Towards a Theoretical Understanding of the Power of Restart in SAT solvers

by

Chunxiao Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Abstract

Restarts are a widely used class of techniques integral to the efficiency of Conflict-Driven Clause Learning (CDCL) SAT solvers. While the utility of such policies has been well-established empirically, to-date we still lack a deep theoretical understanding of why restart policies are crucial to the power of CDCL SAT solvers.

This thesis first presents studies on three classes of formulas which were conjectured to be able to exponentially separate a solver configuration S with restarts and S with no restarts. However we were able to prove that these candidates classes of formulas are not even sufficient to super-polynomially separate the two solver configurations.

We then provide a series of results that theoretically provide evidences for establishing the power of restarts for various models of Boolean SAT solvers. More precisely, we make the following contributions. First, we introduce a new class of *s*atisfiable instances called $Ladder_n$ and use it to construct another formula F_T, and prove that for the *drunk* randomized DPLL SAT solver D (introduced by Alekhnovich and Razborov), the configuration D with restarts can solve $Ladder_n$ formulas in sub-exponential time in size of $Ladder_n$, while D without restarts requires exponential time in the size of $Ladder_n$, with high probability. Two crucial insights enabled us to prove this separation result for restarts: first, we changed the focus from unsatisfiable instances to satisfiable ones; second, we observed that, at least for the models we considered, restart heuristics add proof-theoretic or algorithmic power by compensating for the weaknesses in some other important heuristic like value selection. Second, we introduce a key new notion used in above-mentioned proofs, called *Decision Complexity* $dc(\varphi)$, for DPLL proofs of an unsatisfiable formula φ and show that size of DPLL proofs of φ are lower bounded by $2^{dc(\varphi)}$.

Acknowledgements

I would like to thank my parents and my uncle's family for supporting my study in Canada, and always being there to back me up. I would also like to thank my advisor Vijay Ganesh for spending lots of time patiently guiding me through research and life. I would not be able to complete this thesis without Dr. Ganesh's mentorship.

Dedication

This is dedicated to mom and dad, along with ones I love.

Table of Contents

List of Figures

Chapter 1

Introduction

Boolean SAT solvers have been dominant tools used across variety of domains in computer science and software engineering. Researchers and practitioners encode problems they are interested in into Boolean formulas, and try to solve them using a SAT solver. Even though the SAT problem is NP-complete [11], modern Conflict-Driven Clause Learning (CDCL) SAT solvers [25, 26] have proven to be extremely successful and can solve industrial instances with millions of variables.

While most of the work in the field of SAT solvers has been focused on improving the efficiency of solvers from an empirical point of view, there have also been significant advances in understanding SAT solvers from a proof complexity-theoretic point of view. For instance, in their seminal paper, Pipatsrisawat and Darwiche [30] proved that CDCL SAT solvers with non-deterministic variable and value selection, and with restarts (as well as asserting clause learning schemes) are polynomially equivalent to general resolution. Around the same time, Atserias et al. independently showed that CDCL SAT solvers with sufficient randomness in branching and restarts can polynomially simulate bounded-width resolution with high probability [3]. However, it remains unknown if these polynomial equivalence results still hold when restarts are disabled.

More generally, despite considerable effort for nearly two decades, we still do not have a complete picture of why restarts are so crucial to the efficiency of CDCL SAT solvers. Recently, Bonet, Buss and Johannsen [9] showed that CDCL SAT solvers with no restart (but with non-deterministic variable and value selection) are strictly more powerful than regular resolution, and they also refuted several candidate class of formulas which were conjectured to be separators for restarts. This is a strong result, given the well-known theorem of Alekhnovich that regular resolution is strictly weaker than general resolution

[2].

In this thesis, we address the question of the power of restarts in both the DPLL [14, 13] and CDCL SAT solver settings. Specifically, we present three results on the power of restarts: First, we proved that three classes of formulas which were conjectured to be separators for exponentially separating models of CDCL SAT solver with non-deterministic *dynamic* variable and value selection with and without restart are not sufficient to even superpolynomially separate the two. Second, we introduce a new class of $s$atisfiable instances called $Ladder_n$ and use it to prove that for the *drunk* randomized DPLL SAT solver model (D_{ND_RD}) introduced by Alekhnovich and Razborov [1], the configuration of D_{ND_RD} with restarts can solve $Ladder_n$ formulas in sub-exponential time in size of $Ladder_n$, while the configuration of D_{ND_RD} without restarts requires time exponential in the size of $Ladder_n$, with high probability (w.h.p.). Finally, we show that models of CDCL SAT solvers with non-deterministic *static* variable and value selection, and with restarts are no more powerful from a proof-complexity theoretic point of view than the same configurations without restarts.

As we studied this problem in depth, we made the following observation about restarts, namely, that there seems to be a subtle interplay between various solver heuristics and restarts, wherein the power of restarts becomes apparent only when one or more important heuristics (e.g., variable or value selection) are weakened (e.g., solver configurations where non-deterministic value or variable selection heuristics are replaced by their weaker randomized cousins). Another observation we made was that all previous theoretical work on the power of restarts seems to have focused on treating solvers as proof systems, i.e., on unsatisfiable instances only. For one of our result, we show that one can separate certain solver configurations with and without restarts, if we change our perspective from unsatisfiable to satisfiable instances (see Section 4.4). All our results hold irrespective of the computational overhead of the various heuristics considered in this thesis.

Contribution

1. First, we prove that three classes of formulas which were conjectured to be able to exponentially separate CDCL SAT solvers with non-deterministic variable selection, value selection and restart from the same solver without restart are not sufficient to even superpolynomially separate the two solver configurations. (See Chapter 3 for details.)

2. Second, we show that CDCL SAT solvers with non-deterministic static variable selection, non-deterministic static value selection, and with restarts, are polynomially

equivalent to the same model but without restarts. In fact, our result is stronger, in that, both configurations produce the exact same proof for the same unsatisfiable input formula. Further, their runs are identical for satisfiable instances as well. Finally, the result holds irrespective of the choice of learning scheme. (See Section 4.1 for details.)

3. Third, we introduce the notion of *decision complexity* of an unsatisfiable formula φ, and prove that the size of any DPLL proof for φ is lower bounded by $2^{dc(\varphi)}$. We use this result to prove the separation result mentioned below. Additionally, we believe that this lower bound theorem is of independent interest to the proof complexity theory community (See Section 4.2 for details.)

4. Finally, we prove that DPLL SAT solvers with non-deterministic variable selection, arbitrary value selection, and with restarts are polynomially equivalent to DPLL SAT solvers with the same configuration but with no restarts. (In fact, this result is not surprising at all.) However, very surprisingly, when we shift our focus to satisfiable instances we are able to prove a separation. More precisely, we prove that DPLL SAT solvers with non-deterministic variable selection, restarts, and randomized value selection are exponentially faster than the same model, but without restarts, with high probability. To prove this result we introduce a class of formulas we refer to as $Ladder_n$ formulas, which we believe are of independent interest to the proof complexity theory community. (See Sections 4.3 and 4.4 for details.)

Chapter 2

Background

In this chapter, we present definitions and concepts which we use for the rest of the thesis.

2.1 SAT Solving

CDCL SAT solvers can be viewed as extensions of DPLL SAT solvers. Both DPLL SAT solvers and CDCL SAT solvers are backtracking based algorithms with a handful of key components. At its core, DPLL SAT solvers relies on the following components: *variable selection*, *value selection*, *Boolean constraint propagation*(BCP) and *restart*. On the other hand, on top of all the key components that are also in DPLL SAT solvers, CDCL SAT solvers has one more essential component called *conflict analysis* which gives CDCL SAT solvers exponential speedup comparing to DPLL SAT solvers. On a high level, conflict analysis analyzes why certain partial assignments are not satisfying, and produces so called *learnt clauses* which are later added to the input formula to help speed up the search. In this section, we describe how DPLL and CDCL SAT solver works including input format, heuristics and how does a solver search for satisfying assignments.

2.1.1 Preliminary

A typical SAT solver takes input a Boolean formula in *conjunctive normal form*(CNF). In order to define what CNF is, we start by presenting notions needed to define a formula.

A *Boolean variable* used to define a formula is a variable that can take either the value **true** (or $\top$) or the value **false** (or $\bot$). A *literal* is defined to be either a variable x or its negation $\neg x$. A *clause* is a disjunction of literals, and we say a clause c has width w if c contains w literals. A CNF formula is a conjunction of clauses. And the size of a formula is the number of clauses in it.

Consider a CNF formula φ. An assignment p is a map from variables in φ to truth values. We say p is empty if p is an empty map. We say p is *complete* if p maps all variables in p to truth values, and we say p is *partial* if p does not map some of the variables in φ to truth values. (We sometimes use the term "assign" instead of "map".) A clause c is *satisfied* under a partial assignment p if p assigns some literal in C to $\top$. And c is *falsified* under p if p assigns every literal in C to $\bot$, we also say c is *conflicting*. And we say C is *unit* under p if p assigns all but one literal in C to $\bot$.

Restriction: We use $F[x]$ to denote the restricted (or simplified) formula of F after the variable x is set to $\top$, and similarly, we use $F[\neg x]$ to denote the restricted formula of F after the variable x is set to $\bot$.

2.1.2 DPLL SAT solvers

On a high level, a DPLL solver S starts with an empty assignment p, and then picks a variable x using variable selection function and assigns it a value using value selection function, and we say such x is a *decision variable*. After adding the assignment to the variable to the empty assignment p, S performs BCP and then checks if any clause is falsified under p, and we call the variables that are added during the BCP subroutine *propagated variables*. If there is such a clause, the solver backtracks and re-assigns x to the opposite value (and now the variable x becomes a propagated variable), and then updates p and recursively calls itself with φp. If there is no conflicting clause under p, the solver picks another variable and assigns it a value, and then adds it to p and recursively calls itself with $\varphi[p]$. We first describe the key components in a DPLL SAT solver and then present the pseudocode of the DPLL SAT solving algorithm in Algortihm 1.

Variable selection: Variable selection function takes input the state of the solver and outputs a variable that has not been assigned a value.

Value selection: Value selection function takes input the state of the solver and a variable, then outputs a truth value.

Algorithm 1 The DPLL SAT Solving Algorithm

```
 1: function DPLL($\phi$, $\mu$)
 2:     Input: A CNF formula $\varphi$, and an assignment $p$
 3:     Output: true (SAT) or false (UNSAT)
 4:
 5:     bcp_ret = BCP($\varphi$,$p$);
 6:     if (bcp_ret == CONFLICT) then              ▷ If top-level conflict, return UNSAT
 7:         dpll_ret = false;
 8:     else
 9:         if (All variables have been assigned) then       ▷ If solution found, return SAT
10:             dpll_ret = true;
11:         else (Select decision variable $x$)       ▷ Select an unassigned variable $x$ using
    variable selection function
12:             dpll_ret = (DPLL($\varphi$, $p \cup$ valueselection($x$))
13:                         || DPLL($\varphi$, $p \cup \neg$valueselection($x$)));        ▷ Recurse DPLL
14:         end if
15:     end if
16:     return dpll_ret;
17: end function
```

Boolean constraint propagation(BCP): BCP is a key component in both DPLL and CDCL SAT solvers, and it is believed to be the main workhorse for both DPLL and CDCL SAT solvers. The BCP function takes input a partial assignment p and a CNF formula φ, if φ has a unit clause under p, BCP propagates the unit literal l to $\top$ and adds it to the partial assignment and extends p. We say BCP is formed till "saturation" if the BCP function then recursively invoke BCP with input the extended assignment p and the formula φ.

2.1.3 CDCL SAT solvers

CDCL SAT solvers are extension of DPLL SAT solvers. CDCL SAT solvers kept all the key components in DPLL SAT solvers, but CDCL SAT solvers have an extra component, conflict analysis, which is crucial to the solvers' success. Similar to DPLL SAT solvers, CDCL SAT solvers take input CNF formulas, and start the search by simplifying the formula using BCP. And then generates a decision variable and assign it a value using value selection. And lastly recursively calls itself with a updated partial assignment. However, upon detecting a conflict, CDCL SAT solvers do not simply backtrack the last decision variable, but instead analyze which literals on the current partial assignment contributed to the conflict using conflict analysis. By producing a learnt clause which asserts that the solver should not make the same mistake again, CDCL SAT solver then add the learnt clause to the formula. The pseudocode of the DPLL SAT solving algorithm is presented in Algortihm 2.

2.1.4 Solvers as Proof Systems

All results in this thesis relies on the connection between solvers and proof systems. We are not the first ones to study solvers as proof systems. For instance, in their seminal paper, Knot Pipatsrisawat and Adnan Darwiche [30] proved that CDCL SAT solvers with non-deterministic variable and value selection, and with restarts (as well as asserting clause learning schemes) are polynomially equivalent to general resolution. Around the same time, Albert Atserias, Johannes Klaus Fichte and Marc Thurley independently showed that CDCL SAT solvers with sufficient randomness in branching and restarts can polynomially simulate bounded-width resolution with high probability [3]. However, it remains unknown if these polynomial equivalence results still hold when restarts are disabled.

Just like solvers, proof systems can be viewed as blackboxes to solve the SAT problem. A propositional proof system takes input a Boolean formula, and produces proofs that certify the satisfiability of the formula. And proof complexity is the field of studying the complexity of the size of the proofs generated by a proof system. Fortunately, in proof complexity research literature, researchers have developed tools which are used for comparing the power of proof systems. Please find details about proof systems and proof complexity in Section 2.2.

At its core, proof complexity is about studying the complexity of proof systems, and comparing the power of different proof systems. Researchers do so by either establishing so called *p-equivalence* between proof systems to show two proof systems are as powerful, or prove a separation in the power of proof systems by showing the existence of a class of formulas for which one proof system can produce short proofs where the other solver can only produce large proofs.

2.1.5 SAT solver configurations

To leverage the tools and method used in proof complexity, we first have to first define a solver model mathematically, and we can then think solvers as proof systems.

Variable and Value selection We first present notations we use to characterize different variable and value selection functions.

1. **Non-deterministic Static (NS) Variable Selection Heuristic**: A non-deterministic algorithm that takes as input a formula, and outputs a total ordering (or ranking) of all variables of the input formula prior to solver's execution. During the solver's execution, the NS variable selection heuristic returns the unassigned variable with the highest rank in this total ordering.

2. **Non-deterministic Dynamic (ND) Variable Selection Heuristic**: A non-deterministic algorithm that, upon invocation, outputs an unassigned variable during the run of the solver.

3. **Non-deterministic Static (NS) Value Selection Heuristic**[1]: A non-deterministic algorithm that takes as input a formula and outputs a map from variables of the input formula to truth values, prior to the solver's execution. During the run of the solver, the NS value selection algorithm takes as input a variable, and returns its predetermined value.

4. **Random Dynamic (RD) Value Selection Heuristic**: A randomized algorithm takes as input a variable of the input formula and assigns it a truth value uniformly and independently (i.e., independent of any other invocation of the heuristic) at random.

Notation for SAT Solver Configurations. Below we present notation that is used to describe configurations of SAT solvers that we study in this paper:

1. **Solver Configurations C_{NS_NS} and $C^R_{NS\_NS}$:** By C_{NS_NS} we denote CDCL SAT solvers with non-deterministic static variable selection and value selection with no restarts, and by $C^R_{NS\_NS}$ we denote the same configuration with restarts. (Used in Section 4.1)

2. **Solver Configurations $D_{ND\_arbitrary}$ and $D^R_{ND\_arbitrary}$:** By the term *arbitrary* we mean that the value selection heuristic can be any algorithm whatsoever, that take as input a variable of the input formula, and output a truth value. By $D_{ND\_arbitrary}$ we denote DPLL SAT solvers with non-deterministic dynamic variable selection and

[1]While we use the same abbreviation NS for both non-deterministic variable and value selection, we do not anticipate any confusion since it will be very clear from context which heuristic is being referred to.

arbitrary value selection without restarts, and by $D^R_{ND\_arbitrary}$ we denote the same configuration with restarts. (Used in Section 4.4)

3. **Solver Configurations D_{ND_ND} and $D^R_{ND\_ND}$:** By D_{ND_ND} we denote DPLL SAT solvers with non-deterministic dynamic variable selection and value selection without restarts, and by $D^R_{ND\_ND}$ we denote the same configuration with restarts. (Used in Section 4.4)

4. **Solver Configurations D_{ND_RD} and $D^R_{ND\_RD}$:** By D_{ND_RD} we denote DPLL SAT solvers with non-deterministic dynamic variable selection and random dynamic value selection without restarts[2], and by $D^R_{ND\_RD}$ we denote the same configuration with restarts. We choose to refer to these models as drunk DPLL SAT solvers. (Used in Section 4.4)

2.1.6 Graph representation of DPLL proofs

We now define the graph representation of a DPLL proof, conventionally, DPLL proofs are represented as trees, where internal nodes are labelled with variables. In our definition, we choose to label the edges of a DPLL proof tree to better capture a run of a DPLL SAT solver, such definition allows us to define decision complexity, $dc(\varphi)$, of a formula φ. Further, with decision complexity, we can lower bound the size of DPLL proofs to be $2^{dc(\varphi)}$. The motivation of our lower bound result is as follows: The famous result by Ben-Sasson and Widgerson [6] suggests that one can use the resolution width of a formula to lower bound the size of tree-like/general resolution proofs. But resolution width is a proof-theoretic notion, a natural question one can then ask is whether we can show a similar lower bound result using algorithmic properties of solvers. decision complexity turns out to be one of such algorithmic properties, as it is a notion that naturally captures the behaviours of DPLL-style proof search algorithms. And lastly we think decision complexity can be a powerful tool in analyzing and understanding solver behaviours in CDCL setting as well.

DPLL proof trees with labels. A *DPLL proof tree* with labels is a binary tree used to represent a run of a DPLL SAT solver, where the nodes denote variables of the input formula and edges are labelled as follows: an edge is either labeled with "$d : l$" or "$p : l$", where l is a literal, and "$d : l$" means the literal l is set to $\top$ as a decision, and "$p : l$"

[2]This model is inspired by the drunk model proposed by Alekhnovich and Razborov [2]. In their paper, they defined drunk model as DPLL SAT solvers with arbitrary variable selection and random value selection.

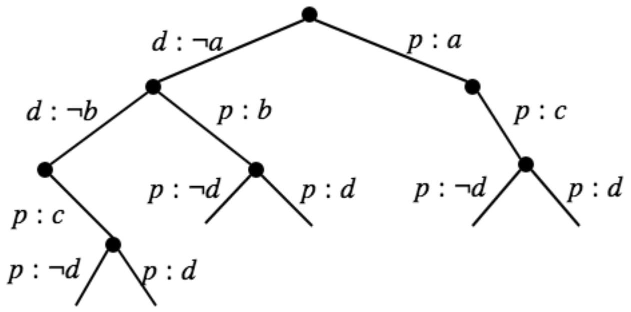

Figure 2.1: A DPLL proof tree with labels for $(a \vee b \vee c) \wedge (b \vee \neg c \vee d) \wedge (a \vee \neg c \vee \neg d) \wedge (a \vee \neg b \vee d) \wedge (\neg b \vee \neg d) \vee (\neg a \vee c) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg a \vee \neg d)$

means the literal l is set to $\top$ as a propagation. A DPLL proof tree is very similar to the tree-like resolution proof of a formula, with the only difference is that the edge labels have an additional tag of being a "decision branch" or a "propagation branch". Figure 2.1 presents a DPLL proof tree for an unsatisfiable (UNSAT) formula. Then from the definition above, if an internal node of the DPLL proof tree has only one child, then the edge connecting the child is labelled with a "$p : l$", for some literal l. And if a node has two children and that node is not a parent of two leaf nodes, then exactly one edge to one of its child is labelled with "$d : l$" and the edge to the other child is labelled with "$p : \neg l$" for some literal l.

Definition 2.1.1. *The decision complexity of a path p in a DPLL proof tree, denoted $dc(p)$, is the number of decision branches in p. The decision complexity of a DPLL proof tree T, denoted $dc(T)$, is the maximal decision complexity of a path in T, that is $dc(T) = max_{p \in T}\{dc(p)\}$. Lastly, the decision complexity of a formula φ, denoted $dc(\varphi)$, is the minimum decision complexity over all DPLL proof trees of φ.*

By our definition of decision complexity, the DPLL proof tree in Figure 2.1 has decision complexity 2, while the formula that it proves has decision complexity 1.

Definition 2.1.2. *We define a size-preserving transformation, label switch, over decision nodes in DPLL proof trees. A node in a DPLL proof tree is a decision node if it has two children that are not leaves. Applying label switch on a decision node x in a DPLL tree, denoted $ls(x)$, relabels the decision branch of x as a propagation branch, and relabels the propagation branch as a decision branch.*

Definition 2.1.3. *A relabelling for a DPLL tree T is defined over a sequence of label switch on decision nodes in T.*

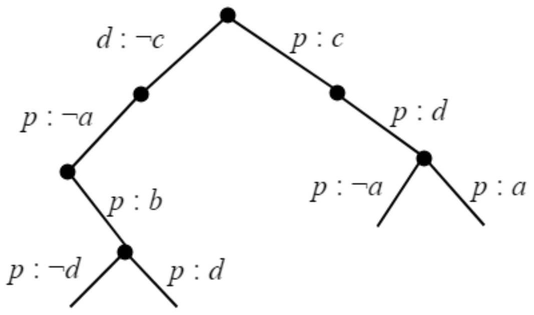

Figure 2.2: A DPLL proof tree of decision complexity 1 for the formula in Figure 2.1.

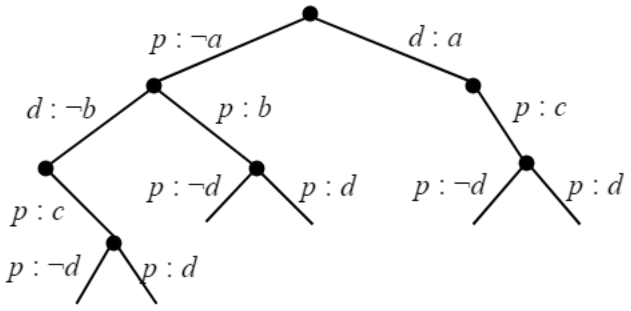

Figure 2.3: A relabelling of the DPLL proof tree in Figure 2.1 into a DPLL proof tree of the same size, but with decision complexity 1.

Figure 2.3 shows a relabelling of the DPLL proof tree in Figure 2.1, the new tree has same size as the original tree, however the new tree has decision complexity 1.

2.2 Proof Complexity

2.2.1 Proof system

In general, in the context of propositional proof systems, for any proof propositional proof system P. P produces proofs of unsatisfiability for the class of formulas $TAUT$, which is the class of all unsatisfiable formulas. P produces proofs of formulas in $TAUT$ using *deduction* with the help of *inference rules.*

2.2.2 p-equivalence

Definition 2.2.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be proof systems [12, 20]. We say that $\mathcal{A}$ polynomially simulates or p-simulates $\mathcal{B}$ (denoted as $\mathcal{A} \leq_p \mathcal{B}$) if for every proof π in $\mathcal{B}$, there is a proof in A of size at most $f(|\pi|)$, where f is a polynomial function.*

Definition 2.2.2. *Let $\mathcal{A}$ and $\mathcal{B}$ be proof systems. We say that $\mathcal{A}$ is polynomially equivalent or p-equivalent to $\mathcal{B}$ (denoted as $\mathcal{A} \sim_p \mathcal{B}$) if $\mathcal{A} \leq_p \mathcal{B}$ and $\mathcal{B} \leq_p \mathcal{A}$.*

2.3 Related Work

To the best of our knowledge, the first paper to discuss restarts in the context of SAT solvers was one by Gomes and Selman [15]. The paper also suggested the "heavy-tailed distribution" explanation for the power of restarts. This explanation is not considered valid anymore in the CDCL setting [22]. The dominant static restart policy today is the one proposed by Luby [24], which is theoretically proven to be the optimal universal restart policy for Las Vegas algorithms. The dominant dynamic restart policy is the one proposed by Glucose in 2012 [4]. The development of dynamic restart policies has attracted many researchers to work on it. To name a few, Biere et al. [7] proposed a variation of the Glucose restart policy; Liang et al. [23] developed machine learning based restart policies; and Nejati et al. [27] used multi-armed bandits in their portfolio based restart policy. In addition to the empirical work on restarts, there has been considerable interest among theorists to better understand why restart policies are so crucial to the success of modern SAT solver, as well as seem to be necessary to carry out certain simulation results [9]. Hypotheses aimed at explaining the power of restarts based on empirical observations have also been also proposed. Examples include, the *heavy-tail* explanation [15], and the "restarts compact assignment trail and hence produce clauses with lower literal block distance (LBD)" perspective [23].

Heavy-tailed explanation. A heavy-tailed distribution was observed for the runtime of randomized DPLL solvers on various satisfiable formulas [15]. Which on a high level means that the probability of a DPLL solver with randomness having a long runtime is non-negligible in practice, and hence the solver could benefit from restarting. One could relate our theoretical analysis of the power of restart to the heavy-tailed explanation, since the core arguments of the heavy-tailed explanation and our proof for the separation results rely on the same observation: after a bad initial partial assignment, the solver gets "stuck" in a hard restricted formula which is unsatisfiable, by having the power of restarts, the solver has the opportunity to choose another potentially better initial partial assignment. Because of the above, one may then argue that we cannot lift our results to a CDCL setting just like the heavy-tailed explanation does not lift to CDCL. However we believe that our result is not a proof-complexity theoretic analog of the heavy-tailed explanation in the DPLL setting. We make no assumptions about independence (or lack thereof) of branching decisions across restarts boundaries. In point of fact, the variable selection in the DPLL model we use is non-deterministic. Only the value selection is randomized. We have arrived at a separation result without relying on the assumptions made by the heavy-tailed distribution explanation, and interestingly we are able to prove that the "solver does get stuck in a bad part of the search space by making bad value selections". Note that in our model the solver is free to go back to "similar parts of the search space across restart boundaries". In fact, in our proof for DPLL with restarts, the solver chooses the same variable order for Ladder formulas across restart boundaries. Further, the reason heavy-tail explanation does not apply to CDCL is because modern variable selection heuristics in CDCL allow the solver to "go back to similar parts of the search space across restart boundaries". Hence, at this point we cannot rule out the possibility that an idea similar to the one we used in our proofs can be successful in proving a restarts separation result for CDCL (in a setting similar to the one we used for DPLL for satisfiable instances).

Algorithm 2 The CDCL SAT Solving Algorithm

```
 1: function CDCL($\varphi$, $p$)
 2:     Input: A CNF formula $\varphi$, and an assignment $p$
 3:     Output: true (SAT) or false (UNSAT)
 4:
 5:     if (CONFLICT == BCP($\varphi$,$p$)) then
 6:         return UNSAT;
 7:     end if                                  ▷ If top-level conflict, return UNSAT
 8:     dl = 0;                                 ▷ : Initialize decision level $dl$ to be 0
 9:     while all variables have NOT been assigned do               ▷ The search loop
10:         $l$ = DecisionHeuristic($\varphi$,$p$); ▷ Variable and value selection heuristic combined,
    $l$ is the literal to be assigned $T$.
11:         dl = dl + 1;                        ▷ : Increment $dl$ for each new decision variable
12:         $p = p \wedge x$;                    ▷ Add literal $l$ to the assignment trail $p$
13:         if CONFLICT == (BCP($\varphi, p$)) then
14:             $c, bt\_level$ = ConflictAnalysis($\varphi, p$);   ▷ Analyze conflict and return a learnt
    clause $c$ and a backjump level $bt\_level$
15:             if $bt\_level < 0$ then
16:                 return UNSAT;                                ▷ Top-level conflict
17:             else
18:                 backtrack($\varphi, p, bt\_level$);         ▷ Backjump to start search again
19:                 dl = $bt\_level$;
20:             end if
21:         end if
22:     end while
23:     return SAT;
24: end function
```

Chapter 3

Refuting candidate instances

In this chapter, we study three classes of instances which are conjectured to be candidates separators to exponentially separate C_{ND_ND} from $C^R_{ND\_ND}$, and prove that none of the three classes of candidate instances are sufficient to even prove super-polynomial separation results. For each section of the chapter, we present the construction of the instances, properties of the instances which were conjectured to be hard for C_{ND_ND}, and follows by proofs of why these classes of instances are not sufficient for the separation result.

3.1 LS vs LSR formulas

LS vs LSR is a class of instances Ed Zulkoski et al. used to exponentially separate the size of Learning-sensitive Backdoor[33] of a SAT solver with and without Restart.

3.1.1 Original construction

The LS vs LSR formulas contain following sets of clauses:

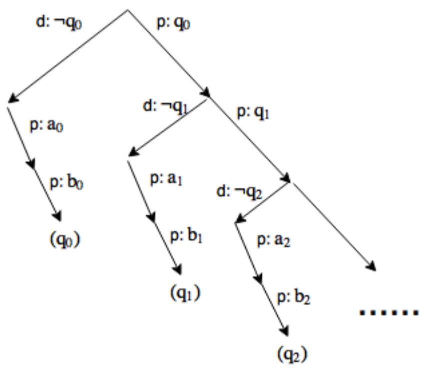

Figure 3.1: A polynomial size proof of a LS vs LSR formula produced by C_{ND_ND}

$$
\begin{gathered}
(x_0 \vee x_1 \vee \cdots \vee x_{n-1} \vee x_n \vee \neg q_0)\wedge \\
\wedge(q_0 \vee a_0) \wedge (q_0 \vee b_0) \wedge (q_0 \vee \neg a_0 \vee \neg b_0)\wedge \\
\wedge(x_0 \vee x_1 \vee \cdots \vee x_{n-1} \vee \neg x_n \vee \neg q_0 \vee \neg q_1 \vee \cdots \vee \neg q_{2^n-1} \vee \neg q_{2^n})\wedge \\
\wedge(q_{2^n} \vee a_{2^n}) \wedge (q_{2^n} \vee b_{2^n}) \wedge (q_{2^n} \vee \neg a_{2^n} \vee \neg b_{2^n})\wedge \\
.... \\
\wedge(\neg x_0 \vee \neg x_1 \vee \cdots \vee \neg x_{n-1} \vee x_n \vee \neg q_0 \vee \neg q_1 \cdots \vee \neg q_{2^n-1})\wedge \\
\wedge(q_{2^n-1} \vee a_{2^n-1}) \wedge (q_{2^n-1} \vee b_{2^n-1}) \wedge (q_{2^n-1} \vee \neg a_{2^n-1} \vee \neg b_{2^n-1})\wedge \\
\wedge(\neg x_0 \vee \neg x_1 \vee \cdots \vee \neg x_{n-1} \vee \neg x_n \vee \neg q_0 \vee \neg q_1)\wedge \\
\wedge(q_1 \vee a_1) \wedge (q_1 \vee b_1) \wedge (q_1 \vee \neg a_1 \vee \neg b_1)
\end{gathered}
$$

Size of this formula, N, equals to 2^n where n is the number of x variables. The idea of above gadget is that one needs to learn the (q_i) clauses in lexicographical order in order to produce a short proof. However the formula by itself turns out to be easy for C_{ND_ND} if the solver branches on all N q_i's lexicographically, each time the solver branches on q_i, it immidiately derives a conflict and learns unit clause $(\neg q_i)$ and backtrack to the root (Figure 3.1), and after learning all such unique clauses for $0 \leq i \leq N$, the solver can start branching on all the x variables to derive UNSAT in linear time in the size of the formula.

3.1.2 XOR-lifting q_i variables

The idea of *XOR-lifting* or *xorification* of a CNF formula is often used in proof complexity[5] to lift the hardness of the formulas for certain underlying proof systems. To XOR-lift variables, we replace each variable by an *xor* of several variables and then rewrite the formula in CNF. This idea has helped lifting an easy formulas for certain proof systems to hard formulas for the same proof system, and thus proving lower bounds. For example, Alekhnovich et al.[2] and Urquhart [31] provided classes of instances that requires exponential size proofs for regular resolution but polynomial size proofs for general resolution. We tried to apply these lifting ideas to the instances we are considering, and study if a C_{ND_ND} solver can still produce a short proof.

Based on the original LS vs LSR formulas, we modify the formulas using XOR-lifting such that it may potentially require a C_{ND_ND} solver to do extra work, and only produces proofs of exponential size. The intuition is as follow: instead of using a single q_i in the clauses, we have two new variables, q_i^s and q_i^t, and replace q_i's with:

$$q_i^{\oplus} = (q_i^s \lor q_i^t) \land (\neg q_i^s \lor \neg q_i^t)$$

In this way, instead of learning a unit clause right away by making just one decision, the solver needs to branch on two variables before learning "useful clauses". Since we only allow backtracking, the solver can only backtrack one of q_i^s or q_i^t after learning a conflict, which blocks the solver from learning the desired clauses in polynomial time.

However simply doing this is not enough, since expanding the new formula into CNF causes exponential blowup in size (for the reason that there exists a clause with $N\ q's$). One way to avoid this problem is to introduce new variables D_i and add the following clauses to the formula:

$$D_0 \Rightarrow (\neg q_0^{\oplus})$$

$$D_1 \Rightarrow (D_0 \lor \neg q_1^{\oplus})$$

$$\ldots$$

$$D_{2^n-1} \Rightarrow (D_{2^n-2} \lor \neg q_{n-1}^{\oplus})$$

$$D_{2^n} \Rightarrow (D_{2^n-1} \lor \neg q_n^{\oplus})$$

Then replace corresponding subclauses in the original formula by the D_i variables.

Resulting formula is shown below:

$$(x_0 \lor x_1 \lor \cdots \lor x_{n-1} \lor x_n \lor D_0) \land$$

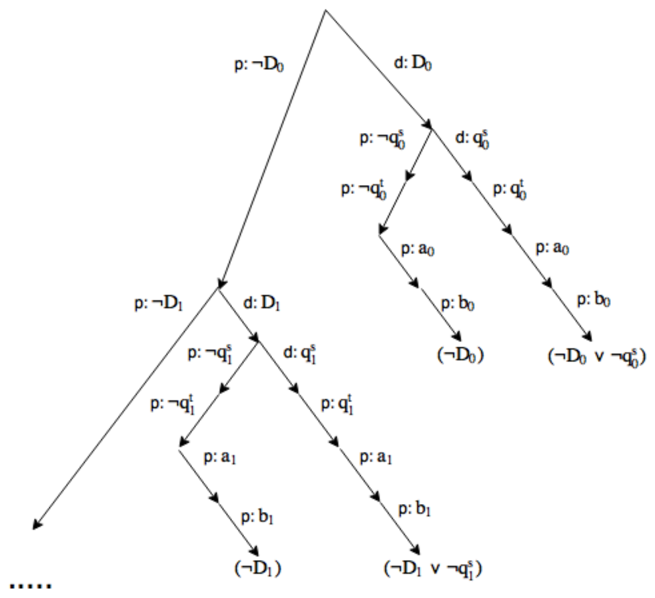

Figure 3.2: C_{ND_ND} proof tree for learning unit clauses $(\neg D_0)$ and $(\neg D_1)$, rest of the tree for learning all other $(\neg D_i)$ clauses are omitted

$$\wedge(q_0^{\oplus} \vee a_0^{\oplus}) \wedge (q_0^{\oplus} \vee b_0^{\oplus}) \wedge (q_0^{\oplus} \vee \neg a_0^{\oplus} \vee \neg b_0^{\oplus})\wedge$$
$$\wedge(x_0 \vee x_1 \vee \cdots \vee x_{n-1} \vee \neg x_n \vee D_{2^n})\wedge$$
$$\wedge(q_{2^n}^{\oplus} \vee a_{2^n}^{\oplus}) \wedge (q_{2^n}^{\oplus} \vee b_{2^n}^{\oplus}) \wedge (q_{2^n}^{\oplus} \vee \neg a_{2^n}^{\oplus} \vee \neg b_{2^n}^{\oplus})\wedge$$
$$\ldots.$$
$$\wedge(\neg x_0 \vee \neg x_1 \vee \cdots \vee \neg x_{n-1} \vee x_n \vee D_{2^n-1})\wedge$$
$$\wedge(q_{2^n-1}^{\oplus} \vee a_{2^n-1}^{\oplus}) \wedge (q_{2^n-1}^{\oplus} \vee b_{2^n-1}^{\oplus}) \wedge (q_{2^n-1}^{\oplus} \vee \neg a_{2^n-1}^{\oplus} \vee \neg b_{2^n-1}^{\oplus})\wedge$$
$$\wedge(\neg x_0 \vee \neg x_1 \vee \cdots \vee \neg x_{n-1} \vee \neg x_n \vee D_2)\wedge$$
$$\wedge(q_1^{\oplus} \vee a_1^{\oplus}) \wedge (q_1^{\oplus} \vee b_1^{\oplus}) \wedge (q_1^{\oplus} \vee \neg a_1^{\oplus} \vee \neg b_1^{\oplus})$$

Unfortunately, we managed to find a polynomial proof (Figure 3.2) for the gadget above by branching on D_i variables and followed by q_i^s and q_i^t, then the solver is able to learn all the unit clauses (D_i) in polynomial time, it turns out that being able to learn all the (D_i) clauses in lexicographical order gives same power as learning $(\neg q_i)$ clauses in the unlifted formula, and thus leads to a polynomial size proof.

3.1.3 XOR-lifting both q_i and D_i variables

One of many reasons that above construction was not sufficient for a separation result is that a C_{ND_ND} solver could still learn unit clauses (D_i) by just lexicographically making decision on D_i variables (along with a constant number of queries to q_i variables), so we tried to apply $XOR-lifting$ to all D_i's as well. Having said that, we replaced all the D_i variables with $D_i^{\oplus}$, in the way it's done with the q_i variables.

Clearly in order to produce a short proof, we need to derive both $(D_i^s \vee \neg D_i^t)$ and $(\neg D_i^s \vee D_i^t)$ for all D variables.

- In a $C_{ND\_ND}^R$ solver, we can start by first branching on $\neg D_0^s$ and D_0^t followed by queries to q_0^s and q_0^t to learn $(D_0^s \vee \neg D_0^t)$, then *restart*, and branch on D_0^s and $\neg D_0^t$ followed by queries to q_0^s and q_0^t to learn $(\neg D_0^s \vee D_0^t)$, then restart, do the similar and continue on learning $(D_1^s \vee \neg D_1^t)$ and $(\neg D_1^s \vee D_1^t)$ and repeat until we learn all $\neg D_i^{\oplus}$ clauses.
- However in a C_{ND_ND} solver, if we follow the strategy we used when only q_i's are XOR-lifted, that is branch on $\neg D_0^s$ and D_0^t followed by queries to q_0^s and q_0^t to learn $(D_0^s \vee \neg D_0^t)$, at this point, we are forced to backtrack, which results in a path with $\neg D_0^s$ and $\neg D_0^t$ on the trail (Figure 3(a)), it's easy to see that branching just on q_0^s and q_0^t will not yield a conflict anymore, so we need to branch on the next D variables, that is D_1^s and D_1^t. However since we only consider Decision Learning Scheme and $\neg D_0^s$ is currently on the trail as a decision variable, future learnt clauses are "polluted" by it (Figure 3(b)), in the sense that $(D_i^s \vee \neg D_i^t)$ and $(\neg D_i^s \vee D_i^t)$ will not be learnt unless we've exhaustively search the subtree. But on the other hand, each time we branch on a new D variable, the number of branches in the proof tree gets doubled (Figure 3(c)), and since the tree has depth N, size of the whole proof tree is exponential in the size of N.

Note that in the $C_{ND\_ND}^R$ proof, we don't need to query any x variables to learn all $\neg D_i^{\oplus}$ clauses. So we decided to focus on a specific part of the formula:

$$D_0^{\oplus} \Rightarrow (\neg q_0^{\oplus})$$

$$D_1^{\oplus} \Rightarrow (D_0^{\oplus} \vee \neg q_1^{\oplus})$$

$$...$$

$$D_{2^n-1}^{\oplus} \Rightarrow (D_{2^n-2}^{\oplus} \vee \neg q_{n-1}^{\oplus})$$

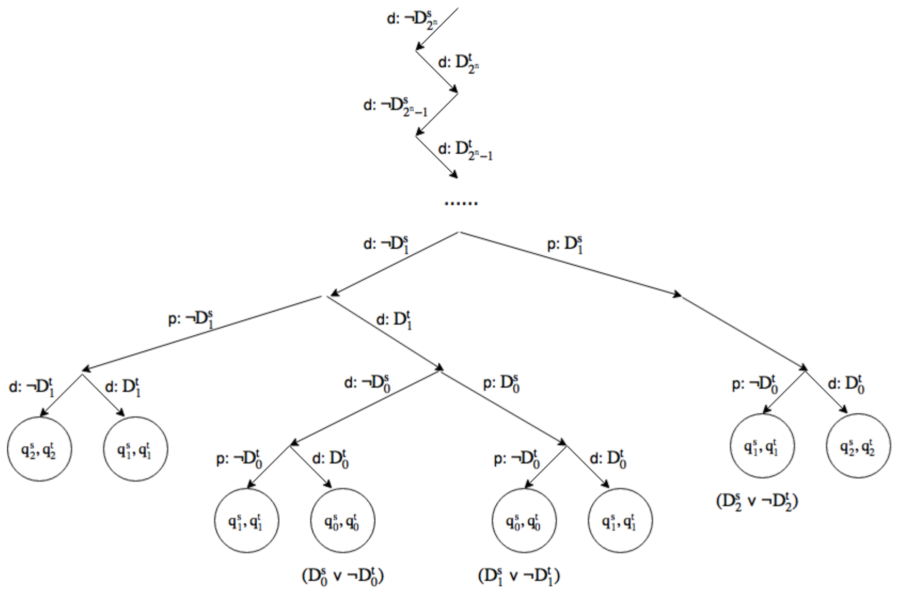

Figure 3.3: Proof tree to learn all $(D^s_i \vee \neg D^t_2)$ clauses (only useful learnt clauses are displayed, unuseful clauses means that the clause is only used once, when backtracking at the conflict happens.)

$$D^{\oplus}_{2^n} \Rightarrow (D^{\oplus}_{2^n-1} \vee \neg q^{\oplus}_n)$$

The problem with the previous strategy is that after we learn $(D^s_0 \vee \neg D^t_0)$ we have to branch on other D varibles, we solved this problem by falsifying all $\neg D^{\oplus}_i$ at the begining in reverse order. (Figure 3.3) The informal intuition is as follow: observe that above fragment forms a chain of implications, which is inductive in nature, so instead of proving from D_0 to D_{2^n}, we "assume" $D_1, D_2, ..., D_{2^n}$ to be false, and try to disprove our "assumption" from bottom up. This intuition is still not concrete at the moment, but we are trying to apply it to other instances for justification. As shown in the proof tree, we only need to pay a constant price for learning a $(D^s_0 \vee \neg D^t_0)$ clause, since we have $2N$ number of $D's$ on the trial, a complete search over the tree in Figure 4 still takes polynomial time. And after we learn all $(D^s_i \vee \neg D^t_i)$, we are at the root of the tree, and can start branching in the same manner and learn all $(\neg D^s_i \vee D^t_i)$ clauses, clearly it will take polynomial time as well. And we have a short C_{ND_ND} proof.

3.2 Pebbling formula

Pebbling formulas are a class of unsatisfiable formulas, there are three kinds of clauses: source clauses, target clauses and precedence clauses. Figure 3.4 presents the high level structure and the clauses in XOR-lifted Pebbling formula. If we consider the unlifted version of the Pebbling formula, that is ignoring all of the $\oplus$ in Figure 3.4, there are n nodes in the graph, which correspond to n distinct variables in the CNF, and consequently depth of a pebbling graph is $log(n)$. We define *levels* incrementally from root to leaves, that is the target node is on level 1, and source nodes are on level 4. We can easily see that the formula is unsatisfiable, since the nodes at level 4 (source nodes) implies the nodes at level 3, which implies the nodes at level 2, and for the same reason, the node at level 1, $10^{\oplus}$, has to be $\top$, however from the target clause, $(\neg 10^{\oplus})$, we have a contradiction. Since unlifted pebbling formulas are trivially easy to solve, that is a C_{ND_ND} solver can derive UNSAT using only linearly number of unit propagations. To lift the hardness of the Pebbling formulas, we consider the XOR-lifted pebbling formula. There is a nice relation between the Pebbling structure and the structure of LS vs LSR formulas. Recall that the hardness of LS vs LSR formulas come from the fact that, a solver needs to learn clauses which correspond to $(\neg D^{\oplus})$, however the formulas are set up in a way that to learn $(\neg D_i^{\oplus})$, the solver needs to first learnt $(\neg D_{i-1}^{\oplus})$. This is the same for Pebbling formulas, except that to learn about a node x in the Pebbling formula, a solver needs to first learn all the nodes that appear in the subtree rooted at x. In other words, the LS vs LSR formulas can be viewed as a single path from the root nodes to a leaf node in the Pebbling formulas.

The short C_{ND_ND} proof of lifted pebbling comes naturally by generalizing the strategy we used to produce our short C_{ND_ND} proof for lifted LS vs LSR formula, that is, find a chain of implication (the idea of chain of implication still needs refinement), and falsify the nodes in the chain in reverse order. (Figure 6)

Note that in the proof tree, we were able to learn all $(5^{\oplus})$, $(6^{\oplus})$, $(7^{\oplus})$ before backtracking to level 3 in the pebbling graph. Having said that, we've reduced the pebbling graph by one level, since we've learnt all fact that can be derived from nodes in level 4, in other words, nodes in level 4 no longer participate in further proof. Additionally, in order to learn each clause, we only need to reason within a constant number of variables at any given time, so reducing pebbling graphs by 1 layer requires only polynomial effort. Since there are only $log(n)$ number of layers, size of our proof tree is polynomial in n.

Another very interesting and important fact is that, even though we restrict C_{ND_ND} to only do backtracking, it backtracked twice consecutively after learning $(6^s \vee 6^t)$ and $(5^s \vee 5^t)$, which essentially simulated backjumping (labeled red in Figure 3.5). This seems

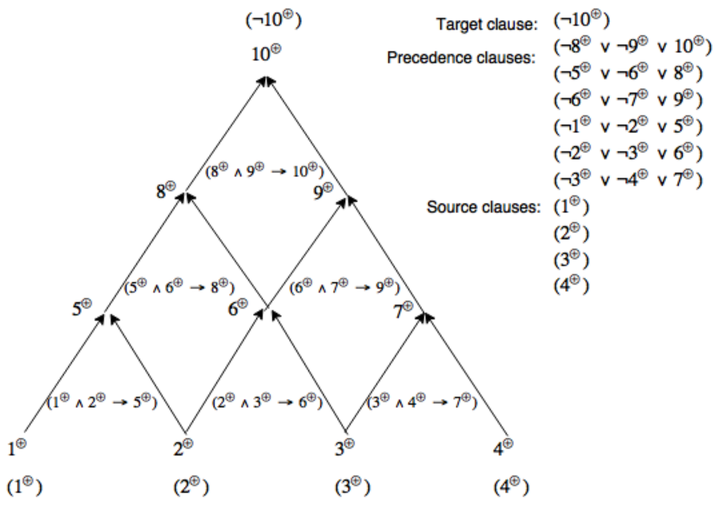

Figure 3.4: Pebbling formula with 4 layers

to be a natural behaviour but somewhat unexpected, recall that the reason we only allow backtracking is to avoid backjumping. As a result, we may need to either restrict our C_{ND_ND} model further so that backjumping is completely disabled, or consider backjumping as part of C_{ND_ND}. However unfortunately we do not have much intuition in either direction yet.

3.3 Possibility of a non-trivial polynomial separation

Due to lack of success in exponentially separating $C^R_{ND\_ND}$ and C_{ND_ND}, we started investigating in weather we can prove a non-trivial polynomial separation.

3.3.1 k-disequality (k-XOR) lifting

We extended the idea of XOR-lifting.

- In XOR-lifting, we replace each variable q by $q^{\oplus}$ where

$$q_i^{\oplus} = (q_i^s \lor q_i^t) \land (\neg q_i^s \lor \neg q_i^t)$$

  which encodes that q^s and q^t cannot have same polarity.

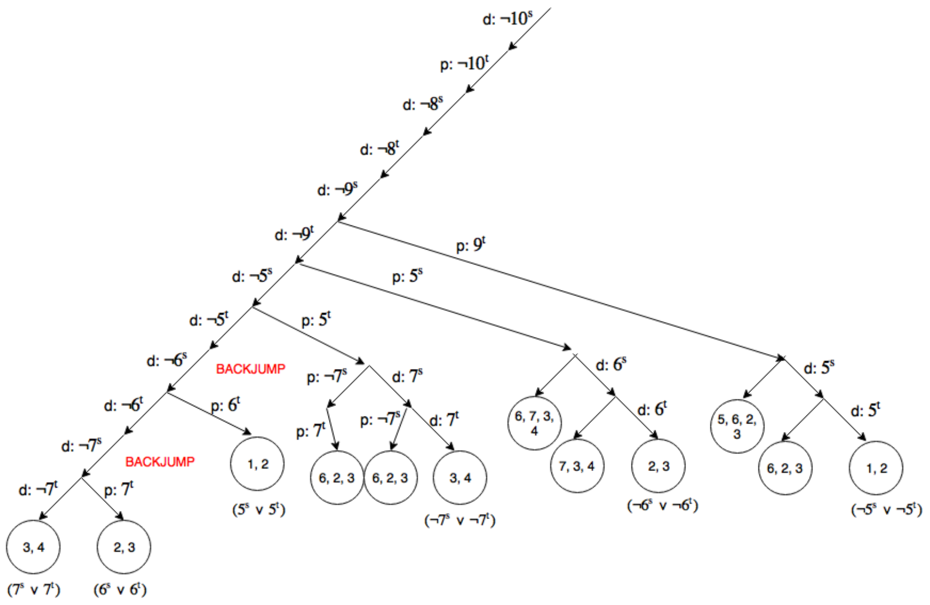

Figure 3.5: A small proof tree for solving the Pebbling formulas.

- In *k-disequality* lifting, we encode the condition such that all k variables cannot have same polarity. For example, for *3-disequality* lifting on variable q, we replace q by $q^{3-disequality}$ where

$$q_i^{3-disequality} = (q_i^s \vee q_i^t \vee q_i^r) \wedge (\neg q_i^s \vee \neg q_i^t \vee \neg q_i^r)$$

The idea of lifting a variable is to add complexity in resolving on that variable. Consider a simple input formula:

$$(q) \wedge (\neg q)$$

This formula is clearly unsatisfiable, and contradiction can be derived within one resolution step. Now consider the 3-disequality lifted version:

$$(q^{3-disequality}) \wedge (\neg q^{3-disequality})$$

It's trivial to expand $(q^{3-disequality})$ to CNF:

$$(q^{3-disequality}) \leftrightarrow ((q^s \vee q^t \vee q^r) \wedge (\neg q^s \vee \neg q^t \vee \neg q^r))$$

However there are many ways to encode $(\neg q^{3-disequality})$, we considered two encodings, where one encodes it using least number of clauses possible, and the second encodes it using most number of clauses possible.

$$(\neg q^{3-disequality}) \leftrightarrow ((q^s \vee \neg q^t) \wedge (q^t \vee \neg q^r) \wedge (q^r \vee \neg q^r))$$

or

$$\begin{aligned}(\neg q^{3-disequality}) \leftrightarrow &((q^s \vee q^t \vee \neg q^r)\wedge \\ &(q^s \vee \neg q^t \vee q^r)\wedge \\ &(q^s \vee \neg q^t \vee \neg q^r)\wedge \\ &(\neg q^s \vee q^t \vee q^r)\wedge \\ &(\neg q^s \vee q^t \vee \neg q^r)\wedge \\ &(\neg q^s \vee \neg q^t \vee q^r))\end{aligned}$$

In the former encoding, only five resolution steps are needed to derive UNSAT, and in the latter, eight resolution steps are required. On the other hand in terms of $C^R_{ND\_ND}$ and C_{ND_ND}, for former encoding, both $C^R_{ND\_ND}$ and C_{ND_ND} can derive UNSAT by branching on any one variable to any polarity and followed by a series of unit propagation, and for latter encoding, both $C^R_{ND\_ND}$ and C_{ND_ND} requires exploring entire search space over the three variables. It seems from these two different encodings that the number of resolution steps directly correlates to the size of the encoding. However at this point, we still don't have a full proof to justify this observation. If this correlation holds, we also prove that k-disequality lifting is not sufficient to polynomially separate $C^R_{ND\_ND}$ and C_{ND_ND}.

Chapter 4

Separation and equivalence between different models of CDCL SAT solvers

From our work presented in Chapter 3, we realized it is really difficult to prove/disprove a separation results for restarts when the underlying CDCL SAT solver has both non-deterministic variable selection and non-deterministic value selection. In this chapter, we approach the problem of understanding the power of restart from a slightly different angle.

In this section, we address the question of the power of restarts in both the DPLL [14, 13] and CDCL SAT solver settings. Specifically, we present two results on the power of restarts: First, we introduce a new class of *s*atisfiable instances called $Ladder_n$ and use it to prove that for the *drunk* randomized DPLL SAT solver model (D_{ND_RD}) introduced by Alekhnovich and Razborov [1], the configuration of D_{ND_RD} with restarts can solve $Ladder_n$ formulas in time polynomial in size of $Ladder_n$, while the configuration of D_{ND_RD} without restarts requires time exponential in the size of $Ladder_n$, with high probability (w.h.p.). Second, we show that models of CDCL SAT solvers with non-deterministic *static* variable and value selection, and with restarts are no more powerful from a proof-complexity theoretic point of view than the same configurations without restarts.

Recall that one of the intuitive reasons that restart give more power to a CDCL SAT solver is that restarts can help a solver escape from "bad" or "hard" search spaces. But what happens if the solver has the power of non-determinism and makes no mistakes in the first place? As we studied this problem in depth, we made the following observation about restarts, namely, that there seems to be a subtle interplay between various solver heuristics and restarts, wherein the power of restarts becomes apparent only when one or more important heuristics (e.g., variable or value selection) are weakened (e.g., solver configurations where non-deterministic value or variable selection heuristics are replaced by their weaker randomized cousins). Another observation we made was that all previous theoretical work on the power of restarts seems to have focused on treating solvers as proof systems, i.e., on unsatisfiable instances only. For one of our result, we show that one can separate certain solver configurations with and without restarts, if we change our perspective from unsatisfiable to satisfiable instances (see Section 4.4). All our results hold irrespective of the computational overhead of the various heuristics considered in this chapter.

Restart and Weak Decision Learning. Before we jump to the results of the section, we would like to state a preliminary observation on restarts[1]. If we consider the Weak Decision Learning Scheme (WDLS) for a CDCL solver defined as follow: Upon deriving a conflict, the solver learns the clause over the disjunction of the negation of the decision variables on the current assignment trail. Clearly, the solver model $C_{\text{ND\_ND}}$ with WDLS is only as powerful as $D_{\text{ND\_ND}}$, since each learnt clause will only be used once for propagation after the solver backtracks immediately after learning the conlict clause, and remains satisfied for the rest of the solver run, and this is exactly what $D_{\text{ND\_ND}}$ does. However on the other hand, it is clear that WDLS is an asserting learning scheme [28], and hence satisfied the conditions in [30] which proved that CDCL with asserting learning scheme and restarts p-simulates general resolution, thus we have $C^R_{ND\_ND}$ with WDLS is exponentially more powerful than the same solver but with no restarts. Given that WDLS is unconventional clause learning scheme which does not quite capture state-of-art CDCL SAT solver's clause learning scheme, we only list this results as a discussion.

4.1 Restarts and Non-Deterministic Static Variable and Value Selection in CDCL SAT solvers

In this section, we show that CDCL SAT solvers with non-deterministic static variable selection, non-deterministic static value selection, and with restarts ($C^R_{\text{NS\_NS}}$), are polynomially equivalent to the same model but without restarts ($C_{\text{NS\_NS}}$).

[1]Personal communication with Robert Robere.

Theorem 4.1.1. $C^R_{NS\_NS} \sim_p C_{NS\_NS}$

Proof. We first show that C_{NS_NS} p-simulates $C^R_{NS\_NS}$. On a high level, we want to show that a run of C_{NS_NS} derives the same set of learnt clauses as $C^R_{NS\_NS}$. More formally, we prove this by induction on the number of restart calls by $C^R_{NS\_NS}$.
Induction base: number of restart calls, $r = 0$. Clearly if $C^R_{NS\_NS}$ does not restart, it is exactly the same as C_{NS_NS}, and they have the same clause database.
Inductive hypothesis: Consider a run of $C^R_{NS\_NS}$ with $r = k$ restarts for some $k \geq 0$. There is a run of C_{NS_NS} which produces the same clause database as $C^R_{NS\_NS}$.
Inductive step: Consider a run of $C^R_{NS\_NS}$ with $r = k + 1$ restarts. Due to our inductive hypothesis, there is a run of C_{NS_NS} which produce the same clause database as $C^R_{NS\_NS}$ right before the $(k+1)^{th}$ restart call. Now consider the assignment trail for $C^R_{NS\_NS}$ up to the asserting literal, l, from the last learnt clause before the $(k+1)^{th}$ restart call. Due to the definition of clause learning and asserting literal, we know the assignment trail for $C^R_{NS\_NS}$ up to l does not cause a conflict, and the truth values for variables assigned from that partial assignment are either due to the static value selection or due to the current clause database. After the last restart call, $C^R_{NS\_NS}$ starts to branch with respect to the static branching and propagates with respect to the clause database (By definition of restart, the clause database does not change before and after a restart call.). And this will produce the same assignment trail up to l, and clearly, the clause database does not change since no conflicts are detected. By inductive hypothesis there is a run of C_{NS_NS} which produces the same clause database. Due to the assumption that $C^R_{NS\_NS}$ only invokes $k+1$ restart calls, $C^R_{NS\_NS}$ will not make another restart call after this point. And hence C_{NS_NS} produces the same run as $C^R_{NS\_NS}$.

Thus C_{NS_NS} p-simulates $C^R_{NS\_NS}$. And the other direction, $C^R_{NS\_NS}$ p-simulates C_{NS_NS} is true by definition. □

Note that in the proof above, we not only argue that C_{NS_NS} is p-equivalent to $C^R_{NS\_NS}$, we also show that the two configurations produce the same run. The crucial observation is that given any state of $C^R_{NS\_NS}$, we can produce a run of C_{NS_NS} which ends in the same state. In other words, our proof not only suggests that $C^R_{NS\_NS}$ is equivalent to C_{NS_NS} from a proof theoretic point of view, it also implies the two configurations are equivalent for satisfiable formulas.

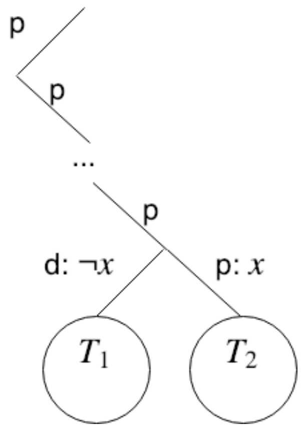

Figure 4.1: The path of *decision complexity* $k + 1$ appears in T_1.

4.2 Decision Complexity and Size Lowerbounds for DPLL Proofs

Theorem 4.2.1. *Consider an unsatisfiable formula φ, the size of any DPLL trees of φ is at least $2^{dc(\varphi)}$.*

Lemma 4.2.2. *Let T be a DPLL proof tree for φ. If there does not exist a relabelling of T into a new tree T' such that $dc(T') < dc(T)$, then $size(T) \geq 2^{dc(T)}$.*

Proof. By induction on $dc(T)$.
Induction base: $dc(T) = 0$. Clearly, any DPLL tree for proving φ is unsatisfiable contains at least one leaf, and we have $size(T) \geq 1 = 2^{DP(T)} = 2^0$.
Inductive hypothesis: For all DPLL trees T where $dc(T) \leq k$ for some k, if there does not exist a relabelling of T into a new DPLL tree T' such that $dc(T') < dc(T)$, then $size(T) \geq 2^{dc(T)}$.
Inductive step: $dc(T) = k + 1$. By definition of *decision complexity*, there exists a path $p \in T$ where $dc(p) = k + 1$. W.l.o.g. consider p appears on the left side of T.

Since the path p where $dc(p) = k + 1$ is in T_1, we know that $dc(T_1) = k$. Now there are two cases for T_1:

Case 1: there does not exist a relabelling of T_1 into a new DPLL tree T_1' such that $dc(T_1') < dc(T_1)$.

Case 2: there exists a relabelling of T_1 into a new DPLL tree T'_1 such that $dc(T'_1) < dc(T_1)$.

For case 1. By inductive hypothesis, $size(T_1) \geq 2^k$. Now if we prove that $size(T_2) \geq 2^k$ then we are done. First we use $\varphi(T_2)$ to denote the formula obtained from restricting φ with the partial assignment before T_2. Note that $dc(\varphi(T_2)) \geq k$, because otherwise, by applying *label switch* on x and relabelling T_2, we have a new tree T' where $dc(T') < k+1$, which is in contradiction with the assumption of T stated in the lemma. Now again there are two subcases:

Case 1.1: $dc(\varphi(T_2)) = k$.

Case 1.2: $dc(\varphi(T_2)) = k+1$. (Note that $dc(\varphi(T_2))$ cannot be larger than $k+1$, since by definition $dc(T) \geq dc(T')$ where T' is a subtree of T.)

For case 1.1, $dc(\varphi(T_2)) = k$. Since $dc(\varphi(T_2)) = k$, by definition of the *decision complexity* of a formula, there does not exist a relabelling of T_2 into a new DPLL tree T'_2 such that $dc(T'_2) < dc(T_2)$. By inductive hypothesis $size(T_2) \geq 2^{dc(\varphi(T_2))} = 2^k$, and combining with $size(T_1) \geq 2^k$, we have $size(T) \geq 2^{k+1}$, lemma is proved.

For case 1.2, $dc(\varphi(T_2)) = k+1$. Then we know there exists a path p' where $p' \in T_2$ and $dc(p') = k+1$. Similar to T, T_2 will have two subtrees T_{21} and T_{22} as in Figure 4.2 (figure on the left), where $p' \in T_{21}$.

And we know that $dc(T_{21}) = k$. If $dc(\varphi(T_{21})) = k$, then by inductive hypothesis we have $size(T_{21}) \geq 2^k$, and combining with $size(T_1) \geq 2^k$, we have $size(T) \geq 2^{k+1}$, the lemma is proved. Other wise, consider T_{22} and continue as T_2. T will be like Figure 4.2 (figure on the right).

Since the height of T is finite, at some point, there will be a subtree $T_{22...222}$ where $dc(\varphi(T_{22...222})) = k$, and we have $size(T_{22...222}) \geq 2^k$. Again, combining $size(T_1) \geq 2^k$, we have $size(T) \geq 2^{k+1}$, the lemma is proved.

Now consider case 2: there exists a relabelling of T_1 into a new DPLL tree T'_1 such that $dc(T'_1) < dc(T_1)$. We first relabel T_1 into T'_1 such that $dc(T'_1) < dc(T_1)$. Then $dc(T_2) = k+1$ and there does not exist a relabelling of T_2 into a new DPLL tree T'_2 such that $dc(T'_2) < dc(T_2)$, due to the assumption on T stated in the lemma. Now we consider the same argument for T_2 instead of T and prove that $size(T_2) \geq 2^{k+1}$, and this proves the lemma (Again, since the height of T is finite, at some point, we will be able to apply case 1.).

□

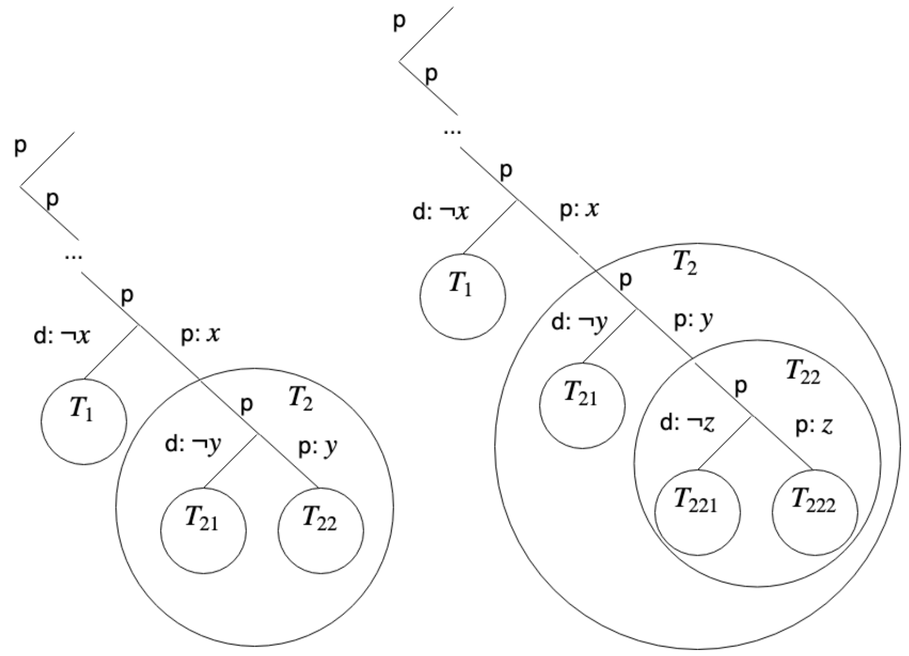

Figure 4.2: Left: The path of *decision complexity* $k+1$ appears in T_{21}. Right: The path of *decision complexity* $k+1$ appears in T_{221}.

proof of Theorem 4.2.1. Consider any DPLL proof tree T for an unsatisfiable formula φ. First relabel T into T' such that T' has the least *decision complexity* among all relabellings of T. Now T' satisfies the assumptions of Lemma 4.2.2. Then $size(T') \geq 2^{dc(T')}$. Since relabelling is size preserving, by definition of $dc(\varphi)$, we have $size(T) = size(T') \geq 2^{dc(T')} \geq 2^{dc(\varphi)}$.

$\square$

Note that decision complexity of a formula is different from the concept of a smallest strong backdoor introduced by Williams, Gomes, and Selman [32]. In general a large smallest strong backdoor does not necessarily imply size lowerbound of proofs. For example, consider the following class of formulas:

$$(x_1 \vee x_2 \vee x_3 \vee ... \vee x_{k-1} \vee x_k \vee PHP_n^1)\wedge$$

$$\wedge(x_1 \vee x_2 \vee x_3 \vee ... \vee x_{k-1} \vee \neg x_k \vee PHP_n^2)\wedge$$

$$\wedge(x_1 \vee x_2 \vee x_3 \vee ... \vee \neg x_{k-1} \vee x_k \vee PHP_n^3)\wedge$$

$(x \vee PHP_n^1) \wedge (\neg x \vee PHP_n^2)$, where PHP_n^1 and PHP_n^2 both encode pigeon hole principle of size n, however over disjoint variables. Clearly, the smallest strong backdoor of the formulas are a lot larger than the decision complexity of the formulas, and it is not clear if smallest strong backdoor can be used to lowerbound proof size at all. On the other hand, it seems that one may be able to relate decision complexity to the width of a formula, however this is still subject to proof.

4.3 $Ladder_n$ Formulas

Inspired by Alekhnovich's work [1], where the authors proved exponential lowerbound for drunk DPLL SAT solvers over a class of satisfiable instances, we studied the behavior of restarts in a drunk style DPLL SAT solver through the lens of satisfiable instances. In this section, we introduce a new class of satisfiable formulas, $Ladder_n$, which is then used to prove our separation result in Section 4.4.

Definition 4.3.1. *$Ladder_n$ contains two sets of variables, q_i^j and d_k variables for $0 \leq i \leq m-1$, $0 \leq j \leq c-1$ and $0 \leq k \leq n-1$ where $m = 2^n$ and c is some large enough constant that is greater or equal to 2 (so that the value of q_i^j cannot be unit propagated by just giving a complete assignment to d_k variables.). We can now construct $Ladder_n$ over q_i^j and d_k variables:*

$$\begin{array}{ll}
Q_0 \Rightarrow \neg D_0 & D_0 \Rightarrow Q_{(0+\frac{m}{2})} \mod m \\
Q_1 \Rightarrow \neg D_1 & D_1 \Rightarrow Q_{(1+\frac{m}{2})} \mod m \\
Q_2 \Rightarrow \neg D_2 & D_2 \Rightarrow Q_{(2+\frac{m}{2})} \mod m \\
\cdots & \cdots \\
Q_{m-3} \Rightarrow \neg D_{m-3} & D_{m-3} \Rightarrow Q_{(m-3+\frac{m}{2})} \mod m \\
Q_{m-2} \Rightarrow \neg D_{m-2} & D_{m-2} \Rightarrow Q_{(m-2+\frac{m}{2})} \mod m \\
Q_{m-1} \Rightarrow \neg D_{m-1} & D_{m-1} \Rightarrow ((\neg q_0^0 \vee q_1^0) \wedge (\neg q_1^0 \vee q_2^0) \wedge \\
 & \qquad \cdots \wedge (\neg q_{m-2}^0 \vee q_{m-1}^0) \wedge (\neg q_{m-1}^0 \vee q_0^0))
\end{array}$$

$$(Q_0 \Leftrightarrow Q_1) \wedge (Q_1 \Leftrightarrow Q_2) \wedge \cdots \wedge (Q_{m-3} \Leftrightarrow Q_{m-2}) \wedge (Q_{m-2} \Leftrightarrow Q_{m-1})$$

- $Q_i \equiv (\bigvee_{j=0}^{c-1} q_i^j) \wedge (\bigvee_{j=0}^{c-1} \neg q_i^j)$*. In other words, Q_i evaluates to $\perp$ if and only if all q_i^j's have the same truth value.*

- *Let $bin(j)$ be the binary representation of j using n bits, and let $bin(j)_k$ denote the $(k+1)'$th Most Significant Bit (MSB) in $bin(j)$. Then each D_j is defined as follows:*

$$D_j \equiv (\bigwedge_{k=0}^{n-1} x_k) \begin{cases} x_k = d_k & \text{if } bin(j)_k = 1 \\ x_k = \neg d_k & \text{if } bin(j)_k = 0 \end{cases}$$

Proposition 1. *The set of satisfying assignments for $Ladder_n$ has the properties: $d_k = \top$, for $0 \leq k \leq bin(m) - 1$; and $q_i^j = q_{i'}^{j'}$, for all possible i, j, i' and j'.*

The core idea for the Ladder formulas is that once a DPLL solver makes a decision that violates the conditions in proposition 1 early on, the restricted formula becomes unsatisfiable and requires exponential size proofs for the underlying DPLL solver. To understand the implication structure of $Ladder_n$ and Proposition 1, we make the following observations:

1. When some Q_i is asserted to $\top$, then the set of clauses $\bigwedge_{i=0}^{m-2} Q_i \Leftrightarrow Q_{i+1}$ implies that all other Q's should also be $\top$. Then the set of clauses $\bigwedge_{i=0}^{m-1}(Q_i \Rightarrow \neg D_i)$ implies that all D's should be $\bot$. However due to the construction of the D's, this is clearly unsatisfiable.

2. When some D_k is asserted to $\top$, then the set of clauses $\bigwedge_{i=0}^{m-1}(Q_i \Rightarrow \neg D_i)$, we know that the corresponding Q_k is implied to be $\bot$. And also due to the set of clauses $\bigwedge_{i=0}^{m-2}(D_i \Rightarrow Q_{i+\frac{m}{2} \mod m})$, some $Q_{k'}$ that is $\frac{m}{2}$ away from Q_k is implied to be $\top$. And finally, the set of clauses $\bigwedge_{i=0}^{m-2} Q_i \Leftrightarrow Q_{i+1}$ which asserts all Q's should be equivalent is contradicting with above.

The proof for the separation result relies on the two observations above, please find the details of the proof in Section 4.4.

4.4 Restarts and Non-Deterministic Variable Selection in DPLL SAT Solvers

4.4.1 $D_{ND\_\textbf{arbitrary}}$ and D_{ND_ND} as Proof Systems

In this sub-section we show that, when considered as a proof system, a DPLL SAT solver with non-deterministic dynamic variable selection, arbitrary value selection and no restart ($D_{ND\_\text{arbitrary}}$) is p-equivalent to DPLL SAT solver with non-deterministic dynamic variable selection, non-deterministic dynamic value selection, and no restarts (D_{ND_ND}), and hence, transitively p-equivalent to tree-like resolution. The point of this result is the following: as long as the variable selection is non-deterministic dynamic, restarts add no proof-theoretic power to DPLL SAT solvers. However, very surprisingly, when we shift our focus to satisfiable instances some of these models with restarts can be separated from their counterparts without restarts (Theorem 4.4.8).

Theorem 4.4.1. $D_{ND\_arbitrary} \sim_p D_{ND\_ND}$

Proof. It is trivial to show that $D_{ND\_ND} \leq_p D_{ND\_arbitrary}$ since every run of $D_{ND\_arbitrary}$ is also a run of D_{ND_ND}.

To show that $D_{ND\_ND} \geq_p D_{ND\_arbitrary}$, we argue that every proof of D_{ND_ND} can be converted to a proof of same size in $D_{ND\_arbitrary}$. Let $\mathcal{F}$ be a tautology. Recall that a run of D_{ND_ND} on $\mathcal{F}$ starts with non-deterministically picking some variable x to branch on, and non-deterministically choose a truth value to x. W.l.o.g. suppose the solver assigns x to $\top$. And after the solver proves $\mathcal{F}[x] = \bot$, it then backtracks and proves $\mathcal{F}[\neg x] = \bot$, and conclude that $\mathcal{F}$ is UNSAT. Now to simulate a run of D_{ND_ND} with $D_{ND\_arbitrary}$, since variable selection is non-deterministic, $D_{ND\_arbitrary}$ also picks variable x to branch on first, if value selection returns x, the solver focus on the restricted formula $\mathcal{F}[x]$, and if value selection returns $\neg x$, the solver focus on the restricted formula $\mathcal{F}[\neg x]$. Because there is no clause learning, the order of which one of $\mathcal{F}[x]$ and $\mathcal{F}[\neg x]$ is searched first does not affect the size of the search space for the other. Now by recursively calling $D_{ND\_arbitrary}$ on $\mathcal{F}[x]$ and $\mathcal{F}[\neg x]$ and their further restricted formulas, $D_{ND\_arbitrary}$ can produce a proof for $\mathcal{F}$ of the same size as a tree-like resolution proof for $\mathcal{F}$. $\square$

4.4.2 D_{ND_RD} and $D^R_{ND\_RD}$ and Satisfiable Formulas

In this section, we present a proof of exponential separation of a DPLL SAT solver with non-deterministic variable selection and dynamic random value selection with restart ($D^R_{\text{ND\_RD}}$) from the same configuration but with no restart ($D_{\text{ND\_RD}}$).

Lemma 4.4.2. *D_{ND_RD} can find satisfying assignment for $Ladder_n[d_0][d_1]...[d_{n-1}]$ in time $\theta(m)$.*

Proof. When all d_k variables are $\top$, we have $D_{m-1} = \top$. Which asserts that $(\bigwedge_{i=0}^{m-1}(\neg q_i^0 \vee q_{i+1 \mod m}^0)) = \top$, by making a decision on q_0^0, if $q_0^0 = \top$, then $q_i^0 = \top$ for all $0 < i \leq m-1$, and if $q_0^0 = \bot$, then $q_i^0 = \bot$ for all $0 < i \leq m-1$. W.l.o.g., consider the case $q_i^0 = \top$ for all $0 \leq i \leq m-1$. Since $D_{m-1} = \top$, the set of clauses $Q_{m-1} \Rightarrow \neg D_{m-1}$ implies $\neg Q_{m-1}$. Since we have $q_{m-1}^0 = \top$, by querying on q_{m-1}^j variables, $D_{\text{ND\_RD}}$ can then exhaustively search over all q_{m-1}^j and assign each q_{m-1}^j to $\top$ for all $0 < j \leq c$ in time $O(2^c)$, which is constant. Then by using the set of clauses $\bigwedge_{i=0}^{m-2} Q_i \Leftrightarrow Q_{i+1}$, the DPLL SAT solver branches on all q_{m-2}^j variables, and the solver will assign $q_{m-2}^j = \top$ for all $0 < j \leq c$ in time $O(2^c)$ (since we already have $q_{m-2}^0 = \top$). By repeatedly querying q_{m-3}^j, q_{m-4}^j, ..., q_1^j and q_0^j variables in order, the solver assigns all q_i^j variables to $\top$ in time $m * O(2^c) = \theta(m)$. And at this point, the solver has assigned a truth value to all variables and the assignment is satisfying since it is consistent with the conditions in Proposition 1.

□

Putting it differently, the set of d_k variables form a weak backdoor for $Ladder_n$ formulas. Further, the following lemma shows that, with probability at least $\frac{1}{2}$, $D^R_{\text{ND\_RD}}$ can exploit this weak backdoor using only $\theta(m)$ number of restart calls. Additionally, the lemma shows that the runtime of the solver is $\theta(mlogm)$, and thus we have a polynomial upper bound on the solver model $D^R_{\text{ND\_RD}}$.

Lemma 4.4.3. *The expected number of restarts for $D^R_{ND\_RD}$ to solve $Ladder_n$ in time $O(m \log m)$ is m.*

Proof. Due to Lemma 4.4.2, if the DPLL SAT solver assigns all d_k variables to $\top$ before assigning any other variables, then the solver can find a satisfying assignment in $\theta(m)$ time with probability 1. Now we use the power of restart to find this desire assignment efficiently. The strategy the solver adopts is as follows: branch on all d_k variables, if at least one of the d_k variables is assigned $\bot$, the solver restarts, repeat this procedure until all d_k variables are assigned $\top$. Now we argue that with probability at least $\frac{1}{2}$, the solver only needs to make $\theta(m)$ restart calls before finding the desire assignment to d_k variables. There are $m = 2^n$ possible assignments to all d_k variables, with probability $\frac{1}{m}$ the solver assigns all d_k variables to $\top$ by randomly assigning truth values to d_k variables. To compute the number of restarts needed to find the desire assignment with probability $\frac{1}{2}$, we use the following formula, where m is the number of possible assignments for d_k variables and r is the number of restart calls needed. (Note that the number of restarts required follows geometric distribution.)

$$1 - (\frac{m-1}{m})^r = \frac{1}{2} \Rightarrow (\frac{m-1}{m})^r = \frac{1}{2} \Rightarrow r = \frac{\log \frac{1}{2}}{\log \frac{m-1}{m}}$$

Now for comparing the growth rate of r and m, we compute $\lim_{m\to\infty}\frac{r}{m}$.

$$\begin{aligned}
\lim_{m\to\infty}\frac{r}{m} = \lim_{m\to\infty}\frac{\frac{\log\frac{1}{2}}{\log\frac{m-1}{m}}}{m} &= \lim_{m\to\infty}\frac{\log\frac{1}{2}}{m\log\frac{m-1}{m}} \\
&= \lim_{t\to 0}\frac{t\log\frac{1}{2}}{\log(1-t)} \quad \text{(Let } t = 1/m\text{)} \\
&= \lim_{t\to 0}\frac{\log\frac{1}{2}}{\frac{1}{t-1}} \quad \text{(L'Hôpital's Rule)} \\
&= \lim_{t\to 0}(t-1)\log\frac{1}{2} = \log 2
\end{aligned}$$

Since $\lim_{m\to\infty}\frac{r}{m} = \log 2$ is a constant, we know r and m grow as fast. In other words, we have $r = \theta(m)$. And we finish the proof by combining the result with Lemma 4.4.2. It takes $\theta(m)$ restart before $D^R_{ND\_RD}$ finds the desire assignments to the d_k variables, during which the solver needs to make at most $n = \log m$ decisions to assign all the d_k variables between each restart, so the solver makes $\theta(m * \log m)$ decisions in total before we can apply Lemma 4.4.2. Combining the result from Lemma 4.4.2, the overall complexity is $\theta(m\cdot\log m) + \theta(m) = \theta(m\cdot\log m)$.

□

With the above proof, we have a polynomial upper bound for the solver model $D^R_{\text{ND\_RD}}$, and we now present the tools needed to prove an exponential lower bound for the solver model $D_{\text{ND\_RD}}$ below.

Lemma 4.4.4. *$Ladder_n[\neg d_a]$ is UNSAT and $dc(Ladder_n[\neg d_a]) = \Omega(m)$ for every $0 \le a \le n-1$.*

Proof. $Ladder_n[\neg d_a]$ is UNSAT since a satisfying assignment should have $d_k = \top$ for all $0 < k < n-1$ as stated in Proposition 1. Since $d_a = \bot$, we know the partial assignment can only be extended to one that satisfy some $D_{i'}$ for $0 \le i' \le m-2$. As a consequence the set of clauses $\bigwedge_{i=0}^{m-1}(Q_i \Rightarrow \neg D_i)$ assert $Q_{i'} = \bot$, and the set of clauses $\bigwedge_{i=0}^{m-2}(D_i \Rightarrow Q_{i+\frac{m}{2} \bmod m})$ assert that $Q_{i'+\frac{m}{2} \bmod m} = \top$. However this is conflicting with the set of clauses $\bigwedge_{i=0}^{m-2}(Q_i \Leftrightarrow Q_{i+1})$. To detect this conflict, a DPLL SAT solver needs to determine the truth value of at least $\frac{m}{2}$ Q_i's along some path of the proof tree, and this requires at least $\frac{m}{2}$ decisions, hence $dc(Ladder_n[\neg d_a]) = \Omega(m)$.

□

Lemma 4.4.5. *$Ladder_n[q_a^b][\neg q_{a'}^{b'}]$ is UNSAT and $dc(Ladder_n[q_a^b][\neg q_{a'}^{b'}]) = \Omega(m)$ if $a \neq a'$ or $b \neq b'$.*

Proof. $Ladder_n[q_a^b][\neg q_{a'}^{b'}]$ is UNSAT since a satisfying assignment should have $q_i^j = q_{i'}^{j'}$ for all possible i, j, i' and j' as stated in Proposition 1. Now consider a DPLL SAT solver run on $Ladder_n[q_a^b][\neg q_{a'}^{b'}]$. There are two cases:

Case 1: $a = a'$ and $b \neq b'$
Case 2: $a \neq a'$

For Case 1, we know that under the restriction, Q_a can only take the value $\top$ as the solver extends the partial assignment. Which in turn logically asserts all Q_i's to be $\top$ from the set of clauses $\bigwedge_{i=0}^{m-2}(Q_i \Leftrightarrow Q_{i+1})$, and this implies that all D_i's will be $\perp$ from the set of clauses $\bigwedge_{i=0}^{m-1}(Q_i \Rightarrow \neg D_i)$. However due to the definition of D_i's, this means no assignments over d_k variables can evaluate all D_i's to $\perp$. To detect this inconsistency, a DPLL SAT solver needs to determine the truth value of all Q_i's along some path, which requires at least $\Omega(m)$ number of decisions.

For Case 2, clearly, as a DPLL SAT solver extends the partial assignment, it is impossible for it to satisfy the sets of clauses $\bigwedge_{i=0}^{m-1}(Q_i \Rightarrow \neg D_i)$, $D_{m-1} \Rightarrow \bigwedge_{i=0}^{m-1}(\neg q_i^0 \vee q_{(i+1) \bmod m}^0)$ and $\bigwedge_{i=0}^{m-2}(Q_i \Leftrightarrow Q_{i+1})$ simultaneously unless some of the d_k variables is assigned to $\perp$. As a result, a DPLL SAT solver has to assign some d_k variable to $\perp$ along some path, but again, by the same argument we used for the proof for Lemma 4.4.4, the solver must determine the value of at least $\frac{m}{2}$ Q_i's along some path of the proof tree, and this requires at least $\frac{m}{2}$ decisions. Combining the result from Case 1, $dc(Ladder_n[q_a^b][\neg q_{a'}^{b'}]) = \Omega(m)$.

□

Lemma 4.4.6. *With probability at least $\frac{3}{4}$, the class of formulas, $Ladder_n$, requires $\Omega(2^m)$ time before finding a satisfying assignment using D_{ND_RD}.*

Proof. Consider the first four variables branched on by $D_{\text{ND\_RD}}$. Observe that, all clauses in $Ladder_n$ have width greater than four for sufficiently large n, so there will be no unit propagations or conflicts after branching on the first four variables, additionally none of the Q_i's value can be evaluated. Now we argue, $Ladder_n$ restricted with any four variables whose polarities are randomly chosen is UNSAT and has decision complexity $\Omega(m)$ with probability at least $\frac{3}{4}$.

Case 1: There are at least two d_k variables among the first four variables.

Case 2: There are at least three q_i^j variables among the first four variables.

For Case 1, with probability at least $\frac{3}{4}$, at least one of the d_k's is assigned $\perp$ (denoted the first such d_k as d_a), and w.l.o.g. assume there is no pair of q variables that are assigned opposite polarity before the assignment of $\neg d_a$. For Case 2, with probability at least $\frac{3}{4}$, at least two q_i^j's are assigned opposite values. W.l.o.g., let q_a^b be the first q_i^j variable assigned to $\top$, and let $q_{a'}^{b'}$ the first one assigned to $\perp$, and assume there is no d_k variables assigned $\perp$ before the assignment of the pair of q's with opposite value.

Recall that in our proof for Lemma 4.4.4 and Lemma 4.4.5, the core argument is that a DPLL SAT solver has to determine the value of at least $\frac{m}{2}$ or at least m Q_i's. And additionally as we stated, with just four restrictions, we cannot determine the value of any Q_i's. Which means our argument for the proof of Lemma 4.4.4 and Lemma 4.4.5 still applies on both Case 1 and Case 2. Thus, the decision complexity for the restricted formula for both Case 1 and Case 2 is $\Omega(m)$. Lastly by applying Theorem 4.2.1, we complete the proof for the lemma.

□

Lemma 4.4.7. *The expected number of restarts needed for $D^R_{ND\_RD}$ to find a satisfying assignment for $Ladder_n$ in polynomial time is m, where D_{ND_RD} will find a satisfying assignment in exponential time with probability at least $\frac{3}{4}$.*

Proof. The proof for Lemma 4.4.7 follows from Lemma 4.4.3 and Lemma 4.4.6.

□

With Lemma 4.4.7, we have shown that with probability at least $\frac{3}{4}$, $Ladder_n$ formula requires exponential time for $D_{\text{ND\_RD}}$ to solve. However this does not give us a separation between $D^R_{\text{ND\_RD}}$ and $D_{\text{ND\_RD}}$, as one could argue if for the remaining $\frac{1}{4}$ probability, $D_{\text{ND\_RD}}$ can solve Ladder formula in polynomial time, then one only needs to run $D_{\text{ND\_RD}}$ over the Ladder formula four times (in other words, a constant number of times), and then one of the four runs is then expected to have polynomial run time. To complete the proof, we construct another class of formulas $\mathcal{F}_T$ using $Ladder_n$ formulas and show that with probability at least $\frac{3}{4}^T$, $\mathcal{F}_T$ requires $\Omega(2^m)$ time for $D_{\text{ND\_RD}}$ to solve, and on the other hand, the expected number of restarts $D_{\text{ND\_RD}}$ needs to solve $\mathcal{F}_T$ in polynomial time is m^T. As a result, if we choose $T = \omega(1)$, and T grows just a little faster than a constant, we can super-polynomially separate $D_{\text{ND\_RD}}$ and $D^R_{\text{ND\_RD}}$.

To construct $\mathcal{F}_n$, we simply take the conjunction over T $Ladder_n$ formulas over disjoint sets of variables:

$$\mathcal{F}_T \equiv Ladder_n^1 \wedge Ladder_n^2 \wedge Ladder_n^3 \wedge ... \wedge Ladder_n^{T-1} \wedge Ladder_n^T$$

Theorem 4.4.8. *The expected number of restarts needed for $D^R_{ND\_RD}$ to find a satisfying assignment for $\mathcal{F}_T$ in polynomial time is m^T, where D_{ND_RD} will find a satisfying assignment in time $\Omega(2^m)$ with probability at least $\frac{3}{4}^T$.*

Proof. We first observe that, since $\mathcal{F}_T$ consist T $Ladder_n$ formulas over distinct variables, a satisfying assignment of $\mathcal{F}_T$ must satisfy every one of the T $Ladder_n$ formulas.

To show an upper bound for $D^R_{\text{ND\_RD}}$, we use a similar argument used in Lemma 4.4.3, where we showed that the expected number of restart required to set the d_k variables to the correct value is m. Now consider the case for $\mathcal{F}_T$, in order for $D^R_{\text{ND\_RD}}$ to solve $\mathcal{F}_T$, instead of setting n d_k variables to the desired value, the solver needs to set $n * T$ d_k variables before it can solver $\mathcal{F}_T$ in polynomial time. Then the expected number of restarts is $2^{Tn} = 2^{n^T} = m^T$.

To show lower bound for $D_{\text{ND\_RD}}$, we simply argue that since the probability of finding a satisfying assignment for one of the T $Ladder_n$ formulas is independent from finding a satisfying assignment of another, with probability at least $\frac{3}{4}^T$, $D_{\text{ND\_RD}}$ will get "stuck" in at least one of the T $Ladder_n$ formulas, and by Lemma 4.4.6, requires $\Omega(2^m)$ time to find a satisfying assignment for $\mathcal{F}_T$. □

Chapter 5

Conclusion

We made four contributions in this thesis. First, we discussed three classes of candidate instances proposed by proof complexity theorists for separating C_{ND_ND} and $C^{R}_{ND\_ND}$, and showed that both of these classes of instances are not able to exponentially separate C_{ND_ND} and $C^{R}_{ND\_ND}$. For the second part of the thesis, we prove results that establish the power of restarts (or lack thereof) for certain models of CDCL and DPLL SAT solvers. We first show that restarts do not provide any extra power for CDCL SAT solvers when we assume that both the variable and value selection heuristics are non-deterministic static. Second, we show that DPLL SAT solvers with non-deterministic dynamic variable selection, randomized dynamic value selection, and restarts are super-polynomially faster on the class of formulas $\mathcal{F}_n$ than the same model without restarts. Third, in order to show the second result we proved a generic result that may have wider applicability, namely, that the size of DPLL proofs of unsatisfiable formula φ is lower bounded by $2^{dc(\varphi)}$, where $dc(\varphi)$ is the decision complexity of φ. Lastly, we showed that CDCL SAT solvers with non-deterministic static variable selection, non-deterministic static value selection, and with restarts, are polynomially equivalent to the same model but without restarts. In fact, our result is stronger, in that, both configurations produce the exact same proof for the same unsatisfiable input formula. Further, their runs are identical for satisfiable instances as well. Finally, the result holds irrespective of the choice of learning scheme.

Crucial to our results were two insights. First, that there is very subtle interplay between restarts and other SAT solver heuristics in the models we studied, i.e., the power of restarts becomes apparent only when some other heuristic is weakened relative to their non-deterministic counterpart (e.g., randomized value selection). Second, we shifted the focus from unsatisfiable instances to satisfiable instances to prove our result vis-a-vis the drunk DPLL SAT solver model. In the future, we plan to lift our ideas to a large number of configurations we have identified as interesting both for restarts and other heuristics, leveraging the lessons learnt in this work.

References

[1] Michael Alekhnovich, Edward A Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of dpll algorithms on satisfiable formulas. In *SAT 2005*, pages 51–72. Springer, 2006.

[2] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 448–456. ACM, 2002.

[3] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, 2011.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[5] Paul Beame, Trinh Huynh, and Toniann Pitassi. Hardness amplification in proof complexity. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 87–96. ACM, 2010.

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

[7] Armin Biere and Andreas Fröhlich. Evaluating cdcl variable scoring schemes. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 405–422. Springer, 2015.

[8] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[9] Maria Luisa Bonet, Sam Buss, and Jan Johannsen. Improved separations of regular resolution from clause learning proof systems. *Journal of Artificial Intelligence Research*, 49:669–703, 2014.

[10] Samuel R Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize dll algorithms with clause learning. *arXiv preprint arXiv:0811.1075*, 2008.

[11] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[12] Stephen A Cook and Robert A Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, 1979.

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

[14] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[15] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000.

[16] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[17] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[18] Shai Haim and Toby Walsh. Restart strategy selection using machine learning techniques. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 312–325. Springer, 2009.

[19] Donald Knuth. *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[20] Jan Krajícek. Proof complexity. In *European Congress of Mathematics Stockholm, June 27–July 2, 2004*, pages 221–232, 2005.

[21] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[22] Jia Hui Liang. *Machine Learning for SAT Solvers*. PhD thesis, University of Waterloo, Canada, 2018.

[23] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 94–110. Springer, 2018.

[24] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[25] Joao P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[26] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[27] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 120–131. Springer, 2017.

[28] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI*, pages 1481–1484, 2008.

[29] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. *CP*, 9:654–668, 2009.

[30] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[31] Alasdair Urquhart. A near-optimal separation of regular and general resolution. *SIAM Journal on Computing*, 40(1):107–121, 2011.

[32] Ryan Williams, Carla P Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI*, volume 3, pages 1173–1178. Citeseer, 2003.

[33] Edward Zulkoski, Ruben Martins, Christoph Wintersteiger, Robert Robere, Jia Liang, Krzysztof Czarnecki, and Vijay Ganesh. Relating complexity-theoretic parameters with sat solver performance. *arXiv preprint arXiv:1706.08611*, 2017.