Resolution Trees with Lemmas: Resolution Refinements that Characterize DLL Algorithms with Clause Learning

Samuel R. Buss*
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093-0112, USA
sbuss@math.ucsd.edu

Jan Hoffmann†
Institut für Informatik
Ludwig-Maximilians Universität
D-80538 München, Germany
hoffmann@cip.ifi.lmu.de

Jan Johannsen
Institut für Informatik
Ludwig-Maximilians Universität
D-80538 München, Germany
jan.johannsen@ifi.lmu.de

October 31, 2008

Abstract

Resolution refinements called *w-resolution trees with lemmas* (WRTL) and with *input lemmas* (WRTI) are introduced. Dag-like resolution is equivalent to both WRTL and WRTI when there is no regularity condition. For regular proofs, an exponential separation between regular dag-like resolution and both regular WRTL and regular WRTI is given.

It is proved that DLL proof search algorithms that use clause learning based on unit propagation can be polynomially simulated by regular WRTI. More generally, non-greedy DLL algorithms with learning by unit propagation are equivalent to regular WRTI. A general form of clause learning, called DLL-Learn, is defined that is equivalent to regular WRTL.

A variable extension method is used to give simulations of resolution by regular WRTI, using a simplified form of proof trace extensions. DLL-Learn and non-greedy DLL algorithms with learning by unit propagation can use variable extensions to simulate general resolution without doing restarts.

Finally, an exponential lower bound for WRTL where the lemmas are restricted to short clauses is shown.

*Supported in part by NSF grants DMS-0400848 and DMS-0700533.
†Supported in part by the Studienstiftung des deutschen Volkes (German National Merit Foundation).

1 Introduction

Although the satisfiability problem for propositional logic (SAT) is NP-complete, there exist SAT solvers that can decide SAT on present-day computers for many formulas that are relevant in practice [23, 21, 20, 4, 5, 6]. The fastest SAT solvers for structured problems are based on the basic backtracking procedures known as DLL algorithms [10], extended with additional techniques such as clause learning.

DLL algorithms can be seen as a kind of proof search procedure since the execution of a DLL algorithm on an unsatisfiable CNF formula yields a tree-like resolution refutation of that formula. Conversely, given a tree-like resolution refutation, an execution of a DLL algorithm on the refuted formula can be constructed whose runtime is roughly the size of the refutation. By this exact correspondence, upper and lower bounds on the size of tree-like resolution proofs transfer to bounds on the runtime of DLL algorithms.

This paper generalizes this exact correspondence to extensions of DLL by clause learning. To this end, we define natural, rule-based resolution proof systems and then prove that they correspond to DLL algorithms that use various forms of clause learning. The motivation for this is that the correspondence between a clause learning DLL algorithm and a proof system helps explain the power of the algorithm by giving a description of the space of proofs which is searched by it. In addition, upper and lower bounds on proof complexity can be transferred to upper and lower bounds on the possible runtimes of large classes of DLL algorithms with clause learning.

We introduce, in Section 3, tree-like resolution refinements using the notions of a resolution tree with lemmas (RTL) and a resolution tree with input lemmas (RTI). An RTL is a tree-like resolution proof in which every clause needs only to be derived once and can be copied to be used as a leaf in the tree (i.e., a lemma) if it is used several times. As the reader might guess, RTL is polynomially equivalent to general resolution.

Since DLL algorithms use learning based on unit propagation, and since unit propagation is equivalent to input resolution (sometimes called "trivial resolution" [3]), it is useful to restrict the lemmas that are used in a RTL to those that appear as the root of input subproofs. This gives rise to proof systems based on resolution trees with input lemmas (RTI). Somewhat surprisingly, we show that RTI can also simulate general resolution.

A resolution proof is called *regular* if no variable is used as a resolution variable twice along any path in the tree. Regular proofs occur naturally in the present context, since a backtracking algorithm would never query the same variable twice on one branch of its execution. It is known that regular resolution is weaker than general resolution [15, 1], but it is unknown whether regular resolution can simulate regular RTL or regular RTI. This is because, in regular RTL/RTI proofs, variables that are used for resolution to derive a clause can be reused on paths where this clause appears as a lemma.

For resolution and regular resolution, the use of a weakening rule does not increase the power of the proof system (by the subsumption principle). However,

for RTI and regular RTL proofs, the weakening rule may increase the strength of the proof system (this is an open question, in fact), since eliminating uses of weak inferences may require pruning away parts of the proof that contain lemmas needed later in the proof. Accordingly, Section 3 also defines proof systems regWRTL and regWRTI that consist of regular RTL and regular RTI (respectively), but with a modified form of resolution, called "w-resolution", that incorporates a restricted form of the weakening rule.

In Section 4 we propose a general framework for DLL algorithms with clause learning, called DLL-L-UP. The schema DLL-L-UP is an attempt to give a short and abstract definition of modern SAT solvers and it incorporates all common learning strategies, including all the specific strategies discussed by Beame et al. [3]. Section 5 proves that, for any of these learning strategies, a proof search tree can be transformed into a regular WRTI proof with only a polynomial increase in size. Conversely, any regular WRTI proof can be simulated by a "non-greedy" DLL search tree with clause learning, where by "non-greedy" is meant that the algorithm can continue decision branching even after unit propagation could yield a contradiction.

In Section 6 we give another generalization of DLL with clause learning called DLL-Learn. The algorithm DLL-Learn can simulate the clause learning algorithm DLL-L-UP. More precisely, we prove that DLL-Learn p-simulates, and is p-simulated by, regular WRTL. The DLL-Learn algorithm is very similar to the "pool resolution" algorithm that has been introduced by Van Gelder [25] but differs from pool resolution by using the "w-resolution" inference in place of the "degenerate" inference used by Van Gelder (the terminology "degenerate" is used by Hertel et al. [2]). Van Gelder has shown that pool resolution can simulate not only regular resolution, but also any resolution refutation which has a regular depth-first search tree. The latter proof system is the same as the proof system regRTL in our framework, therefore the same holds for DLL-Learn. It is unknown whether DLL-Learn or DLL-L-UP can p-simulate pool resolution or vice versa.

Sections 4-6 prove the equivalence of clause learning algorithms with the two proof systems regWRTI and regWRTL. Our really novel system is regWRTI: this system has the advantage of using input lemmas in a manner that closely matches the range of clause learning algorithms that can be used by practical DLL algorithms. In particular, the regWRTI proof system's use of input lemmas corresponds directly to the clause learning strategies of Silva and Sakallah [23], including first-UIP, relsat, and other clauses based on cuts, and including learning multiple clauses at a time. Van Gelder [25] shows that pool resolution can also simulate these kinds of clause learning (at least, for learning single clauses), but the correspondence is much more natural for the system regWRTI than for either pool resolution or DLL-Learn.

It is known that DLL algorithms with clause learning and restarts can simulate full (non-regular, dag-like) resolution by learning every derived clause, and doing a restart each time a clause is learned [3]. Our proof systems, regWRTI and DLL-Learn, do not handle restarts; instead, they can be viewed as capturing what can happen between restarts. Another approach to simu-

lating full resolution is via the use of "proof trace extensions" introduced by Beame et al. [3]. Proof trace extensions allow resolution to be simulated by clause learning DLL algorithms, and a related construction is used by Hertel et al. [2] to show that pool resolution can "effectively" p-simulate full resolution. These constructions require introducing new variables and clauses in a way that does not affect satisfiability, but allow a clause learning DLL algorithm or pool resolution to establish non-satisfiability. However, the constructions by Beame et al. [3] and the initially circulated preprint of Hertel et al. [2] had the drawback that the number of extra introduced variables depends on the size of the (unknown) resolution refutation.

Section 7 introduces an improved form of proof trace extensions called "variable extensions". Theorem 19 shows that variable extensions can be used to give a p-simulation of full resolution by regWRTI (at the cost of changing the formula that is being refuted). Variable extensions are simpler and more powerful than proof trace extensions. Their main advantage is that a variable extension depends only on the number of variables, not on the size of the (unknown) resolution proof. The results of Section 7 were first published in the second author's diploma thesis [17]; the subsequently published version of the article of Hertel et al. [2] gives a similarly improved construction (for pool resolution) that does not depend on the size of the resolution proof and, in addition, does not use degenerate resolution inferences.

One consequence of Theorem 19 is that regWRTI can effectively p-simulate full resolution. This improves on the results of Hertel et al. [2] since regWRTI is not known to be as strong as pool resolution. It remains open whether regWRTI or pool resolution can p-simulate general resolution without variable extensions.

Section 8 proves a lower bound that shows that for certain hard formulas, the pigeonhole principle PHP_n, learning only small clauses does not help a DLL-algorithm. We show that resolution trees with lemmas require size exponential in $n \log n$ to refute PHP_n when the size of clauses used as lemmas is restricted to be less than $n/2$. This bound is asymptotically the same as the lower bound shown for tree-like resolution refutations of PHP_n [18]. On the other hand, there are regular resolution refutations of PHP_n of size exponential in n [8], and our results show that these can be simulated by DLL-L-UP. Hence the ability of learning large clauses can give a DLL-algorithm a superpolynomial speedup over one that learns only short clauses.

2 Preliminaries

Propositional logic. Propositional formulas are formed using Boolean connectives $\neg$, $\wedge$, and $\vee$. However, this paper works only with formulas in conjunctive normal form, namely formulas that can be expressed as a set of clauses. We write $\overline{x}$ for the negation of x, and $\overline{\overline{x}}$ denotes x. A *literal* l is defined to be either a variable x or a negated variable $\overline{x}$. A clause C is a finite set of literals, and is interpreted as being the disjunction of its members. The empty clause is denoted $\Box$. A *unit* clause is a clause containing a single literal. A set F of

clauses is interpreted as the conjunction of its clauses, i.e., a conjunctive normal form formula (CNF).

An assignment α is a (partial) mapping from the set of variables to $\{0,1\}$, where we identify 1 with *True* and 0 with *False*. The assignment α is implicitly extended to assign values to literals by letting $\alpha(\overline{x}) = 1-\alpha(x)$, and the domain, $dom(\alpha)$, of α is the set of literals assigned values by α. The *restriction* of a clause C under α is the clause

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

The *restriction* of a set F of clauses under α is

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

If $F|_\alpha = 1$, then we say α *satisfies* F.

An assignment is called *total* if it assigns values to all variables. We call two CNFs F and F' *equivalent* and write $F \equiv F'$ to indicate that F and F' are satisfied by exactly the same total assignments. Note, however, that $F \equiv F'$ does not always imply that they are satisfied by the same partial assignments.

If $\epsilon \in \{0,1\}$ and x is a variable, we define x^ϵ by letting x^0 be x and x^1 be $\overline{x}$.

Resolution. Suppose that C_0 and C_1 are clauses and x is a variable with $x \in C_0$ and $\overline{x} \in C_1$. Then the *resolution rule* can be used to derive the clause $C = (C_0 \setminus \{x\}) \cup (C_1 \setminus \{\overline{x}\})$. In this case we write $C_0, C_1 \vdash_x C$ or just $C_0, C_1 \vdash C$.

A *resolution proof* of a clause C from a CNF F consists of repeated applications of the resolution rule to derive the clause C from the clauses of F. If $C = \Box$, then F is unsatisfiable and the proof is called a *resolution refutation*.

We represent resolution proofs either as graphs or as trees. A *resolution dag* (RD) is a dag $G = (V, E)$ with labeled edges and vertices satisfying the following properties. Each node is labeled with a clause and a variable, and, in addition, each edge is labeled with a literal. There must be a single node of out-degree zero, labeled with the conclusion clause. Further, all nodes with in-degree zero are labeled with clauses from the initial set F. All other nodes must have in-degree two and are labeled with a variable x and a clause C such that $C_0, C_1 \vdash_x C$ where C_0 and C_1 are the labels on the the two immediate predecessor nodes and $x \in C_0$ and $\overline{x} \in C_1$. The edge from C_0 to C is labeled $\overline{x}$, and the edge from C_1 to C is labeled x. (The convention that that $x \in C_0$ and $\overline{x}$ is on the edge from C_0 might seem strange, but it allows a more natural formulation of Theorem 4 below.)

A resolution dag G is *x-regular* iff every path in G contains at most one node that is labeled with the variable x. G is *regular* (or a regRD) if G is x-regular for every x.

We define the *size* of a resolution dag $G = (V, E)$ to be the number $|V|$ of vertices in the dag. $Var(G)$ is the set of variables used as resolution variables

in G. Note that if G is a resolution proof rather than a refutation, then $Var(G)$ may not include all the variables that appear in clause labels of G.

A *resolution tree* (RT) is a resolution dag which is tree-like, i.e., a dag in which every vertex other then the conclusion clause has out-degree one. A regular resolution tree is called a regRT for short.

The notion of (p-)simulation is an important tool for comparing the strength of proof systems. If $\mathcal{Q}$ and $\mathcal{R}$ are refutation systems, we say that $\mathcal{Q}$ *simulates* $\mathcal{R}$ provided there is a polynomial $p(n)$ such that, for every $\mathcal{R}$-refutation of a CNF F of size n there is a $\mathcal{Q}$-refutation of F of size $\leq p(n)$. If the $\mathcal{Q}$-refutation can be found by a polynomial time procedure, then this called a *p-simulation*. Two systems that simulate (resp, p-simulate) each other are called *equivalent* (resp, *p-equivalent*). Some basic prior results for simulations of resolution systems include:

Theorem 1.

a. [24] *Regular tree resolution (regRT) p-simulates tree resolution (RT).*

b. [15, 1] *Regular resolution (regRD) does not simulate resolution (RD).*

c. [7] *Tree resolution (RT) does not simulate regular resolution (regRD).*

Weakening and w-resolution. The *weakening* rule allows the derivation of any clause $C' \supseteq C$ from a clause C. However, instead of using the weakening rule, we introduce a *w-resolution* rule that essentially incorporates weakening into the resolution rule. Given two clauses C_0 and C_1, and a variable x, the *w-resolution rule* allows one to infer $C = (C_0 \setminus \{x\}) \cup (C_1 \setminus \{\overline{x}\})$. We denote this condition $C_0, C_1 \vdash_x^w C$. Note that $x \in C_0$ and $\overline{x} \in C_1$ are not required for the w-resolution inference.

We use the notations WRD, regWRD, WRT, and regWRT for the proof systems that correspond to RD, regRD, RT, and regRT (respectively) but with the resolution rule replaced with the w-resolution rule. That is, given a node labeled with C, an edge from C_0 to C labeled with $\bar{x}$ and an edge from C_1 to C labeled with x, we have $C = (C_0 \setminus \{x\}) \cup (C_1 \setminus \{\overline{x}\})$.

Similarly, we use the notations RDW and RTW for the proof systems that correspond to RD and RT, but with the general weakening rule added. In an application of the weakening rule, the edge connecting a clause $C' \supseteq C$ with its single predecessor C does not bear any label.

The resolution and weakening rules can certainly p-simulate the w-resolution rule, since a use of the w-resolution rule can be replaced by weakening inferences that derive $C_0 \cup \{x\}$ from C_0 and $C_1 \cup \{\overline{x}\}$ from C_1, and then a resolution inference that derives C. The converse is not true, since w-resolution cannot completely simulate weakening; this is because w-resolution cannot introduce completely new variables that do not occur in the input clauses. According to the well-known subsumption principle, weakening cannot increase the strength of resolution though, and the same reasoning implies the same about w-resolution; namely, we have:

Proposition 2. *Let R be a WRD proof of C from F of size n. Then there is an RD proof S of C' from F of size $\leq n$ for some $C' \subseteq C$. Furthermore, if R is regular, so is S, and if R is a tree, so is S.*

Proof. The proof of the theorem is straightforward. Writing R as a sequence $C_0, C_1, \ldots, C_n = C$, define clauses $C'_i \subseteq C_i$ by induction on i so that the new clauses form the desired proof S. For $C_i \in F$, let $C'_i = C_i$. Otherwise C_i is inferred by w-resolution from C_j and C_k w.r.t. a variable x. If $x \in C_j$ and $\overline{x} \in C_k$, let C'_i be the resolvent of C'_j and C'_k as obtained by the usual resolution rule; if not, then let C'_i be C'_j if $x \notin C'_j$, or C'_k if $\overline{x} \notin C'_k$. It is easy to check that each $C'_i \subseteq C_i$ and that, after removing duplicate clauses, the clauses C'_j form a valid resolution proof S. If R is regular, then so is S, and if R is a tree so is S. □

Essentially the same proof shows the same property for the system with the full weakening rule:

Proposition 3. *Let R be a RDW proof of C from F of size s. Then there is an RD proof S of C' from F of size $\leq s$ for some $C' \subseteq C$. Furthermore, if R is regular, so is S, and if R is a tree, so is S.*

There are several reasons why we prefer to work with w-resolution, rather than with the weakening rule. First, we find it to be an elegant way to combine weakening with resolution. Second, it works well for using resolution trees (with input lemmas, see the next section) to simulate DLL search algorithms. Third, since weakening and resolution together are stronger than w-resolution, w-resolution is a more refined restriction on resolution. Fourth, for regular resolution, using w-resolution instead of general weakening can be a quite restrictive condition, since any w-resolution inference $C_0, C_1 \vdash^w_x C$ "uses up" the variable x, making it unavailable for other resolution inferences on the same path, even if the variable does not occur at all in C_0 and C_1. The last two reasons mean that w-resolution can be rather weak; this strengthens our results below (Theorems 11 and 13) about the existence of regular proofs that use w-resolution.

The following simple theorem gives some useful properties for regular w-resolution.

Theorem 4. *Let G be a regular w-resolution refutation. Let C be a clause in G.*

a. *Suppose that C is derived from C_0 and C_1 with the edge from C_0 (resp. C_1) to C labeled with $\overline{x}$ (resp. x). Then $\overline{x} \notin C_0$, and $x \notin C_1$.*

b. *Let α be an assignment such that for every literal l labeling an edge on the path from C to the final clause, $\alpha(l) = True$. Then $C|_\alpha = 0$.*

Proof. The proof of part a. is based on the observation that if $\overline{x} \in C_0$, then also $\overline{x} \in C$. However, by the regularity of the resolution refutation, every clause on the path from C to the final clause $\Box$ must contain $\overline{x}$. But clearly $\overline{x} \notin \Box$.

Part b. is a well-known fact for regular resolution proofs. It holds for similar reasons for regular w-resolution proofs: the proof proceeds by induction on clauses in the proof, starting at the final clause $\Box$ and moving up towards the leaves. Part a. makes the induction step trivial. $\Box$

Directed acyclic graphs We define some basic concepts that will be useful for analyzing both resolution proofs and conflict graphs (which are defined below in Section 4). Let $G = (V, E)$ be a dag. The set of leaves (nodes in V of in-degree 0) of G is denoted V_G^0. The *depth* of a node u in V is defined to equal the maximum number of edges on any path from a leaf of G to the node u. Hence leaves have depth 0. The subgraph rooted at u in G is denoted G_u; its nodes are the nodes v for which there is a path from v to u in G, and its edges are the induced edges of G.

3 w-resolution trees with lemmas

This section first gives an alternate characterization of resolution dags by using *resolution trees with lemmas*. We then refine the notion of lemmas to allow only *input lemmas*. For non-regular derivations, resolution trees with lemmas and resolution trees with input lemmas are both proved below to be p-equivalent to resolution. However, for regular proofs, the notions are apparently different. (In fact we give an exponential separation between regular resolution and regular w-resolution trees with input lemmas.) Later in the paper we will give a tight correspondence between resolution trees with input lemmas and DLL search algorithms.

The intuition for the definition of a resolution tree with lemmas is to allow any clause proved earlier in the resolution tree to be reused as a leaf clause. More formally, assume we are given a resolution proof tree T, and further assume T is *ordered* in that each internal node has a left child and a right child. We define $<_T$ to be the post-ordering of T, namely, the linear ordering of the nodes of T such that if u is a node in T and v is in the subtree rooted at u's left child, and w is in the subtree rooted at u's right child, then $v <_T w <_T u$. For F a set of clauses, a *resolution tree with lemmas* (RTL) proof from F is an ordered binary tree such that (1) each leaf node v is labeled with either a member of F or with a clause that labels some node $u <_T v$, and (2) each internal node v is labeled with a variable x and a clause C, such that C is inferred by resolution w.r.t. x from the clauses labeling the two children of v, and (3) the unique out-degree zero node is labeled with the conclusion clause D. If $D = \Box$, then the RTL proof is a refutation.

w-resolution trees with lemmas (WRTL) are defined just like RTL's, but allowing w-resolution in place of resolution, and *resolution trees with lemmas and weakening* (RTLW) are defined in the same way, but allowing the weakening rule in addition to resolution.

An RTL or WRTL proof is *regular* provided that no path in the proof tree contains more than one (w-)resolution using a given variable x. Note that paths

follow the tree edges only; any maximal path starts at a leaf node (possibly a lemma) and ends at the conclusion.

It is not hard to see that resolution trees with lemmas (RTL) and resolution dags (RD) p-simulate each other. Namely, an RD can be converted into an RTL by doing a depth-first, leftmost traversal of the RD. In addition, it is clear that regular RTL's p-simulate regular RD's. The converse is open, and it is false for regular WRTL, as we prove in Section 5: intuitively, the problem is that when one converts an RTL proof into an RD, new path connections are created when leaf clauses are replaced with edges back to the node where the lemma was derived.

We next define resolution trees with input lemma (RTI) proofs. These are a restricted version of resolution trees with lemmas, where the lemmas are required to have been derived earlier in the proof by *input proofs*. Input proofs have also been called *trivial proofs* by Beame et al. [3], and they are useful for characterizing the clause learning permissible for DLL algorithms.

Definition An *input resolution tree* is a resolution tree such that every internal node has at least one child that is a leaf. Let v be a node in a tree T and let T_v be the subtree of T with root v. The node v is called an *input-derived node* if T_v is an input resolution tree.

Often the node v and its label C are identified. In this case, C is called an *input-derived clause*. In RTI proofs, input-derived clauses may be reused as lemmas. Thus, in an RTI proof, an input-derived clause is derived by an input proof whose leaves either are initial clauses or are clauses that were already input-derived.

Definition A *resolution tree with input lemmas* (RTI) proof T is an RTL proof with the extra condition that every lemma in T must appear earlier in T as an input-derived clause. That is to say, every leaf node u in T is labeled either with an initial clause from F or with a clause that labels some input-derived node $v <_T u$.

The notions of w-resolution trees with input lemmas (WRTI), regular resolution trees with input lemmas (regRTI), and regular w-resolution trees with input lemmas (regWRTI) are defined similarly.[1]

It is clear that the resolution dags (RD) and resolution trees with lemmas (RTL) p-simulate resolution trees with input lemmas (RTI). Somewhat surprisingly, the next theorem shows that the converse p-simulation holds as well.

Theorem 5. *Let G be a resolution dag of size s for the clause C from the set F of clauses. Let d be the depth of C in G. Then there is an RTI proof T for C from F of size $< 2sd$. If G is regular then T is also regular.*

[1] A small, but important point is that w-resolution inferences are not allowed in input proofs, even for input proofs that are part of WRTI proofs. We have chosen the definition of input proofs so as to make the results in Section 5 hold that show the equivalence between regWRTI proofs and DLL-L-UP search algorithms. Although similar results could be obtained if the definition of input proof were changed to allow w-resolution inferences, it would require also using a modified, and less natural, version of clause learning.

Proof. The dag proof G can be unfolded into a proof tree T', possibly exponentially bigger. The proof idea is to prune clauses away from T' leaving a RTI proof T of the desired size.

Without loss of generality, no clause appears more than once in G; hence, for a given clause C in the tree T', every occurrence of C in T' is derived by the same subproof T'_C. Let d_C be the depth of C in the proof, i.e., the height of the tree T'_C. Clauses at leaves have depth 0. We give the proof tree T' an arbitrary left-to-right order, so that it makes sense to talk about the i-th occurrence of a clause C in T'.

We define the j-th occurrence of a clause C in T' to be *leafable*, provided $j > d_C$. The intuition is that the leafable clauses will have been proved as a input clause earlier in T, and thus any leafable clause may be used as a lemma in T.

To form T from T', remove from T' any clause D if it has a successor that is leafable, so that every leafable occurrence of a clause either does not appear in T or appears in T as a leaf. To prove that T is a valid RTI proof, it suffices to prove, by induction on i, that if C has depth $d_C = i > 0$, then the i-th occurrence of C is input-derived in T. Note that the two children C_0 and C_1 of C must have depth $< d_C$. Since every occurrence of C is derived from the same two clauses, these occurrences of C_0 and C_1 must be at least their i-th occurrences. Therefore, by the induction hypothesis, the children C_0 and C_1 are leafable and appear in T as leaves. Thus, since it is derived by a single inference from two leaves, the i-th occurrence of C is input-derived.

It follows that T is a valid RTI proof. If the proof G was regular, clearly T is regular too.

To prove the size bound for T, note that G has at most $s-1$ internal nodes. Each one occurs at most d times as an internal node in T, so T has at most $d(s-1)$ internal nodes. Thus, T has at most $2d \cdot (s-1) + 1 < 2sd$ nodes in all. □

The following two theorems summarize the relationships between our various proof systems. We write $\mathcal{R} \equiv \mathcal{Q}$ to denote that $\mathcal{R}$ and $\mathcal{Q}$ are p-equivalent, and $\mathcal{Q} \leq \mathcal{R}$ to denote that $\mathcal{R}$ p-simulates $\mathcal{Q}$. The notation $\mathcal{Q} < \mathcal{R}$ means that $\mathcal{R}$ p-simulates $\mathcal{Q}$ but $\mathcal{Q}$ does not simulate $\mathcal{R}$.

Theorem 6. *RD* $\equiv$ *WRD* $\equiv$ *RTI* $\equiv$ *WRTI* $\equiv$ *RTL* $\equiv$ *WRTL*

Proof. The p-equivalences RD $\equiv$ WRD and RTI $\equiv$ WRTI and RTL $\equiv$ WRTL are shown by (the proof of) Proposition 2. The simulations RTI $\leq$ RTL $\equiv$ RD are straightforward. Finally, RD $\leq$ RTI is shown by Theorem 5. □

For regular resolution, we have the following theorem.

Theorem 7. *regRD* $\equiv$ *regWRD* $\leq$ *regRTI* $\leq$ *regRTL* $\leq$ *regWRTL* $\leq$ *RD and* *regRTI* $\leq$ *regWRTI* $\leq$ *regWRTL.*

Proof. regRD $\equiv$ regWRD and regWRTL $\leq$ RD follow from the definitions and the proof of Proposition 2. The p-simulations regRTI $\leq$ regRTL $\leq$ regWRTL

and regRTI $\leq$ regWRTI $\leq$ regWRTL follow from the definitions. The p-simulation regRD $\leq$ regRTI is shown by Theorem 5. $\square$

Below, we prove, as Theorem 14, that regRD $<$ regWRTI. This is the only separation in the hierarchy that is known. In particular, it is open whether regRD $<$ regRTI, regRTI $<$ regRTL, regRTL $<$ regWRTL, regWRTL $<$ RD or regWRTI $<$ regWRTL hold. It is also open whether regWRTI and regRTL are comparable.

4 DLL algorithms with clause learning

4.1 The basic DLL algorithm

The DLL proof search algorithm is named after the authors Davis, Logeman and Loveland of the paper where it was introduced [10]. Since they built on the work of Davis and Putnam [11], the algorithm is sometimes called the DPLL algorithm. There are several variations on the DLL algorithm, but the basic algorithm is shown in Figure 1. The input is a set F of clauses, and a partial assignment α. The assignment α is a set of ordered pairs (x, ϵ), where $\epsilon \in \{0, 1\}$, indicating that $\alpha(x) = \epsilon$. The DLL algorithm is implemented as a recursive procedure and returns either `UNSAT` if F is unsatisfiable or otherwise a satisfying assignment for F.

```
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) and ε ∈ {0,1}
6     β ←DLL(F, α ∪ {(x, ε)})
7     if β ≠ UNSAT then
8          return β
9     else
10         return DLL(F, α ∪ {(x, 1 − ε)})
```

Figure 1: The basic DLL algorithm.

Note that the DLL algorithm is not fully specified, since line 5 does not specify how to choose the branching variable x and its value ϵ. Rather one can think of the algorithm either as being nondeterministic or as being an algorithm schema. We prefer to think of the algorithm as an algorithm schema, so that it incorporates a variety of possible algorithms. Indeed, there has been extensive research into how to choose the branching variable and its value [13, 22].

There is a well-known close connection between regular resolution and DLL algorithms. In particular, a run of DLL can be viewed as a regular resolution tree, and vice-versa. This can be formalized by the following two propositions.

Proposition 8. *Let F be an unsatisfiable set of clauses and α an assignment. If there is an execution of* $\mathrm{DLL}(F, \alpha)$ *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 $Var(T) \cap dom(\alpha) = \varnothing$.*

The converse simulation of Proposition 8 holds, too, that is, a regular resolution tree can be transformed directly in a run of DLL.

Proposition 9. *Let F be an unsatisfiable set of clauses. Suppose that C has a regular resolution proof 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* $\mathrm{DLL}(F, \alpha)$*, that returns* `UNSAT` *after at most $s - 1$ recursive calls.*

The two propositions are based on the following correspondence between resolution trees and a DLL search tree: first, a leaf clause in a resolution tree corresponds to a clause falsified by α (so that $F|_\alpha = 0$), and second, a resolution inference with respect to a variable x corresponds to the use of x as a branching variable in the DLL algorithm. Together the two propositions give the following well-known exact correspondence between regular resolution trees and DLL search.

Theorem 10. *If F is unsatisfiable, then there is an execution of* $\mathrm{DLL}(F, \varnothing)$ *that executes with $< s$ recursive calls if and only if there exists a regular refutation tree for F of size $\leq s$.*

4.2 Learning by unit propagation

Two of the most successful enhancements of DLL that are used by most modern SAT solvers are unit propagation and clause learning. *Unit clause propagation* (also called Boolean constraint propagation) was already part of the original DLL algorithm and is based on the following observation: If α is a partial assignment for a set of clauses F and if there is a clause $C \in F$ with $C|_\alpha = \{l\}$ a unit clause, then any $\beta \supset \alpha$ that satisfies F must assign l the value *True*.

There are a couple of methods that the DLL algorithm can use to implement unit propagation. One method is to just use unit propagation to guide the choice of a branching variable by modifying line 5 so that, if there is a unit clause in $F|_\alpha$, then x and ϵ are chosen to make the literal true. More commonly though, DLL algorithms incorporate unit propagation as a separate phase during which the assignment α is iteratively extended to make any unit clause true until there are no unit clauses remaining. As the unit propagation is performed, the DLL algorithm keeps track of which variables were set by unit propagation and which clause was used as the basis for the unit propagation. This information is then useful for clause learning.

Clause learning in DLL algorithms was first introduced by Silva and Sakallah [23] and means that new clauses are effectively added to F. A learned clause D must be implied by F, so that adding D to F does not change the space of satisfying assignments. In theory, there are many potential methods

for clause learning; however, in practice, the only useful method for learning clauses is based on unit propagation as in the original proposal [23]. In fact, all deterministic state of the art SAT solvers for structured (non-random) instances of SAT are based on clause learning via unit propagation. This includes solvers such as Chaff [21], Zchaff [20] and MiniSAT [12].

These DLL algorithms apply clause learning when the set F is falsified by the current assignment α. Intuitively, they analyze the *reason* some clause C in F is falsified and use this reason to infer a clause D from F to be learned. There are two ways in which a DLL algorithm assigns values to variables, namely, by unit propagation and by setting a branching variable. However, if unit propagation is fully carried out, then the first time a clause is falsified is during unit propagation. In particular, this happens when there are two unit clauses $C_1|_\alpha = \{x\}$ and $C_2|_\alpha = \{\overline{x}\}$ requiring a variable x to be set both *True* and *False*. This is called a *conflict*.

The reason for a conflict is analyzed by building a conflict graph. Generally, this is done by maintaining an *unit propagation graph* that tracks, for each variable which has been assigned a value, the reason that implies the setting of the variable. The two possible reasons are that either (a) the variable was set by unit propagation when a particular clause C became a unit clause, in which case C is the reason, or (b) the variable was set arbitrarily as a branching variable. The unit propagation graph G has literals as its nodes. The leaves of G are literals that were set true as branching variables, and the internal nodes are variables that were set true by unit propagation. If a literal l is an internal node in G, then it was set true by unit propagation applied to some clause C. In this case, for each literal $l' \neq l$ in C, $\overline{l'}$ is a node in G and there is an edge from $\overline{l'}$ to l. If the unit propagation graph contains a conflict it is called a *conflict graph*. More formally, a conflict graph is defined as follows.

Definition A *conflict graph* G for a set F of clauses under the assignment α is a dag $G = (V \cup \{\Box\}, E)$ where V is a set of literals and where the following hold:

a. For each $l \in V$, either (i) l has in-degree 0 and $\alpha(l) = 1$, or (ii) there is a clause $C \in F$ such that $C = \{l\} \cup \{l' : (\overline{l'}, l) \in E\}$. For a fixed conflict graph G, we denote this clause as C_l.

b. There is a unique variable x such that $V \supseteq \{x, \overline{x}\}$.

c. The node $\Box$ has only the two incoming edges $(x, \Box)$ and $(\overline{x}, \Box)$.

d. The node $\Box$ is the only node with outdegree zero.

Let V_G^0 denote the nodes in G of in-degree zero. Then, letting $\alpha_G = \{(x, \epsilon) : x^\epsilon \in V_G^0\}$, the conflict graph G shows that every vertex l must be made true by any satisfying assignment for F that extends α. Since for some x, both x and $\overline{x}$ are nodes of G, this implies α cannot be extended to a satisfying assignment for F. Therefore, the clause $D = \{\overline{l} : l \in V_G^0\}$ is implied by F, and D can be taken as a learned clause. We call this clause D the *conflict clause* of G and denote it $CC(G)$.

There is a second type of clause that can be learned from the conflict graph G in addition to the conflict clause $CC(G)$. Namely, let $l \neq \square$ be any non-leaf node in G. Further, let $V^0_{G_l}$ be the set of leaves l' of G such that there is a path from l' to l. Then, the clauses in F imply that if all the leaves $l' \in V^0_{G_l}$ are assigned true, then l is assigned true. Thus, the clause $D = \{l\} \cup \{\overline{l'} : l' \in V^0_{G_l}\}$ is implied by F and can be taken as a learned clause. This clause D is called the *induced clause* of G_l and is denoted $IC(l, G)$. In the degenerate case where G_l consists of only the single literal l, this would make $IC(l, G)$ equal to $\{l, \overline{l}\}$; rather than permit this as a clause, we instead say that the induced clause does not exist.

In practice, both conflict clauses $CC(G)$ and induced clauses $IC(l, G)$ are used by SAT solvers. It appears that most SAT solvers learn the *first-UIP* clauses [23], which equal $CC(G)$ and $IC(l, G')$ for appropriately formulated G and G'. Other conflict clauses that can be learned include *all-UIP* clauses [26], *rel-sat* clauses [19], *decision* clauses [26], and *first cut* clauses [3]. All of these are conflict clauses $CC(G)$ for appropriate G. Less commonly, multiple clauses are learned, including clauses based on the cuts advocated by the mentioned works [23, 26], which are a type of induced clauses.

In order to prove the correspondence in Section 5 between DLL with clause learning and regWRTI proofs, we must put some restrictions on the kinds of clauses that can be (simultaneously) learned. In essence, the point is that for DLL with clause learning to simulate regWRTI proofs it is necessary to learn multiple clauses at once in order to learn all the clauses in a regular input subproof. But on the other hand, for regWRTI to simulate DLL with clause learning, regWRTI must be able to include regular input proofs that derive all the learned clauses so as to have them available for subsequent use as input lemmas. Thus, we define a notion of "compatible clauses" which is a set of clauses that can be simultaneously learned. For this, we define the notion of a series-parallel decomposition of a conflict graph G.

Definition A graph $H = (W, E')$ is a *subconflict graph* of the conflict graph $G = (V, E)$ provided that H is a conflict graph with $W \subseteq V$ and $E' \subseteq E$, and that each non-leaf vertex of H (that is, each vertex in $W \setminus V^0_H$) has the same in-degree in H as in G.

H is a *proper* subconflict graph of G provided there is no path in G from any non-leaf vertex of H to a vertex in V^0_H.

Note that if l is a non-leaf vertex in the subconflict graph H of G, then the clause C_l is the same whether it is defined with respect to H or with respect to G.

Definition Let G be a conflict graph. A *decomposition* of G is a sequence $H_0 \subset H_1 \subset \cdots \subset H_k$, $k \geq 1$, of distinct proper subconflict graphs of G such that $H_k = G$ and H_0 is the dag on the three nodes $\square$ and its two predecessors x and $\overline{x}$.

A decomposition of G will be used to describe sets of clauses that can be simultaneously learned. For this, we put a structure on the decomposition that describes the exact types of clauses that can be learned:

Definition A *series-parallel decomposition* $\mathcal{H}$ of G consists of a decomposition $H_0, \ldots, H_k$ plus, for each $0 \leq i < k$, a sequence $H_i = H_{i,0} \subset H_{i,1} \subset \cdots \subset H_{i,m_i} = H_{i+1}$ of proper subconflict graphs of G. Note that the sequence

$$H_0 = H_{0,0}, H_{0,1}, H_{0,2}, \ldots, H_{0,m_0} = H_1 = H_{1,0}, H_{1,1}, \ldots, H_{k-1,m_{k-1}} = H_k$$

is itself a decomposition of G. However, we prefer to view it as a two-level decomposition. A *series* decomposition is a series-parallel decomposition with trivial parallel part, i.e., with $k = 1$. A *parallel* decomposition is series-parallel decomposition in which $m_i = 1$ for all i. Note that we always have $H_i \neq H_{i+1}$ and $H_{i,j} \neq H_{i,j+1}$.

Figure 2 illustrates a series-parallel decomposition.

Definition For $\mathcal{H}$ a series-parallel decomposition, the set of *learnable clauses*, $CC(\mathcal{H})$, for $\mathcal{H}$ consists of the following induced clauses and conflict clauses:

- For each $1 \leq j \leq m_0$, the conflict clause $CC(H_{0,j})$, and
- For each $0 < i < k$ and $0 < j \leq m_i$ and each $l \in V^0_{H_i} \setminus V^0_{H_{i,j}}$, the induced clause $IC(l, H_{i,j})$.

It should be noted that the definition of the parallel decomposition incorporates the notion of "cut" used by Silva and Sakallah [23]. The DLL algorithm shown in Figure 3 chooses a single series-parallel decomposition $\mathcal{H}$ and learns some subset of the learnable clauses in $CC(\mathcal{H})$. It is clear that this generalizes all of the clause learning algorithms mentioned above.

The algorithm schema DLL-L-UP that is given in Figure 3 is a modification of the schema DLL. In addition to returning a satisfying assignment or `UNSAT`, it returns a modified formula that might include learned clauses. If F is a set of clauses and α is an assignment then DLL-L-UP(F, α) returns (F', α') such that $F' \supseteq F$ and F' is equivalent to F and such that α' either is `UNSAT` or is a satisfying assignment for F.[2]

The DLL-L-UP algorithm as shown in Figure 3 does not explicitly include unit propagation. Rather, the use of unit propagation is hidden in the test on line 2 of whether unit propagation can be used to find a conflict graph. In practice, of course, most algorithms set variables by unit propagation as soon as possible and update the implication graph each time a new unit variable is set. The algorithm as formulated in Figure 3 is more general, and thus covers

[2]Our definition of DLL-L-UP is slightly different from the version of the algorithm as originally defined in Hoffmann's thesis [17]. The first main difference is that we use series-parallel decompositions rather the compatible set of subconflict graphs of Hoffmann [17]. The second difference is that our algorithm does not build the implication graph incrementally by the use of explicit unit propagation; instead, it builds the implication graph once a conflict has been found.

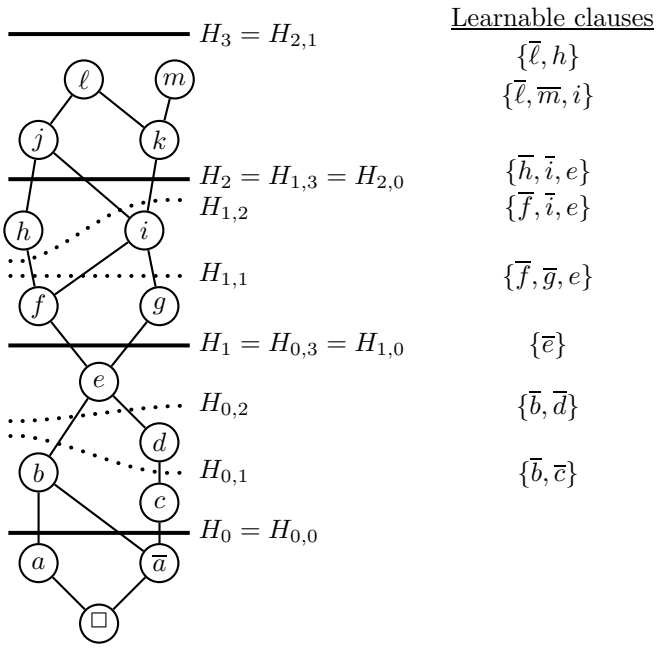

Figure 2: A series-parallel decomposition. Solid lines define the sets H_i of the parallel part of the decomposition, and dotted lines define the sets $H_{i,j}$ in the series part. Each line (solid or dotted) defines the set of nodes that lie below the line. The learnable clauses associated with each set are shown in the right column.

more possible implementations of DLL-L-UP, including algorithms that may change the implication graph retroactively or may pick among several conflict graphs depending on the details of how F can be falsified. There is at least one implemented clause learning algorithm that does this [14].

As shown in Figure 3, if $F|_\alpha$ is false, then the algorithm must return `UNSAT` (lines 2-6). Sometimes, however, we use instead a "non-greedy" version of DLL-L-UP. For the non-greedy version it is optional for the algorithm to immediately return `UNSAT` once F has a conflict graph. Thus the non-greedy DLL-L-UP algorithm can set a branching variable (lines 7-11) even if F has already been falsified and even if there are unit clauses present. This non-greedy version of DLL-L-UP will be used in the next section to simulate regWRTI proofs.

The constructions of Section 5 also imply that DLL-L-UP is p-equivalent to the restriction of DLL-L-UP in which only series decompositions are allowed. That is to say, DLL-L-UP with only series decompositions can simulate any run of DLL-L-UP with at most polynomially many more recursive calls.

```
DLL-L-UP(F, α)
1     if F|α = 1 then return (F, α)
2     if there is a conflict graph for F under α then
3          choose a conflict graph G for F under α
4               and a series-parallel decomposition H of G
5          choose a subset S of CC(H)     -- the learned clauses
6          return (F ∪ S, UNSAT)
7     choose x ∈ Var(F|α) and ε ∈ {0, 1}
8     (G, β)←DLL-L-UP(F, α ∪ {(x, ε)})
9     if β ≠ UNSAT then
10         return (G, β)
11    return DLL-L-UP(G, α ∪ {(x, 1 − ε)})
```

Figure 3: DLL with Clause Learning.

5 Equivalence of regWRTI and DLL-L-UP

5.1 regWRTI simulates DLL-L-UP

We shall prove that regular WRTI proofs are equivalent to non-greedy DLL-L-UP searches. We start by showing that every DLL-L-UP search can be converted into a regWRTI proof. As a first step, we prove that, for a given series-parallel decomposition $\mathcal{H}$ of a conflict graph, there is a single regWRTI proof T such that every learnable clause of $\mathcal{H}$ appears as an input-derived clause in T. Furthermore, T is polynomial size; in fact, T has size at most quadratic in the number of distinct variables that appear in the conflict graph.

This theorem generalizes earlier, well-known results of Chang [9] and Beame et al. [3] that any individual learned clause can be derived by input resolution (or, more specifically, that unit resolution is equivalent to input resolution). The theorem states a similar fact about proving an entire set of learnable clauses simultaneously.

Theorem 11. *Let G be a conflict graph of size n for F under the assignment α. Let $\mathcal{H}$ be a series-parallel decomposition for G. Then there is a regWRTI proof T of size $\leq n^2$ such that every learnable clause of $\mathcal{H}$ is an input-derived clause in T. The final clause of T is equal to $CC(G)$. Furthermore, T uses as resolution variables, only variables that are used as nodes (possibly negated) in $G \setminus V_G^0$.*

First we prove a lemma. Let the subconflict graphs $H_0 \subset H_1 \subset \cdots \subset H_k$ and $H_{0,0} \subset H_{0,1} \subset \cdots \subset H_{k-1,m_{k-1}}$ be as in the definition of series-parallel decomposition.

Lemma 12.

a. *There is an input proof T_0 from F which contains every conflict clause $CC(H_{0,j})$, for $j = 1, \ldots, m_0$. Every resolution variable in T_0 is a non-leaf node (possibly negated) in H_1.*

b. *Suppose that* $1 \leq i < k$ *and* u *is a literal in* $V^0_{H_i}$. *Then there is an input proof* T^u_i *which contains every (existing) induced clause* $IC(u, H_{i,j})$ *for* $j = 1, \ldots, m_i$. *Every resolution variable in* T^u_i *is a non-leaf node (possibly negated) in the subgraph* $(H_{i+1})_u$ *of* H_{i+1} *rooted at* u.

Proof. We prove part a. of the lemma and then indicate the minor modifications needed to prove part b. The construction of T_0 proceeds by induction on j to build proofs $T_{0,j}$; at the end, T_0 is set equal to T_{0,m_0}. Each proof $T_{0,j}$ ends with the clause $CC(H_{0,j})$ and contains the earlier proof $T_{0,j-1}$ as a subproof. In addition, the only variables used as resolution variables in $T_{0,j}$ are variables that are non-leaf nodes (possibly negated) in $H_{0,j}$.

To prove the base case $j = 1$, we must show that $CC(H_{0,1})$ has an input proof $T_{0,1}$. Let the two immediate predecessors of $\square$ in G be the literals x and $\overline{x}$. Define a clause C as follows. If x is not a leaf in $H_{0,1}$, then we let $C = C_x$; recall that C_x is the clause that contains the literal x and the negations of literals that are immediate predecessors of x in the conflict graph. Otherwise, since $H_{0,1} \neq H_0$, $\overline{x}$ is not a leaf in $H_{0,1}$, and we let $C = C_{\overline{x}}$. By inspection, C has the property that it contains only negations of literals that are in $H_{0,1}$. For $l \in C$, define the $\{0,1\}$-depth of l as the maximum length of a path to $\overline{l}$ from a leaf of $H_{0,1}$. If all literals in C have $\{0,1\}$-depth equal to zero, then $C = CC(H_{0,1})$, and C certainly has an input proof from F (in fact, since $C = C_x$ or $C = C_{\overline{x}}$, we must have $C \in F$).

Suppose on the other hand, that C is a subset of the nodes of $H_{0,1}$ with some literals of non-zero $\{0,1\}$-depth. Choose a literal l in C of maximum $\{0,1\}$-depth d and resolve C with the clause $C_{\overline{l}} \in F$ to obtain a new clause C'. Since $C_{\overline{l}} \in F$, the resolution step introducing C' preserves the property of having an input proof from F. Furthermore, the new literals in $C' \setminus C$ have $\{0,1\}$-depth strictly less than d. Redefine C to be the just constructed clause C'. If this new C is a subset of $CC(H_{0,1})$ we are done constructing C. Otherwise, some literal in C has non-zero $\{0,1\}$-depth. In this latter case, we repeat the above construction to obtain a new C, and continue iterating this process until we obtain $C \subset CC(H_{0,1})$.

When the above construction is finished, C is constructed as a clause with a regular input proof $T_{0,1}$ from F (the regularity follows by the fact that variables introduced in C' have $\{0,1\}$ depth less than that of the resolved-upon variable). Furthermore $C \subset CC(H_{0,1})$. In fact, $C = CC(H_{0,1})$ must hold, because there is a path, in $H_{0,1}$, from each leaf of $H_{0,1}$ to $\square$. That completes the proof of the $j = 1$ base case.

For the induction step, with $j > 1$, the induction hypothesis is that we have constructed an input proof $T_{0,j}$ such that $T_{0,j}$ contains all the clauses $CC(H_{0,p})$ for $1 \leq p \leq j$ and such that the final clause in $T_{0,j}$ is the clause $CC(H_{0,j})$. We are seeking to extend this input proof to an input proof $T_{0,j+1}$ that ends with the clause $CC(H_{0,j+1})$. The construction of $T_{0,j+1}$ proceeds exactly like the construction above of $T_{0,1}$, but now we start with the clause $C = CC(H_{0,j})$ (instead of $C = C_x$ or $C_{\overline{x}}$), and we update C by choosing the literal $l \in C$ of maximum $\{0, j+1\}$-depth and resolving with $C_{\overline{l}}$ to derive the next C. The

rest of the construction of $T_{0,j+1}$ is similar to the previous argument. For the regularity of the proof it is essential that $H_{0,j}$ is a proper subconflict graph of $H_{0,j+1}$. By inspection, any literal l used for resolution in the new part of $T_{0,j+1}$ is a non-leaf node in $H_{0,j+1}$ and has a path from l to some leaf node of $H_{0,j}$. Since $H_{0,j}$ is proper, it follows that l is not an inner node of $H_{0,j}$ and thus is not used as a resolution literal in $T_{0,j}$. Thus $H_{0,j+1}$ is regular. This completes the proof of part a.

The proof for part b. is very similar to the proof for part a. Fixing $i > 0$, let u be any literal in $V^0_{H_{i,0}}$. We need to prove, for $1 \leq j \leq m_i$, there is an input proof $T^u_{i,j}$ from F such that (a) $T^u_{i,j}$ contains every existing induced clause $IC(u, H_{i,k})$ for $1 \leq k < j$, and (b) $T^u_{i,j}$ ends with the induced clause $IC(u, H_{i,j})$, and (c) the resolution variables used in $T^u_{i,j}$ are all non-leaf nodes (possibly negated) of $V_{(H_{i,j})_u}$. The proof is by induction on j. One starts with the clause $C = C_u$. The main step of the construction of $T^u_{i,j+1}$ from $T^u_{i,j}$ is to find the literal $v \neq u$ in C of maximum $\{i, j\}$-depth, and resolve C with $C_{\overline{v}}$ to obtain the next C. This process proceeds iteratively exactly like the construction used for part a. This completes the proof of Lemma 12. □

We now can prove Theorem 11. Lemma 12 constructed separate regular input resolution proofs $T_{0,m_0} = T_0$ and $T^u_{i,m_i} = T^u_i$ that included all the learnable clauses of $\mathcal{H}$. To complete the proof of Theorem 11, we combine all these proofs into one single regWRTI proof. For this, we construct proofs T^*_i of the clause $CC(H_i)$. T^*_1 is just T_0. The proof T^*_{i+1} is constructed from T^*_i by successively resolving the final clause of T^*_i with the final clauses of the proofs T^u_i, using each $u \in V^0_{H_i} \setminus V^0_{H_{i+1}}$ as a resolution variable, taking the u's in order of increasing $\{i, m_i\}$-depth to preserve regularity. Letting $T = T^*_k$, it is clear that T^*_k contains all the clauses from $CC(\mathcal{H})$, and, by construction, T^*_k is regular.

To bound the size of T, note that any regular input proof S has size $2r + 1$ where r is the number of distinct variables used as resolution variables in S. Since T is regular, and is formed by combining the regular input proofs T_0, T^u_i in a linear fashion, the total size of T is less than $n + \sum_{k=0}^{n-1}(2k + 1) = n^2 + 1$. This completes the proof of Theorem 11. □

Note that, since the final clause of T contains only literals from V^0_G, T does not use any variable that occurs in its final clause as a resolution variable.

We can now prove the first main result of this section, namely, that regWRTI proofs polynomially simulate DLL-L-UP search trees.

Theorem 13. *Suppose that F is an unsatisfiable set of clauses and that there is an execution of a (possibly non-greedy)* DLL-L-UP *search algorithm on input F that outputs* `UNSAT` *with s recursive calls. Then there is a regWRTI refutation of F of size at most $s \cdot n^2$ where $n = |Var(F)|$.*

Proof. Let S be the search tree associated with the DLL-L-UP algorithm's execution. We order S so that the DLL-L-UP algorithm effectively traverses S in a depth-first, left-to-right order. We transform S into a regWRTI proof tree T

as follows. The tree T contains a copy of S, but adds subproofs at the leaves of S (these subproofs will be derivations of learned clauses). For each internal node in S, if the corresponding branching variable was x and was first set to the value x^ϵ, then the corresponding node in T is labeled with x as the resolution variable, and its left incoming edge is labeled with x^ϵ and its right incoming edge is labeled with $x^{1-\epsilon}$. For each node u in S, let α_u be the assignment at that node that is held by the DLL-L-UP algorithm upon reaching that node. By construction, α_u is equivalently defined as the assignment that has $\alpha_u(l) = 1$ for literal l that labels an edge on the path (in T) between u and the root of T.

For a node u that is a leaf of S, the DLL-L-UP algorithm chooses a conflict graph G_u with a series-parallel decomposition $\mathcal{H}_u$ such that every leaf node l of G_u is a literal set to true by α_u. Also, let F_u be the set F of original clauses augmented with all clauses learned by the DLL-L-UP algorithm before reaching node u. By Theorem 11, there is a proof T_u from the clauses F_u such that every learnable clause of $\mathcal{H}_u$ appears in T_u as in input-derived clause. Hence, of course, every clause learned at u by the DLL-L-UP algorithm appears in T_u as an input-derived clause. The leaf node u of S is then replaced by the proof T_u in T. Note that by Theorem 11 and the definition of conflict graphs, the final clause C_u of T_u is a clause that contains only literals falsified by α_u.

So far, we have defined the clauses C_u that label nodes u in T only for leaf nodes u. For internal nodes u, we define C_u inductively by letting v and w be the immediate predecessors of u in T and defining C_u to be the clause obtained by (w-)resolution from the clauses C_v and C_w with respect to the branching variable x that was picked at node u by the DLL-L-UP algorithm. Clearly, using induction from the leaves of S, the clause C_u contains only variables that are falsified by the assignment α_u. This makes T a regWRTI proof.

Let r be the root node of S. Since α_r is the empty assignment, the clause C_r must equal the empty clause $\Box$. Thus T is a regWRTI refutation of F and Theorem 13 is proved. $\Box$

Since DLL clause learning based on first cuts has been shown to give exponentially shorter proofs than regular resolution [3], and since Theorem 13 states that regWRTI can simulate DLL search algorithms (including ones that learn first cut clauses), we have proved that regRD does not simulate regWRTI:

Theorem 14. *regRD $<$ regWRTI.*

Hoffmann [17] gave a direct proof of Theorem 14 based on the variable extensions described below in Section 7.

5.2 DLL-L-UP simulates regWRTI

We next show that the non-greedy DLL-L-UP search procedure can simulate any regWRTI proof T. The intuition is that we split T into two parts: the *input parts* are the subtrees of T that contain only input-derived clauses. The *interior part* of T is the rest of T. The interior part will be simulated by a DLL-L-UP search procedure that traverses the tree T and at each node,

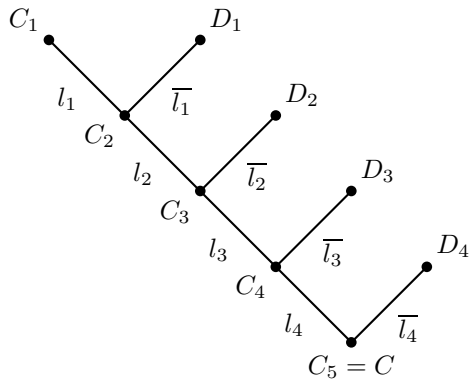

Figure 4: A regular input proof of C. Edges are labeled l_i or $\overline{l_i}$. The C_i's and D_i's are clauses.

chooses the resolution variable as the branching variable and sets the branching variable according to the label on the left incoming edge. In this way, the tree T is traversed in a depth-first, left-to-right order. The input parts of T are not traversed however. Once an input-derived clause is reached, the DLL-L-UP search learns all the clauses in that input subproof and backtracks returning `UNSAT`.

The heart of the procedure is how a conflict graph and corresponding series-parallel decomposition can be picked so as to make all the clauses in a given input subproof learnable. This is the content of the next lemma.

Lemma 15. *Let T be a regular input proof of C from a set of clauses F. Suppose that α falsifies C, that is, $C|_\alpha = 0$. Further suppose no variable in C is used as a resolution variable in T. Then there is a conflict graph G for F under α and a series decomposition $\mathcal{H}$ for G such that the set of learnable clauses of $\mathcal{H}$ is equal to the set of input-derived clauses of T.*

Recall that a series decomposition just means a series-parallel decomposition with a trivial parallel part, i.e, $k = 1$ in the definition of series-parallel decompositions.

Proof. Without loss of generality, F is just the set of initial clauses of T. Let the input proof T contain clauses $C_{m+1} = C, C_m, \ldots, C_1, D_m, \ldots, D_1$ as illustrated in Figure 4 with $m = 4$. Each C_{i+1} is inferred from C_i and D_i by resolution on l_i, where $\overline{l_i} \in C_i$ and $l_i \in D_i$. For each i, we have $D_i = \{l_i\} \cup D_i'$, where $D_i' \subseteq C_{i+1}$. Likewise, $C_i = \{\overline{l_i}\} \cup C_i'$, where $C_i' \subseteq C_{i+1}$.

As illustrated in Figure 5, we construct conflict graphs $H_{0,0} = \{\square, l_1, \overline{l_1}\} \subset H_{0,1} \subset \cdots \subset H_{0,m} = G$ which form a series decomposition of G. $H_{0,i}$ will be a conflict graph from the set of clauses $\{C_1, D_1, \ldots, D_i\}$ under α_i where α_i is the assignment that falsifies all the literals in C_{i+1}. Indeed, the leaves of $H_{0,i}$ are precisely the negations of literals in C_{i+1}. For $i > 0$, the non-leaf nodes of $H_{0,i}$ are $\overline{l_1}$ and $l_1, \ldots, l_i$. The predecessors of $\overline{l_1}$ are defined to be the literals u with

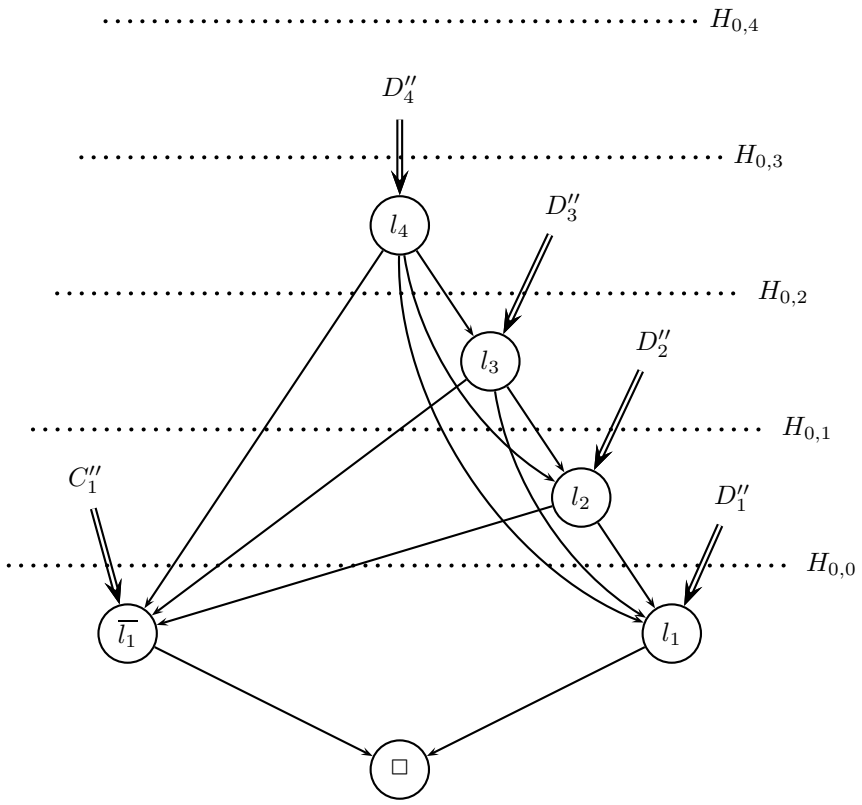

Figure 5: A conflict graph and a series decomposition. The solid lines and arcs indicate edges that may or may not be present. The notations C_1'' and D_i'' indicate zero or more literals, and the double lines indicate an edge from each literal in the set. The dashed lines indicate cuts, and thereby the sets $H_{0,i}$ in the series decomposition. Namely, the set $H_{0,i}$ contains the nodes below the corresponding dotted line.

$\overline{u} \in C_1'$, that is $C_{\overline{l_1}} = C_1$. Likewise, the predecessors of l_i are the literals u with $\overline{u} \in D_i'$ so that $C_{l_i} = D_i$.

To start with, we define $H_{0,0}$ to equal $\{\square, l_1, \overline{l}_1\}$. Let $H_{0,i}$ be already constructed. Then we have $\overline{l}_{i+1} \in C_{i+1}$ since C_{i+2} is inferred by resolution on l_{i+1} from C_{i+1}. It follows that $\alpha_i(l_{i+1}) = 1$ and that l_{i+1} is a leaf in $H_{0,i}$. We obtain $H_{0,i+1}$ from $H_{0,i}$ by adding the predecessors of l_{i+1} (i.e., the literals u with $\overline{u} \in D_{i+1}'$) to $H_{0,i}$. The leaves of $H_{0,i+1}$ are now exactly the negations of the literals in the clause C_{i+2}'. Finally the graph $H_{0,m} = G$ and the series decomposition $\mathcal{H}$ defined by the graphs $H_{0,i}$ is as wanted. This completes the proof of Lemma 15. $\square$

We can now finish the proof that DLL-L-UP simulates regWRTI.

Theorem 16. *Suppose that F has a regWRTI proof of size s. Then there is an execution of the non-greedy* DLL-L-UP *algorithm with the input $(F, \varnothing)$ that makes $< s$ recursive calls.*

Proof. Let T be a regWRTI refutation of F. The DLL-L-UP algorithm works by traversing the proof tree T in a depth-first, left-to-right order. At each non-input-derived node u of T, labeled with a clause C, the resolution variable for that clause is chosen as the branching variable x, and the variable x is assigned the value 1 or 0, corresponding to the label on the edges coming into u. By part b. of Theorem 4, the clause C is falsified by the assignment α. At each input-derived node of T, the DLL-L-UP algorithm learns the clauses in the input subproof above u by using the conflict graph and series decomposition given by Lemma 15. Since the DLL-L-UP search cannot find a satisfying assignment, it must terminate after traversing the (non-input) nodes in the regWRTI refutation tree. The number of recursive calls will equal twice the number of non-input-derived nodes of T, which is less than s. □

6 Generalized DLL with clause learning

6.1 The algorithm DLL-Learn

This section presents a new formulation of DLL with learning called DLL-Learn. This algorithm differs from DLL-L-UP in two important ways. First, unit propagation is no longer used explicitly (although it can be simulated). Second, the DLL-Learn algorithm uses more information that arises during the DLL search process, namely, it can infer clauses by resolution at each node in the search tree. This makes it possible for DLL-Learn to simulate regular resolution trees with full lemmas; more specifically, DLL-Learn is equivalent to regWRTL.

The DLL-Learn algorithm is very similar to the pool resolution system introduced by Van Gelder [25]. Furthermore, our Theorem 17 is similar to results obtained by Van Gelder for pool resolution. Our constructions differ mostly in that we use w-resolution in place of the degenerate resolution inference of Van Gelder [25]. Loosely speaking, Van Gelder's degenerate resolution inference is a method of allowing resolution to operate on any two clauses without any weakening. Conversely, our w-resolution is a method for allowing resolution to operate on any two clauses, but with the maximum reasonable amount of weakening.

The idea of DLL-Learn is to extend DLL so that it can learn a new clause C at each node in the search tree. As usual, the new clause will satisfy $F \equiv F \cup \{C\}$. At leaves, DLL-Learn does not learn a new clause, but marks a preexisting falsified clause as "new". At internal nodes, after branching on a variable x and making two recursive calls, the DLL-Learn algorithm can use w-resolution to infer a new clause, $C_{DLL(F,\alpha)}$, from the two identified new clauses, C_0 and C_1 returned by the recursive calls. Since x does not have to occur in $Var(C_0)$ and $Var(C_1)$, C is obtained by a w-resolution instead of resolution.

The DLL-Learn algorithm shown in Figure 6 uses non-greedy detection of contradictions. Namely, the "`optionally do`" on line 2 of Figure 6 allows the algorithm to continue to branch on variables even if the formula is already unsatisfied. This feature is needed for a direct proof of Theorem 17. In addition, it could be helpful in an implementation of the algorithm: Think of a call of DLL(F, α) such that $F|_\alpha = 0$ and suppose that all of the falsified clauses $C \in F$ are very large and thus undesirable to learn. It might, for example, be the case that $F|_\alpha$ contains two conflicting unit clauses $C_0|_\alpha = \{x\}$ and $C_1|_\alpha = \{\neg x\}$, where C_0 and C_1 are small. In that case, it could be better to branch on the variable x and to learn the resolvent of C_0 and C_1.

There is one situation where it is not optional to execute lines 3-4; namely, if α is a total assignment and has assigned values to all variables, then the algorithm must do lines 3-4.

Note that it is possible to remove C_0 and C_1 from F in line 13 if they were previously learned. Additionally, in an implementation of DLL-Learn it could be helpful to tag C_i as the new clause in H in line 13 if $C_i \subseteq C$ for an $i \in \{0, 1\}$ instead of learning C — this would be essentially equivalent to using Van Gelder's degenerate resolution instead of w-resolution.

```
DLL-Learn(F, α)
1     if F|α = 1 then return (F, α)
2     if F|α = 0 then optionally do
3          tag a C ∈ F with C|α = 0 as the new clause
4          return (F, UNSAT)
5     choose x ∈ Var(F) \ dom(α) and a value ϵ ∈ {0, 1}
6     (G, β)←DLL-Learn(F, α ∪ {(x, ϵ)})
7     if β ≠ UNSAT then return (G, β)
8     (H, γ)←DLL-Learn(G, α ∪ {(x, 1 − ϵ)})
9     if γ ≠ UNSAT then return (H, γ)
10    select the new C0 ∈ G and the new C1 ∈ H
11    C ← (C0 − {x^(1−ϵ)}) ∪ (C1 − {x^ϵ})
12    H ← H ∪ {C}                                  -- learn a clause
13    tag C as the new clause in H.
14    return (H, UNSAT)
```

Figure 6: DLL with a generalized learning.

It is easy to verify that, at any point in the DLL-Learn algorithm, when a clause C is tagged as new, then $C|_\alpha = 0$.

There is a straightforward, and direct, translation between executions of the DLL-Learn search algorithm on input $(F, \varnothing)$ and regWRTL proofs of F. An execution of DLL-Learn$(F, \varnothing)$ can be viewed as traversing a tree in depth-first, left-to-right order. If there are $s-1$ recursive calls to DLL-Learn, the tree has s nodes. Each node of the search tree is labeled with the clause tagged in the corresponding call to DLL-Learn. Thus, leaves of the tree are labeled with clauses that either are from F or were learned earlier in the tree. The clause on an internal node of the tree is inferred from the clauses on the two children

using w-resolution with respect to the branching variable. Finally, the clause C labeling the root node, where $\alpha = \varnothing$, must be the empty clause, since α must falsify C. In this way the search algorithm describes precisely a regWRTL proof tree. Conversely, any regWRTL refutation of F corresponds exactly to an execution of the DLL-Learn$(F, \varnothing)$.

This translation between DLL-Learn and regWRTI proof trees gives the following theorem.

Theorem 17. *Let F be a set of clauses. There exists a regWRTL refutation of F of size s if and only if there is an execution of* DLL-Learn$(F, \varnothing)$ *that performs exactly $s - 1$ recursive calls.*

It follows as a corollary of Theorems 7 and 17 that DLL-Learn can polynomially simulate DLL-L-UP.

7 Variable Extensions

This section introduces the notion of a *variable extension* of a CNF formula. A variable extension augments a set F of clauses with additional clauses such that modified formula $VEx(F)$ is satisfiable if and only if F is satisfiable. Variable extensions will be used to prove that regWRTI proofs can simulate resolution dags, in the sense that if there is an RD refutation of F, then there is a polynomial size regWRTI refutation of $VEx(F)$. Hence, DLL-Learn and the non-greedy version of DLL-L-UP can simulate full (non-regular) resolution in the same sense.

Our definition of variable extensions is inspired by the proof trace extensions of Beame et al. [3] that were used to separate DLL with clause learning from regular resolution dags. A similar construction was used by Hertel et al. [2] to show that pool resolution can simulate full resolution. Our results strengthen and extend the prior results by applying directly to regWRTI proofs. More importantly, in contrast to proof trace extensions, variable extensions do not depend on the size of a (possibly unknown) resolution proof but only on the number of variables in the formula.

Definition Let F be a set of clauses and $|Var(F)| = n$. The set of *extension variables* of F is $EVar(F) = \{q, p_1, \ldots, p_n\}$, where q and p_i are new variables. The *variable extension* of F is the set of clauses

$$VEx(F) \;=\; F \cup \big\{\{q, \bar{l}\} : l \in C \in F\big\} \cup \big\{\{p_1, p_2, \ldots, p_n\}\big\}.$$

Obviously $VEx(F)$ is satisfiable if and only if F is. Furthermore, $|VEx(F)| = O(|F|)$.

Suppose that G is a resolution dag (RD) proof from F. We can reexpress G as a sequence of (derived) clauses $C_1, C_2, \ldots, C_t$ which has the following properties: (a) C_t is the final clause of G, and (b) each C_i is inferred by resolution from two clauses D and E, where each of D and E either are in F or appear earlier in the sequence as C_j with $j < i$. Basically, the sequence is an ordinary resolution refutation, but with the clauses from F omitted.

Lemma 18. *Suppose that $D, E \vdash_x C$. Then, there is an input resolution proof tree T_C of the clause $\{q\}$ from $VEx(F) \cup \{D, E\}$ such that C appears in T_C and such that $|T_C| = 2 \cdot |C| + 3$.*

Proof. The proof T_C starts by resolving D and E to yield C. It then resolves successively with the clauses $\{q, \overline{l}\}$, for $l \in C$, to derive $\{q\}$. □

Theorem 19. *Let F be a set of clauses, $n = |Var(F)|$, and let C be a clause. Suppose that G is a resolution dag proof of C from F of size s. Then, there is a regWRTI proof T of C from $VEx(F)$ of size $\leq 2s \cdot (d+2) + 1$ where $d = \max\{|D| : D \in G\} \leq n$.*

Proof. Let $C_1, \ldots, C_t$ be a sequence of the derived clauses in G as above. Without loss of generality, $t < 2^n$ since F also has a regular resolution tree refutation, and this has depth at most n, and thus has $< 2^n$ internal nodes. Let T' be a binary tree with t leaves and of height $h = \lceil \log_2 t \rceil \leq n$. For each node u in T', let $l(u)$ be the level of u in T', namely, the number of edges between u and the root. Label u with the variable $p_{l(u)}$. Also, label every node u in T' with the clause $\{q\}$. T' will form the middle part of a regWRTI proof: each clause $\{q\}$ at level i is inferred by w-resolution from its two children clauses (also equal to $\{q\}$) with respect to the variable p_i.

Now, we expand T' into a regWRTI proof tree T''. For this, for $1 \leq i \leq t$, we replace the i-th leaf of T' with a new subproof T_{C_i} defined as follows. Letting C_i be as above, let D_i and E_i be the two clauses from which C_i is inferred in G. Then replace i-th leaf of T' by the input proof T_{C_i} from Lemma 18 which contains C_i and ends with the clause $\{q\}$. Note that each of D_i and E_i either is in F or appeared as an input clause in a proof, T_{D_i} or T_{E_i}, inserted at an earlier leaf of T'. Therefore T'' is a valid regWRTI proof of $\{q\}$ from $VEx(F)$. Since there are at most $s-1$ internal nodes in T' and each T_{C_i} has size $\leq 2d+3$, T'' has size at most $(s-1) + s \cdot (2d+3)$.

Finally, we form a regWRTI proof of C by modifying T'' by adding a new root labeled with the clause C and the resolution variable q. Let the left child of this new root be the root of T'', and let the right child be a new node labeled also with C. (This is permissible since C is input-derived in T''.) Label the left edge coming to the new root with the literal $\overline{q}$, and the right edge with the literal q. This makes C inferred from $\{q\}$ and C by w-resolution with respect to q. T is a valid regWRTI of size at most $s+1+s\cdot(2d+3) = 2s\cdot(d+2)+1$. □

Since DLL-L-UP and DLL-Learn simulate regWRTI, Theorem 19 implies that these two systems p-simulate full resolution by the use of variable extensions:

Corollary 20. *Suppose that F has a resolution dag refutation of size s. Then both* DLL-L-UP *and* DLL-Learn*, when given $VEx(F)$ as input, have executions that return* `UNSAT` *after at most $p(s)$ recursive calls, for some polynomial p.*

We now consider some issues about "naturalness" of proofs based on resolution with lemmas. Beame et al. [3] defined a refutation system to be natural

provided that, whenever F has a refutation of size s, then $F|_\alpha$ has a refutation of size at most s. We need a somewhat relaxed version of this notion:

Definition Let $\mathcal{R}$ be a refutation system for sets of clauses. The system $\mathcal{R}$ is *p-natural* provided, there is a polynomial $p(s)$, such that, whenever a set F has an $\mathcal{R}$-refutation of size s, and α is a restriction, then $F|_\alpha$ has an $\mathcal{R}$-refutation of size $\leq p(s)$.

The next proposition is well-known.

Proposition 21. *Resolution dags (RD) and regular resolution dags (regRD) are natural proof systems.*

As a corollary to Theorem 19 we obtain the following theorem.

Theorem 22.

a. *regWRTI is equivalent to RD if and only if regWRTI is p-natural.*

b. *regWRTL is equivalent to RD if and only if regWRTL is p-natural.*

Proof. Suppose that regWRTI $\equiv$ RD. Then, since RD is natural, we have immediately that regWRTI is p-natural.

Conversely, suppose that regWRTI is p-natural. By Theorem 7, RD p-simulates regWRTI. So it suffices to prove that regWRTI p-simulates RD. Let F have an RD refutation of size s. By Theorem 19, $VEx(F)$ has a regWRTI proof of size $2s(s+2)+1$. Let α be the assignment that assigns the value 1 to each of the extension variables q and $p_1, \ldots, p_n$. Since $VEx(F)|_\alpha$ is F and since regWRTI is p-natural, F has a regWRTI proof of size at most $p(2s(s+2)+1)$. This proves that regWRTI p-simulates RD, and completes the proof of a.

The proof of b. is similar. □

Theorem 22 is stated for the equivalence of systems with RD. It could also be stated for *p-equivalent* but then one needs an "effective" version of p-natural, where the $\mathcal{R}$-refutation of $F|_\alpha$ is computable in polynomial time from α and a $\mathcal{R}$-refutation of F.

8 A Lower Bound for RTLW with short lemmas

In this section we prove a lower bound showing that learning only short clauses does not help a DLL algorithm for certain hard formulas. The proof system corresponding to DLL algorithms with learning restricted to clauses of length k is, according to Section 5, regWRTI with the additional restriction that every used lemma is a clause of length at most k. We prove a lower bound for a stronger proof system that allows arbitrary lemmas instead of just input lemmas, drops the regularity restriction, and uses the general weakening rule instead of just w-resolution, i.e., RTLW as defined in Section 3. We define RTLW(k) to be the restriction of RTLW in which every lemma used, i.e., every leaf label that does not occur in the initial formula, is of size at most k.

The hard example formulas we prove the lower bound for are the well-known Pigeonhole Principle formulas. This principle states that there can be no 1-to-1 mapping from a set of size $n+1$ into a set of size n. In propositional logic, the negation of this principle gives rise to an unsatisfiable set of clauses PHP_n in the variables $x_{i,j}$ for $1 \leq i \leq n+1$ and $1 \leq j \leq n$. The variable $x_{i,j}$ is intended to state that i is mapped to j. The set PHP_n consists of the following clauses:

- the *pigeon clause* $P_i = \{x_{i,j}\,;\, 1 \leq j \leq n\}$ for every $1 \leq i \leq n+1$.
- the *hole clause* $H_{i,j,k} = \{\bar{x}_{i,k}, \bar{x}_{j,k}\}$ for every $1 \leq i < j \leq n+1$ and $k \leq n$.

It is well-known that the pigeonhole principle requires exponential size dag-like resolution proofs: Haken [16] shows that every RD refutation of PHP_n is of size $2^{\Omega(n)}$. Note that the number of variables is $O(n^2)$, so that this lower bound is far from maximal. In fact, Iwama and Miyazaki [18] prove a larger lower bound for tree-like refutations.

Theorem 23 (Iwama and Miyazaki [18]). *Every resolution tree refutation of PHP_n is of size at least $(n/4)^{n/4}$.*

We will show that for $k \leq n/2$, RTLW(k) refutations of PHP_n are asymptotically of the same size $2^{\Omega(n \log n)}$ as resolution trees. On the other hand, it is known [8] that dag-like resolution proofs need not be much larger than Haken's lower bound: there exist RD refutations of PHP_n of size $2^n \cdot n^2$. These refutations are even regular, and thus can be simulated by regWRTI. Hence PHP_n can be solved in time $2^{O(n)}$ by some variant of DLL-L-UP when learning arbitrary long clauses, whereas our lower bound shows that any DLL algorithm that learns only clauses of size at most $n/2$ needs time $2^{\Omega(n \log n)}$.

In fact, we will prove our lower bound for the weaker *functional* pigeonhole principle $FPHP_n$, which also includes the following clauses:

- The functional clause $F_{i,j,k} = \{\bar{x}_{i,j}, \bar{x}_{i,k}\}$ for every $1 \leq i \leq n+1$ and every $1 \leq j < k \leq n$.

While the lower bound of Iwama and Miyazaki is only stated for the clauses PHP_n, it is easily verified that their proof works as well when the functional clauses are added to the formula.

Our lower bound proof uses the fact that resolution trees with weakening (RTW) are natural, i.e., preserved under restrictions in the following sense:

Proposition 24. *Let R be a RTW proof of C from F of size s, and ρ a restriction. There is an RTW proof R' for $C|_\rho$ from $F|_\rho$ of size at most s.*

We denote the resolution tree R' by $R|_\rho$. Since this proposition is well-known a proof will not be given.

Next, we need to bring refutations in RTLW(k) to a certain normal form. First, we show that it is unnecessary to use clauses as lemmas that are subsumed by axioms in the refuted formula.

Lemma 25. *If there is a RTLW(k) refutation of some formula F of size s, then there is a RTLW(k) refutation of F of size at most $2s$ in which no clause C with $C \supseteq D$ for some clause D in F is used as a lemma.*

Proof. If a clause C with $C \supseteq D$ for some $D \in F$ is used as a lemma, replace every leaf labeled C by a weakening inference of C from D. □

Secondly, we need the fact that an RTLW(k) refutation does not need to use any tautological clauses, i.e., clauses of the form $C \cup \{x, \bar{x}\}$ for a variable x.

Lemma 26. *If there is a RTLW(k) refutation of some formula F of size s, then there is a RTLW(k) refutation of F of size at most s that contains no tautological clause.*

Proof. Let P be an RTLW(k)-refutation of F of size s that contains t occurrences of tautological clauses. We transform P into a refutation P' of size $|P'| \leq s$ such that P' contains fewer than t occurrences of tautological clauses. Finitely many iterations of this process yields the claim.

We obtain P' as follows. Since the final clause of P is not tautological, if $t > 0$, there must be a tautological clause $C \cup \{x, \bar{x}\}$ which is resolved with a clause $D \cup \{x\}$ to yield a non-tautological clause $C \cup D \cup \{x\}$. The idea is to cut out the subtree T_0 that derives the clause $C \cup \{x, \bar{x}\}$, and derive $C \cup D \cup \{x\}$ by a weakening from $D \cup \{x\}$. This gives a "proof" P_0 with fewer tautological clauses than P. However, P_0 may not be a valid proof, since some of the clauses in T_0 might be used as lemmas in P_0. To fix this, we shall extract parts of T_0 and plant them onto P_0 so that all lemmas used are derived. In order to make this construction precise, we need the notion of trees in which some of the used lemmas are not derived.

A *partial RTLW* from F is defined to be a tree T which satisfies all the conditions of an RTLW, except that some leaves may be labeled by clauses that occur neither in F nor earlier in T; these are called the *open leaves* of T.

We construct P' in stages by defining, for $i \geq 0$, a partial RTLW refutation P_i of F and a partial RTLW derivation T_i of $C \cup \{x, \bar{x}\}$ from F with the following properties:

- All open leaves in P_i appear in T_i. The first open leaf in P_i is denoted C_i.
- All open leaves in T_i appear in P_i before C_i.
- $|P_i| + |T_i| = |P|$.

P_0 and T_0 were defined above and certainly satisfy the two properties. Given P_i and T_i, we construct P_{i+1} and T_{i+1} as follows: We locate the first occurrence of C_i in T_i and let T_i^* be the subtree of T_i rooted at this occurrence. We form T_{i+1} by replacing in T_i the subtree T_i^* by a leaf labeled C_i. And, we form P_{i+1} by replacing the first open leaf, C_i, in P_i by the tree T_i^*.

The invariants are easily seen to be preserved. Obviously, $|P_{i+1}| + |T_{i+1}| = |P_i| + |T_i| = |P|$. The open leaves of T_i^* appear in P_i before C_i, and therefore, any open leaf in P_{i+1}, and in particular, C_{i+1} if it exists, must occur after the

(formerly open leaf) clause C_i. New open leaves in T_i are C_i and possibly some lemmas derived in T_i^*, and these all occur in P_{i+1} before C_{i+1}.

Since P_{i+1} contains fewer open leaves than P_i for every i, there is an m such that P_m contains no open leaves, and thus is an RTLW refutation. We then discard T_m and set $P' := P_m$. Each lemma used in P' was a lemma in P, thus P' is also an RTLW(k) refutation.

Note that the total number of occurrences of tautological clauses in P_{i+1} and T_{i+1} combined is the same as in P_i and T_i combined. This is also equal to the number of tautological clauses in P. Furthermore, T_m must contain at least one tautological clause, namely its root $C \cup \{x, \bar{x}\}$. It follows that P' has fewer tautological clauses than P. □

A matching ρ is a set of pairs $\{(i_1, j_1), \ldots, (i_k, j_k)\} \subset \{1, \ldots, n+1\} \times \{1, \ldots, n\}$ such that all the i_ν as well as all the j_ν are pairwise distinct. The size of ρ is $|\rho| = k$. A matching ρ induces a partial assignment to the variables of PHP_n as follows:

$$\rho(x_{i,j}) = \begin{cases} 1 & \text{if } (i,j) \in \rho \\ 0 & \text{if there is } (i,j') \in \rho \text{ with } j \neq j' \\ & \text{ or } (i',j) \in \rho \text{ with } i \neq i' \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We will identify a matching and the assignment it induces. The crucial property of such a matching restriction ρ is that $FPHP_n|_\rho$ is – up to renaming of variables – the same as $FPHP_{n-|\rho|}$.

The next lemma states that a short clause occurring as a lemma in an RTLW refutation can always be falsified by a small matching restriction.

Lemma 27. *Let C be a clause of size $k \leq n/2$ such that*

- *C is not tautological,*
- *$C \not\supseteq H_{i,i',j}$ for any hole clause $H_{i,i',j}$,*
- *$C \not\supseteq F_{i,j,j'}$ for any functional clause $F_{i,j,j'}$.*

Then there is a matching ρ of size $|\rho| \leq k$ such that $C|_\rho = \Box$.

Proof. First, we let ρ_1 consist of all those pairs (i, j) such that the negative literal $\bar{x}_{i,j}$ occurs in C. By the second and third assumption, these pairs form a matching. All the negative literals in C are set to 0 by ρ_1, and by the first assumption, no positive literal in C is set to 1 by ρ_1.

Now consider all pigeons $i_1, \ldots, i_r$ mentioned in positive literals in C that are not already set to 0 by ρ_1, i.e., that are not mentioned in any of the negative literals in C. Pick $j_1, \ldots, j_r$ from the $n/2$ holes not mentioned in C, and set $\rho_2 := \{(i_1, j_1), \ldots, (i_r, j_r)\}$. This matching sets the remaining positive literals to 0, thus for $\rho := \rho_1 \cup \rho_2$, we have $C|_\rho = \Box$. Clearly the size of ρ is at most k since we have picked at most one pair for each literal in C. □

Finally, we are ready to put all ingredients together to prove our lower bound.

Theorem 28. *For every $k \leq n/2$, every RTLW(k)-refutation of $FPHP_n$ is of size $2^{\Omega(n \log n)}$.*

Proof. Let R be an RTLW(k)-refutation of $FPHP_n$ of size s. By Lemmas 25 and 26, R can be transformed into R' of size at most $2s$ in which no clause is tautological and no clause used as a lemma is subsumed by a clause in $FPHP_n$. Let C be the first clause in R' which is used as a lemma; C is of size at most k. The subtree R_C of R' rooted at C is a resolution tree for C from $FPHP_n$.

By Lemma 27, there is a matching restriction ρ of size $|\rho| \leq k$ such that $C|_\rho = \Box$. Then $R_C|_\rho$ is a resolution tree with weakening refutation of $FPHP_n|_\rho$, which is the same as $FPHP_{n-k}$. By Proposition 3, applications of the weakening rule can be eliminated from $R_C|_\rho$ without increasing the size. Therefore by Theorem 23, R_C is of size

$$\left(\frac{n-k}{4}\right)^{\frac{n-k}{4}} \geq \left(\frac{n}{8}\right)^{\frac{n}{8}}$$

and hence the size of R is at least

$$s \geq \frac{1}{2}|R_C| \geq 2^{\Omega(n \log n)}.$$

□

References

[1] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3:81–102, 2007.

[2] Fahim Bacchus, Philipp Hertel, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simlulate general propositional resolution. Submitted for publication, 2008.

[3] 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.

[4] Daniel Le Berre and Laurent Simon. The essentials of the SAT 2003 competition. In *Proc. 6th International Conference on Theory and Applications of Satisfiability (SAT 2003)*, LNCS 2919, pages 452–467. Springer, 2003.

[5] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004*, LNCS 3542, pages 321–344. Springer, 2004.

[6] 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:i–xiv, 2005.

[7] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution restrictions and cutting planes proof systems. *SIAM Journal on Computing*, 30:1462–1484, 2000.

[8] Samuel R. Buss and Toniann Pitassi. Resolution and the weak pigeonhole principle. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic, 11th International Workshop CSL '97*, pages 149–156. Springer LNCS 1414, 1998.

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

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

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

[12] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, LNCS 3569, pages 61–75. Springer, 2005.

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

[14] Zhaohui Fu, Yogesh Mahajan, and Sharad Malik. New features of the SAT'04 version of zChaff. SAT Competition 2004 – Solver Description, `http://www.princeton.edu/ chaff/zchaff/sat04.pdf`, 2004.

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

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

[17] Jan Hoffmann. Resolution proofs and DLL-algorithms with clause learning. Diploma Thesis, LMU München, 2007. `http://www.tcs.ifi.lmu.de/~hoffmann` .

[18] Kazuo Iwama and Shuichi Miyazaki. Tree-like resolution is superpolynomially slower than dag-like resolution for the pigeonhole principle. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC)*, pages 133–142, 1999.

[19] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solver real-world SAT instances. In *Proc. 14th Natl. Conference on Artificial Intelligence*, pages 203–208, 1997.

[20] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004*, LNCS 3542, pages 360–375. Springer, 2004.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

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

[23] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 220–227, 1996.

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

[25] Allen Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), LNAI 3835*, pages 580–594, Montego Bay, Jamaica, 2005. Springer-Verlag.

[26] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 279–285, 2001.