Pool Resolution and its Relation to Regular Resolution and DPLL with Clause Learning

Allen Van Gelder

University of California, Santa Cruz CA 95060, USA,
WWW home page: http://www.cse.ucsc.edu/~avg

Abstract. Pool Resolution for propositional CNF formulas is introduced. Its relationship to state-of-the-art satisfiability solvers is explained. Every regular-resolution derivation is also a pool-resolution derivation. It is shown that a certain family of formulas, called $\mathrm{NT}^{**}(n)$ has polynomial sized pool-resolution refutations, whereas the shortest regular refutations have an exponential lower bound. This family is a variant of the $\mathrm{GT}(n)$ family analyzed by Bonet and Galesi (FOCS 1999), and the $\mathrm{GT}'(n)$ family shown to require exponential-length regular-resolution refutations by Alekhnovitch, Johannsen, Pitassi and Urquhart (STOC 2002). Thus, Pool Resolution is exponentially stronger than Regular Resolution. Roughly speaking a general-resolution derivation is a pool-resolution derivation if its directed acyclic graph (DAG) has a depth-first search tree that satisfies the regularity restriction: on any path in this tree no resolution variable is repeated. In other words, once a clause is derived at a node and used by its tree parent, its derivation is forgotten, and subsequent uses of that clause treat it as though it were an input clause. This policy is closely related to DPLL search with recording of so-called conflict clauses. Variations of DPLL plus conflict analysis currently dominate the field of high-performance satisfiability solving. The power of Pool Resolution might provide some theoretical explanation for their success.

1 Introduction

The reader is assumed to be generally familiar with the propositional satisfiability problem, CNF formulas, and resolution derivations. Some definitions are briefly reviewed in Section 2, but are not comprehensive.

The history of propositional resolution is interesting. In 1960, Davis and Putnam published an algorithm for deciding whether a propositional CNF formula is unsatisfiable [11]. They did not use the term "resolution" but after Robinson introduced the term in 1965, subsequent literature recognized that what Davis and Putnam were doing was a particular policy for propositional resolution. In 1962, Davis, Logemann and Loveland published the search algorithm that is well known today [10]. Interestingly, they described it as an *optimization* of the 1960 algorithm to conserve memory.

It appears to have been "folklore" in the theory community that it is possible to extract a resolution refutation from the 1962 search algorithm of Davis,

Logemann and Loveland. This fact was probably known to Tseitin in 1968 and to Galil soon after, although we are not able to pinpoint any published statement. About this time, the 1962 search algorithm began to be referred to as the "Davis-Putnam algorithm" or simply "DP" in the literature. This must have been rather irritating to Logemann and Loveland, particularly since the wrong 1960 paper was consistently cited as the source. This misnomer was not discovered until the 1990s, when the 1962 paper was rediscovered and the acronym DPLL was proposed for the search algorithm "to recognize the contributions of all four authors" [26].

In any case, Tseitin introduced the *regularity* restriction on resolution derivations, that no variable may be resolved upon more than once on any path in the derivation, and demonstrated a super-polynomial lower bound for *regular resolution* [23]. Both DP (resolution) and DPLL (search with resolution extracted) fell under the umbrella of *regular resolution*.

In the 1980's practical experience emerged that showed that DPLL (then called DP) performed unexpectedly well *as a propositional decision procedure* in comparison to known published strategies for resolution [20, 21]. In the 1990's further practical experience showed that DPLL performed poorly when used "as is" but could be greatly enhanced with additional "preorder" reasoning [14]. Meanwhile, it was recalled or rediscovered that DPLL was able to extract a resolution refutation from its search [16, 25].

The last decade has apparently seen theory and practice marching in opposite directions. DPLL produces a resolution derivation that is tree-like, whereas DP produces a resolution derivation that is a directed acyclic graph (DAG) and also is *ordered* (i.e., clashing variables appear in the same order along every DAG path). Although DP has achieved very little practical success [12], DPLL has been the work-horse for high performance satisfiability solvers [27, 3, 18, 19, 28, 13]. Yet theory has shown that the best tree-like resolution derivation may be exponential in length when a DAG resolution derivation is polynomial in length [5, 6]; indeed the separation holds even when the DAG resolution derivation is required to be *ordered* or *regular* [7, 8].

1.1 Summary of Results

The purpose of this paper is to show that the *practice* of satisfiability solving has unknowingly been moving in the direction indicated by *proof complexity theory*, away from the limitations of tree-like computations.

We introduce and formalize a new resolution strategy, called *Pool Resolution*. We show that pool resolution is "exponentially stronger" than regular resolution. Specifically, we show that pool resolution can linearly simulate regular resolution on all formulas. Then we exhibit a family of formulas for which there are polynomial-length pool refutations, but only exponential-length regular refutations. The exponential lower bound for regular refutations on this family was already shown by Alekhnovitch, Johannsen, Pitassi and Urquhart [1]; this paper demonstrates the existence of polynomial-length pool refutations. Beame, Kautz, and Sabharwal provide an excellent discussion on comparing reasoning systems

Table 1. Summary of notations.

$a, \ldots, z$	Literal; i.e., propositional variable or negated propositional variable.
$\neg x$	Complement of literal x; $\neg\neg x$ is not distinguished from x.
$\|x\|$	The propositional variable in literal x; i.e., if a is a variable, $\|a\| = \|\neg a\| = a$.
$A, \ldots, Z$	Disjunctive clause, or set of literals, depending on context.
$\mathcal{A}, \ldots, \mathcal{H}$	CNF formula, or set of literals, depending on context.
π	Resolution derivation DAG.
σ	Total assignment, represented as the set of true literals.
$[p_1, \ldots, p_k]$	Clause consisting of literals $p_1, \ldots, p_k$.
$\bot$	The *empty clause*, which represents *false*.
$\top$	The *tautologous clause*, which represents *true*; (see Definition 2.3).
$\alpha, \ldots, \delta$	Subclause, in the notation $[p, q, \alpha]$, denoting a clause with literals p, q, and possibly other literals, α.
C^-	Read as "C, or some clause that subsumes C".
p	Where context makes it clear, $[p]$ may be abbreviated to p.
C, p	In a context where a formula is expected, $\{C\}$ may be abbreviated to C and $\{[p]\}$ may be abbreviated to p.
$\mathbf{res}(q, C, D)$	Resolvent of C and D, where q and $\neg q$ are the clashing literals (see Definition 2.3).

through proof complexity and how it applies to DPLL with "clause learning" [4].

Based on proof-complexity comparisons, it was known that exponential separations supported the following strict order: tree-like $<$ regular $<$ general resolution. This paper shows that regular $<$ pool $\leq$ general resolution. Finally, we show that pool resolution provides a framework that encompasses most, if not all, of the well-known satisfiability solvers based on DPLL and some form of conflict analysis, also known as "clause learning," "clause recording," "nonchronological backtracking," "postorder lemmas," and similar terms. The main difference is that pure pool resolution remembers all derived clauses so they are available for re-use, whereas implemented satisfiability solvers only remember some of their derived clauses.

2 Preliminaries

2.1 Notation

This section collects notations and definitions used throughout the paper. Standard terminology for conjunctive normal form (CNF) formulas is used. Notations are summarized in Table 1. Although the general ideas of resolution and derivations are well known, there is no standard notation for many of the technical aspects, so it is necessary to specify our notation in detail. As usual, a finite set of *propositional variables* is assumed (*variables* for short) and a *literal* is either a variable or a negated variable.

Definition 2.1. (clause, formula, *Lits*, mention) A *clause* is either a *regular clause* or the unique *tautologous clause*, denoted by $\top$. A *regular clause* is a (possibly empty) consistent set of literals, which are logically connected disjunctively. A regular clause C is said to *mention* a literal q if either $q \in C$ or $\neg q \in C$. A *CNF formula* (*formula* for short) is a finite (possibly empty) sequence of clauses, which are logically connected conjunctively. The set of all literals that can be constructed from the variables in formula $\mathcal{F}$ is denoted by $\mathit{Lits}(\mathcal{F})$. This set is assumed to have some fixed linear order. □

There are technical reasons for defining a formula as a sequence, rather than a set, of clauses. First, this permits duplicate clauses. Second, when a procedure derives clauses sequentially, some or all of the derived clauses can be appended to the input formula, the structure remains a *formula*, and the order of derivation is preserved.

Definition 2.2. (assignment, satisfaction, model) A partial assignment is a partial function from the set of variables into $\{\mathit{false}, \mathit{true}\}$. This partial function is extended to literals, clauses, and formulas in the standard way. If the partial assignment is a total function, it is called a *total assignment*, or simply an *assignment*.

A clause or formula is *satisfied* by a partial assignment if it is mapped to *true*; A partial assignment that satisfies a formula is called a *model* of that formula. □

A partial assignment is conventionally represented by the (necessarily consistent) set of *unit clauses* that are mapped into *true* by the partial assignment. Note that this representation is a very simple formula.

2.2 Resolution as a Total Function

The standard definition of *resolution* is a binary operation on two clauses that contain a distinguished pair of *clashing literals*; i.e., one clause contains x and the other contains $\neg x$. It is convenient to extend the definition to *all* pairs of clauses and all literals, making resolution a total function. Recall that all tautologous clauses are considered to be indistinguishable and are denoted by $\top$.

Definition 2.3. (resolution, subsumption, useless clause) *Resolution* is an operation that takes as parameters a literal, called the *clashing literal*, and two clauses; it produces a clause as its result, called the *resolvent*. In all cases, the *resolvent* is independent of the order of the clause operands C and D, and is independent of the polarity of the clashing literal q:

$$\mathbf{res}(q, C, D) = \mathbf{res}(q, D, C) = \mathbf{res}(\neg q, C, D) = \mathbf{res}(\neg q, D, C).$$

The variable $|q|$ is called the *clashing variable* and $\neg q$ is also called the clashing literal.

If $C = [q, \alpha]$ and $D = [\neg q, \beta]$ are two regular clauses (α and β are subclauses), then

$$\mathbf{res}(q, C, D) = \begin{cases} [\alpha \cup \beta] & \text{if } \alpha \cup \beta \text{ is consistent;} \\ \top & \text{otherwise.} \end{cases}$$

This defines the standard *resolution* operation.

Resolution is extended to include $\top$ as an identity element:

$$\mathbf{res}(q, C, \top) = C.$$

Resolution is further extended to apply to any two regular clauses and any literals, as follows. Fix a total order on the clauses definable with the current set of propositional variables such that $\bot$ is smallest, $\top$ is largest, and wider clauses are "bigger" than narrower clauses. The smaller of two equally wide clauses is the one whose literals are lexicographically smaller in the fixed literal ordering (recall Definition 2.1).

If $C = [\alpha]$ does not contain q and does not contain $\neg q$, and $D = [\neg q, \beta]$ or $D = [q, \beta]$, then

$$\mathbf{res}(q, C, D) = [\alpha]\,.$$

If $C = [\alpha]$ and $D = [\beta]$ and neither contains q or $\neg q$, then

$$\mathbf{res}(q, C, D) = \text{the smaller of } C \text{ and } D.$$

If clause $C \subset D$, we say C *properly subsumes* D; if $C \subseteq D$, we say C *subsumes* D. Also, any regular clause properly subsumes $\top$. Notation D^- is read as "D, or some clause that subsumes D". Alternatively, D^- may be read as "some clause that logically implies D".

A clause is said to be *useless* for formula $\mathcal{F}$ if it is subsumed by a clause in $\mathcal{F}$; $\top$ is always useless. (Normally, tautologous resolvents are discarded.) □

Definition 2.4. (derivation, refutation) A *derivation* (short for *propositional resolution derivation*) from formula $\mathcal{F}$ is a directed acyclic graph (DAG) in which each vertex is labeled with a clause and possibly with a clashing literal. Let D be the clause label of vertex v. If $D = C \in \mathcal{F}$, then v has no out-edges and no clashing literal, and is called a *leaf*. Otherwise v is called a *resolution vertex*, has two out-edges, say to vertices with clause labels D_0 and D_1. The edge to D_1 is labeled with some literal, say q, and the edge to D_0 is labeled with $\neg q$. The vertex v is also labeled with the clashing literal q and the clause D such that

$$D = \mathbf{res}(q, D_0, D_1),$$

where **res** is the total function defined in Definition 2.3. When the derivation contains $\bot$, it is called a *refutation*. □

In most analyses a derivation is a *rooted* DAG, and a derivation is said to derive its root clause. In actual computation, some clauses might be derived that turn out to be useless, yet remain in the DAG. In much of the discussion, vertices are referred to by their clause labels. However, it is possible for the same clause to label several vertices and in such cases further specification of the vertex is needed.

3 The Pool Resolution Procedure

In its general form, pool resolution can be regarded as a procedure **PoolRes** that takes a clause P, called the current *pool*, and an input formula $\mathcal{F}$ as parameters, and determines whether $\mathcal{F} \models P$ ($\mathcal{F}$ logically implies P). Of course, $\mathcal{F} \models \bot$ means that $\mathcal{F}$ is unsatisfiable.

For simplicity of description, $\mathbf{PoolRes}(P, \mathcal{F})$ operates on a global proof structure G. If $\mathcal{F} \models P$, **PoolRes** modifies G to be a derivation of some clause $D \subseteq P$ and returns D. Otherwise, **PoolRes** creates a (global) partial assignment $\mathcal{A}$ that demonstrates that P is not logically implied by $\mathcal{F}$ and returns the special value **SAT** that is not a clause. Note that **PoolRes** might not be able to derive P exactly, but can derive P^- whenever $\mathcal{F} \models P$.

The global proof structure G is a resolution DAG (Definition 2.4) that initially consists of one vertex C_i for each clause in $\mathcal{F}$ and no edges. This is the state of G when the top-level call $\mathbf{PoolRes}(P, \mathcal{F})$ occurs. To produce a refutation, the top-level call is $\mathbf{PoolRes}(\bot, \mathcal{F})$. The procedure modifies G as the computation proceeds. Pseudocode for a recursive implementation of **PoolRes** is shown in Figure 1.

PoolRes$(P, \mathcal{F})$

```
 1) If F has no eligible clauses:
 2)       construct partial assignment A = ¬(P);
 3)       return SAT.
 4) If G contains an acceptable clause D ⊆ P:
 5)       return some such clause D.

 6) // (If no base case applies, expand the pool P.)
 7) Choose a clashing literal q not mentioned in P.

 8) D0 = PoolRes([P, ¬q], F).
 9) If D0 = SAT, return SAT.
10) If ¬q ∉ D0, return D0.

11) D1 = PoolRes([P, q], F).
12) If D1 = SAT, return SAT.
13) // (If q ∉ D1, res(q, D0, D1) = D1.)

14) Create a new vertex for G labeled with D = res(q, D0, D1).
    Create an edge labeled ¬q from D to D0 and an edge labeled q from D to D1.
15) Return D.
```

Fig. 1. Pseudocode for the general version of **PoolRes**. To produce a refutation, the top-level call is $\mathbf{PoolRes}(\bot, \mathcal{F})$. A clause C is *eligible* with respect to pool P if $C \cup P$ is consistent. For "pure" **PoolRes** all clauses are *acceptable*; other options are discussed in the text.

Several remarks about **PoolRes** may be made before analyzing its completeness and performance.

1. An arbitrary clause C is said to be *eligible* with respect to pool P if $C \cup P$ is consistent; otherwise, it is *ineligible* with respect to P. The idea is that only eligible clauses might be useful to **PoolRes** for deriving P^-. For a clause C to be useful to **PoolRes** for deriving P^-, it must eventually play the role of D on line 4 in the current procedure invocation or in some recursive invocation. The first parameter of **PoolRes** is called the *pool parameter*. But all pool parameters for these invocations are consistent supersets of P (possibly P itself), so no clause containing a literal that is complementary to some literal of P can be a subset of P or the pool parameter of any recursive invocation.
   Note that the set of eligible clauses shrinks as recursion depth increases. For example, when $\neg q$ is added to the pool at line 8, all clauses containing q become ineligible in that recursive call.
2. At line 1, suppose there are no eligible clauses in $\mathcal{F}$. Then $\mathcal{A} = \neg(P)$ satisfies all clauses of $\mathcal{F}$. By the soundness of resolution $\mathcal{A}$ must satisfy all clauses in G, so every clause in G has some literal that is complementary to some literal of P, and there are no eligible clauses in G, either.
3. There are many possible policies for what is an *acceptable* clause on line 4. Bookkeeping not shown in the pseudocode might be needed to decide whether a clause is "acceptable" under a particular policy. As discussed above, $D \subseteq P$ is possible only if D is *eligible* with respect to P, so it does not matter whether ineligible clauses are "acceptable." For "pure" **PoolRes**, all clauses in G are "acceptable." To force tree-like derivations to be produced, make all *derived* clauses unacceptable.
   The only restriction on acceptable-clause policies needed to ensure completeness is that, if all variables are mentioned in P and there are any eligible clauses, then *some* eligible clause must be acceptable.
   By formulating **PoolRes** to allow an arbitrary policy for "acceptable" clauses, it is easier to show that some instantiation of **PoolRes** is able to simulate, or imitate, other reasoning systems. With this flexibility, **PoolRes** is able to reject clauses that would generate a base case, and continue to line 7 to derive a better clause.
4. Line 10 is an optimization. Due to resolution being a total function, line 14 would define D to be either D_0 or D_1 if line 10 were omitted.
5. At line 14, if $q \notin D_1$, then D is in a separate vertex from the vertex of D_1, even though D and D_1 are the same clause. In this case, the vertex containing D has an edge to the vertex containing D_0. This technicality ensures that the final G is rooted (unless **SAT** is returned).
6. At line 5, there might be several clauses that could be returned. The selection could greatly influence the future course of the computation.
7. At line 7, there are normally many variables to choose among, then two polarities for the literal to be called q. The policy for this choice is the major determinant of the procedure's practical ability to construct small

derivations. The theoretical (non-deterministic) power of pool resolution is determined (in part) by assuming this choice is always made optimally.

It is quite straightforward to show that **PoolRes** behaves "correctly"; that is, it is sound and complete.

Theorem 3.1. Let $\mathcal{F}$ be a formula; let G be a resolution DAG with the clauses of $\mathcal{F}$ as leaves; let P be a regular clause, $P \subset \mathit{Lits}(\mathcal{F})$. Then $\mathbf{PoolRes}(P, \mathcal{F})$ as given in Figure 1 returns a clause P^- if and only if $\mathcal{F} \models P$.

Proof. The proof is by structural induction on the call graph of **PoolRes**. The callgraph is a directed tree in which each procedure invocation is a vertex and there is an edge from each procedure invocation to its immediate recursive calls, if any. The leaves are procedure invocations that make no recursive calls. In these cases, either line 1 is true or line 4 is true.

Case 1: line 1 is true. Then $\mathbf{PoolRes}(P, \mathcal{F})$ does not return P^-. Also $\mathcal{A}$ satisfies all clauses in $\mathcal{F}$ and falsifies P, so $\mathcal{F} \models P$ is false.

Case 2: line 4 is true. Then $\mathcal{F} \models D$ by the soundness of resolution and $D = P^-$, so $D \models P$. Thus the theorem holds for the base cases.

For procedure invocations that make at least one recursive call, we assume the theorem holds for the immediate recursive calls, by the inductive hypothesis. Line 1 is false, so there is some eligible clause. Line 4 is false, so every clause $D \in G$ is not acceptable or is not a subset of P. If every variable is mentioned in P, the acceptable-clause policy is required to make some eligible clause acceptable, and any eligible clause is a subset of P. It follows that some variable is not mentioned in P, so it is possible to choose some literal q at line 7.

If **SAT** is returned by a recursive call, then (by the inductive hypothesis) some superset of P is not logically implied by $\mathcal{F}$, hence neither is P. In this case, **SAT** is returned by the current invocation, as required by the theorem.

If **SAT** is not returned by either immediate recursive call, then by the inductive hypothesis, at line 8, $\mathcal{F} \models D_0 = [P, \neg q]^-$, and at line 11 (if reached), $\mathcal{F} \models D_1 = [P, q]^-$. If $\neg q \notin D_0$ at line 8, then D_0 is returned at line 10 and, $D_0 = P^- \models P$. If $q \notin D_1$ at line 11, then $D_1 = P^- \models P$ and at line 14 $D = D_1$ and D is returned.

Finally, if $\neg q \in D_0$ and $q \in D_1$, then D is a standard resolvent at line 14 and $\mathcal{F} \models D$ by soundness of resolution. But $D = P^-$, so $\mathcal{F} \models P$. □

4 Pool Resolution Graphs

By the time the pool resolution procedure **PoolRes** given in Figure 1 has exited at top level, assuming it did not return **SAT**, it has produced a *rooted* resolution DAG G. Essentially, all rooted resolution DAGs can be characterized according to whether some instantiation of a pool resolution procedure could produce them. For analysis, we are only concerned with *refutation* DAGS, i.e., those whose root is $\bot$. This section shows that there is a close connection, and an important difference, between refutation DAGs produced by regular resolution and those produced by pool resolution.

Definition 4.1. A rooted resolution DAG based on input formula $\mathcal{F}$ is called a *pool resolution DAG* if it can be produced by some sequence of choices in the pool resolution procedure **PoolRes** given in Figure 1. These choices include:

1. Which clauses are "acceptable" at line 4 (without loss of generality, we can assume at most one clause is deemed "acceptable" each time line 4 is executed);
2. Which literal to choose as q at line 7.

□

Definition 4.2. Let G be a subgraph of a resolution DAG (but not necessarily a resolution DAG in its own right). A path in G is a *regular path* if no clashing variable occurs twice among the vertices of the path. The subgraph G is said to be a *regular DAG* if every path in G is regular. □

As defined by Tseitin [23], *regular resolution* is the resolution system that produces resolution DAGs that are regular DAGs, in accordance with Definition 4.2. Our use of the term *regular* is simply extended to include DAGs that are not resolution derivations.

Theorem 4.3. If a rooted resolution refutation DAG G (based on input formula $\mathcal{F}$) is a pool resolution DAG, then there is some depth-first search of G (beginning at its root) whose depth-first search tree (DFST) is a regular DAG.

Proof. By hypothesis, G can be produced by the execution of some pool resolution procedure. Each derived vertex v of G is created in exactly one invocation of **PoolRes**. Let q be the clashing literal for that procedure invocation. When the depth-first search visits v it processes the outgoing edge corresponding to $\neg q$ before the edge corresponding to q. For every path in the resulting DFST, there is a path in the call graph in which the vertices created at line 14 appear in the same order. But the same clashing variable cannot appear twice on any path in the call graph. Therefore, the DFST is a regular DAG. □

Theorem 4.4. Let G be a rooted resolution refutation DAG (based on input formula $\mathcal{F}$) that contains only standard resolution operations (i.e., the clashing literal q is present in one operand and $\neg q$ is present in the other operand and $\top$ never occurs). If there is some depth-first search of G (beginning at its root) whose depth-first search tree (DFST) is a regular DAG, then G is a pool resolution DAG.

Proof. By hypothesis, there is a DFST that is a regular DAG. Order the edges leaving any vertex of G to be in the same order as those edges were processed during the depth-first search. Let the pool resolution procedure imitate the depth-first traversal. The top-level call is **PoolRes**$(\bot, \mathcal{F})$; each recursive invocation of **PoolRes** corresponds to an edge of G and they occur in the same order that the edges are processed by the depth-first search (recall that depth-first search *processes* both tree and non-tree edges, but only traverses across tree edges).

We need to show that this pool resolution procedure constructs (an isomorphic copy of) G.

Case 1: Suppose the traversal is *visiting* vertex v; i.e., the edge $p \to v$ has just been traversed, where p is the DFST parent of v. The pool parameter P for current invocation of **PoolRes** consists of the clashing literals on the edges of the DFST path from the root of G to v (if v is the root, $P = \bot$). Assuming v is not the root, $\mathbf{PoolRes}(P, \mathcal{F})$ corresponds to the tree edge $p \to v$.

Case 1-A: Suppose the operation at v is $D = \mathbf{res}(q, D_0, D_1)$, so that v has edges to w_0, labeled with D_0 and to w_1, labeled with D_1. Without loss of generality, assume the depth-first search processes the edge $v \to w_0$, which is labeled with $\neg q$, before processing $v \to w_1$. By the regularity of the DSFT, q is not mentioned in P. So the procedure invocation $\mathbf{PoolRes}(P, \mathcal{F})$ considers all clauses to be "not acceptable" at line 4 and chooses q as the clashing literal at line 7. By induction on the structure of G, we may assume **PoolRes** recursively derives D_0 at line 8, recursively derives D_1 at line 11. It then resolves them at line 14, deriving D, and creates (an isomorphic copy of) v, $v \to w_0$, and $v \to w_1$.

Case 1-B: Now suppose v is a leaf vertex of G, i.e., is labeled with a clause $C \in \mathcal{F}$. The sequence of resolutions on the path back to the root of G must remove all literals of C because the root contains $\bot$. Thus $C \subseteq P$. Let the acceptable-clause policy accept C at line 4. So C is returned at line 5 and the parent invocation will create (an isomorphic copy of) the edge $p \to v$.

Case 2: Suppose the depth-first traversal is processing a non-tree edge $u \to v$; i.e., v has been visited earlier and u is being visited currently. Let the operation at u be $D = \mathbf{res}(q, D_0, D_1)$, where v is labeled with D_0 and $u \to v$ is labeled with $\neg q$. Thus $\neg q \in D_0$. (It is easy to see that u cannot be the root of G in this case because then there would be a regular DFST path from u to v whose first edge label is q.)

As in Case 1, the pool parameter P for the **PoolRes** invocation corresponding to the tree-edge to $p \to u$ consists of the clashing literals on the edges of the DFST path from the root to u. This invocation calls $\mathbf{PoolRes}([P, \neg q], \mathcal{F})$ at line 8. The latter invocation corresponds to the non-tree edge $u \to v$ in G. Because D_0 labels v, as in Case 1-B, the sequence of resolutions on the backward path that begins $v \to u$ and continues back to the root by reversing the DSFT path to u must remove all literals of D_0. So $D_0 \subseteq [P, \neg q]$. Let the acceptable-clause policy for $\mathbf{PoolRes}([P, \neg q], \mathcal{F})$ accept D_0 at line 4. So D_0 is returned at line 5 and the parent invocation will create (an isomorphic copy of) the edge $u \to v$. The case where v corresponds to D_1 is similar. □

Corollary 4.5. Let G be a rooted resolution refutation DAG (based on input formula $\mathcal{F}$) that contains only standard resolution operations (i.e., the clashing literal q is present in one operand and $\neg q$ is present in the other operand and $\top$ never occurs). If G was produced by regular resolution, then G is a pool resolution DAG.

Proof. Any DFST for G is a regular DAG. Apply Theorem 4.4. □

5 Exponential Separation of Pool Resolution from Regular Resolution

In this section we consider a family of graphs $\mathrm{NT}^*(n)$, which has $N = n(n-1)$ variables. It is known that there is a positive constant α such that any regular refutation DAG for $\mathrm{NT}^*(n)$ has more than $2^{\alpha n}$ vertices, for large enough n, whereas general refutations of length $\Theta(n^3)$ are known (see discussion of Theorem 5.7 below). An empirical test indicates that `zChaff` [28] takes time proportional to $e^{0.75\,n}$ on this family, although it is not limited by the regularity restriction. We introduce a related family $\mathrm{NT}^{**}(n)$ whose regular refutations are at least as long, but has a pool refutation DAG with $O(n^3)$ vertices. The name NT is an abbreviation for "no triangles."

5.1 The Family $\mathrm{NT}^*(n)$ and Related Formulas

The definition of the family $\mathrm{NT}^*(n)$ is facilitated by some terminology. For this section, let n be a fixed positive integer, and let $N = n(n-1)$. Some expressions will depend implicitly on n.

For all of the formulas considered, there is an underlying semantic interpretation that guides our understanding. We suppose there is a set W whose elements are denoted w_i, $1 \le i \le n$. The propositional variables of $\mathrm{NT}^*(n)$ and related formulas correspond to possible directed edges between distinct elements of this set. A variable is true if the edge is present.

Definition 5.1. Let $\langle i, j\rangle$, where i and j are distinct integers in the range $[1, n]$ (1 through n) denote a propositional variable (the semantic interpretation is $w_i \to w_j$). Define $V = \{\langle i, j\rangle\}$. □

Definition 5.2. A *qualifying triple* is an ordered triple of distinct positive integers (i, j, k) in the range 1 through n, such that i is the maximum of the three; i.e., $1 \le j < i \le n$, $1 \le k < i$, and $j \ne k$. The set of all qualifying triples is denoted by Q. There are $n(n-1)(n-2)/3$ qualifying triples.

An integer-valued function $f(i, j, k) : Q \to [0, N-1]$ is called *β-fair* if it maps at least $\beta\, n$ qualifying triples into each value in its range. □

Definition 5.3. Let $\pi : V \to [0, N-1]$ be the permutation of V that arranges its elements in lexicographical order. Define $s : [0, N-1] \to V$ by the equation $s(x) = \pi^{-1}(x)$. For example, $s(0) = \langle 1, 2\rangle$, $s(2n) = \langle 3, 4\rangle$, etc. (The results hold for $\pi(\langle i, j\rangle)$ being *any* permutation of V, but this degree of generality is not important.) □

Definition 5.4. Clauses are named as follows for indexes indicated. In clause types A_0 and A_1 $r(i, j, k)$ is some function whose range is $[0, N-1]$ (see Definition 5.2 and (8) for particulars).

$$C(j) \equiv [\langle 1, j\rangle, \ldots, \langle j-1, j\rangle, \langle j+1, , j\rangle, \ldots, \langle n, j\rangle] \qquad 1 \le j \le n \qquad (1)$$

$$B(i, j) \equiv [\neg\langle i, j\rangle, \neg\langle j, i\rangle] \qquad 1 \le i < j \le n \qquad (2)$$

$$B^+(i,j) \equiv [\langle i,j\rangle, \langle j,i\rangle] \qquad 1 \le i < j \le n \tag{3}$$

$$A_0(i,j,k) \equiv [\neg\langle i,j\rangle, \neg\langle j,k\rangle, \neg\langle k,i\rangle, \neg s(r(i,j,k))] \qquad (i,j,k) \in Q \tag{4}$$

$$A_1(i,j,k) \equiv [\neg\langle i,j\rangle, \neg\langle j,k\rangle, \neg\langle k,i\rangle, \ s(r(i,j,k))] \qquad (i,j,k) \in Q \tag{5}$$

$$A(i,j,k) \equiv [\neg\langle i,j\rangle, \neg\langle j,k\rangle, \neg\langle k,i\rangle] \qquad (i,j,k) \in Q \tag{6}$$

$$T(i,j,k) \equiv [\neg\langle i,j\rangle, \neg\langle j,k\rangle, \neg\langle k,i\rangle] \qquad 1 \le i,j,k \le n \text{ and } i,j,k \text{ distinct.} \tag{7}$$

The $C(j)$ are called *long clauses*; the others are *short clauses.* □

Definition 5.5. Formulas are named as follows:

Formula name	*Clauses included*
$\mathrm{NT}^*(n)$	$C(j)$, $B(i,j)$, $B^+(i,j)$, $A_0(i,j,k)$, $A_1(i,j,k)$
$\mathrm{NT}(n)$	$C(j)$, $B(i,j)$, $B^+(i,j)$, $A(i,j,k)$
$\mathrm{GT}(n)$	$C(j)$, $B(i,j)$, $T(i,j,k)$

□

Note that $A_0(i,j,k)$ and $A_1(i,j,k)$ can be resolved to produce $A(i,j,k)$, after which $A_0(i,j,k)$ and $A_1(i,j,k)$ are subsumed and can be discarded. Further resolutions with B^+ clauses produce *transitivity* clauses $T(i,j,k)$ for all distinct (i,j,k) triples. Thus any model of the short clauses of $\mathrm{NT}^*(n)$ must be a complete linear order, where $x \to y$ is interpreted as $x > y$. The same holds for the short clauses of $\mathrm{NT}(n)$ and the short clauses of $\mathrm{GT}(n)$. But with this interpretation, $C(j)$ states that w_j is not a maximal element. Thus $\mathrm{NT}^*(n)$, $\mathrm{NT}(n)$ and $\mathrm{GT}(n)$ are unsatisfiable. The earlier work on the proof complexity of these families is reviewed here.

Theorem 5.6. ([22, 8, 1]) The families $\mathrm{GT}(n)$ and $\mathrm{NT}(n)$ have regular refutations of length $O(n^3)$. □

The family $\mathrm{NT}^*(n)$ is a variant of the family $\mathrm{GT}'(n)$ invented by Alekhnovitch, Johannsen, Pitassi and Urquhart [1]; the modifications are introduced to avoid some possible minor technical problems with the original definitions. They prove very ingeniously that if $r(i,j,k)$ is β-fair, then any regular refutation of $\mathrm{NT}^*(n)$ requires at least $2^{0.1\,\beta\,n}$ resolution steps. They define (in effect—their notation is different) a particular $r(i,j,k)$ and claim that it is 1-fair; but the claim was not proved and turns out to be incorrect.[1] However, the idea of the proof is perfectly sound and only needs to be adjusted for an achievable value of β.

Theorem 5.7. ([1]) Any regular refutation of $\mathrm{NT}^*(n)$ requires at least $2^{0.02\,n}$ resolution steps.

Proof. (Sketch) The following function can be shown to be β-fair for $\beta = 0.2$ and $n \ge 50$:

$$r(i,j,k) = ((n+1)\,i + 2\,n\,j + k) \bmod N \tag{8}$$

[1] Their function was actually 0-fair, as it did not map *any* distinct triples to n or $2n+1$, among other values.

To show that any x in the range $[0, N-1]$ is mapped to by at least $\beta\, n$ qualifying triples, it is convenient to consider four cases according to whether $\lfloor x/n \rfloor < n/2$ and whether $x \bmod n < n/2$. Then the analysis of each case is straightforward. The rest of the proof is the same as in the cited paper [1]. □

5.2 The Family $\mathrm{NT}^{**}(n)$

To demonstrate an exponential separation between regular resolution and pool resolution, we introduce the family of formulas $\mathrm{NT}^{**}(n)$, inspired by Beame *et al.* [4]. The formula $\mathrm{NT}^{**}(n)$ contains all variables in $\mathrm{NT}^{*}(n)$, plus the variables $x_{i,j,k}$ and $y_{i,j,k}$ for each qualifying triple $(i,j,k) \in Q$. The formula $\mathrm{NT}^{**}(n)$ consists of all clauses in $\mathrm{NT}^{*}(n)$, plus the following

$$\begin{aligned} &[x_{i,j,k}, \neg y_{i,j,k}]\,;\ [\neg x_{i,j,k}, y_{i,j,k}]\,; \\ &[x_{i,j,k}, \langle i,j\rangle]\,;\ [x_{i,j,k}, \langle j,k\rangle]\,;\ [x_{i,j,k}, \langle k,i\rangle]\,; \qquad (i,j,k) \in Q \end{aligned} \tag{9}$$

The variables $x_{i,j,k}$ are called *proof-trace* variables [4]. The clauses containing $y_{i,j,k}$ are added so that the $x_{i,j,k}$ are not pure literals.

Theorem 5.8. Any regular refutation of $\mathrm{NT}^{**}(n)$ requires at least $2^{0.02\,n}$ resolution steps.

Proof. By setting all $x_{i,j,k} = 1$ in $\mathrm{NT}^{**}(n)$, and simplifying, the resulting clauses are those in $\mathrm{NT}^{*}(n)$, plus unit clauses $[y_{i,j,k}]$, which are useless for refuting $\mathrm{NT}^{*}(n)$. By Definition 2 and Proposition 1 of Beame *et al.* [4], the shortest regular refutation of $\mathrm{NT}^{**}(n)$ is at least as long as the shortest regular refutation of $\mathrm{NT}^{*}(n)$. The lower bound follows by Theorem 5.7. □

The idea for the polynomial-length pool refutation of $\mathrm{NT}^{**}(n)$ is to use the proof-trace variables $x_{i,j,k}$ to derive $A(i,j,k)$ from $A_0(i,j,k)$ and $A_1(i,j,k)$ for all $(i,j,k) \in Q$ without leaving any variables of $\mathrm{NT}(n)$ in the pool. Then the pool refutation can proceed as it would for $\mathrm{NT}(n)$ (no asterisks), treating $A(i,j,k)$ as input clauses. The resulting entire refutation is not regular because some non-tree paths leading to $A(i,j,k)$ have $r(s(i,j,k))$ or $\neg r(s(i,j,k))$ as a clashing literal.

Theorem 5.9. The formula $\mathrm{NT}^{**}(n)$ has a pool refutation with $O(n^3)$ steps.

Proof. Beginning with an empty pool, derive $A(i,j,k)$ for each $(i,j,k) \in Q$ as indicated in Figure 2; after each derivation $\neg x_{i,j,k}$ and $\neg y_{i,j,k}$ are added to the pool on the right branch. This part of the derivation requires $O(n^3)$ steps.

After all $A(i,j,k)$ have been derived, attach a pool refutation of $\mathrm{NT}(n)$ of size $O(n^3)$ to the lowest vertex on the rightmost branch. Such a refutation exists by Theorem 5.6 and Corollary 4.5. The derived clause at this vertex is $\bot$, so it is propagated back up to the root of the entire derivation. □

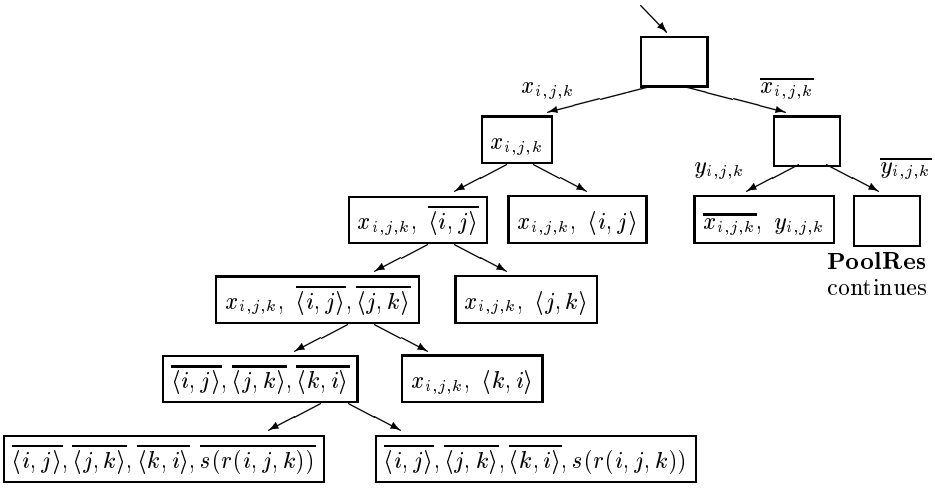

Fig. 2. Derivation of all $A(i,j,k)$ without leaving any V variables in the pool. Overbars denote negation.

6 Relation of Pool Resolution to DPLL with Clause Learning

Recall the pseudocode of **PoolRes** in Figure 1. Any execution of a DPLL-style search, including popular methods of "clause learning," can be simulated by **PoolRes** by following two basic principles:

1. True assigned literals in DPLL are the *negations* of pool literals in **PoolRes**.
2. "Learned clauses" in the DPLL version are *the same as* (a subset of) clauses derived by **PoolRes**.

Assignment of a literal $q = 1$ as a "decision" (backtrackable "guess," use of splitting rule) in DPLL corresponds to adding $\neg q$ to the pool at line 8. Backtracking to the assignment $q = 0$ in DPLL corresponds to adding q to the pool at line 11.

When DPLL deletes clauses that are satisfied, this corresponds to such clauses becoming ineligible in **PoolRes**. When DPLL shortens clauses due to complements of true literals, the remaining literals are just the nonpool literals in **PoolRes**.

Unit clause propagation in DPLL, say assigning $x = 1$, following a "guess" or "backtrack" assignment to q is simulated in **PoolRes** by adding x to the pool at line 8 and adding $\neg x$ to the pool at line 11. See Section 6.1.

However, if DPLL is enhanced with *preorder reasoning* operations, the corresponding operations would need to be added to **PoolRes** to enable it to continue simulating the search procedure. Examples of *preorder reasoning* include

equivalent-literal recognition, binary-clause reasoning, subsumption resolution, etc. Such operations are difficult to combine efficiently with clause learning and most leading satisfiability solvers do not implement them.

Several clause learning schemes have been analyzed by Zhang *et al.* [28], and more formally by Beame *et al.* [4]. They are primarily outgrowths of the GRASP scheme [18], and much of the terminology originates from that paper. Please see these papers for details. We show how pool resolution can simulate them.

6.1 Pool Expansion Strategies

Recall the pseudocode of **PoolRes** in Figure 1. As mentioned, the policy for choosing q at line 7 is crucial for both theoretical and practical performance. It is useful, at least in practice, to define the *nonpool literals* in an eligible clause C to be those literals in C that are not in the pool P. The *nonpool count* for C is the number of such nonpool literals, i.e., $|C - P|$. If the nonpool count is 0 (and C is "acceptable"), the base case of lines 4 and 5 applies.

If the nonpool count is 1, C is analogous to a unit clause in DPLL. In this case, let $\neg q$ be the sole nonpool literal of C and choose q as the clashing literal at line 7. The recursive call at line 8 returns immediately, with $D_0 = C$ or some other "acceptable" clause whose only nonpool literal was $\neg q$. Then for the second recursive call, at line 11, the set of eligible clauses is reduced by discarding all clauses containing $\neg q$. Thus the problem has been simplified without branching.

Another practical strategy is analogous to the pure literal rule of DPLL. If q is a nonpool literal and no eligible clause contains $\neg q$, choose q as the clashing literal at line 7. Now $\neg q$ is added to the pool at line 8, causing all clauses containing q to become ineligible. In addition, line 10 (if reached) must be true. Thus the problem has been simplified without branching.

6.2 Correspondence with RelSat Learning

The learning procedure of RelSat [3] has the simplest correspondence with **PoolRes**. Say $q = 1$ was a "decision" assignment and $\bot$ was derived, possibly after some additional assignments by unit-clause propagation. RelSat "learns" a clause of the form $[\neg q, \alpha]$ where the subclause α consists of complements of (some of the) literals that were assigned true before the $q = 1$ "decision." Using the straightforward simulation described above, **PoolRes** derives the same clause in the procedure invocation where q was chosen as the clashing literal at line 7.

6.3 Correspondence with GRASP, First UIP

Marques-Silva and Sakallah [18] introduced the term *unique implication point* (UIP) to refer to a vertex, say x, in their implication graph such that all paths from the decision literal, say q, to $\bot$ pass through x. Van Gelder and Okushi [24] independently analyzed similar structures, used the term *articulation point*, and gave a linear-time algorithm for detecting them.

If x is a UIP, then a clause $[\neg x, \alpha]$ can be inferred from the implication graph, where subclause α consists of complements of (some of the) literals that were assigned true before the $q = 1$ "decision." This clause is called the *UIP clause*. The decision literal is always a UIP and gives rise to the RelSat clause, as in Section 6.2.

Marques-Silva and Sakallah studied the scheme consisting of learning the UIP clause of the *first UIP*, i.e., the one closest to $\perp$ in the implication graph. Suppose $q = 1$ is the decision literal, $x \neq q$ is the first UIP, and $[\neg x, \alpha]$ is the UIP clause. Using the straightforward simulation described above, **PoolRes** derives the same clause in the procedure invocation that chooses $\neg x$ as the clashing literal at line 7, to simulate the unit-clause propagation assignment $x = 1$ in DPLL. The call at line 8 returns the same D_0 that was used for the antecedent edges of x in the implication graph; this applies to all literals that DPLL assigns through unit-clause propagation. Then $\neg x$ is added to the pool and $D_1 = [\neg x, \alpha]$ is derived and returned at line 11.

However, GRASP and some other search engines have an option to learn *only* the first UIP clause. When $x \neq q$, this means they do not backtrack to the assignment $q = 0$. Instead, they erase all assignments at the current "decision level," i.e., those at and after $q = 1$, then they learn (assert) the first UIP clause, $[\neg x, \alpha]$. At this point all literals of α are false, so $\neg x$ becomes a *failure driven assertion*.

For **PoolRes** to simulate first UIP, it needs to "look ahead" at the point where $q = 1$ is the "decision" and anticipate that x will become the first UIP. So **PoolRes** skips choosing q as the clashing literal, and skips subsequent steps until $x = 1$ is assigned by unit-clause propagation. At this point it chooses x as the clashing literal. That is, $\neg x$ is added to the pool at line 8 as though x were a decision variable in DPLL. Next it "plugs in" the same derivation that occurred in the RelSat style simulation, except that $\neg x$ was added to the pool at line 11 in that simulation, as described a few paragraphs above. Thus $D_0 = [\neg x, \alpha]$ is derived and returned. Then it adds x to the pool at line 11, which simulates GRASP's failure-driven assertion of $\neg x$.

6.4 Correspondence with Decision Learning

The *decision learning* strategy requires the learned clause to contain only the negations of "decision" literals. For **PoolRes** to simulate this strategy, it only simulates "decision" assignments, and defers all unit-clause propagations until a decision assignment has been made that allows unit-clause propagation to derive $\perp$. Suppose that decision is $q = 1$. The decision-literal UIP clause can be derived, and contains only decision literals. All derived clause are available for later use, so those that depend only on decision literals at early levels can potentially be used in many branches.

6.5 Correspondence with FirstNewCut

The *First New Cut strategy* was proposed recently by Beame *et al.* [4]. We refer the reader to that paper for details. Relying on their Proposition 4, the clause specified as First New Cut, can be derived by what they define as a *trivial resolution*. This involves a series of resolution steps in a chain each one with a different clashing literal.

Such a *trivial resolution* is easy to simulate with pool resolution: just add the clashing literals to the pool in the reverse order of the trivial resolution, and derive the clauses on the way back out of the recursions.

The idea to simulate the *First New Cut strategy* is to simulate only the decision assignments. The pool P contains their complements. When a contradiction can be derived, say after the decision $q = 1$, let the pool be $[P, \neg q]$. Simulate the implication graph by adding literals to the pool in a topological order consistent with the implication graph, such that all literals on the *opposite* side of the cut from the empty clause are added before any on the same side. This policy derives the First New Cut clause. Continue derivations based on the implication graph until $[P, \neg q]^-$ has been derived. Continue with the pool $[P, q]$.

7 Conclusion

We introduce a system called *pool resolution* and show that it simulates regular resolution linearly and has exponentially shorter refutations on at least one family of formulas. This paper draws heavily on earlier work in proof complexity [1, 4]. Thus pool resolution is one of the strongest known refinements of general resolution. Whether it has the full power of general resolution (within a polynomial factor) is unknown, but seems unlikely.

We also show that pool resolution is able to simulate several strategies for clause learning within DPLL. These simulations are natural enough that they provide some hope that most of the power of pool resolution can be realized, at least on practical problems, by some form of DPLL with clause learning. Beame *et al.* have obtained related results for their *First New Cut* learning strategy [4].

References

1. Alekhnovich, M., Johannsen, J., Pitassi, T., Urquhart, A.: An exponential separation between regular and unrestricted resolution. In: Proc. 34th ACM Symposium on Theory of Computing. (2002) 448–456
2. Anderson, R., Bledsoe, W.W.: A linear format for resolution with merging and a new technique for establishing completeness. Journal of the ACM **17** (1970) 525–534
3. Bayardo, Jr., R.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proc. AAAI. (1997) 203–208
4. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351

5. Beame, P., Pitassi, T.: Simplified and improved resolution lower bounds. In: Proc. 28th ACM Symposium on Theory of Computing. (1996)
6. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow — resolution made simple. JACM **48** (2001) 149–168
7. Bonet, M., Galesi, N.: A study of proof search algorithms for resolution and polynomial calculus. In: Proc. 40th Symposium on Foundations of Computer Science. (1999) 422–432
8. Bonet, M., Galesi, N.: Optimality of size-width tradeoffs for resolution. Computational Complexity **10** (2001) 261–276
9. Clegg, M., Edmonds, J., Impagliazzo, R.: Using the Groebner basis algorithm to find proofs of unsatisfiability. In: Proc. 28th ACM Symposium on Theory of Computing. (1996) 174–183
10. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
11. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the Association for Computing Machinery **7** (1960) 201–215
12. Dechter, R., Rish, I.: Directional resolution: the davis-putnam procedure, revisited. In: Proc. 4th Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'94), Morgan Kaufmann, San Francisco (1994) 134–145
13. Goldberg, E., Novikov, Y.: Berkmin: a fast and robust sat-solver. In: Proc. Design, Automation and Test in Europe. (2002) 142–149
14. Johnson, D.S., Trick, M.A., eds.: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996)
15. Krishnamurthy, B.: Short proofs for tricky formulas. Acta Informatica **22** (1985) 253–274
16. Lee, S.J., Plaisted, D.A.: Eliminating duplication with the hyper-linking strategy. Journal of Automated Reasoning **9** (1992) 25–42
17. Letz, R., Mayr, K., Goller, C.: Controlled integration of the cut rule into connection tableau calculi. Journal of Automated Reasoning **13** (1994) 297–337
18. Marques-Silva, J.P., Sakallah, K.A.: GRASP–a search algorithm for propositional satisfiability. IEEE Transactions on Computers **48** (1999) 506–521
19. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 39th Design Automation Conference. (2001)
20. Plaisted, D.A. (private communication) (1984)
21. Plaisted, D.A.: The search efficiency of theorem proving strategies. In: 12th International Conference on Automated Deduction, Springer-Verlag (1994) 57–71
22. Stälmarck, G.: Short resolution proofs for a sequence of tricky formulas. Acta Informatica **33** (1996) 277–280
23. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In Slisenko, A.O., ed.: Seminars in Mathematics v. 8: Studies in Constructive Mathematics and Mathematical Logic, Part II. Steklov Math. Inst., Leningrad (1968) 115–125 (English trans., 1970, Plenum.).
24. Van Gelder, A., Okushi, F.: Lemma and cut strategies for propositional model elimination. Annals of Mathematics and Artificial Intelligence **26** (1999) 113–132
25. Van Gelder, A., Tsuji, Y.K.: Incomplete thoughts about incomplete satisfiability procedures. In: Second DIMACS Challenge Workshop: Cliques, Coloring and Satisfiability. (1993) (also at `ftp://ftp.cse.ucsc.edu/pub/avg/incomplete.ps.Z`).

26. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. In Johnson, D.S., Trick, M., eds.: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society (1996) (also at
`ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.Z`).
27. Zhang, H., Stickel, M.E.: Implementing the davis-putnam method. Journal of Automated Reasoning **24** (2000) 277–296
28. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: ICCAD. (2001)