3.1 (DLL with Learning)

```
DLL-LEARN(F, α)
  1      if  F|α = 1 then return (F, α)
  2      if  F|α = 0 then do optionally              -- cont. learning?
  3          tag a  C ∈ F with C|α = 0 as new
  4          return (F, UNSAT)
  5      if  Var(F) − dom(α) = ∅ then                -- F|α = 0
  6          tag a  C ∈ F with C|α = 0 as new
  7          return (F, UNSAT)
  8      choose x ∈ Var(F) − dom(α) and a value ε ∈ {0,1}
  9      (G, β)←DLL-LEARN(F, α ∪ {(x, ε)})
 10      if  β ≠ UNSAT then return (G, β)
 11      if  G|α = 0 then optionally return (G, UNSAT)   -- fast backtracking
 12      (H, γ)←DLL-LEARN(G, α ∪ {(x, 1 − ε)})
 13      if  γ ≠ UNSAT then return (H, γ)
 14      select the newest C₀ ∈ G and the newest C₁ ∈ H
 15      C ← (C₀ − {x^{1−ε}}) ∪ (C₁ − {x^ε})
 16      H ← H ∪ {C}                                 -- learn a clause
 17      if  C₀ ∉ F then do optionally H ← H − {C₀}   -- keep clauses?
 18      if  C₁ ∉ F then do optionally H ← H − {C₁}
 19      return (H, UNSAT)
```

In the above algorithm the clause $C_{DLL(F,\alpha)}$ that is learned in line 16 in an execution of DLL-LEARN(F,α) is always included in F until it has been used to derive the clause $C_{DLL(F,\alpha')}$ that is learned on the next higher level in the recursion tree during the execution of DLL-LEARN(F,α') with $\alpha' = \alpha - \{(x,\epsilon)\}$ that has called DLL-LEARN(F,α) in line 9 or in line 12. Thereafter one can decide to keep it or to delete it from F in the lines 17 and 18.

Since the maximal height of the recursion tree is $n = |Var(F)|$ there are at most n learned clauses $C_{DLL(F,\alpha)}$ that have to be stored to derive other clauses in an execution of DLL-LEARN.

The algorithm schema DLL-LEARN is a generalization of the basic DLL algorithm schema DLL since every execution of DLL can be simulated by DLL-LEARN in the following way. Chose in DLL-LEARN the same branching variables as in DLL, always delete clauses in the lines 16 and 17 and never do fast backtracking or continued learning.

It is proved in Section 3.5 that DLL-LEARN can also simulate DLL-L-UP, i.e.,

if there is an execution of DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ that performs s recursive calls then there is an execution of DLL-LEARN$(F, \varnothing)$ that does at most $2n(s+2)$ recursive calls whereby $n = \min\{|Var(F)|, s\}$.

The soundness and completeness of DLL-LEARN are proved in Section 3.4.

Chapter 3

Tree-Like Resolution

3.1 Resolution Trees

Proof systems that are based on *resolution* are some of the best known and most studied in the literature. One can use resolution directly to prove a given CNF formula to be unsatisfiable. Thus resolution proofs are often called *resolution refutations*. But since a formula F is tautological if and only if the CNF formula $CNF(\neg F)$ is unsatisfiable, one can consider a resolution refutation for $CNF(\neg F)$ as a proof for $\vDash F$ and therefore resolution as a proof system in terms of Section 1.3.

Resolution refutations are based one a single inference rule which is called *resolution rule*.

Definition (Resolution Rule) Let C_0, C_1 and C be clauses and let x be a variable. C can be obtained by the *resolution rule* from C_0 and C_1 according to x if $x \in C_0$, $\neg x \in C_1$ and $C = (C_o - \{x\}) \cup (C_1 - \{\neg x\})$.

In that case we write $C_0, C_1 \vdash_x C$ or $C_0, C_1 \vdash C$ and call C a *resolvent* of C_0 and C_1.

A resolution refutation of a CNF F consists of repeated application of the resolution rule to the clauses of F and the resulting resolvents such that finally the empty clause $\square$ appears as a resolvent. Because $\vDash (C_0 \wedge C_1) \rightarrow C$ if $C_0, C_1 \vdash C$, it follows $\vDash F \rightarrow 0$ and hence F is unsatisfiable.

The following lemma states the correctness of the resolution rule.

Lemma 3.1.1 Let α be an assignment and let C_0, C_1 and C be clauses with $C_0, C_1 \vdash_x C$. If $C_0|_\alpha = C_1|_\alpha = 1$ then $C|_\alpha = 1$.

PROOF Let $x \in C_0$, $\neg x \in C_1$ and $C = (C_o - \{x\}) \cup (C_1 - \{\neg x\})$. Let α be an assignment with $C_0|_\alpha = C_1|_\alpha = 1$. It holds $x \notin dom(\alpha)$, $\alpha(x) = 0$ or

$\alpha(\neg x) = 0$. Therefore there is a literal $l \in (C_0 \cup C_1) - \{x, \neg x\}$ with $\alpha(l) = 1$. But $(C_0 \cup C_1) - \{x, \neg x\} = C$ and thus $C|_\alpha = 1$.   ∎

To the best of the author's knowledge the first resolution based proof system was suggested 1930 by Herbrand [23] and it should be mentioned that resolution can also be used as a proof system for first order logic due to results that are based on the work of Löwenheim, Herbrand and others (see [31] and [16]). Many first order theorem provers and *logic programming* tools like *Prolog interpreters* are based on resolution.

In the framework of proof complexity, resolution proofs were studied initially by Tseitin [33] and there are different ways of both, representing a resolution proof and measuring its size (see also Chapter 5). A recapitulation of the different types of resolution and their relation has been given by Rachinsky [30].

In this chapter resolution proofs are considered to be trees. This means that every resolvent is used exactly once to derive another resolvent and if a clause C is used k times for a resolution in a proof then it has to be derived k times from F. Proofs in which a resolvent can be used arbitrarily often are called dag-like resolution proofs and discussed in Chapter 4.

A *resolution tree* for a CNF F as defined below is a binary tree such that the leaves are labeled with clauses from F and every inner node is a resolvent of its children.

Definition (Resolution Tree) Let F be a CNF formula, C a clause and T a binary tree in which the nodes are labeled with clauses and the edges are labeled with variables. T is a *resolution tree* for C from F if

- the root of T is labeled with the clause C,

- each leaf of T is a labeled with a clause $D \in F$,

- if the inner node is labeled with D and has two children that are labeled with D_0 and D_1 then $D_0, D_1 \vdash_x D$ for a variable x and the edges between the node and its children are labeled with x.

The *size* $|T|$ of T is the number of nodes in T.

$Var(T) = \{\, x \mid$ there is an edge labeled with x in $T \,\}$ is the set of the variables in T and $Cl(T) = \{\, C \mid C$ is the label of a node in $T\}$ is the *set of clauses* in T.

Remark 3.1.2 A node and its label are often identified in a resolution tree. In that terms a resolution tree T for C from F can be defined as follows. C is the root of T, every leaf in T is a clause in F and $D_0, D_1 \vdash_x D$ if D is the parent of D_0 and D_1.
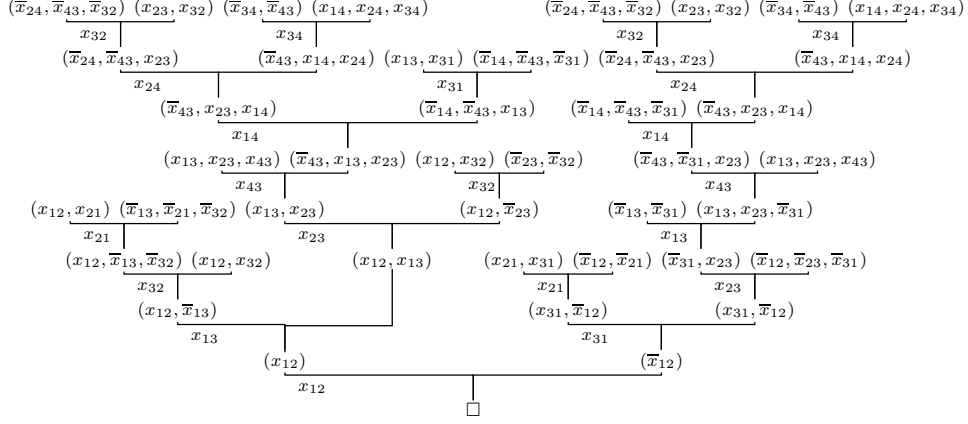
Figure 3.1: A resolution tree of $\square$ from $\mathrm{OP}_4 \wedge (x_{21} \vee x_{31}) \wedge (x_{12} \vee x_{32})$ the additional clauses can be resolved from OP_4 analogously to $(x_{13} \vee x_{23})$.

The above definition of the size $|T|$ of a resolution tree T for the CNF F does not depend on the size of clause labels in the tree. Even though, that is a little imprecise, it suffices since polynomial factors are of less interest in this paper and the clause size is bounded by $|Var(F)|$ since $Var(Cl(T)) \subseteq Var(F)$. Thus every clause label in T can be stored in space $O(|Var(F)|)$.

Note also that the size of the clauses in $Cl(T)$ and the size of T itself are highly related, i.e., T is large if and only if T contains large clauses. This result applies for dag-like and tree-like resolution and details can be found in [6].

As mentioned above, resolution trees can be regarded as a proof system in the following way.

Definition (The Proof System RT) Let F be a formula. The proof system RT is defined through: $\mathrm{RT}(F, T)$ if and only if T is a resolution tree for $\square$ from $CNF(\neg F)$.

The next lemma shows the soundness of RT. The completeness of RT is shown in Section 3.2. Since one can verify in polynomial time whether $C_0, C_1 \vdash C$ for clause C_0, C_1 and C, it is also computable in time $p(|T| + |F|)$ for a polynomial p if a tree T is a resolution tree for $\square$ from a given CNF formula F. Thus, and since $CNF(\neg F)$ is computable in polynomial time, RT is a well defined proof system.

Lemma 3.1.3 Let F be a CNF formula and let T be a resolution tree for the clause C from F. Then $C|_\alpha = 1$ for every assignment α with $F|_\alpha = 1$.

PROOF The statement is proved by induction on $d = Depth_T(C)$.

If C is a leaf then $C \in F$ and $C|_\alpha = 1$ follows from $F|_\alpha = 1$ by definition.

If $d > 0$ then C has two children C_0 and C_1 with $C_0, C_1 \vdash_x D$ for a variable x. Since $Depth_T(C_0) < d$ and $Depth_T(C_1) < d$, it follows by induction that $C_0|_\alpha = C_1|_\alpha = 1$. But then $C|_\alpha = 1$ by Lemma 3.1.1.   ∎

Corollary 3.1.4 (Soundness of `RT`) Let F be a CNF formula and let T be a resolution tree for $\square$ from F. Then F is unsatisfiable.

PROOF Suppose for the purpose of contradiction that α is a satisfying assignment for F. Then $F|_\alpha = 1$ and, by Lemma 3.1.3, $\square|_\alpha = 1$ which contradicts the definition of $\square$.   ∎

3.2   DLL Algorithms and Regular Resolution Trees

There is a strong connection between resolution and DLL algorithms that has been well known already as the first DLL algorithm has been presented by Davis, Logemann and Loveland in [15]. In fact DLL has been introduced there as a variation of the resolution based proof search presented in [16].

Particularly a run DLL can be considered as a resolution tree with an additional property: On every path from a leaf to the root there is for each variable x at most one edge that is labeled with x, i.e., one resolves at most once on a variable per path. A resolution tree with this property is called *regular*. A proof system based on regular resolution proofs was considered first in terms of complexity by Tseitin in [33].

Definition (Regular Resolution Tree) Let T be a resolution tree. T is called *x-regular* if every path from the root to a leaf contains at most one edge labeled with x. T is called *regular* if T is x-regular for every variable x.

Definition (`regRT`) Let F be a formula. The proof system `regRT` is defined through: `regRT`(F, T) if and only if T is a regular resolution tree for $\square$ from $CNF(\neg F)$.

The resolution tree in Figure 3.1 on the preceding page is regular.

Since it is easy to check with a depth first search in polynomial time whether a resolution tree is regular, the relation `regRT`(F, T) can be decided in polynomial time. The soundness of `regRT` follows from Lemma 3.1.3 and the completeness is shown below. Thus `regRT` is a proof system.

The next proposition shows that a resolution tree can be transformed into a regular resolution tree without any increase of the tree-size.

Lemma 3.2.1 If there exists a resolution tree of size s for $\square$ from F then there is also a regular resolution tree of size at most s for $\square$ from F.

PROOF Let F be a CNF formula and let T be a resolution tree of size s for $\square$ from F. Let $x \in Var(T)$ (if $Var(T) = \varnothing$ then T is already regular). It suffices to show by induction that T can be transformed into a x-regular tree T_x of size at most s such that T_x is y-regular for each variable $y \neq x$ for which T is y-regular . Therefore let d be the number of edges e that are labeled with x such that there is another edge labeled with x on the path from e to $\square$.

If $d = 0$ then T is already x-regular.

Let $d > 0$. Then there is an edge e in T that is labeled with x such that there is another edge labeled with x on the path from e to $\square$. Let e correspond to the resolution inference $C_0, D_0 \vdash_x C_1$ and let $C_1, C_2, \ldots, C_n = \square$ be the sequence of clauses that appear on the path from e to $\square$. Let furthermore e' be the first edge on the path that is labeled with x and let e' correspond to the resolution inference $C_k, D_k \vdash_x C_{k+1}$.

T is now transformed into a resolution tree T' for $\square$ from F that does not contain the inference $C_0, D_0 \vdash_x C_1$ anymore. Assume therefore w.l.o.g. that $C_1 = (C_0 - \{x\}) \cup (D_0 - \{\neg x\})$ and $C_{k+1} = (C_k - \{x\}) \cup (D_k - \{\neg x\})$ (the other cases are symmetric). To start with, remove D_0 and all its predecessors (i.e., its derivation) from T and substitute the subtree T_{C_0} of T with root C_0 (i.e., the derivation of C_0) for C_1.

To make T' a valid resolution tree, the clauses C_i are either deleted or replaced by a clause C'_{i+1} for $i = 1, \ldots, n$ such that $C'_{i+1} \subseteq C_{i+1} \cup \{x\}$ for $i < k$ and $C'_{i+1} \subseteq C_{i+1}$ for $i \geq k$ as follows. Let $C_i, D_i \vdash_{x_i} C_{i+1}$ be the resolution inference of C_{i+1} in T and $C_{i+1} = (C_i - \{x_i^\epsilon\}) \cup (D_i - \{x_i^{1-\epsilon}\})$. If $x_i^\epsilon \in C'_i$ then let $C'_{i+1} = (C'_i - \{x_i^\epsilon\}) \cup (D_i - \{x_i^{1-\epsilon}\}$ and replace C_{i+1} by C'_{i+1}. Else let $C'_{i+1} = C'_i$, delete D_i together with its derivation (i.e., the subtree T_{D_i} of T) and replace C_{i+1} by the subtree $T'_{C'_i}$ of the modified tree T'.

The resulting tree T' is a resolution tree. Since at least two clauses have been deleted from T, $s' = |T'| < s$ and by induction T' can be transformed into a regular resolution tree for $\square$ from F of size at most s'. ∎

Theorem 3.2.2 follows immediately from Lemma 3.2.1.

Theorem 3.2.2 `regRT ≡ RT`

The following lemma shows how a regular resolution tree can be constructed from an execution of DLL.

Lemma 3.2.3 Let F be a CNF formula and α an assignment. If there is an execution of $\text{DLL}(F, \alpha)$ that returns `UNSAT` and performs s recursive calls then

there exists a clause C with $C|_\alpha = 0$ such that C has a regular resolution tree T from F with $|T| \leq s + 1$ and $Var(T) \cap dom(\alpha) = \varnothing$.

PROOF By induction on s. Fix an execution of $DLL(F, \alpha)$ that returns UNSAT after s recursive calls. Recall that by Lemma 2.1.3 s is even.

If $s = 0$ then the algorithm performs no recursive calls and since it returns UNSAT there must be a clause C in F with $C|_\alpha = 0$. Thus the resolution tree that consists only of a root labeled with C satisfies the conditions.

For the induction step assume that $s = r + 2$ for an even integer r. Since $DLL(F, \alpha)$ does not terminate within the first 4 lines, there is a branching variable x which is chosen in line 5. Because the algorithm returns UNSAT there is a call of $DLL(F, \alpha_0)$ in line 6 that execute s_0 recursive calls and a call of $DLL(F, \alpha_1)$ in line 10 that execute s_1 recursive calls, whereby $\alpha_i = \alpha \cup \{(x, i)\}$ and $s = s_0 + s_1 + 2$. Since both of the calls return UNSAT the induction hypothesis states that there are clauses C_0, C_1 such that $C_i|_{\alpha_i} = 0$ and C_i has a regular resolution tree T_i with $|T_i| \leq s_i + 1$ and $Var(T_i) \cap dom(\alpha_i) = \varnothing$ for $i = 0$ and $i = 1$. If $x \notin C_0$ or $\neg x \notin C_1$ one can put $T = T_0$ or $T = T_1$, respectively and it follows immediately that T is as stated in the lemma. Otherwise note that T_0 and T_1 do not contain edges labeled with x. Hence a regular resolution tree T can be derived from T_0 and T_1 by connecting the roots of T_0 and T_1 with a the new root $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ and labeling the edges with $\{x\}$. By choice $\neg x \notin C_0$ and $x \notin C_1$. So $C \subseteq (C_1 \cup C_2) - \{x, \neg x\}$ and therefore $C|_\alpha = 0$. Since $|T| = |T_0| + |T_1| + 1 \leq (s_0 + 1) + (s_1 + 1) + 1 = s + 1$, T is a regular resolution tree as wanted. ∎

The other direction of Lemma 3.2.3 holds, too, i.e., a given regular resolution tree can be transformed directly in a run of DLL. While Lemma 3.2.3 appears often in literature (for example in [5]) Lemma 3.2.4 has not been seen by the author yet but is well-known among experts. The idea of the proof is to use the structure of the resolution tree as a variable branching heuristic in DLL.

Lemma 3.2.4 Let F be a CNF formula and let C be a clause that has a regular resolution tree T of size s from F. Let α be an assignment with $C|_\alpha = 0$ and $Var(T) \cap dom(\alpha) = \varnothing$. Then there is an execution of $DLL(F, \alpha)$, that returns UNSAT after at most $s - 1$ recursive calls.

PROOF The lemma is proved by induction on s. Note therefore that every resolution tree has an odd number of nodes.

Let for the induction basis $s = 1$. Then by definition $C \in F$ and $Var(T) = \varnothing$. Let now α be an assignment with $C|_\alpha = 0$. Then $F|_\alpha = 0$ and $DLL(F, \alpha)$ terminates without any recursive calls and returns UNSAT in line 2.

Suppose for the induction step that that T has size $s + 2$ for an odd integer s. Removing the root C from T generates two sub trees T_0, T_1 of T with

$|T_0|+|T_1|+1 = |T|$ and roots C_0, C_1, respectively. Let x be the label of the edges between C and C_0, C_1 and assume w.l.o.g. that $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ (otherwise swap T_0 and T_1). Suppose that α is an assignment with $C|_\alpha = 0$ and $Var(T) \cap dom(\alpha) = \varnothing$. Consider a call of $\mathrm{DLL}(F, \alpha)$. Since $x \in Var(T)$ it follows $x \notin dom(\alpha)$.
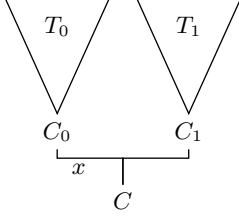
If $D|_\alpha = 0$ for a clause $D \in F$ then $\mathrm{DLL}(F, \alpha)$ returns `UNSAT` in line 2 without any recursive calls and the statement holds.

If $F|_\alpha \neq 0$ then it is shown by contradiction that $x \in Var(F|_\alpha)$. Suppose therefore $x \notin Var(F|_\alpha)$. Then every clause D with $x \in Var(D)$ is satisfied by α. But it follows that $x \in Var(C_0)$ and thus there is a leaf L in T that contains x and is therefore satisfied by α. Let $l \in L$ be the literal that is satisfied by α. Since $Var(T) \cap dom(\alpha) = \varnothing$ it follows $l \in C$ which leads to a contradiction because $C|_\alpha = 0$. Thus $x \in Var(F|_\alpha)$ and it is possible to chose x in line 5 of the DLL algorithm.

The algorithm calls $\mathrm{DLL}(F, \alpha_0)$ in line 6, with $\alpha_0 = \alpha \cup \{(x, 0)\}$. By the definition of resolution trees $C_0, C_1 \vdash_1 C$ and therefore (w.l.o.g.) $C_0 \subseteq C \cup \{x\}$ and $C_1 \subseteq C \cup \{\neg x\}$. Hence it follows that $C_0|_{\alpha_0} = 0$. Since T is regular, $x \notin Var(T_i)$ and $Var(T_i) \cap dom(\alpha) = \varnothing$ hold. So it follows by induction that there is an execution of $\mathrm{DLL}(F, \alpha_0)$ that performs less than $|T_0| - 1$ recursive calls and returns `UNSAT`.

Therefore $\mathrm{DLL}(F, \alpha_1)$ is called in line 6 with $\alpha_1 = \alpha \cup \{(x, 1)\}$. Again it follows by induction that there is an execution of $\mathrm{DLL}(F, \alpha_1)$ that performs less than $|T_1| - 1$ recursive calls and returns `UNSAT`. Therefore there exists an execution of of $\mathrm{DLL}(F, \alpha)$ that makes overall $2 + |T_0| - 1 + |T_1| - 1 = s + 1$ recursive calls and returns `UNSAT` in line 10. ∎

Overall it has been proved that the algorithm schema DLL corresponds exactly to the proof system `RT` is the following way.

Theorem 3.2.5 If F is a CNF formula then there is an execution of $\mathrm{DLL}(F, \varnothing)$ that executes less than s recursive calls if and only if there exists a regular resolution tree for $\square$ of size less than $s + 1$ from F.

PROOF "$\Leftarrow$": Follows from Lemma 3.2.4 with $C = \square$ and $\alpha = \varnothing$.
"$\Rightarrow$": Because $\square$ is the only clause C with $C|_\varnothing = 0$ this is a direct conclusion from Lemma 3.2.3 applied to $\alpha = \varnothing$. ∎

Corollary 3.2.6 (Completeness of `regRT`) For every unsatisfiable CNF formula F with $n = |Var(F)|$ there is a regular resolution tree of size at most 2^n for $\square$ from F.

PROOF Let F be an unsatisfiable CNF formula. Then $DLL(F, \varnothing)$ returns
UNSAT by Theorem 2.1.2. Thus there is a regular resolution tree T for $\square$ from
F by Theorem 3.2.5. It follows directly from the definition of regularity that
$|T| \leq 2^n$.   ∎

Since regRT is complete, RT is also complete.

The idea of the proof of Lemma 3.2.3 can be used to define a version of the
DLL algorithm schema that returns either a satisfying assignment or are regular
resolution tree for $\square$ from the input formula.

Algorithm 3.2.7

```
DLL⁺(F, α)
1       if  F|α = 0 then return a  C ∈ F with  C|α = 0
2       if  F|α = 1 then return α
3       choose  x ∈ Var(F|α)
4       β ←  DLL⁺(F, α ∪ {(x,0)})
5       if  β is an assignment then return  β
6       else
7            δ ←  DLL⁺(F, α ∪ {(x,1)})
8            if  δ is an assignment then return  δ
9            let  C₀ be the root of  β and  C₁ the root of  δ
10           if  C₀|α = 0 then return  β
11           if  C₁|α = 0 then return  δ
12           return the tree  T that arises out of  β and  δ by adding
13           a new root  C = (C₀ − {x}) ∪ (C₁ − {¬x}) and an edge from
14           C to  Cᵢ that is labeled with  {x} for  i = 0 and  i = 1
```

Note that there is an polynomial time execution of the basic DLL algorithm
$DLL(F)$ if and only if there is a polynomial time execution of $DLL^+(F)$.

3.3   Resolution Trees with Lemmas

As mentioned in Section 3.1, a clause in a resolution tree is derived from scratch
every time it is used in a resolution inference i.e., if a clause C is used k times
for resolution in the proof the it has to be derived k times from F.

In this section *resolution trees with lemmas* are defined to remove that short-
coming of resolution trees such that every clause needs only to be derived once
regardless of the number of resolution inferences it is used in. The idea is, to
have the possibility to reuse clauses that appear "earlier" in the proof.

Thereby a resolution tree for a formula F can be visualized as a proof that is
generated step by step from left to right. One starts with clauses from F as
leaves and then builds the tree from left to right using the resolution rule. If
a clause C that has been already derived and is needed for another resolution

inference during the construction then one can include C just as a new leaf instead of giving another proof for C.

Definition Let T be a binary tree. T induces a total order $<_T$ on the nodes V of T as follows. Let v be an inner node of T, let l be a successor of v in the left subtree below v and let r be a successor of v in the right subtree below v. Then $l <_T r <_T v$.

Definition (RTL) Let F be a CNF formula, C a clause and T a binary tree in which the nodes are labeled with clauses and the edges are labeled with variables. Identify the nodes of T with their labels. T is a *resolution tree with lemmas (RTL)* for C from F if

- C is the root of T,

- if D is a leaf in T then $D \in F$ or $D = D'$ for some node D' with $D' <_T D$,

- if D is an inner node and the two children of D are D_0 and D_1 then $D_0, D_1 \vdash_x D$ for a variable x and the edges between D and its children are labeled with x.

The *size* $|T|$ of T is the number of nodes in T and the *regularity* of T, $Var(T)$ and $Cl(T)$ are defined as for resolution trees.

Remark 3.3.1 A node and its label are identified in RTL. So if it is written $C = D$ for some nodes C and D then this usually means that the labels of the nodes are identical, not the nodes. Of course $D <_T C$ always refers to the nodes and $C, C' \vdash_x D$ always refers to the labels of the nodes.

(Regular) resolution trees with lemmas have been considered first by Reinhold Letz [25] and it is shown in Chapter 4 that a RTL is nothing else then a different view on a dag-like resolution proof, i.e., dag-like resolution proofs and resolution trees with lemmas can be transformed into each other with a constant blow up.

Every resolution tree T for C from F is also a RTL for C from F. Thus it is trivial that every CNF formula F that has a resolution tree of size s for a clause C has also a RTL of size s for C.

Proposition 3.3.2 Let F be a CNF formula and C a clause. If C has a (regular) resolution tree of size s from F then C has also a (regular) RTL of size s from F.

Definition (RTL) Let F be a formula. The proof system RTL is defined through: $\mathtt{RTL}(F, T)$ if and only if T is a resolution tree with lemmas for $\square$ from $CNF(\neg F)$.
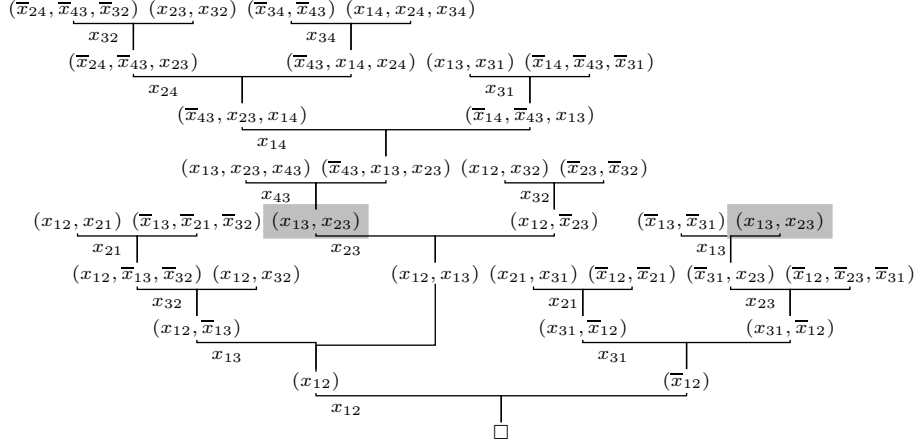
Figure 3.2: A regular RTL of $\square$ from $\mathrm{OP}_4 \wedge (x_{21} \vee x_{31}) \wedge (x_{12} \vee x_{32})$.
Compare it to the regular resolution tree in Figure 3.1. The clause
$(x_{13} \vee x_{23})$ has been used as lemma.

With a similar algorithm as described above for `RT`, `RTL` it is decidable in poly-
nomial time. The completeness of `RTL` follows from Corollary 3.2.6 and the
soundness can be proved analogous to `RT`. Thus `RTL` is a proof system.

Proposition 3.3.3 (Soundness of `RTL`) Let F be a CNF formula and let T
be a RTL for $\square$ from F. Then F is unsatisfiable.

PROOF Analogous to Lemma 3.1.3 and Corollary 3.1.4.   ∎

Corollary 3.3.4 (Completeness of `RTL`) For every unsatisfiable CNF for-
mula F there is a regular resolution tree with lemmas of F for $\square$.

PROOF Since every resolution tree is also a resolution tree with lemmas, the
completeness of `RTL` follows directly form Corollary 3.2.6.   ∎

In Section 3.2 it has been shown that there is a correspondence between DLL
algorithms and regular resolution trees in terms of complexity: There is a poly-
nomial p such that for a given unsatisfiable CNF formula F the difference
between the fastest execution time of $\mathrm{DLL}(F, \alpha)$ and the size of the smallest
RTL for $\square$ from F is not bigger then $p(|F|)$.

In the next section it is shown that there is the same connection between DLL
algorithms with learning and a generalization of regular resolution trees with
lemmas, namely *regular weak resolution trees*.

Regular weak RTL are based on a generalization of the resolution rule that
is called *weak resolution*. It is defined in the same way the resolution rule is

defined except that the assumptions must contain the variable according to which is resolved.

Definition (Weak Resolution Rule) Let C_0, C_1 and C be clauses and let x be a variable. C can be obtained by the *resolution rule* from C_0 and C_1 with respect to x iff $C = (C_o - \{x\}) \cup (C_1 - \{\neg x\})$ or $C = (C_o - \{\neg x\}) \cup (C_1 - \{x\})$. Write $C_0, C_1 \vdash_x^w C$ in that case.

Regardless of the fact that the weak resolution rule is a generalization of the resolution rule it is called weak because it combines the resolution rule with an inference rule that is widely called *weakening rule*.

The weakening rule can be used to infer arbitrary clauses C' with $C' \supseteq C$ from a clause C. In combination with the resolution rule it is possible to simulate a weak resolution inference $C_0, C_1 \vdash_x^w C$ by an application of the resolution rule and two applications of the weakening rule as follows: $C_0 \vdash C_0 \cup \{x\}$, $C_1 \vdash C_1 \cup \{\neg x\}$, $C_0 \cup \{x\}, C_1 \cup \{\neg x\} \vdash_x C$.

Weak RTL differ from RTL only in one way: in a weak RTL the weak resolution rule is used instead of the resolution rule.

Definition (WRTL) Let F be a CNF formula, C a clause and T a binary tree in which the nodes are labeled with clauses and the edges are labeled with variables. Identify the nodes of T with their labels. T is a *weak resolution tree with lemmas (WRTL)* for C from F if

- the root of T is labeled with C,

- if D is a leaf in T then $D \in F$ or $D = D'$ for a node D' with $D' <_T D$,

- if D is an inner node and the two children of D are D_0 and D_1 then $D_0, D_1 \vdash_x^w D$ for a variable x and the edges between D and its children are labeled with x.

$Var(T)$, $Cl(T)$ and the *size* and the *regularity* of T are defined analogously to RTL.

Definition (WRTL and regWRTL) Let F be a formula. The proof system WRTL (regWRTL) is defined through: $\text{WRTL}(F, T)$ ($\text{reg WRTL}(F, T)$) if and only if T is a weak (regular) resolution tree with lemmas for $\Box$ from $CNF(\neg F)$.

The following proposition follows directly from the definition since every resolution inference according the resolution rule is also an inference according to the weak resolution rule.

Proposition 3.3.5 Let F be a CNF formula and C a clause. If C has a (regular) RTL of size s from F then C also has a weak (regular) RTL of size s from F.

For non-regular RTL the other direction of Proposition 3.3.5 holds, too. The proof is is given in Section 4.1.

Theorem 3.3.6 Let F be a CNF formula and C a clause. If C has a WRTL of size s from F then C also has a RTL of size s from F.

PROOF See Theorem 4.1.4 on page 61.   ∎

It is an open question whether Theorem 3.3.6 also holds for regular WRTL or, more exactly, whether there is a polynomial p such that for a regular WRTL for C of size s from F there is always a regular RTL for C from F of size smaller than $p(s)$.

Corollary 3.3.7 `WRTL` and `regWRTL` are sound and complete.

PROOF Follows from Proposition 3.3.5 and Theorem 3.3.6.   ∎

In particular the following lemma holds.

Lemma 3.3.8 Let F be a CNF formula and let T be a WRTL for the clause C from F. Then $C|_\alpha = 1$ for every assignment α with $F|_\alpha = 1$.

PROOF Analogous to Lemma 3.1.1 and Lemma 3.1.3.   ∎

Corollary 3.3.9 `RT` $\leq$ `regRTL` $\leq$ `regWRTL` $\leq$ `WRTL` $\equiv$ `RTL`

In Chapter 4 it is shown that `RT` $\not\equiv$ `regRTL`. The questions whether `regRTL` $\equiv$ `regWRTL` and whether `regWRTL` $\equiv$ `WRTL` are open.

3.4   DLL with Learning and Regular Weak RTL

The goal of this section is to show that there is a correspondence between regular WRTL and DLL algorithms with learning. It is proved that there is a polynomial p such that for every unsatisfiable CNF formula F the difference between the fastest execution time of DLL-LEARN(F, α) and the size of the smallest regular WRTL for $\Box$ from F is not bigger then $p(|F|)$.

First it is shown that if one removes the root from a given regular WRTL T from a given formula F then the right remaining subtree of T is a regular WRTL from F conjoined with the set of clauses of the left subtree of T. The other direction of that statement holds too. Both are helpful in the future proofs in this section.

Lemma 3.4.1 Let F be a CNF formula and let C_0, C_1 and C be clauses.

(a) Let T be a regular WRTL for C from F and let T_0 be the left and T_1 be the right subtree of T that arises out of T by removing C. If C_0 is the root of T_0 and C_1 is the root of T_1 then T_0 is a regular WRTL for C_0 from F and T_1 is a regular WRTL for C_1 from $F \cup Cl(T_0)$.

(b) Let T_0 be a regular WRTL for C_0 from F, T_1 a regular WRTL for C_1 from $F \cup Cl(T_0)$ and $x \notin Var(T_0) \cup Var(T_1)$. Let T be the tree that arises out of T_0 and T_1 by adding the the new node $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ and two edges from C to C_0 and C_1 that are labeled with x such that $C_0 \leq_T C_1$. Then T is a regular WRTL for C from F.

PROOF (a) It is obvious that T_0 is a regular WRTL for C_0 from F. To prove that T_1 is a regular WRTL for C_1 from $F \cup Cl(T_0)$, the three parts of the definition of a WRTL have to be shown. The first and the third property hold since T is a WRTL. To show the second property let L be a leaf in T_1. Since T is a WRTL there is a clause D in $Cl(T) - \{C\}$ with $D \leq_T L$ and $D = L$. If $D \in Cl(T_1)$ then $D \leq_{T_1} L$. Otherwise $D \in Cl(T_0)$. In both cases the second property holds for L. Since T is regular, T_1 is regular, too. Thus T_1 is as stated.

(b) It is clear that the first and the third property of the definition of a WRTL hold. It follows also that the second property holds for all leaves of T_0. So let L be a leaf in T_1. By the definition of T_1, $L \in Cl(T_0)$, $L \in F$ or there is a $D \in Cl(T_1)$ with $D \leq_{T_1} L$ and $D = L$. But since $Cl(T_0) \subseteq Cl(T)$ therefore $L \in F$ or $D \leq_T L$ and $D = L$ for a $D \in Cl(T)$. Finally it follows directly from the assumptions that T is regular. ∎

Second it is shown that for a given regular WRTL of size s there is a run of DLL-LEARN (see page 26) that does exactly $s - 1$ recursive calls. The proof is similar to the proof of Lemma 3.2.4.

Lemma 3.4.2 Let F be an unsatisfiable formula in CNF and C a clause that has a regular WRTL T of size s from F. Let α be an assignment with $C|_\alpha = 0$ and $dom(\alpha) \cap Var(T) = \varnothing$. Then there is an execution of DLL-LEARN(F, α) that performs exactly $s - 1$ recursive calls and returns $(F \cup Cl(T), \texttt{UNSAT})$ so that C is tagged as the newest clause in $F \cup Cl(T)$.

PROOF The lemma is shown by induction on s.

Induction basis: $s = 1$. Then T consists only of the clause C and by definition of a WRTL $C \in F$. Since $C|_\alpha = 0$ it follows $F|_\alpha = 0$ and DLL-LEARN(F, α) can terminate in line 4 without any recursive calls and C can be chosen and tagged as new in line 3.

Induction step: $|T| = s + 2$. Let then T_0 be the left and T_1 be the right subtree of T that arises after removing the root C from T. Let C_0 be the root of T_0, C_1 the root of T_1 and let x be the label of the edges between C and C_0, C_1, i.e.,

$C_0, C_1 \vdash^w_x C$. Since the other case is fully symmetric assume for the purpose of simplicity that $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$.

Because $x \in Var(T)$ and $Var(T) \cap dom(\alpha) = \varnothing$, $x \in Var(F) - dom(\alpha)$. Thus it is possible to skip a termination without recursive calls in line 4 or line 7 and to choose x as branching variable in line 8.

Choose 0 as the first value of ϵ in line 8 and let $\alpha_i = \alpha \cup \{(x, i)\}$ for $i = 0$ and $i = 1$. Then DLL-LEARN(F, α_0) is called in line 9. Since $C|_\alpha = \varnothing$ and $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ by assumption it follows $C_0|_{\alpha_0} = \varnothing$. Because T is regular, T_0 is a regular WRTL and $dom(\alpha_0) \cap Var(T_0) = \varnothing$. Hence the induction hypothesis states that there is an execution of DLL-LEARN(F, α_0) that makes $|T_0| - 1$ recursive calls and returns $(G, \texttt{UNSAT})$ such that $G = G \cup Cl(T_0)$ and C_0 is the newest clause in G.
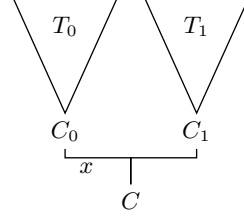
Since the execution of line 11 is optional, it is possible to call DLL-LEARN(G, α_1) in line 12. By Lemma 3.4.1 (a) T_1 is a regular WRTL for C_1 from G. Since T is regular and by the assumption it follows again that $dom(\alpha_1) \cap Var(T_1) = \varnothing$ and that $C_1|_{\alpha_1} = 0$. Thus there exists by induction an execution of DLL-LEARN(G, α_1) that makes $|T_1| - 1$ recursive calls and returns $(H, \texttt{UNSAT})$ such that $H = G \cup Cl(T_1)$ and C_1 is the newest clause in H.

Because of the lines 14,15 and 16 of the algorithm DLL-LEARN(F, α) returns then $(H \cup \{C\}, \texttt{UNSAT})$ and $H \cup \{C\} = G \cup Cl(T_1) \cup \{C\} = F \cup Cl(T_0) \cup Cl(T_1) \cup \{C\}$.

Therefore there is an execution of DLL-LEARN(F, α) that makes overall $|T_0| + |T_1| - 2 + 2 = |T| - 1$ recursive calls and returns $(F \cup Cl(T), \texttt{UNSAT})$ whereby C is the newest clause in H. ∎

The next lemma shows that the opposite direction of Lemma 3.4.2 holds too: for an execution of DLL-LEARN that returns $(F', \texttt{UNSAT})$ there exists a regular RTL of size at most $2s + 1$. The blowup by the factor 2 in this direction is caused by the fast backtracking step in line 11 of DLL-LEARN.

Note that *continued learning* is used in the proof. It is open whether continued learning is needed for the proof of Lemma 3.4.3.

Lemma 3.4.3 Let F be a CNF formula and let α be an assignment such that there exists an execution of DLL-LEARN(F, α) that makes s recursive calls and returns $(G, \texttt{UNSAT})$. Let C be the newest clause in G. Then $C|_\alpha = 0$ and there is a regular WRTL T for C from F with $|T| \leq 2s + 1$, $Var(T) \cap dom(\alpha) = \varnothing$ and $G \subseteq F \cup Cl(T)$.

PROOF The lemma is proved by induction on s. Fix an execution of DLL-LEARN(F, α) that makes s recursive calls and returns $(G, \texttt{UNSAT})$ and let C be the newest clause in G.
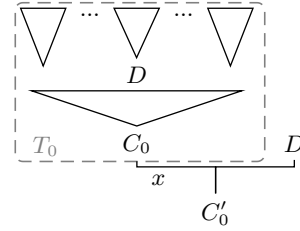
Induction Basis: $s = 0$, i.e., DLL-LEARN(F, α) makes no recursive calls. This means that the algorithm returns $(G, \texttt{UNSAT})$ in line 4 or line 7. In both cases $F|_\alpha = 0$ and a $C \in F$ with $C|_\alpha = 0$ is tagged as the newest clause in line 3 or 6, respectively. Thus the WRTL T that consists only of the clause C is as stated in the lemma.

Induction Step: DLL-LEARN(F, α) makes $s > 0$ recursive calls. Then a branching variable $x \in Var(F) - dom(\alpha)$ and a value $\epsilon \in \{0, 1\}$ is chosen in line 8. Since the other case is symmetric it is assumed that $\epsilon = 0$. Let $\alpha_i = \alpha \cup \{(x, i)\}$ for $i = 0$ and $i = 1$.

By the choice of x and ϵ, DLL-LEARN(F, α_0) is called in line 9. The execution after that call consists of $s_0 < s$ recursive calls and $(G, \texttt{UNSAT})$ is returned by DLL-LEARN(F, α_0). Let C_0 be the newest clause in G. By the induction hypotheses it follows that $C_0|_{\alpha_0} = 0$ and there is a regular WRTL T_0 for C_0 from F with $|T_0| \leq 2s_0 + 1$, $Var(T_0) \cap dom(\alpha) = \varnothing$ and $G \subseteq F \cup Cl(T_0)$.

Now there is the possibility to do fast backtracking and two cases have to be distinguished.

Case 1: DLL-LEARN(F, α) returns $(G, \texttt{UNSAT})$ in line 11. Then $G|_\alpha = 0$. If $x \notin C_0$ then $C_0|_\alpha = 0$ and since $(G, \texttt{UNSAT})$ is returned after $s_0 + 1$ recursive calls T_0 is a regular weak RTL as wanted. If otherwise $x \in C_0$ then $D|_\alpha = 0$ for a $D \in G$ with $D \neq C_0$. Thus $D \in F \cup Cl(T_0)$ and the tree T_0' that arises out of T_0 by adding a new leaf D and a new root $C_0' = (C_0 - \{x\}) \cup D$ as well as edges from C_0' to C_0 and D is a regular WRTL for C_0' from F. It holds $Cl(T_0') \subseteq Cl(T_0)$, $C_0'|_\alpha = 0$ and $|T_0'| = |T_0| + 2$ and since there are $s = s_0 + 1$ recursive calls in the whole execution is follows $|T_0'| \leq 2s_0 + 1 + 2 = 2s + 1$ and T_0' is as wanted.
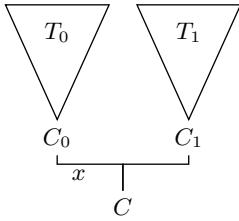
Case 2: DLL-LEARN(G, α_1) is called in line 12. The execution of that call does overall $s_1 = s - (s_0 + 2) < s$ recursive calls and returns $(H, \texttt{UNSAT})$. Let C_1 be the newest clause in H. By induction it follows $C_1|_{\alpha_1} = 0$ and there is a regular WRTL T_1 for C_1 from G with $|T_1| \leq 2s_1 + 1$, $Var(T_1) \cap dom(\alpha) = \varnothing$ and $H \subseteq F \cup Cl(T_1)$.

Let T be the tree that arises out of T_0 and T_1 by adding a new root $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ and edges from C to C_0 and C_1 that are labeled with x such that $C_0 \leq_T C_1$. Since $x \in dom(\alpha_i)$, $x \notin Var(T_i)$ for $i = 0$ and $i = 1$. Thus it follows by Lemma 3.4.1 (b) that T is a regular WRTL for C from F. Furthermore $C|_\alpha = 0$, $Var(T) \cap dom(\alpha) = \varnothing$, $|T| = |T_0| + |T_1| + 1 \leq 2s_0 + 2s_1 + 1 \leq 2s + 1$ and $H \subseteq G \cup Cl(T_1) \subseteq F \cup Cl(T)$. By definition DLL-LEARN(F, α) returns $(H \cup \{C\}, \texttt{UNSAT})$. And since $C \in Cl(T)$, $H \cup \{C\} \subseteq F \cup Cl(T)$ and T is as stated. ∎

Theorem 3.4.4 Let F be an unsatisfiable CNF formula.

(a) If there exists a regular WRTL for $\Box$ from F of size s then there is an execution of DLL-LEARN$(F, \varnothing)$ that performs exactly $s-1$ recursive calls.

(b) If there exists an execution of DLL-LEARN$(F, \varnothing)$ that performs s recursive calls then there is a regular WRTL for $\Box$ from F of size at most $2s + 1$.

PROOF (a) Apply Lemma 3.4.2 to F, $C = \Box$ and $\alpha = \varnothing$.

(b) Apply Lemma 3.4.3 to F and $\alpha = \varnothing$. Then $C = \Box$ since the empty clause $\Box$ is the only clause that is falsified by α.   ∎

Compare Theorem 3.4.4 also to Theorem 4.3.7 in Section 4.3 which states that DLL-LEARN can also simulate general resolution in a less efficient way.

Now the correctness of DLL-LEARN follows easily form Theorem 3.4.4.

Corollary 3.4.5 (Correctness of DLL-Learn) Let F be a CNF formula. Then every execution of DLL-LEARN$(F, \varnothing)$ terminates. It returns (F', α) for a partial satisfying assignment α and a formula $F' \supseteq F$ if F is satisfiable and $(F', \texttt{UNSAT})$ otherwise.

PROOF It is easy to see that DLL-LEARN$(F, \varnothing)$ always terminates and by Lemma 3.4.3 it follows that F is unsatisfiable if the algorithm returns $(F', \texttt{UNSAT})$.

Therefore it suffices to show that that $F|_{\alpha'} = 1$ if DLL-LEARN(F, α) returns (F', α') for an assignment α.

It is shown by induction on the number of recursive calls s that $F' \equiv F$ if DLL-LEARN(F, α) returns (F', α') and $F|_{\alpha'} = 1$ if $\alpha' \neq \texttt{UNSAT}$.

Induction basis: If there are no recursive calls then $F' = F$, $\alpha = \alpha'$ and $F|_{\alpha} = 1$ if $\alpha \neq \texttt{UNSAT}$.

Induction step: Let DLL-LEARN(F, α) perform $s > 0$ recursive calls. If (F', α') is the return value of a recursive call in line 9 that is returned in line 10 or line 11 then the statement follows by induction since these calls perform less recursive calls.

Otherwise DLL-LEARN(F, α) performs two recursive calls. It follows by induction that $F \equiv G$ for the return value $(G, \texttt{UNSAT})$ of the first recursive call in line 9. Furthermore it follows by induction that $G \equiv H$ if (H, γ) is the value returned by the second recursive call DLL-LEARN(G, α') in line 12. Thus $F \equiv H$. By induction $G|_{\alpha'} = 1$ if $\alpha' \neq \texttt{UNSAT}$ holds and the statement is proved in that case. If $\alpha' = \texttt{UNSAT}$ then DLL-LEARN returns $(F', \texttt{UNSAT})$ for $F' = H \cup \{C\}$ and a new clause C in line 19. Since C is the newest clause in F', holds by Lemma 3.4.3 that there is a regular WRTL for C from F. But then $\vDash F \to C$ by Lemma 3.3.8 and therefore $F \equiv F \cup \{C\} \equiv H \cup \{C\}$ since $F \equiv H$.   ∎

As in Section 3.2 DLL-LEARN(F, α) can be improved to return a regular WRTL instead of UNSAT if the restricted formula $F|_\alpha$ is unsatisfiable.

Algorithm 3.4.6

```
DLL-LEARN'(F, α)
 1      if F|α = 1 then return (F, α)
 2      if F|α = 0 then do optionally              -- cont. learning?
 3           tag a C ∈ F with C|α = 0 as new
 4           return (F, C)
 5      if Var(F) − dom(α) = ∅ then               -- F|α = 0
 6           tag a C ∈ F with C|α = 0 as new
 7           return (F, C)
 8      choose x ∈ Var(F) − dom(α) and a value ε ∈ {0, 1}
 9      (G, β)←DLL-LEARN'(F, α ∪ {(x, ε)})
10      if β is not a reg.  WRTL then return (G, β)
11      if G|α = 0 then do optionally             -- fast backtracking
12           let C₀ be the root of β
13           if C₀|α = 0 then return (G, β)
14           fix a clause D ∈ G with D|α = 0
15           let T be the tree that arises out of β by adding a
16           new root C = (C₀ − {x¹⁻ᵉ}) ∪ D, a new leaf D and edges
17           from C to C₀ and D that are labeled with {x}
18           return (G, T)
19      (H, γ)←DLL-LEARN'(G, α ∪ {(x, 1 − ε)})
20      if γ is not a reg.  WRTL then return (H, γ)
21      select the newest C₀ ∈ G and the newest C₁ ∈ H
22      C ← (C₀ − {x¹⁻ᵉ}) ∪ (C₁ − {xᵉ})
23      H ← H ∪ {C}                               -- learn a clause
24      if C₀ ∉ F then do optionally H ← H − {C₀}  -- keep clauses?
25      if C₁ ∉ F then do optionally H ← H − {C₁}
26      let T be the tree that arises out of β and γ by adding
27      a new root C = (C₀ − {x}) ∪ (C₁ − {¬x}) and an edge from
28      C to Cᵢ that is labeled with {x} for i = 0 and i = 1
29      return (H, T)
```

Note that the above algorithm can run in polynomial time if and only if DLL-LEARN can run in polynomial time.

3.5 Learning by Unit Propagation and Regular WRTI

A natural, resolution based proof system that corresponds directly to DLL-L-UP one to one such as regular WRTL to DLL-LEARN has not been found. But it is proved in this section that every execution of DLL-L-UP can be transformed with polynomial blowup into regular WRTL that only uses *learning*

by input resolution. Learning by input resolution means that all lemmas that are used in the RTL occur earlier as the root of a linear tree.

As an application of this theorem, it is shown that the algorithm schema DLL-LEARN can simulate the schema DLL-L-UP, i.e., for every execution of DLL-L-UP there is an execution of DLL-LEARN so that the difference between the running times of the executions is bounded by a polynomial.

To start with it is shown that a conflict graph can be transformed into a simple linear resolution tree for the conflict clause, namely an *input resolution tree.* See Figure 3.3 on page 48 for an example of the construction. This result has been proved earlier in [12] and [2]. In these papers one can also find proofs for the other direction of Lemma 3.5.1, i.e., transforming a regular input resolution proof into a conflict graph.

Definition (Input Resolution Tree) A resolution tree T is called *input resolution tree* iff every inner node of T has a child that is a leaf.

For the next lemma recall the definitions on page 19.

Lemma 3.5.1 Let F be a CNF formula, $G = (V, E)$ a conflict graph for F with $|V_{G\restriction_\square}| > 3$. Then $C_G = \overline{V^0_{G\restriction_u}}$ has a regular input resolution tree T_G from F with $Var(T_G) \subseteq Var(V_{G\restriction_\square}) - Var(C_G) \subseteq Var(V) - Var(V^0_G)$.

PROOF By induction on $k = |V - V^0_G|$, i.e., the number of inner nodes of G. Let x be the conflict variable of G.

Induction basis: $k = 2$. Then $x \notin V^0_G$ or $\neg x \notin V^0_G$. Assume w.l.o.g. $x \notin V^0_G$. Then there is by definition a clause C_x in F with $C_x = \{x\} \cup \{ \bar{v} \mid (v, x) \in E \}$. But since $V^0_G = \{\neg x\} \cup \{ \bar{v} \mid (v, x) \in E \}$, $C_x = C_G$ follows and the input resolution tree that consists only of $C_G \in F$ is as required.

Induction step: $k > 2$. Then there is a $l \in V$ with $Depth_G(l) = 1$ such that $G' = (V', E')$ with $V' = V$ and $E' = E - \{ (v, l) \mid v \in V, (v, l) \in E \}$ is a conflict graph with $k - 1$ inner nodes. Let w.l.o.g. $l = y$ for a variable y ($l = \neg y$ is analogous).

By definition there is a clause $C_l \in F$ with $C_l = \{l\} \cup \{ \bar{v} \mid (v, l) \in E \}$. By induction there exists an input resolution tree $T_{G'}$ for $C_{G'} = (C_G \cup C_l) - \{y, \neg y\}$ with $Var(T_{G'}) = Var(V_{G'\restriction_\square}) - Var(C_{G'})$. Thus $y \notin Var(T_{G'})$ and the tree T_G that arises out of $T_{G'}$ by adding the new root C_G, a new leaf C_l and edges from C_0 to $C_{G'}$ and C_l that are labeled with y is a regular input resolution tree for C_G from F. It holds that $Var(T_G) = Var(T_{G'}) \cup \{y\} \subseteq (Var(V_{G'\restriction_\square}) - Var(C_{G'})) \cup \{y\} \subseteq Var(V_{G\restriction_\square}) - Var(C_G)$. ∎

At next it is shown that a regular input resolution tree that corresponds to a proper sub conflict graph can be extended to a regular input resolution tree for

the conflict clause of the super conflict graph. Note that Lemma 3.5.2 does not in general apply to non-proper sub conflict graphs.

Lemma 3.5.2 Let F be a CNF formula and let $n \geq 1$. Let $H_1, \ldots, H_s = G$ be conflict graphs for F such that $|V_{H_1 \lceil \square}| > 3$ and H_i is a proper sub conflict graph of H_{i+1} for each $1 \leq i < s$. Then there is a regular input resolution tree T for $C_G = \overline{V^0_{G \lceil \square}}$ from F with $C_{H_i} \in Cl(T)$ for every $1 \leq i \leq s$ and $Var(T) \subseteq Var(V_{G \lceil \square}) - Var(C_G)$.

PROOF To start with, let $s = 1$. By Lemma 3.5.1 there is a regular input resolution tree T_1 for $C_{H_1} = \overline{V^0_{H_1 \lceil \square}}$ from F with $Var(T_1) \subseteq Var(V_{H_1 \lceil \square}) - Var(C_{H_1})$. Then $T = T_1$ is already as wanted.

Let now $s > 1$, $H = H_{s-1}$ and $G = H_s$. Let T_H be a a regular input resolution tree for $C_H = \overline{V^0_{H \lceil \square}}$ from F with $C_{H_i} = \overline{V^0_{H_i \lceil \square}} \in Cl(T_H)$ for every $1 \leq i < s$ and $Var(T_H) \subseteq Var(V_{H \lceil \square}) - Var(C_H)$.

It is shown by induction on $d = |V_{G \lceil \square}| - |V_{H \lceil \square}|$ that T_H can be extended to a regular input resolution tree T for C_G from F. T is then as required.

Induction basis: $d = 0$. Then $G \lceil \square = H \lceil \square$ and $T = T_H$ is already as wanted.

Induction step: $d > 0$. Then $m = \max\{ Depth_G(v) \mid v \in V^0_{H \lceil \square} \} > 0$. Pick a $l = x^\epsilon \in V^0_{H \lceil \square}$ with $Depth_G(l) = m$. Since H is a proper sub conflict graph of G, $V_{G \lceil v} \cap V^0_{H \lceil \square} = \{v\}$ for all $v \in V^0_{H \lceil \square}$.

Let $\tilde{H} = (\tilde{V}, \tilde{E})$ with $U = \{ v \mid (v, l) \in E \}$, $\tilde{V} = V' \cup U$ and $\tilde{E} = E' \cup \{ (u, v) \in E \mid v = l \}$. $\tilde{H}$ is a sub conflict graph of G.

To show that $\tilde{H}$ is proper for G let $v_0 \in V^0_{\tilde{H} \lceil \square}$. Then $v_0 \in V^0_{\tilde{H} \lceil \square} - \{l\}$ or $v_0 \in U$. If $v_0 \in V^0_{\tilde{H} \lceil \square} - \{l\}$ then $\tilde{V} \cap V_{G \lceil v_0} \subseteq U \cup V^0_{H \lceil \square} \subseteq (V^0_{H \lceil \square} \cup U) - \{l\} \subseteq V^0_{\tilde{H} \lceil \square}$. If $v_0 \in U$ then $\tilde{V} \cap V_{G \lceil v_0} \subseteq U \cup (V^0_{H \lceil \square} - \{l\}) \subseteq V^0_{\tilde{H} \lceil \square}$. Hence $\tilde{H}$ is a proper sub conflict graph of G with $\tilde{d} = |V_{G \lceil \square}| - |V_{\tilde{H} \lceil \square}| < d$.

Define $T_{\tilde{H}}$ to be tree that arises out of T_H by by adding the new root $C_{\tilde{H}}$, a new leaf C_l and edges from C_H to $C_{\tilde{H}}$ and $C_l = \{l\} \cup \{ \bar{v} \mid (v, l) \in E \}$ that are labeled with x. Because $x \in C_H$ holds $x \notin Var(T_H)$ and by definition of a conflict graph $C_l \in F$. Thus and since $C_H = (C_{\tilde{H}} - x^{1-\epsilon}) \cup (C_l - \{x^\epsilon\})$, $T_{\tilde{H}}$ is a regular input resolution tree for $C_{\tilde{H}}$ from F and $Var(T_{\tilde{H}}) = Var(T_H) \cup \{x\} \subseteq (Var(V_{H \lceil \square}) - Var(C_H)) \cup \{x\} = Var(V_{\tilde{H} \lceil \square}) - Var(C_{\tilde{H}})$.

Therefore the induction hypothesis applies and $T_{\tilde{H}}$ can be extended to a regular input resolution tree for C_G from F with $Var(T) \subseteq Var(V_{G \lceil \square}) - Var(C_G)$. But since $T_{\tilde{H}}$ is an extension of T_H, T is also an extension of T_H. ∎

Analogous to a conflict graph, a subgraph $G \lceil_u$ of a UP-graph G can be transformed into a regular input resolution tree T for the induced clause $C_{G \lceil_u}$. There are more general versions of this result that may appear in a future work with
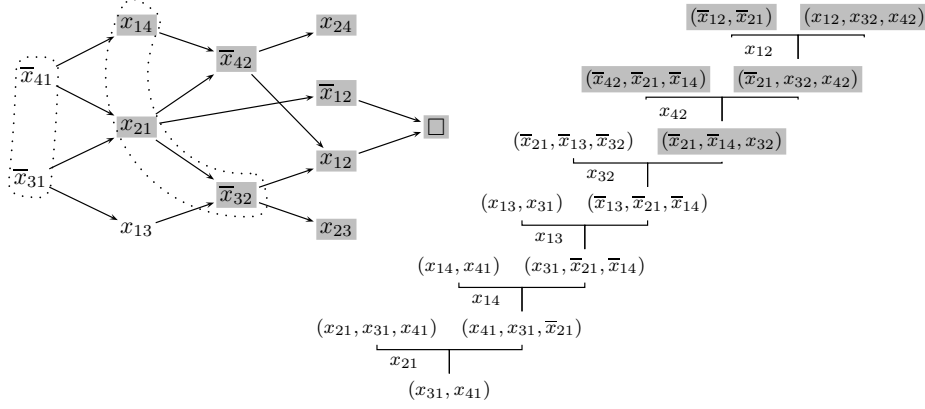
Figure 3.3: A conflict graph for OP_4 with a proper sub conflict graph and a regular input resolution tree for the conflict clause $(\bar{x}_{41} \lor \bar{x}_{21} \lor x_{32})$ of the sub graph that is extended to a tree of the conflict clause $(x_{31} \lor x_{41})$.

Sam Buss. But for the definition of compatible subgraphs, that is used in this thesis, Lemma 3.5.3 is sufficient.

Lemma 3.5.3 Let F be a CNF formula, $G = (V, E)$ a UP-graph for F and $u \in V - V_G^0$. Then $C_{G\lceil u} = \overline{V_{G\lceil u}^0} \cup \{u\}$ has a regular input resolution tree T_u from F with $Var(T_u) \subseteq Var(V_{G\lceil u}) - Var(C_{G\lceil u})$.

PROOF By induction on $k = |V_{G\lceil u} - V_{G\lceil u}^0|$, i.e., on the number of inner nodes of $G\lceil u$.

Induction basis: $k = 1$. Then $V_{G\lceil u} = \{u\} \cup \{ v \mid (v, u) \in E_{G\lceil u} \}$ and $C_{G\lceil u} = \{u\} \cup \{ \bar{v} \mid (v, u) \in E_{G\lceil u} \} \in F$ is allready an input resolution tree as required.

Induction step: $k > 1$. Let $V_{G\lceil u}^1 = \{ v \in V_{G\lceil u} \mid Depth_{G\lceil u}(v) = 1 \}$. Since $k > 1$ there is a $l \in V_{G\lceil u}^1$ with $l \neq u$. Let w.l.o.g $l = y$ for a variable y ($l = \bar{y}$ is symmetric).

Put $V' = V_{G\lceil u} - \{y\}$, $E' = E_{G\lceil u} \cap (V' \times V')$ and $H = (V', E')$. Then $H\lceil u = H$ and $|V' - V_H^0| = k - 1$. Thus it follows by induction that there is a input resolution tree T_H for $C_H = \overline{V_H^0} \cup \{u\}$ from F with $Var(T_H) \subseteq Var(V') - Var(C_H)$.

T_H can be extended to an input resolution tree T_u for $C_{G\lceil u} = (C_H - \{y\}) \cup \{ \bar{v} \mid (v, y) \in E_{G\lceil u} \}$ as follows: By definition of $G\lceil u$ there is a clause $C_y \in F$ with $C_y = \{y\} \cup \{ \bar{v} \mid (v, y) \in E_{G\lceil u} \}$. Let T_u be the tree that arises out of T_H by adding the new leaf C_y, the new root $C_{G\lceil u}$ and edges from $C_{G\lceil u}$ to C_y and C_H that are labeled with y. Since $C_{G\lceil u} = (C_H - \{\bar{y}\}) \cup (C_y - \{y\})$, T_u is an input

resolution tree and since $y \in Var(C_H)$ it follows $y \notin Var(T_H)$ and thus T_u is regular. Furthermore $Var(T_u) = Var(T_H) \cup \{y\} \subseteq (Var(V') - Var(C_H)) \cup \{y\} \subseteq Var(V_{G\lceil u}) - Var(C_{G\lceil u})$. Thus T_u is as wanted. ∎

Below is the definition of *(weak) resolution trees with input lemmas* which are (W)RTL that satisfy a restriction on the lemmas that are used in the tree: the only clauses that can be used in the leaves of the tree are clauses $C \in F$ or clauses that are *input nodes* of the tree. One can think of an resolution tree with input lemmas as a generalization of *linear resolution*.

Definition (Input Node) Let T be a WRTL and let C be the label of a node v of T. Then the tree T_v is the subtree of T that consists of v and all successors of v.
v is called an *input node* iff T_v is an input resolution tree.

Often the node v and its label C are identified. Then T_C is written instead of T_v and C is called an input node.

Definition ((W)RTI) Let F be a CNF formula, C a clause and T a binary tree in which the nodes are labeled with clauses and the edges are labeled with variables. Identify the nodes of T with their labels. T is a *(weak) resolution tree with input lemmas ((W)RTI)* for C from F if

- the root of T is labeled with C,

- if D is a leaf in T then $D \in F$ or $D = D'$ for an input node D' with $D' <_T D$,

- if D is an inner node and the two children of D are D_0 and D_1 then $D_0, D_1 \vdash_x D$ ($D_0, D_1 \vdash_x^w D$) for a variable x and the edges between D and its children are labeled with x.

$Var(T)$, $Cl(T)$ and the *size* and the *regularity* of T are defined analogously to RTL.
$ICl(T) = \{ C \in Cl(T) \mid C$ is a input node $\}$ is the set of clauses that are labels of input nodes.

Definition (regWRTI) Let F be a formula. The proof system `regWRTI` is defined through: `regWRTI`(F,T) if and only if T is a regular WRTI for $\square$ from $CNF(\neg F)$.
The proof systems `RTI`, `WRTI` and `regRTI` are defined analogous to `RTL`, `WRTL` and `regRTL`.

`regWRTI` is decidable in polynomial time. The soundness follows from the fact that every WRTI is also a WRTL and `regWRTI` is complete because every regular resolution tree is also a regular WRTI. From similar arguments the same follows for the proof systems `RTI`, `WRTI` and `regRTI`.

Corollary 3.5.4 The proof systems `regWRTI`,`RTI`, `WRTI` and `regRTI` are sound and complete.

The following proposition follows directly from the definition.

Proposition 3.5.5 `regRT` $\leq$ `regRTI` $\leq$ `regWRTI` $\leq$ `regWRTL` and `regRTI` $\leq$ `regRTL`.

In Section 4.1 it is proved that `RTI` $\equiv$ `WRTI` $\equiv$ `RTL` $\equiv$ `WRTL` and in Section 4.3 it is shown that `RT` $<$ `regWRTI`. On the other side the questions whether `regRTL` $\equiv$ `regWRTI`, `regRTL` $\equiv$ `regRTI` or `regWRTI` $\equiv$ `regWRTL` are open.

In order to construct a regular WRTI for a a given execution of DLL-L-UP, Lemma 3.5.6 shows that there is a regular RTI of size $|G|^2$ that contains the conflict clauses of a given compatible set of subgraphs of G as input clauses. There are more general versions of this lemma that perhaps will be published in a future paper with Sam Buss. Figure 3.4 illustrates the proof of Lemma 3.5.6.

For the following lemma recall the definition of compatible subgraphs on page 21.

Lemma 3.5.6 Let G be a CNF formula, let $G = (V, E)$ be a conflict graph for F and let $\mathcal{C}$ be a compatible set of subgraphs of G. Then there exists a regular RTI T for C_G from F such that $|T| \leq 2(|V| - |C_G|)^2$, $Var(T) \subseteq Var(V_{G\lceil_\square}) - Var(C_G)$ and $CC(\mathcal{C}) \subseteq ICl(T)$.

PROOF Let $\mathcal{C} = \{H_1, \ldots H_s\} \cup \{G\lceil_{v_1}, \ldots, G\lceil_{v_t}\}$ with sub conflict graphs H_i and UP-graphs $G\lceil_{v_j}$ be as defined in Section 2.2 on page 18.

Let $Var(v_j) = x_j$ for all $1 \leq j \leq t$ and let w.l.o.g. $Depth_G(v_j) \leq Depth_G(v_k)$ for $j < k$. Then $x_k \notin Var(G\lceil_{v_j})$ for every $k > j$.

We construct inductively regular RTI T_0 for $C_0, \ldots, T_t$ for C_t from F such that for every $1 \leq k \leq t$:

(a) $C_{H_i} \in ICl(T_k)$ for every $1 \leq i \leq s$

(b) $C_{G\lceil_{v_j}} \in ICl(T_k)$ for every $1 \leq j \leq k$

(c) $\overline{V^0_{G\lceil_{v_1}}} \cup \ldots \cup \overline{V^0_{G\lceil_{v_k}}} \cup \{\bar{v}_{k+1}, \ldots, \bar{v}_t\} \subseteq C_k \subseteq \{\bar{v}_{k+1}, \ldots, \bar{v}_t\} \cup C_G$

(d) $\{x_{k+1}, \ldots, x_t\} \cap Var(T_k) = \varnothing$

(e) $|T_k| \leq 2(|V| - |C_G|) \cdot (k+1)$

By Lemma 3.5.2 there is a regular input resolution tree T_0 for C_{H_s} from F with $C_{H_i} \in Cl(T_0)$ for every $1 \leq i \leq s$ and $Var(T_0) \subseteq Var(V_{H_s \lceil \square}) - Var(C_{H_s})$. T_0 satisfies (a) - (e).

Let $k > 0$ and let T_{k-1} be already constructed. By Lemma 3.5.3 there is a a regular input resolution tree T_{v_k} for the induced clause $C_{G \lceil v_k} = \overline{V^0_{G \lceil v_k}} \cup \{v_k\}$ from F with $Var(T_{v_k}) \subseteq Var(V_{G \lceil v_k}) - Var(C_{G \lceil v_k})$. It follows that $|T_{v_k}| \leq 2(|V| - C_G)$ and, from the assumption on the order of the v_i, $\{x_{k+1}, \ldots, x_t\} \cap Var(T_k) = \varnothing$.

Let T_k be the tree that arises out of T_{k-1} and T_{v_k} by adding the new root $C_k = (C_{k-1} - \{\bar{v}_k\}) \cup (C_{G \lceil v_k} - \{v_k\})$ and edges labeled with v_k from C_k to C_{k-1} and $C_{G \lceil v_k}$. Since $v_k \in C_{G \lceil v_k}$ it follows that $x_k \notin Var(T_{v_k})$ and by induction $x_k \notin Var(T_{k-1})$ and $\bar{v}_k \in C_{k-1}$. Thus T_k is a regular RTI from F. (a) (b) and (c) for T_k follow immediately from (a), (b) and (c) for T_{k-1}. Because $\{x_{k+1}, \ldots, x_t\} \cap Var(T_k) = \varnothing$ and $\{x_k, \ldots, x_t\} \cap Var(T_{k-1}) = \varnothing$, $\{x_{k+1}, \ldots, x_t\} \cap Var(T_k) = \varnothing$. Since $|T_{k-1}| \leq 2(|V| - |C_G|) \cdot k$ and $|T_{v_k}| \leq 2(|V| - |C_G|)$ it follows $|T_k| \leq 2|V| \cdot (k+1)$.

Finally $T_t = T$ is a regular RTI for C_G as stated in the lemma. ∎
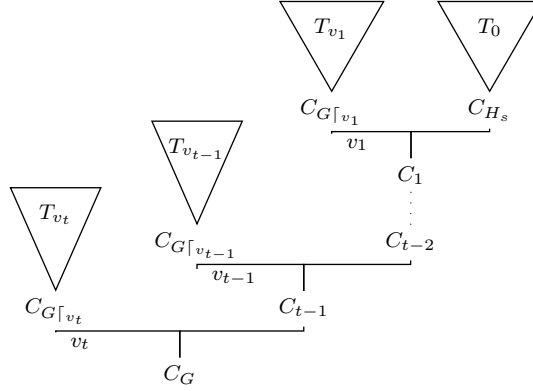
Figure 3.4: The construction of a regular RTI T corresponding to the proof of preflem:compatible-¿regRTI. T contains the set of conflict clauses of a compatible set of subgraphs as input clauses.

Before the proof of the main result of the section is given the next lemma makes a analogous statement as Lemma 3.4.1 for RTL.

Lemma 3.5.7 Let F be a CNF formula and let C_0, C_1 and C be clauses. Let T_0 be a regular WRTI for C_0 from F, T_1 a regular WRTI for C_1 from $F \cup ICl(T_0)$ and $x \notin Var(T_0) \cup Var(T_1)$. Let T be the tree that arises out of T_0 and T_1 by

adding the new node $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ and two edges from C to C_0 and C_1 that are labeled with x such that $C_0 \leq_T C_1$. Then T is a regular WRTI for C from F.

PROOF Analogously to Lemma 3.4.1.   ∎

Now it is shown that an execution of DLL-L-UP can be transformed into a regular WRTI.

Lemma 3.5.8 Let F be a CNF formula and $H = (V, E)$ a UP-graph for F. Let there be an execution of DLL-L-UP(F, H, α_H) that returns (F', UNSAT) and performs s recursive calls and let $n = \min\{|Var(F)|, s+1\}$. Then there exists a clause $C \subseteq \overline{V_H^0}$ (i.e., $C|_{\alpha_H} = 0$) and a regular WRTI T for C from F with $|T| \leq 2n^2 \cdot (s+2)$, $F' \subseteq F \cup ICl(T)$ and $Var(T) \cap Var(V_H^0) = \varnothing$.

PROOF Fix an execution of DLL-L-UP(F, H, α_H) that performs s recursive calls and returns (F', UNSAT). Let $\alpha = \alpha_H$. The statement is proved by induction on s.

Induction basis: $s = 0$, i.e., there are no recursive calls. Then DLL-L-UP(F, H, α) returns (F', UNSAT) in line 9. Let C_1 and C_2 be the clauses with $C_1|_\alpha = \{y\}$ and $C_2|_\alpha = \{\neg y\}$ that are chosen in line 4 of the algorithm. Let $N = \{\ (\bar{v}, y) \mid v \in C_1, v \neq y\ \} \cup \{\ (\bar{v}, \neg y) \mid v \in C_2, v \neq \neg y\ \}$ and $H' = (V', E')$ whereby $V' = V \cup \{y, \bar{y}, \square\}$ and $E' = E \cup N$. Since H is a UP-graph for F it follows immediately that H' is a a conflict graph for F.

If $N = \varnothing$ then $\{y\}, \{\neg y\} \in F$ and no clause is learned, i.e., $F' = F$. In that case the RTI T with $Var(T) = \{y\}$ that consists of the two leaves $\{y\}$, $\{\neg y\}$ and the root $\varnothing$ is as wanted because $y \notin V_H^0$, $\varnothing \subseteq \overline{V_H^0}$ and $|T| = 3 < 4n^2$.

If otherwise $N \neq \varnothing$ then let $\mathcal{C}$ be the compatible set of subgraphs of H' that is chosen in line 8. Then it follows by Lemma 3.5.6 that there is a regular RTI T' for $C_{H'} = \overline{V_{H'\restriction_\square}}$ with $CC(\mathcal{C}) \subseteq Cl(T') = ICl(T')$, $|T'| \leq 2(|V'| - |C_{H'}|)^2$ and $Var(T') \subseteq Var(V_{H'\restriction_\square}) - Var(C_{H'})$.

If $y^\epsilon \notin C_{H'}$ for $\epsilon = 0$ and $\epsilon = 1$ then $C_{H'} \subseteq \overline{V_H^0}$ and one can put $T = T'$ and $C = C_{H'}$. If otherwise $y^\epsilon \in C_{H'}$ for a $\epsilon \in \{0, 1\}$ then $\{y^{1-\epsilon}\} \in F$. In that case define T to be the regular RTI that arises out of T' be adding the new nodes $\{y^{1-\epsilon}\}$ and $C = C_{H'} - \{y^\epsilon\}$ together with edges from C to $\{y^{1-\epsilon}\}$ and $C_{H'}$ that are labeled with y. Then $C \subseteq \overline{V_H^0}$.

In both cases follows $|T| \leq 2|V|^2 \leq 2n^2$ and $Var(T) \cap Var(V_H^0) = \varnothing$. Thus, and since $F' \subseteq F \cup \{C_K\} \subseteq F \cup ICl(T)$, T is a regular WRTI as required.

*Induction step*: $s > 0$. *Case 1*: There is one recursive call of DLL-L-UP in line 15. Let then D be the clause with $D|_\alpha = \{x^\epsilon\}$ that is chosen in line 12 of the algorithm. Let $V' = V \cup \{x^\epsilon\}$, $E' = E \cup \{\ (\bar{v}, x^\epsilon) \mid v \in C, v \neq l\ \}$, $H' = (V', E')$ and $\alpha' = \alpha_{H'}$. Since H is a UP-graph from F, H' is also a UP-graph from F.

Then DLL-L-UP(F, H', α') is called in line 15 and performs $s' = s-1$ recursive calls after which $(F', \mathtt{UNSAT})$ is returned. By induction there is therefore a $C \subseteq \overline{V_{H'}^0}$ and a regular WRTI T for C from F with $|T| \leq 2n^2(s+1)$, $F' \subseteq F \cup ICl(T)$ and $Var(T) \cap Var(V_{H'}^0) = \varnothing$. If $D \in F$ is no unit clause then $V_H^0 = V_{H'}^0$ and thus T is already as required. If $D \in F$ is the unit clause $D = \{x^\epsilon\}$ then $V_{H'}^0 = V_H^0 \cup \{x^\epsilon\}$ and the tree that arises out of T by resolving C and $\{x^\epsilon\}$ on x is as wanted.

Case 2: There are two recursive calls of DLL-L-UP in the lines 17 and 20. Let then x be the branching variable that is chosen in line 16 and let w.l.o.g. $\epsilon = 0$ ($\epsilon = 1$ is analogous). For $i = 0$ and $i = 1$ let $V_i = V \cup \{x_i\}$, $H_i = (V_i, E)$ and $\alpha_i = \alpha_{H_i}$. Since H is a UP-graph from F, H_0 and H_1 are UP-graphs from F, too.

Then DLL-L-UP(F, H_0, α_0) is called in line 17, does $s_0 < s$ recursive calls and returns $(G, \mathtt{UNSAT})$. Thereafter DLL-L-UP(G, H_1, α_1) is called in line 20 and does $s_1 < s - (2 + s_0)$ recursive calls after which $(F', \mathtt{UNSAT})$ is returned.

By induction there is a clause $C_0 \subseteq \overline{V_{H_0}^0}$ and a regular WRTI T_0 from F with $|T_0| \leq 2n^2(s_0 + 2)$, $G \subseteq F \cup ICl(T_0)$ and $Var(T_0) \cap Var(V_{H_0}^0) = \varnothing$. Furthermore it follows that there is a $C_1 \in \overline{V_{H_1}^0}$ and a regular WRTI T_1 from G with $|T_1| \leq 2n^2(s_1 + 2)$, $F' \subseteq G \cup ICl(T_1)$ and $Var(T_1) \cap Var(V_{H_1}^0) = \varnothing$.
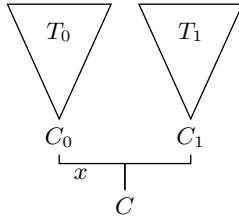
Let T be the tree that arises out of T_0 and T_1 by adding the the new node $C = (C_0 - \{x\}) \cup (C_1 - \{\neg x\})$ and two edges from C to C_0 and C_1 that are labeled with x such that $C_0 \leq_T C_1$. Since $x \notin Var(T_0) \cup Var(T_1)$ it follows by Lemma 3.5.7 that T is a regular WRTI for C from F. It holds $C \subseteq (\overline{V_{H_0}^0} \cup \overline{V_{H_1}^0}) - \{x, \neg x\} = \overline{V_H^0}$, $|T| = |T_0| + |T_1| + 1 \leq 2n^2(s+2)$, $F' \subseteq G \cup ICl(T_1) \subseteq F' \cup ICl(T_0) \cup ICl(T_1) = F' \cup ICl(T)$ and $Var(T) \cap Var(V_H^0) = (Var(T_0) \cup \{x\} \cup Var(T_1)) \cap (Var(V_{H_0}^0) - \{x\}) = \varnothing$ (since $Var(V_{H_0}^0) = Var(V_{H_1}^0)$). Thus T is a RTI as required.

Case 3: There is one recursive call of DLL-L-UP in line 17 and $(G, \mathtt{UNSAT})$ is returned in line 19. Let H_0 and α_0 be as in case 2. Then DLL-L-UP(F, H_0, α_0) is called in line 17, does $s_0 = s - 1$ recursive calls and returns $(G, \mathtt{UNSAT})$ for a $G \supseteq F$ with $G|_\alpha = 0$.

By induction there is a clause $C_0 \subseteq \overline{V_{H_0}^0}$ and a regular WRTI T_0 for C_0 from F with $|T_0| \leq 2n^2(s_0 + 2)$, $G \subseteq F \cup ICl(T_0)$ and $Var(T_0) \cap Var(V_{H_0}^0) = \varnothing$. Since $F|_\alpha \neq 0$, there is a $D \in F' - F$ with $D|_\alpha = 0$. If $C_0|_\alpha = 0$ then T_0 is already as wanted.

Otherwise $C_0|_\alpha = \{x^\epsilon\}$ and we need to resolve on x to get a tree as wanted. Since $D|_\alpha = 0$, $D \subseteq \overline{V_0}$. Let $D \cap (\overline{V_0} - \overline{V_{H_0}^0}) = \{v_1, \ldots, v_t\}$. Analogous to the proof of Lemma 3.5.3 we will construct a regular RTI T_D for the clause

$D' = (D \cap \overline{V_{H_0}^0}) \cup V_{H_0 \lceil v_1}^0 \cup \ldots \cup V_{H_0 \lceil v_t}^0$ from F such that $|T_D| \leq 2n^2$ and $Var(T_D) \cap Var(V_H^0) = \varnothing$.

Let therefore w.l.o.g $Depth_{H_0}(v_i) \leq Depth_{H_0}(v_j)$ if $1 \leq i < j \leq k$. Let T_0 be the RTI that consists of the single clause D. Let $0 < k \leq t$. Let T_{k-1} be already constructed and let D_{k-1} be the root of T_{k-1}.

By Lemma 3.5.3 there is a a regular input resolution tree T_{v_k} for the induced clause $C_{v_k \lceil H_0} = \overline{V_{H_0 \lceil v_k}^0} \cup \{v_k\}$ from F with $Var(T_{v_k}) \subseteq Var(V_{H_0 \lceil v_k}) - Var(C_{H_0 \lceil v_k})$. Let T_k be the tree that arises out of T_{k-1} and T_{v_k} by adding the new root $D_k = (D_{k-1} - \{\bar{v}_k\}) \cup (C_{G \lceil v_k} - \{v_k\})$ and edges labeled with v_k from D_k to D_{k-1} and $C_{G \lceil v_k}$.
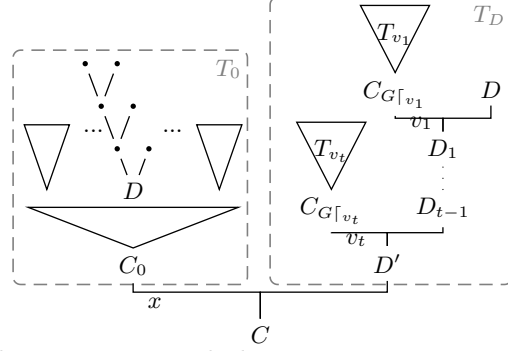
Analogous to the proof of Lemma 3.5.3 it follows that $T_D = T_t$ is a regular RTI as stated above.

Let now T be the tree that arises out of T_0 and T_D by adding a new root $C = (C_0 - \{x^\epsilon\}) \cup D'$ as well as edges from C to C_0 and D' that are labeled with x. Since $x \notin Var(T_0)$ and $x \notin Var(T_D)$, T is a regular WRTI for C from F with $C \subseteq \overline{V_H^0}$.

Because also $|T| \leq 2n^2(s+2)$, $F' \subseteq F \cup ICl(T)$ and $Var(T) \cap Var(V_H^0) = \varnothing$, T is a RTI as wanted.   ∎

Theorem 3.5.9 Let F be an unsatisfiable CNF formula, $s \geq 0$ and $n = \min\{|Var(F)|, s+1\}$. If there exists an execution of DLL-L-UP$(F, (\varnothing, \varnothing)\varnothing)$ which performs s recursive calls then there is a regular weak resolution tree with input lemmas for $\square$ from F of size at most $2n^2(s+2)$.

PROOF By Lemma 3.5.8 there is clause C with $C|_\varnothing = 0$ that has a regular WRTI of size $2n^2(s+2)$. But $C = \square$.   ∎

Now the proof of Lemma 2.2.5 on page 23 in Section 2.2 follows directly from Lemma 3.5.8.

Proof of Lemma 2.2.5 Let F be a CNF formula, H a UP-graph for F and α an assignment. Let DLL-L-UP(F, H, α) return (F', UNSAT).

By Lemma 3.5.8 there is WRTL T for a C with $C|_\alpha = 0$ from F. Thus it follows from Lemma 3.3.8 $F \equiv F \cup \{C\}$.

Suppose for the purpose of contradiction that β is a satisfying assignment for $F|_\alpha$. Then one can assume w.l.o.g. that $\alpha \subseteq \beta$. But then $\beta \models F$ (by Lemma 1.2.7 (a)) and $\beta \nvDash F \cup \{C\}$ and this is a contradiction to $F \equiv F \cup \{C\}$.   ∎

As mentioned above Theorem 3.5.9 can be used to proof the fact that DLL-LEARN can simulate DLL-L-UP.

Theorem 3.5.10 Let F be an unsatisfiable CNF formula, $s \geq 0$ and $n = \min\{|\mathit{Var}(F)|, s+1\}$. If there exists an execution of DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ which performs s recursive calls then there is an execution of DLL-LEARN$(F, \varnothing)$ that performs at most $2n^2(s+2)$ recursive calls.

PROOF If DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ returns $(F', \mathtt{UNSAT})$ then it follows from Theorem 3.5.9 that there is a regular weak resolution tree with input lemmas for $\square$ from F of size at most $2n^2(s+2)$. But then holds by Theorem 3.4.4 (a) that is an execution of DLL-LEARN(F, α) that performs $2n^2(s+2)-1$ recursive calls.

If otherwise DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ returns (F', α') for an assignment α' then DLL-LEARN can simulate DLL-L-UP by selecting all variables $x \in \mathit{dom}(\alpha')$ together with the values $\epsilon_x = \alpha'(x)$ as branching variables in an arbitrary order. DLL-LEARN$(F, \varnothing)$ performs then $|\mathit{dom}(\alpha')| \leq |\mathit{Var}(F)|$ recursive calls. $\blacksquare$

Another way to simulate DLL-L-UP with DLL-LEARN if DLL-L-UP$(F, (\varnothing, \varnothing), \varnothing)$ returns (F', α') for an assignment α' would be to select variables as branching variables in the same order as the variables are selected in the execution of DLL-L-UP (for unit propagation or as branching variables) and to keep every learned clause. A proof can be given with the help of Lemma 3.5.8 and Lemma 3.4.2.

Chapter 4

Dag-Like Resolution

4.1 Resolution Dags, RTL and Regularity

In Chapter 3 resolution proofs have been defined to be trees. A different view on
a resolution proof is to define it as a dag and, as mentioned earlier in Chapter
3, this does not differ from the notion of a resolution tree with lemmas in
terms of proof complexity. Particularly it is shown in this section that every
resolution dag can be transformed into a RTI with linear blow-up of size and
that every RTL can be transformed into a resolution dag of the same or smaller
size. Moreover, every resolution dag that uses the weak resolution rule can be
transformed into a resolution dag that uses the ordinary resolution rule only.
This means that the proof systems RD and RTL are equivalent.

On the other side, the concept of regularity seems to differ in the proof sys-
tems resolution dags, RTL, WRTL, RTI and WRTI and many of the questions
concerning simulations and separations between these systems are open.

To begin with, the definition of a resolution dag is given. The soundness as well
as the completeness of the corresponding proof system follow from Corollary
4.1.6.

Definition ((Weak) Resolution Dag) A *(weak) resolution dag* ((W)RD)
for a clause C from a CNF formula F is a dag $G = (V, E)$ such that the
vertices are labeled with clauses and the edges are labeled with variables, C is
the label of a vertex $v_0 \in V$ and for every $v \in V$ exactly one of the following
statements holds.

- v has in-degree 0 and v is labeled with a clause $D \in F$.

- v has in-degree 2 and the edges between v and its predecessors u_0 and
  u_1 are labeled with a variable x and $D_0, D_1 \vdash_x D$ ($D_0, D_1 \vdash_x^w D$) for the
  labels D of v, D_0 of u_0 and D_1 of u_1.

The vertices and their labels are often identified.

Let x be a variable. G is *x-regular* iff every path in G contains at most one edge that is labeled with x. G is *regular* iff G is x-regular for every variable $x \in \mathcal{V}$.

The *size* $|G|$ of G is defined to be the number of vertices $|V|$ in G.

$Cl(G)$ is the set of clauses in G, i.e., $Cl(G) = \{C \mid \text{there is a } v \in V \text{ that is labeled with } C\}$.

Definition Let F be a formula. The proof system RD(regRD) is defined through: $RD(F,G)$ ($regRD(F,G)$) if and only if G is a (regular) resolution dag for $\square$ from $CNF(\neg F)$.

The proof systems WRD and regWRD are defined analogously.

The next lemma shows how to transform a resolution dag G into a RTI of size $O(|G|^2)$. If G is regular then the constructed RTI is also regular. Recall the definition of the depth of a vertex on page 19.

Lemma 4.1.1 Let F be a CNF formula, C a clause and G a resolution dag of size s for C from F. Then there is a RTI T of size $2s \cdot d + 1$ for C from F where $d < s$ is the maximal depth of a vertex in G. If G is regular then T is also regular.

PROOF The construction of T is performed in two steps. At first G is unfolded to a possibly exponentially big resolution tree T' in which every node has the same derivation as in G. In the second step T' is reduced to a RTI T so that every vertex v of G occurs at most $Depth_G(v)$ times as an inner node of T.

Let $G = (V,E)$.

Step 1: Let $v \in V$ be a vertex with label D. The resolution tree T_v for D from F is defined recursively on $Depth_G(v)$. Every path from a leaf to the root of T_v corresponds to a path in G that ends in v and since one needs to keep track of the vertices of G, every node in T_v is labeled with a clause and a vertex $u \in V$.

If $Depth_G(v) = 0$ then T_v is the tree that consists of the single node (D,v).

If $Depth_G(v) = d > 0$ then v has two predecessors u_0 and u_1 with labels D_i and $Depth_G(u_i) < d$ for $i = 0$ and $i = 1$. Let x be the label of the edges (u_0,v) and (u_1,v). T_v is then the tree that arises out of T_{u_0} and T_{u_1} by adding the new root (D,v) and edges labeled with x from (D_0,u_0) and (D_1,u_1) to (D,v). Since every path in G that ends in u_0 or in u_1 can be extended to a path that ends in v, every path from a leaf to (D,v) in T_v corresponds to a path in G.

Finally one gets a resolution tree $T' = T_{v_0}$ for C from F. Since every path in T' corresponds to a path in G, T' is regular if G is regular.

Step 2: By deleting redundant subtrees, T' is now iteratively transformed into a RTI T. Therefore, subtrees $T_1, \ldots, T_s$ of T' are constructed such that T_i is a

RTI for C from F that has the following property: If $v \in V$, $i > Depth_G(v)$ and a is an inner node of T_i that is labeled with v then $|\{b <_{T_i} a \mid b$ is labeled with $v\}| < Depth_G(v)$,i.e., at most the first $Depth_G(v)$ nodes of T_i that are labeled with v are inner nodes.

Note that every inner node of T' that is labeled with a vertex $v \in V$ has always two predecessors that are labeled with fixed $u_0 \in V$ and $u_1 \in V$.

To begin with put $T_1 = T'$. From the construction of T' it follows that every $v \in V$ with $Depth_G(v) = 0$ occurs only as a label of the leaves in T'. Thus T_1 is as required.

Let now $1 \leq i < s$ and let T_i be already constructed. Define T_{i+1} to be the RTI that arises out of T_i by deleting the subtree above every node a labeled with (D, v) if there is an input node $b <_{T_i} a$ in T_i that is labeled with the clause D.

To show the induction statement for T_{i+1} let $v \in V$ be a vertex and let u_0 and u_1 be the predecessors of v in G. If $Depth_G(v) < i$ then by induction at most the first (with respect to $<_{T_i}$) $Depth_G(v)$ nodes that are labeled with v are inner nodes in T_i and, since there are only nodes deleted in the construction of T_{i+1}, the same applies to T_{i+1}. If $Depth_G(v) = i$ then every inner node of T_i that is labeled with v has always predecessors that are labeled with u_0 and u_1. Because $Depth_G(u_0) < i$ and $Depth_G(u_1) < i$ it follows for $k = 0$ and $k = 1$ that at most the first $i - 1$ nodes that are labeled with u_k are inner nodes of T_i. Therefore at last the ith node in T_i that is labeled with v is the predecessor of two leaves and hence an input node. Thus it follows from the construction of T_{i+1} that at most the first i nodes of T_{i+1} that are labeled with v are inner nodes.

Finally put $T = T_s$. Since $Depth_G(v) \leq s$ for each $v \in V$ it follows $|T_s| \leq 2 \cdot (\sum_{v \in V} Depth_G(v)) + 1 \leq s \cdot d + 1$ with $d = \max\{ Depth_G(v) \mid v \in V \}$ and thus T is a RTI as wanted.

Because T is a subtree of T', T is regular if G is regular. ∎

Note that it is impossible to construct a regular RTI (or a regular RTL) for C from a non-regular resolution dag G for C with the same type of construction as in the above proof. If G contains a clause D that is obtained by resolving on x and x is for example used to resolve the final clause C then D can not occur as lemma in a construction that is similar to the above.

It is easy to transform a WRTL into a weak resolution dag. Just replace every lemma in the WRTL by an edge from the corresponding inner node that is labeled with the lemma to the successor of the lemma. The regularity of a WRTL (or a RTL) is not preserved by that construction.

Lemma 4.1.2 Let F be a formula and let C be a clause. If C has a WRTL T of size s from F then C has a weak resolution dag G of size at most s from F. If T is a RTL then G is a resolution dag.

PROOF To obtain the weak resolution dag G from the WRTL T, all leaves D of T with $D \notin F$ are deleted in the following way (see Figure 4.1).

Let D_0 be a leaf of T with $D_0 \notin F$. Let D be the successor of D_0, let D_1 be the sibling of D_0 and let x be the label of the edges between D_i and D, i.e., $D_0, D_1 \vdash_x^w D$. Then T contains a node $v <_T D_0$ with $v = D_0$. Delete D_0 and add an edge from v to D.

After all the leaves $D_0 \notin F$ are deleted, one has transformed T into a weak resolution dag for C from F.

If T is a RTL then only the resolution rule is used in G and thus G is a resolution dag in that case.   ∎
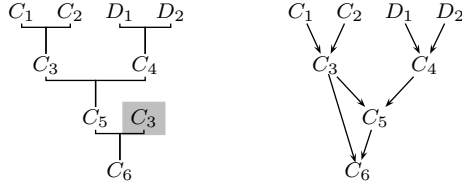

Figure 4.1: A RTL for C_6 with a lemma C_3 and the corresponding resolution dag for C_6.

The regularity of a RTL T is not preserved by the construction in the prior proof. The reason is that every variable can be used on a path from a leaf to the root in T. Thus one can create a non-regular path by adding new edges as above.

At next it is proved that the weak resolution rule is not needed in a resolution dag: every weak resolution dag for a clause C can be transformed into a resolution dag of the same or smaller size for a clause $C' \subseteq C$. The regularity of the dag is preserved by the construction.

Lemma 4.1.3 Let F be a CNF formula and let C be a clause. If C has a weak resolution dag G of size s from F then there is a clause $C' \subseteq C$ that has a a resolution dag G' of size at most s from F.

If G is regular then G' is regular.

PROOF By induction on $s = |G|$. Let $G = (V, E)$.

Induction basis: $s = 1$. Then G consists of one single vertex and is therefore already a regular weak resolution dag.

Induction step: $s > 1$. If G is already a resolution dag then there is nothing to show. Otherwise there are vertices D_0, D_1 and D in V with edges $(D_0, D) \in E$ and $(D_1, D) \in E$ that are labeled with x such that $D_0, D_1 \vdash_x^w D$ but $D_0, D_1 \nvdash_x D$. Then a resolution dag $\tilde{G}$ for C from F is constructed from G as follows.

Let w.l.o.g. $D = D_0 \cup (D_1 - \{\bar{x}\})$, i.e., $x \notin D_0$. Then remove D_1 and replace D by D_0. Furthermore replace successively every successor B of D with $(B_0, B) \in E$, $(B_1, B) \in E$ and $B = (B_0 - \{y\}) \cup (B_1 - \{\neg y\})$ with $B' = (B'_0 - \{y\}) \cup (B_1 - \{\neg y\})$, $B' = (B_0 - \{y\}) \cup (B'_1 - \{\neg y\})$ or $B' = (B'_0 - \{y\}) \cup (B'_1 - \{\neg y\})$, respectively, depending on whether B_0, B_1 or B_0 and B_1 are successors of D_0 (or D_0 itself).

$\tilde{G}$ is then a weak resolution dag for a clause $\tilde{C} \subseteq C$ and $\tilde{G}$ is regular if G is regular. Because $|\tilde{G}| < |G|$, $\tilde{G}$ can be transformed by induction into a resolution dag G' for a clause $C' \subseteq \tilde{C} \subseteq C$ with $|G'| < |G|$. G' is regular if $\tilde{G}$ is regular and the latter follows if G is regular. ∎

The following two theorems summarize Lemma 4.1.1, Lemma 4.1.2 and Lemma 4.1.3.

Theorem 4.1.4 $\mathtt{RD} \equiv \mathtt{WRD} \equiv \mathtt{RTI} \equiv \mathtt{WRTI} \equiv \mathtt{RTL} \equiv \mathtt{WRTL}$

PROOF $\mathtt{WRD} \leq \mathtt{RD}$ follows from Lemma 4.1.3, $\mathtt{RD} \leq \mathtt{RTI}$ is shown by Lemma 4.1.1, $\mathtt{RTI} \leq \mathtt{WRTI}$ and $\mathtt{WRTI} \leq \mathtt{WRTL}$ follow by definition and $\mathtt{WRTL} \leq \mathtt{WRD}$ from Lemma 4.1.2.

Finally $\mathtt{RTL} \leq \mathtt{RD}$ is proved by Lemma 4.1.2 and $\mathtt{RD} \leq \mathtt{RTL}$ follows from Lemma 4.1.1 since $\mathtt{RTI} \leq \mathtt{RTL}$. ∎

Theorem 4.1.5 $\mathtt{regRD} \equiv \mathtt{regWRD} \leq \mathtt{regRTI} \leq \mathtt{regRTL} \leq \mathtt{regWRTL} \leq \mathtt{RD}$ and $\mathtt{regRTI} \leq \mathtt{regWRTI} \leq \mathtt{regWRTL}$.

PROOF $\mathtt{regRD} \leq \mathtt{regWRD}$ follows from the definition and $\mathtt{regWRD} \leq \mathtt{regRD}$ from Lemma 4.1.3.

$\mathtt{regRD} \leq \mathtt{regRTI}$ is proved by Lemma 4.1.1. $\mathtt{regRTI} \leq \mathtt{regRTL} \leq \mathtt{regWRTL}$ as well as $\mathtt{regRTI} \leq \mathtt{regWRTI} \leq \mathtt{regWRTL}$ follow directly from the definitions and $\mathtt{regWRTL} \leq \mathtt{RD}$ from Lemma 4.1.2 and Lemma 4.1.3. ∎

In Section 4.3 a proof for $\mathtt{regRD} < \mathtt{regWRTI}$ is given and this is the only separation in the hierarchy that is known, i.e., the questions whether $\mathtt{regWRD} < \mathtt{regRTI}$, $\mathtt{regRTI} < \mathtt{regRTL}$, $\mathtt{regRTL} < \mathtt{regWRTL}$, $\mathtt{regWRTL} < \mathtt{RD}$ and $\mathtt{regWRTI} < \mathtt{regWRTL}$ are open.

However, it is proved in Section 4.3 that the proof system $\mathtt{regWRTI'}$ which is based on regular WRTI and *variable extensions* is equivalent to $\mathtt{RD}$.

It is also unknown if $\mathtt{regWRTI}$ and $\mathtt{regRTL}$ are comparable or not.

Theorem 4.1.4 also proves Theorem 3.3.6 on page 40 and with Corollary 3.3.7 the next corollary follows.

Corollary 4.1.6 The proof systems $\mathtt{RD}$ and $\mathtt{WRD}$ are sound and complete.

4.2   Known Separations and a Lower Bound for Resolution

In this short section an exponential lower bound for resolution as well as separations between `RD`, `regRD` and `RT` are presented without proofs. Proofs for the results can be found for example in [30].

The first exponential lower bound for resolution was proved by Armin Haken 1985 in [22]. It is based on the pigeonhole principle that formalizes the fact that there is no total one-to-one mapping from the set $\{1, \ldots, m\}$ into the set $\{1, \ldots, n\}$ if $m > n > 1$. The pigeonhole principle can be formalized in propositional logic as follows.

Definition (Pigeonhole Principle) Let $m > n > 1$. The CNF formula PHP_n^m uses the variables $p_{i,j}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ with the intended meaning of putting the pigeon i into the hole j (i.e., mapping i to j) iff $\alpha(p_{i,j}) = 1$ for an assignment α. PHP_n^m consists of the following clauses:

$$P_i = \bigvee_{1 \leq j \leq n} p_{i,j} \qquad\qquad \text{for every } i \in \{1, \ldots m\}$$

$$H_{i,j,k} = \bar{p}_{i,k} \vee \bar{p}_{j,k} \qquad \text{for every } 1 \leq i < j \leq m \text{ and } k \in \{1, \ldots n\}$$

The pigeon clauses P_i state that every pigeon is put into a hole (totality) and the hole clauses $H_{i,j,k}$ state that not more then one pigeon is put into a hole k (one-to-one).

Lemma 4.2.1 states that PHP_n^{n+1} has resolution refutations that are exponential in n. A proof is given in [11].

Lemma 4.2.1 (Buss-Pitassi [11]) There exists a resolution dag G_n for $\square$ from PHP_n^{n+1} of size $|G_n| = O(n^3 2^n)$.

Corollary 4.2.2 The CNF formula PHP_n^{n+1} is unsatisfiable for every $n > 0$.

The proof of Theorem 4.2.3 is based on a technique that is called *bottleneck counting*.

Theorem 4.2.3 (Haken [22], Beame-Pitassi [3]) If G is a resolution dag for $\square$ from PHP_{n-1}^n then $|G| \geq 2^{n/20}$.

The next theorem follows from Theorem 4.2.3, Theorem 4.1.5, Theorem 3.2.5, Theorem 3.4.4 and Theorem 3.5.9.

Theorem 4.2.4 Every execution of $\mathrm{DLL}(\mathrm{PHP}_{n-1}^n, \varnothing)$, DLL-LEARN$(\mathrm{PHP}_{n-1}^n, \varnothing)$ and DLL-L-UP$(\mathrm{PHP}_{n-1}^n, (\varnothing, \varnothing)\varnothing)$ performs at least $2^{n/20}$ recursive calls.

There are different formulas that show $\mathtt{RT} < \mathtt{regRD}$. Here the ordering principle OP_n that has been defined as an example on page 6 in Section 1.2 is used. Lemma 4.2.5 states that OP_n has regular resolution dags of polynomial size.

Lemma 4.2.5 (Bonet-Galesi [10]) There exists a regular resolution dag G_n for $\Box$ from OP_n of size $O(n^3)$.

On the other hand every resolution tree for OP_n has an exponential size in n. The following theorem is proved in [30].

Theorem 4.2.6 There is an integer N such that for every $n \geq N$ a resolution tree $|T_n|$ for $\Box$ from OP_n has size $|T_n| \geq 2^{\frac{n}{2}-1}$.

From Lemma 4.2.5 and Theorem 4.2.6 one derives the separation.

Corollary 4.2.7 $\mathtt{RT} < \mathtt{regRD}$

The results above together with Theorem 3.2.5 and Theorem 3.4.4 imply Theorem 4.2.8.

Theorem 4.2.8 There is an execution of DLL-LEARN($\mathrm{OP}_n, \varnothing$) that performs $O(n^3)$ recursive calls but every execution of DLL($\mathrm{OP}_n, \varnothing$) performs $\Omega(2^{\frac{n}{2}-1})$ recursive calls.

$\mathtt{regRD} < \mathtt{RD}$ has been shown first in [21]. Here, an exponential separation that has been given by of Alekhnovich et al. in [1] is presented. It is based on a modified ordering principle $\mathrm{OP}'_{n,\rho}$.

Definition (Modified Ordering Priciple) Let n be an integer. Define S to be the set $S := \{(i,j,k) \mid i,j,k \in \{1,\ldots n\}, i \neq j \neq k\}$ and let ρ be a function $\rho : S \rightarrow Var(\mathrm{OP}_n) = \{ x_{ij} \mid i,j \in \{1,\ldots,n\}, i \neq j \}$. Then the *modified ordering principle* $\mathrm{OP}'_{n,\rho}$ for n and ρ consists of the following clauses.

$$
\begin{aligned}
(x_{ij} \vee x_{ji}), (\bar{x}_{ij} \vee \bar{x}_{ji}) \qquad & 1 \leq i < j \leq n \\
\bar{x}_{i_1 i_2} \vee \bar{x}_{i_2 i_3} \vee \bar{x}_{i_3 i_1} \vee \rho(i_1, i_2, i_3) \qquad & \text{for } \bar{x}_{i_1 i_2} \vee \bar{x}_{i_2 i_3} \vee \bar{x}_{i_3 i_1} \in \mathrm{OP}_n \\
\bar{x}_{i_1 i_2} \vee \bar{x}_{i_2 i_3} \vee \bar{x}_{i_3 i_1} \vee \neg\rho(i_1, i_2, i_3) \qquad & \text{for } \bar{x}_{i_1 i_2} \vee \bar{x}_{i_2 i_3} \vee \bar{x}_{i_3 i_1} \in \mathrm{OP}_n \\
\bigvee_{1 \leq k \leq n, k \neq j} x_{kj} \qquad & j \in \{1, \ldots, n\}
\end{aligned}
$$

Each transitivity axiom in OP_n has been replaced by two axioms in $\mathrm{OP}'_{n,\rho}$, one with the additional literal $\rho(i_1, i_2, i_3)$, the other with $\neg\rho(i_1, i_2, i_3)$.

As proved in [30], there are polynomial size resolution dags for $\mathrm{OP}'_{n,\rho}$.

Lemma 4.2.9 Let $n \in \mathbb{N}$ and let $\rho_n : \{(i,j,k) \mid i,j,k \in \{1,\ldots n\}, i \neq j \neq k\} \to \{ x_{ij} \mid i,j \in \{1,\ldots,n\}, i \neq j \}$. Then OP'_{n,ρ_n} has a resolution dag for $\square$ of size $O(n^3)$.

On the other side the following lower bound holds.

Theorem 4.2.10 (Alekhnovich et al, [1]) There is a $N \in \mathbb{N}$ such that for each $n \geq N$ there is a ρ_n such that every regular resolution dag for $\square$ from OP'_{n,ρ_n} has a size greater than $2^{n/200}$.

Together with Lemma 4.2.9 the next corollary follows from Theorem 4.2.10.

Corollary 4.2.11 `regRD` < `RD`

4.3   Variable Expansions

In this section the notion of a *variable expansion* of a CNF formula is introduced. On the one hand it is used to show that `regRD` < `regWRTI`. The proof is inspired by a construction of Beame et al. that has been used in [2] to separate regular resolution dags from the proof system that is defined by DLL with learning by unit propagation.

On the other hand, variable expansions are used to define a variant of the proof system `regWRTI` that simulates dag-like resolution. As an application it is shown that the algorithm schema DLL-LEARN can simulate general resolution.

First it is shown that every restriction $F|_\alpha$ of an unsatisfiable CNF formula F has regular resolution dags for $\square$ whose size is as most as big as the shortest regular resolution dag for $\square$ from F. A proof system with this property is called *natural*.

For the proof of Lemma 4.3.1 the following definition is helpful. Recall that a clause C is called *tautological* iff there is a variable x with $\{x, \neg x\} \subseteq C$.

Definition Let $G = (V, E)$ and H be resolution dags. A *path* in G is a sequence $(v_1, u_1), (v_2, u_2), \ldots (v_n, u_n)$ of edges $(v_i, u_i) \in E$ such that $u_i = v_{i+1}$ for $1 \leq i < n$.

Let $\pi = f_1, \ldots, f_k$ be a path in G and let $\pi' = e_1, \ldots e_n$ be a path in H. π' is a subpath of π iff there is a one-to-one function $F : \{e_1, \ldots, e_n\} \to \{f_1, \ldots, f_k\}$ such that $F(e_i)$ and e_i are labeled with the same variable for all $i \in \{1, \ldots n\}$.

Lemma 4.3.1 Let F be a CNF formula, C a clause and α an assignment. If F has a resolution dag G for C of size s then there is a a clause $C' \subseteq C|_\alpha$ and a resolution dag G' for C' from $F|_\alpha$ of size at most s.

If C is non-tautological then G' does not contain any tautological clauses. If G is regular then G' is regular.

PROOF Let $G = (V, E)$. To proof the lemma, graphs $G_0, \ldots G_d$ are inductively constructed such that $G_i = (V_i, E_i)$ has the following properties

(1) G_i is a resolution dag from $F|_\alpha$

(2) $|G_i| \leq d_i = |\{ v \in V \mid Depth_G(v) \leq i \}|$

(3) If a $v \in V$ with $Depth_G(v) \leq i$ is labeled with a D such that $D|_\alpha \neq 1$ then there is a $v' \in V_i$ that is labeled with a $D' \subseteq D|_\alpha$.

(4) Every path π' in G_i that ends in v' is a subpath of a path π in G that ends in v.

(5) If $(v_1, v_2) \in E_i$, D_1 is the label of v_1 and D_2 is the label of v_2 then $D_1 \nsubseteq D_2$.

To begin with, define G_0 to be the resolution dag that contains a leaf v' labeled with $D|_\alpha$ for every leaf v of G that is labeled with D such that $D|_\alpha \neq 1$. Then (1)-(5) apply trivially to G_0.

Let now G_i be already constructed. Then G_i can be extended to G_{i+1} in the following way. Choose one after another every $v \in V$ with $Depth_G(v) = i + 1$ that is labeled with a D such that $D|_\alpha \neq 1$. v has two predecessors u_0 and u_1 in G that are labeled with D_0 and D_1, respectively. By definition $Depth_G(u_i) \leq i$ and $D_0, D_1 \vdash_x D$ for the variable x that is the label of the edges (u_0, v) and (u_1, v). Thus, and because $D|_\alpha \neq 1$, it follows $D_0|_\alpha \neq 1$ or $D_1|_\alpha \neq 1$. Let w.l.o.g $D = (D_0 - \{x\}) \cup (D_1 - \{\neg x\})$. There are 2 different cases.

Case 1: $x \in dom(\alpha)$. Let w.l.o.g. $\alpha(x) = 0$. Then $D_0|_\alpha \neq 1$ since otherwise $D|_\alpha = 1$. By induction there is a vertex $u_0' \in V_i$ that is labeled with a clause $D_0' \subseteq D_0|_\alpha \subseteq D|_\alpha$. Hence one can put $v' = u_0'$ and so (3) and (4) apply trivially to v and v'.

Case 2: $x \notin dom(\alpha)$. By induction there is a vertex $u_0' \in V_i$ that is labeled with a clause $D_0' \subseteq D_0|_\alpha$ and a a vertex $u_1' \in V_i$ that is labeled with a clause $D_1' \subseteq D_1|_\alpha$. If $D_0' \subseteq D$ or $D_1' \subseteq D$ then put $v' = u_0'$ or $v' = u_1'$, respectively. Otherwise $x \in D_0'$ and $\neg x \in D_1'$. Then add a new vertex v' labeled with $D' = (D_0' - \{x\}) \cup (D_1' - \{\neg x\}) \subseteq D|_\alpha$ and edges (u_0', v) and (u_1', v) labeled with x. It follows immediately that (5) holds for the new edges and since (4) holds for u_0, u_0' and u_1, u_1', it also holds for v and v'.

After this construction it is clear that (1), (3), (4) and (5) apply to G_{i+1}. Furthermore there has been at most one vertex added to G_i for every $v \in V$ with $Depth_G(v) = i + 1$. Thus $|G_{i+1}| \leq d_i + |\{ v \in V \mid Depth_G(v) = i + 1 \}| = d_{i+1}$ and (2) holds also for G_{i+1}.

Finally G_d is a resolution dag for a clause $C' \subseteq C|_\alpha$ from $F|_\alpha$. By (4) it follows that G_d is regular if G is regular. Fix a vertex $v_0 \in V_d$ that is labeled with C'

and define G' to by the subgraph of G_d that arises out of G_d by deleting every vertex v from which v_0 is not reachable in G_d. G' is then a resolution dag for C' from $F|_\alpha$ that is regular if G is regular.

Let C be a non-tautological clause. Then C' is also non-tautological. Suppose, for the purpose of a contradiction that G' contains a tautological clause C_0. By construction of G' there is a path π from C_0 to C. There must be an edge $e = (D_0, D)$ in π such that D_0 is tautological and D is non-tautological. Let x be the label of e and D_1 the other predecessor of D, i.e., $D_0, D_1 \vdash_x D$. But then $\{x, \neg x\} \subseteq D_0$ and thus $D_1 \subseteq D$ which is a contradiction to (5).

Thus G' has no tautological clauses if C is non-tautological and is therefore as wanted.   ■

From Lemma 4.3.1 with $C = \square$, Corollary 4.3.2 follows and with $C = \square$ and $\alpha = \varnothing$ follows Corollary 4.3.3.

Corollary 4.3.2 Let F be a CNF formula and α an assignment. If F has (regular) resolution dag for $\square$ of size s then $F|_\alpha$ has a (regular) resolution dag for $\square$ of size at most s.

Corollary 4.3.3 Let F be a CNF formula. If F has a resolution dag for $\square$ of size s then F has also a resolution dag for $\square$ of size at most s that contains no tautological clauses.

The next goal of the section is to modify the CNF formulas $\text{OP}'_{n,\rho}$ such that every regular resolution dag for $\square$ has still at least $2^{n/200}$ vertices and that there is a polynomial size regular WRTI for the modified formulas.

Therefore *variable extensions* of CNF formulas are introduced. They are a simplification of the *proof trace extension* that has been defined in [2].

The idea is to use a resolution dag G for $\square$ from a formula F to add new clauses to F that allow to transfer G into a regular WRTI for $\square$ from the new formula F' such that the size of the RTI is polynomial in $|G|$. Since every added clause in F' contains a new variable positively there is an assignment α of the new variables such that $F'|_\alpha = F$. That is why it follows from Corollary 4.3.2 that F' has no smaller regular resolution dags than F. On the other side F' has small regular RTI if F has small resolution dags and therefore the proof trace extension of the modified ordering principle will lead to the separation `regRD < regWRTI`.

Definition (Variable Extension) Let F be a CNF formula and $|Var(F)| = n$.

The set of *extension variables* of F is $EVar(F) = \{q, p_1, \ldots, p_n\}$. Let w.l.o.g $Var(F) \cap EVar(F) = \varnothing$.

The *variable extension* of F is the CNF formula $VEx(F) = F \cup \{ \{q, \bar{l}\} \mid l \in C$ and $C \in F \} \cup \{p_1, \dots, p_n\}$.

The *extension assignment* according to G is the assignment $\alpha_{VEx(G)} = \{ (x, 1) \mid x \in EVar(F) \}$.

Note that $VEx(F)|_{\alpha_{VEx(G)}} = F$ and $|VEx(F)| \le |F| + 2|Var(F)| + 1 = O(|F|)$. The next two lemmas show that a resolution dag G for a clause C from a CNF F can be transformed into a regular WRTI for C from $VEx(F)$ with polynomial blow-up.

Lemma 4.3.4 Let F be CNF formula and let G be a resolution dag that contains no tautological clauses. Let $C \in Cl(G) - F$, $C \ne \square$ and let C_0 and C_1 be the predecessors of C in G. Then there exists a regular input resolution tree T_C for q from $VEx(F) \cup \{C_0, C_1\}$ with $C \in ICl(T_C)$, $|T_C| = 2(|C| + 1)$ and $Var(T_C) \subseteq Var(F)$.

PROOF Let $C = \{x_1, \dots, x_k\}$ and let x be the label on the edges (C_0, C) and (C_1, C). Then $C_0, C_1 \vdash_x C$. Since C_0 and C_1 are not tautological, $x \notin Var(C)$. Therefore the tree T_C with $Var(T_C) = \{x_1, \dots, x_k, x\}$ as given in Figure 4.2 is regular and hence as wanted. ∎
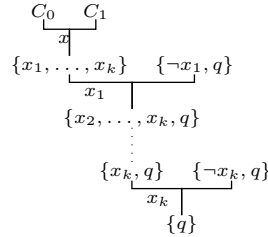

Figure 4.2: The regular input resolution tree T_C for $\{q\}$ that contains the clause $C = \{x_1, \dots, x_x\}$ from Lemma 4.3.4.

For the proof of Lemma 4.3.5 the notion of the *level* of a node in a tree is helpful. The proof is illustrated in Figure 4.3 on page 69.

Definition Let T be a binary tree and let v be a node of T.
The *level* of v is

$$Lev(v) = \begin{cases} 0 & \text{if } v \text{ is the root of } T \\ n+1 & \text{if } v \text{ is the child of the node } u \text{ with } Lev(u) = n \end{cases}$$

The *height* of T is $\max\{ Lev(v) \mid v \text{ is a node of } T \}$.

Note that the minimal height of a binary tree with n leaves is $\lceil \log_2(n) \rceil$.

Lemma 4.3.5 Let F be a CNF formula, $n = |Var(F)|$ and let C be a non-tautological clause. If F has a resolution dag G for C of size s then $VEx(F)$ has a regular WRTI T of size at most $4(d+1)s + 2$ for C where $d = \max\{ |D| \mid D \in Cl(G) \} \leq n$.

PROOF Let $G = (V, E)$. Since C is not tautological it can be assumed w.l.o.g by Corollary 4.3.3 that G contains no tautological clauses. Furthermore it can be assumed that for every clause D there is a most one $v \in V$ that is labeled with D. It follows from Corollary 3.2.6 and Theorem 4.1.5 that one can also assume that $s \leq 2^n$. Identify the vertices V with their labels.

Let $\{C_1, C_2, \ldots C_t\} = Cl(G) - F$ be an ordering of $Cl(G)$ such that $Depth_G(C_i) \leq Depth_G(C_j)$ if $i < j$. One can assume w.l.o.g that $C_t = C$ and $C_i \neq \square$ for $1 \leq i < t$. Let $EVar(G) = \{q, p_1, \ldots, p_n\}$.

Let T_0 be a binary tree of height $h = \lceil \log_2(t) \rceil \leq n$ with t leaves and $2t - 1$ nodes. Label every vertex in T_0 with the clause $\{q\}$ and label every edge that joins a node at level i and a node level $i+1$ with p_{i+1}. T_0 is then a regular WRTI for $\{q\}$ from $VEx(F) \cup \{q\}$ such that $Var(T_0) \subseteq EVar(F)$.

Let $v_1 <_{T_0} \ldots <_{T_0} v_t$ be the order of the leaves v_j of T_0 and let $1 \leq i \leq t$. By Lemma 4.3.4 there exists integers $1 \leq j, k < i$ and a regular input resolution tree T_{C_i} for q from $VEx(F) \cup \{C_j, C_k\}$ with $C \in ICl(T_{C_i})$, $|T_{C_i}| = 2(|C_i| + 1)$ and $Var(T_{C_i}) \subseteq Var(F)$.

Let T' be the regular WRTI for $\{q\}$ from $VEx(F)$ that arises out of T_0 by replacing the leaf v_j with the tree T_{C_j} for every $1 \leq j \leq t$. It follows that $|T'| \leq 2t + t \cdot 2(d+1) \leq 4s(d+1)$.

Define T to be the regular WRTI for C from $VEx(F)$ that arises out of T' by adding the new root C, a new leaf C and edges labeled with q from the root $\{q\}$ of T' and the leaf C to the new root C. See also Figure 4.3 on the facing page.

Since $C = C_t \in ICl(T')$ and $q \notin Var(T')$, T is a regular RTI of size at most $4(d+1)s + 2$ as required.   ∎

Lemma 4.3.5 is now applied to $F = OP'_{n,\rho}$ to separate regular resolution dags from `regWRTI`. It can be used analogous to separate every natural refutation of dag-like resolution from `regWRTI` if the natural refutation is separated from `RD`. Therefore one can separate *negative*, *semantic*, *tree-like* and *ordered* resolution from `regWRTI` in the same way. A proof of Lemma 4.3.1 for these refinements as well as a proof of the separations from dag-like resolution can be found in [30].

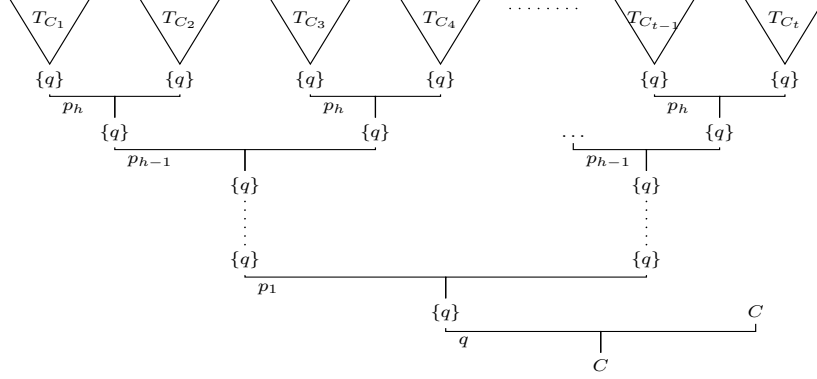**Theorem 4.3.6** `regRD` $<$ `regWRTI`

Figure 4.3: The regular RTI T for $C = C_t$ that is is constructed in Lemma 4.3.5 from a resolution dag G for C from F with $Cl(G) - F = \{C_1, \ldots, C_t\}$ and $h = \lceil \log_2(t) \rceil$. T contains the regular input trees T_{C_k} that are defined in Lemma 4.3.4.

PROOF Theorem 4.2.10 states that there is a $N \in \mathbb{N}$ such that for each $n \geq N$ there is a ρ_n such that every regular resolution dag for $\square$ from OP'_{n,ρ_n} has a size greater than $2^{n/200}$. Let G_n be a regular resolution dag for $\square$ from OP'_{n,ρ_n}. Let $n \geq N$. Suppose for the purpose of a contradiction that there is a resolution dag H_n for $\square$ from $VEx(\mathrm{OP}'_{n,\rho_n})$ with $|H_n| = s < 2^{n/200}$. Then it would follow from Corollary 4.3.2 that OP'_{n,ρ_n} has a regular resolution dag for $\square$ of size s, too. But this contradicts Theorem 4.2.10. Thus every regular resolution dag for $\square$ from $VEx(\mathrm{OP}'_{n,\rho_n})$ has a size greater than $2^{n/200}$ if $n \geq N$.

On the other side it follows from Lemma 4.2.9 that OP'_{n,ρ_n} has a resolution dag G_n for $\square$ of size $O(n^3)$. Therefore Lemma 4.3.5 states that $VEx(\mathrm{OP}'_{n,\rho_n})$ has regular WRTI for $\square$ of size $O(n^3 \cdot |Var(\mathrm{OP}'_{n,\rho_n})|) = O(n^5)$.

Since $|VEx(\mathrm{OP}'_{n,\rho_n})| = O(|\mathrm{OP}'_{n,\rho_n}|)$ this proofs `regRD` < `regWRTI`. ∎

Note that a similar proof of this separation also works to show that DLL-L-UP$(VEx(\mathrm{OP}'_{n,\rho_n}), (\varnothing, \varnothing), \varnothing)$ can run in polynomial time while the length of the smallest regular resolution dag for $\square$ from $VEx(\mathrm{OP}'_{n,\rho_n})$ is exponential in n. For details see [2].

The second application of Lemma 4.3.5 is Theorem 4.3.7 which shows that the algorithm schema DLL-LEARN can simulate dag-like resolution.

Theorem 4.3.7 Let F be a CNF formula and let G be a resolution dag of size s for $\square$ from F. Then there exists an execution of DLL-LEARN$(VEx(F), \varnothing)$ that performs less then $4s^2 + 1$ recursive calls.

PROOF It follows from Lemma 4.3.5 that $VEx(F)$ has a regular WRTI T of size at most $4(d+1)s+2$ for C where $d = \max\{\ |C| \ | \ C \in Cl(G)\ \} \le n$. Since T is a WRTI for $\square$, $d \le s$ and thus $|T| \le 4s^2 + 2$. But then it follows from Theorem 3.4.4 that there is an execution of DLL-LEARN($VEx(F), \varnothing$) that performs $4s^2 + 1$ recursive calls. ∎

It is also possible to define a proof system based on regular WRTI that is equivalent to dag-like resolution.

Definition (regWRTI′) Let F be a formula. The proof system regWRTI′ is defined through: regWRTI′(F,T) if and only if T is a regular WRTI for $\square$ from $VEx(CNF(\neg F))$.

Since $VEx(F)$ can be computed in polynomial time for a CNF formula F, regWRTI′ is decidable in polynomial time. The soundness and the correctness follow from Theorem 4.3.8 and Corollary 4.1.6.

Theorem 4.3.8 regWRTI′ ≡ RD

PROOF Let F be a CNF formula.

To show regWRTI′ $\le$ RD let T be a regular WRTI of size s for $\square$ from $VEx(F)$. Then it follows from Corollary 4.3.2 that $F = F|_{\alpha_{VEx(F)}}$ has a resolution dag for $\square$ of size at most s.

To prove RD $\le$ regWRTI′ let G be a resolution dag of size s for $\square$ from F. Then there exists a regular RTI of size $4s^2 + 2$ for $\square$ from $VEx(F)$ by Lemma 4.3.5. ∎

Another result that follows from Lemma 4.3.5 is that regWRTI (regWRTL) simulates RD iff regular WRTI (regular WRTL) are *natural*.

Definition Regular WRTI (regular WRTL) are *natural* if there is a polynomial p such that there exists a regular WRTI of size at most $p(|T|)$ for $\square$ from $F|_\alpha$ for every assignment α and for every CNF formula F that has a regular WRTI (regular WRTL) T for $\square$.

Theorem 4.3.9   (a) regWRTI ≡ RD if and only if regular WRTI are natural.

  (b) regWRTL ≡ RD if and only if regular WRTL are natural.

PROOF (a): If regWRTI ≡ RD then it follows from Corollary 4.3.2 that regular WRTI are natural.

Let regular WRTI be natural. Let F be a CNF formula and let G be a resolution dag of size s for $\square$ from F. Then it follows from Lemma 4.3.5 that $VEx(F)$ has a regular RTI T of size $O(s^2)$ for $\square$ (note that $\max\{\ |C| \ | \ C \in Cl(G)\ \} \le s$ because T is a RTI for $\square$). But since regular WRTI are natural and

$VEx(F)|_{\alpha_{VEx(G)}} = F$, F has a regular RTI for $\square$ of size $p(s^2)$ for a polynomial p. Thus $\texttt{RD} \leq \texttt{regWRTI}$.

$\texttt{regWRTI} \leq \texttt{RD}$ follows from Theorem 4.1.5.

(b): Since every regular RTI is also a regular RTL the proof is analogous to (a). ∎

Chapter 5

On the Representation of Resolution Proofs

5.1 Sequence-Like Resolution Proofs

In Chapter 3 resolution proofs are represented by binary trees and in Chapter 4 they are represented by dags. In this section *resolution sequences*, a third common representation of resolution proofs, are analyzed.

Definition (Resolution Sequence) Let F be a CNF formula and let C be a clause. A *resolution sequence* for C from F is a sequence $S = C_1, \ldots C_s$ such that for every $1 \leq k \leq s$ $C_k \in F$ or there are i and j with $1 \leq i, j < k$ such that $C_i, C_j \vdash C_k$.
The size of S is $|S| = s$.

It is easy to prove that resolution sequences are equivalent to resolution dags in terms of proof complexity.

Proposition 5.1.1 Let F be a formula and let C be a clause. F has a resolution sequence for C of size at most s if and only if F has a resolution dag of size at most s.

PROOF For the *if* part let $G = (V, E)$ be a resolution dag for C from F with $|V| \leq s$. Fix a vertex $v \in V$ that is labeled with C and let $G' = (V', E')$ be the dag that arises out of G by deleting every vertex $u \in V$ with $Depth_G(u) > Depth_G(v)$. G' is still a resolution dag for C from F. Let $\{v_1, \ldots, v_t\} = V$ be an ordering of V such that $v_t = v$ and $i < j$ if $Depth_G(v_i) < Depth_G(v_j)$. Let C_i be the label of v_i for $1 \leq i \leq t$. Then $C_1, \ldots, C_t$ is a resolution sequence for C of size $t \leq s$.

Let for the *only-if* part $S = C_1, \ldots, C_t$ be a resolution sequence for $C = C_t$ of size $t \leq s$. Let $V = \{v_1, \ldots, v_t\}$ be a set of vertices and label v_i with C_i for

every $i \in \{1, \ldots, t\}$. Let $i \in \{1, \ldots, t\}$ with $C_i \notin F$. Define $1 \leq i_0 < i$ and $1 \leq i_1 < i$ to be two numbers with $C_{i_0}, C_{i_1} \vdash C_i$ and let $E = \{(v_{i_0}, v_i), (v_{i_1}, v_i) \mid v_i \in V$ and $C_i \notin F\}$ be a set of edges. Then $G = (V, E)$ is a resolution dag for C from F.   ∎

Note that both constructions in the proof of Proposition 5.1.1 are non-deterministic. For a given resolution dag G one can receive multiple resolution sequences. On the other side, there are in general many resolution dags that can be constructed from a given resolution sequence. The reason is that there are in general different choices for the numbers i_0 and i_1 for a given i in the proof above. That is why it is not easy to define proof systems in terms of Section 1.3 by using resolution sequences. The author has been made aware of this problem by Nicolas Rachinsky.

5.2   Regular Resolution Sequences

Maybe there are other possible ways to define regularity for resolution sequences $S = C_1, \ldots, C_s$, but the probably most natural way to do it, is to call S *regular* if there is a regular resolution dag that arises out of S by taking the clauses C_i in S as vertices and letting there be directed edges from any non-initial clause to a pair of clauses from which it is inferred. It is shown in this section that it is NP-complete to decide whether a given resolution proof is regular in terms of this definition. This result is due to Sam Buss.

Definition Let F be a CNF formula and let $S = C_1, \ldots, C_s$ be a resolution sequence for $C_s = C$ from F. G is an *S-compatible dag* if $G = (V, E)$ is a resolution dag for C from F with $Cl(G) = \{C_1, \ldots, C_s\}$ such that $i < j$ for every edge $(C_i, C_j) \in E$.
S is *regular* if there exists an S-compatible dag that is regular.

Definition (regRS) Let F be a formula. The proof system regRS is defined through: regRS(F, S) if and only if S is a regular resolution sequence for $\square$ from $CNF(\neg F)$.

From Section 4.1 it follows that regRS is sound and complete. But it is shown below that it is NP-complete to decide regRS. Thus regRS is a proof system according to the definition in Section 1.3 if and only if P=NP.

Proposition 5.2.1 regRS is decidable in non-deterministic polynomial time.

PROOF Let F be a CNF formula and let S be a string. To decide regRS(F, S) proceed as follows. If S is not a sequence of clauses then return FALSE. Otherwise let $S = C_1, \ldots, C_s$. Let furthermore $V = \{v_1, \ldots, v_s\}$, label v_i with C_i and let $E = \varnothing$.

If there is a $C_k \notin F$ for which there are no $i, j < k$ with $C_i, C_j \vdash C_k$ then S is not a regular resolution sequence.

Otherwise choose non-deterministically for every v_k with $C_k \notin F$ two vertices v_i and v_j with $i, j < k$ such that $C_i, C_j \vdash_x C_k$ and add two edges $(v_i, v_k), (v_j, v_k)$ labeled with x to E. S is a regular resolution sequence if and only if $G = (V, E)$ is a regular resolution dag. ∎

To show that it is NP-hard to decide `regRS`, we use a reduction from the well known NP-complete problem *vertex cover*.

Definition (Vertex Cover) Let $G = (V, E)$ be an undirected graph. A *vertex cover* for G of size k is a set $U \subseteq V$ with $|U| = k$ such that $e \cap U \neq \varnothing$ for every $e \in E$.

The algorithmic problem *vertex cover* is to decide for a given undirected graph G and an integer k whether G has a vertex cover of size at most k.

Theorem 5.2.2 It is NP-hard to decide for a given resolution sequence whether it is regular.

PROOF Given $G = (V, E)$, we will construct a sequence-like refutation R. R will be regular if and only if G has a vertex cover of size $\leq k$. The refutation will have two stages: One stage makes sure that at most k vertices are pebbled, and the other makes sure that each edge is covered by at least one vertex. There will be propositional variables v_i that correspond to vertices $v_i \in V$. For each vertex v_i, R will have a derivation of the singleton clause $\{v_i\}$: this will be derived by a resolution either with a variable p_j (indicating that the i-th vertex is covered by the j-th pebble) or with a variable q_i (indicating that the i-th vertex is uncovered). In the second stage, R will use variables e_i that ensure that the i-th edge has at least one vertex pebbled.

The first part of R consists of the following clauses. As initial clauses, the first clauses in R are the singleton clauses $\{\overline{p}_j\}$ and $\{\overline{q}_i\}$, for $i = 1, \ldots, |V|$ and $j = 1, \ldots k$. There is also an initial clause $\{v_0\}$: here v_0 does not correspond to any vertex of V, but rather helps with the base case. Then, for each $i = 1, 2, \ldots, |V|$, R contains the $k+1$ initial clauses $\{\overline{v_{i-1}}, v_i, p_1\} \ldots \{\overline{v_{i-1}}, v_i, p_k\}$ and $\{\overline{v_{i-1}}, v_i, q_i\}$ followed the $k + 2$ non-initial clauses $\{v_i, p_1\} \ldots \{v_i, p_k\}$, $\{v_i, q_i\}$ and $\{v_i\}$. The clause $\{v_i\}$ can be derived by an inference of the form

$$\frac{\dfrac{\{v_{i-1}\} \qquad \{\overline{v_{i-1}}, v_i, x\}}{\{v_i, x\}} \qquad \{\overline{x}\}}{\{v_i\}}$$

where x is one of $p_1 \ldots p_k$ or q_i. There are $k + 1$ possible ways of deriving $\{v_i\}$, but the intent is that $\{v_i\}$ is derived with the aid of resolving on the variable p_j

as x if the j-th pebble is placed on the i-th vertex. If, however, resolution with q_i is used to derive $\{v_i\}$, then the i-th vertex is unpebbled. Since the clauses $\{v_i\}$ are derived sequentially, any dag-like regular resolution can use resolution with each fixed p_j only once: this corresponds to the fact that the j-th pebble can be placed on at most one vertex.

The second stage of R is designed so that it can be regular only if all edges have a pebbled vertex. Let the s-th edge in E join the i-th and i'-th vertices. Then, R includes two initial clauses $\{e_s, \overline{v_i}, q_i\}$ and $\{e_s, \overline{v_{i'}}, q_{i'}\}$, and the three non-initial clauses $\{e_s, q_i\}$, $\{e_s, q_{i'}\}$, and $\{e_s\}$. The intent is that, if the i-th vertex is pebbled, then the inference structure for these five clauses is as follows (using the the clauses $\{v_i\}$ and $\{v_{i'}\}$ derived in the first stage of R):

$$\frac{\dfrac{\{e_s, \overline{v_i}, q_i\} \qquad \{v_i\}}{\{e_s, q_i\}} \qquad \{\overline{q_i}\}}{\{e_s\}} \qquad\qquad\qquad \frac{\{e_s, \overline{v_{i'}}, q_{i'}\} \qquad \{v_{i'}\}}{\{e_s, q_{i'}\}}$$

Note that this inference structure leaves the clause $\{e_s, q_{i'}\}$ unused in R. If the i-th vertex was pebbled then the resolution on q_i that is used to derive $\{e_s\}$ is a regular inference, but otherwise it is not. If the i'-th vertex was pebbled then the intent is the inference structure in R should be as above but with the roles of i and i' interchanged.

The refutation R ends with one further initial clause $\{\overline{e_1}, \overline{e_2}, \ldots, \overline{e_{|E|}},\}$ and then concludes with resolution inferences using the clauses $\{e_i\}$, $i = 1, \ldots, |E|$, to obtain the empty clause.

It is not hard to verify that R can be be given a dag-structure that makes it regular if and only if the the graph G has a vertex cover of size k. First, because the clauses $\{v_i\}$ are derived sequentially from each other, there can be only one use of the resolution rule on a given variable $\{p_j\}$ in a regular proof. This corresponds to the fact that each of the k pebbles can be used only once. Second, each edge clause $\{e_s\}$ cannot be derived if both of the its endpoint clauses $\{v_i\}$ and $\{v_{i'}\}$ were derived using resolution on the clauses $\{q_i\}$ and $\{q_{i'}\}$ that indicate they are not pebbled. Thus, the refutation can be made regular if and only if the graph has a vertex cover of size k. ∎

From Theorem 5.2.2 and Proposition 5.2.1 the corollaries below follow immediately.

Corollary 5.2.3 It is NP-complete to decide `regRS`.

Corollary 5.2.4 `regRS` is a proof system if and only if P=NP.

Open Questions

There are several question left open for further research by this paper.

It has been shown in Chapter 3 that the algorithm schema DLL-LEARN can simulate DLL-L-UP, which is the basis of most of the fastest deterministic SAT-solvers. On the other hand it is not clear if it is possible to develop fast real-world SAT-solver that are based on the algorithm schema DLL-LEARN.

A related problem is the question whether there is a separation between the proof systems `regWRTL` and `regWRTI` which would show that there are CNF formulas that can be decided by DLL-LEARN in polynomial time while DLL-L-UP needs super-polynomial time. Furthermore it is open whether `regWRD` < `regRTI`, `regRTI` < `regRTL`, `regRTL` < `regWRTL`, `regWRTL` < `RD` and `regWRTI` < `regWRTL`.

Finally, it is unclear whether the algorithm schema DLL-L-UP can simulate regular weak RTI or regular weak RTL. It seems to be necessary to include some new features in DLL-L-UP to get such a simulation and there is work in progress that will properly appear in a separate paper.

Bibliography

[1] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *STOC*, pages 448–456, 2002.

[2] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[3] Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *FOCS*, pages 274–282, 1996.

[4] Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present and future. In *Current Trends in Theoretical Computer Science*, pages 42–70. 2001.

[5] Arnold Beckmann and Jan Johannsen. *Bounded Arithmetic and Resolution-Based Proof Systems*, volume 7 of *Collegium Logicum*. Kurt Gödel Society, 2004.

[6] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2):149–169, 2001.

[7] Daniel Le Berre and Laurent Simon. The essentials of the sat 2003 competition. In *SAT*, pages 452–467, 2003.

[8] Daniel Le Berre and Laurent Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *SAT (Selected Papers)*, pages 321–344, 2004.

[9] Daniel Le Berre and Laurent Simon. Preface to the special volume on the sat 2005 competitions and evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, 2005.

[10] Maria Luisa Bonet and Nicola Galesi. Optimality of size-width tradeoffs for resolution. *Computational Complexity*, 10(4):261–276, 2001.

[11] Samuel R. Buss and Toniann Pitassi. Resolution and the weak pigeonhole principle. In *CSL*, pages 149–156, 1997.

[12] C. L. Chang. The unit proof and the input proof in theorem proving. *J. ACM*, 17(4):698–707, 1970.

[13] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[14] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[15] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[16] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[17] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[18] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[19] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[20] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Verlag L. Nebert, Halle/Saale, 1879.

[21] Andreas Goerdt. Regular resolution versus unrestricted resolution. *SIAM J. Comput.*, 22(4):661–683, 1993.

[22] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[23] Jacques Herbrand. *Recherches sur la théorie de la Démonstration.* PhD thesis, École Normale Supérieure Paris, 1930.

[24] Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman. Dynamic restart policies. In *AAAI/IAAI*, pages 674–681, 2002.

[25] Reinhold Letz. Unpublished note, 2001.

[26] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In *SAT (Selected Papers)*, pages 360–375, 2004.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, 2001.

[28] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University of Jerusalem, Israel, 2002.

[29] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[30] Nicolas Rachinsky. The complexity of resolution refinements and satisfiability algorithms. Diplomarbeit, LMU München, 2007.

[31] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[32] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[33] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

[34] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.