Prioritized Unit Propagation and Extended Resolution Techniques for SAT Solvers

by

Jonathan Chung

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, 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Jonathan Chung was the sole author for the entire thesis with the exception of the opening section of Chapter 3, Section 3.1, and Section 3.2, which are adapted from manuscripts that were originally written for publication. This thesis was written under the supervision of Dr. Vijay Ganesh.

Jonathan Chung was the lead author for the research presented in Chapter 3. Jonathan Chung performed this research in consultation with Dr. Vijay Ganesh, who was the primary investigator, and Dr. Samuel Buss, who contributed ideas and input to algorithmic design. Jonathan Chung performed all the coding, experimentation, and analysis, and wrote the draft manuscripts, to which all co-authors contributed intellectual input.

Abstract

NP-complete problems like the Boolean Satisfiability (SAT) Problem are ubiquitous in computer science, mathematics, and engineering. Consequently, researchers have developed algorithms such as Conflict-Driven Clause-Learning (CDCL) SAT solvers, aimed at determining the satisfiability of Boolean formulas. As the result of decades of research in the development of CDCL SAT solvers, these algorithms solve real-life SAT instances surprisingly quickly, performing well despite the fact that the SAT problem is believed to be intractable in general. While modern CDCL SAT solvers are efficient for many real-world applications, there is continual demand for ever more powerful heuristics for newer applications. This demand in turn provides the impetus for research in solver heuristics. In this thesis, we address this need by proposing a new heuristic for Boolean Constraint Propagation (BCP), a key component of CDCL SAT solvers, and a novel, extensible, architectural design of an Extended Resolution (ER) SAT solver, a class of solvers that is more powerful than CDCL solvers.

The impressive performance of CDCL SAT solvers on real-life Boolean instances is, in part, made possible by a combination of logical reasoning rules and heuristics integrated into different components of the solvers. Given that such combinations are currently the most successful paradigm in SAT solving, it is natural to ask how such combinations can be made even more efficient. We observe that there are two different approaches that can be taken to improve SAT solvers: one approach is to modify individual components within the SAT solving algorithm, and the other approach is to change the overall structure of the algorithm. We explore both approaches in this thesis.

Following the first approach, we examine a critical component of CDCL: the Boolean Constraint Propagation (BCP) algorithm, which systematically finds implications of variable assignments made by the solver. In most implementations of BCP, variable values are propagated *greedily* – the values of implied variables are set immediately after they are detected. This observation suggests that there could be a smarter way to perform BCP by prioritizing part of the search space rather than propagating implied variables immediately after they are encountered. In this work, we develop an algorithm which allows BCP to prioritize propagations, choose a heuristic priority ordering of the variables, and demonstrate a class of instances where our prioritized BCP algorithm, combined with this heuristic ordering, is able to outperform the traditional BCP algorithm.

For the second approach, we note that solvers are fundamentally mathematical proof systems, and that CDCL produces proofs in the Resolution proof system, which is theoretically weaker than Extended Resolution (ER), a related proof system. Hence, it is natural

to try integrating ER techniques into the CDCL algorithm, thus rendering it more powerful. However, it is well known that automating the ER proof system deterministically can be very challenging. Instead of proposing a single set of techniques to implement the ER proof system, we develop a programmatic framework (and an associated set of techniques) that enables one to upgrade CDCL solvers into an ER-based SAT solver. More precisely, we add three new major programmatic components: extension variable addition, extension variable substitution, and extension variable deletion. These components can be easily extended to test various ER ideas and heuristics. One of our considered heuristics is shown to be generally competitive with the baseline CDCL solver while improving upon the baseline for a specific class of cryptographic instances.

Acknowledgements

I would like to thank Professor Vijay Ganesh for his guidance and mentorship during my bachelor's and master's degrees, and for his valuable advice on the direction of my career. I would also like to thank Professor Samuel Buss for his excellent input, and I offer appreciation to my colleagues in my research group for their constructive feedback. I would further like to thank Professors Arie Gurfinkel and Derek Rayside for reading my thesis and providing their feedback.

Finally, I am extremely grateful to my friends and family for their constant support and for reminding me to eat, sleep, and take breaks from working. They made my time as a graduate student much more enjoyable.

Dedication

To the Earth.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is perhaps no problem in computer science more famous or frustrating than the question of P vs. NP. At its most fundamental level, computer science is the study of algorithms, which makes it especially compelling that such a seemingly simple question about algorithms has remained unresolved for more than half a century [26]: what makes a problem "easy" or "hard" for an algorithm to solve? Informally, P refers to the group of problems where solutions are easy to find, and NP refers to the group of problems where solutions are easy to verify. NP-complete problems are the most difficult problems in NP; finding an algorithm which efficiently solves one NP-complete problem would immediately yield efficient algorithms for solving *every* problem in NP. In this sense, all NP-complete problems are "equally difficult". These kinds of problems appear abundantly in many subdomains of mathematics [38, 51], engineering [14, 17], and computer science [47], so progress in improving our algorithms for solving NP-complete problems is not only of great theoretical interest, but also enormous practical interest.

The Boolean Satisfiability (SAT) problem, as the archetypal NP-complete problem [26], lies at the heart of P vs. NP. Although it has not yet been theoretically resolved whether NP-complete problems can be solved efficiently, the majority of computer science practitioners suspect that efficient algorithms for solving a general instance of the SAT problem do not exist. Despite this, gigantic real-world SAT instances containing millions of variables and constraints are regularly solved in astonishingly small time frames across many disparate fields thanks to modern Conflict-Driven Clause-Learning (CDCL) SAT solvers, which automatically search for solutions for the SAT problem. In theory, there should not be anything special about these algorithms – they are subject to the same underlying limitations as any algorithm for the SAT problem, and there are still many instances of the SAT problem where these solvers perform poorly. However, decades of cumulative research and

development have revealed various techniques and heuristics which work well in practice, even on problems for which the solvers were not originally designed. Some examples of this include fields such as cryptanalysis [66], program analysis [14], and mathematics [38, 51].

The remarkable performance of modern CDCL SAT solvers is made possible by a small number of core components [48]: backtracking search, efficient constraint propagation, heuristic-controlled decision branching, conflict analysis and conflict-driven clause learning, intermittent search restarts, and preprocessing/inprocessing. Historically, progress in improving the performance of CDCL SAT solvers has come from improving one or more of these components. Thus, it is natural to ask whether existing work in the literature has already identified all potential design spaces within these components, and whether we can augment solvers with additional components to improve their performance.

1.1 Problem Statement

The overarching problem addressed in this thesis is the question of how the traditional CDCL SAT solver framework can be modified to improve its performance. We explore two main ideas: first, a modification of the traditional propagation algorithm to follow a priority ordering of the variables; and second, an extension of the CDCL framework with additional components to incorporate the Extended Resolution (ER) proof system [83], which is theoretically known to be capable of producing exponentially smaller proofs than Resolution (Res), the proof system underlying traditional CDCL SAT solvers [6, 27, 49]. We take a similar approach to both ideas: first, we develop general frameworks to facilitate the testing of heuristics for these unexplored design spaces in current state-of-the-art SAT solvers; then, we implement our frameworks over existing SAT solvers; thirdly, using our implementations of our frameworks, we develop heuristics to improve the overall performance of the solvers; and finally, we conduct empirical evaluations of our heuristics to identify those which improve upon the original solvers.

To facilitate the discovery of useful heuristics, we first impose the following design requirements on our frameworks:

1. First, the frameworks should be extensible – they should support the usage of a wide range of heuristics. Particularly, for the case of ER, where there has already been some work in developing ER-based solvers, the framework must also support the implementation of comparable methods in the literature.

2. Second, the computational overhead incurred by the frameworks should not significantly deteriorate the performance of the underlying SAT solver.

We further require our discovered heuristics to have the following properties:

1. There must be at least one class of problem instances where the heuristics improve over the performance of the base solver; i.e., the average solving time of the solver augmented with the heuristic should be less than the average solving time of the original solver, and the number of instances solved should be larger.

2. The heuristics must not cause the performance of the base solver to degrade significantly over other practical classes of problem instances.

1.2 Summary of Contributions

Prioritized Boolean Constraint Propagation: We first propose a new priority-based BCP algorithm called Delayed BCP, and explain it using the idea of a *variable assignment lifecycle*, where each variable transitions through a set of states during the run of a CDCL solver. In particular, we identify the "queued" state, which previously had not been explicitly identified as a state. The Delayed BCP algorithm acts as a framework for testing various heuristic priority orderings. We implement the Delayed BCP method and empirically evaluate its performance, and find that it improves solver performance on some classes of instances while degrading performance for other instance classes. Then, we use Reinforcement Learning (RL) methods with online learning to dynamically switch between our priority-based BCP method and the traditional BCP method, and demonstrate a smaller performance deterioration while maintaining improved performance on a select class of instances. A slightly more detailed explanation of each contribution follows:

1. **Delayed BCP:** We introduce the **Delayed BCP** algorithm, which differs from the traditional (Immediate BCP) algorithm in terms of the order in which variables are assigned. In Immediate BCP, queued variables are assigned as soon as a unit clause is discovered ("immediately"), and propagation is done in the same order as the variables are assigned ("in-order"). By contrast, in Delayed BCP, variables that are intended to be assigned a value are first queued, and then they are assigned values according to a desired variable priority order ("delayed" and "out-of-order"). This delay in assigning values to variables changes the order in which variables get propagated during BCP, as opposed to Immediate BCP, which in turn changes how conflicts arise during unit propagation.

2. **Dynamic BCP Algorithm Selection with Thompson Sampling:** We model the BCP algorithm selection problem as a Multi-Armed Bandit (MAB) problem [70] and use a Thompson sampling approach [80, 81] as an online learning method to choose between the different BCP algorithms. Since we expect the best choice of BCP algorithm to change throughout a run of the solver, we modify the Thompson sampling method to encourage the decision agent to explore different choices.

3. **Extensive Experimental Evaluation:** We implement the Delayed BCP technique in the `MapleSAT` [55], `MapleLCM` [60], and `CaDiCaL` [10] solvers, and perform extensive empirical evaluations against the corresponding baseline solvers over a wide selection of benchmark instances. We show empirically that despite the fact that Delayed BCP often requires additional unit propagations to detect a conflict, and despite the additional overhead associated with maintaining a priority queue which slows down unit propagation even when the solver is not in a conflicting state, our method outperforms the baseline solvers over some classes of instances generated from cryptographic applications. It is especially notable that despite performing similarly to the baseline solver in terms of CPU time, our method performs a smaller number of propagations overall, indicating that priority-based BCP performs "smarter" propagations than the traditional method.

Extended Resolution SAT Solvers: We propose a framework to augment existing CDCL SAT solvers with Extended Resolution (ER) techniques, building upon previous work in the literature [3, 43]. In the implementation of our framework, we introduce various data structures and algorithms for the efficient management of extension variables and their definitions. Using this framework, we identify a handful of possible heuristics for each component of the method, and conduct an empirical evaluation to determine whether our techniques are helpful. To illustrate the versatility of our method, we also implement some existing ER techniques from the literature [3, 13] using our framework and evaluate them experimentally. A slightly more detailed explanation of each contribution follows:

1. **ER Solver Development Framework:** Our framework for developing ER-based SAT solvers consists of three major components: variable definition, variable substitution, and variable deletion. We further divide variable definition into three minor subcomponents: clause filtering/selection, definition generation, and clause introduction. We identify common tasks that are necessary for the implementation of any practical ER solver and provide algorithms for managing extension variables and definition clauses, extracting common functionality to minimize the work required

to develop new ER solver heuristics. In particular, we provide algorithms for introducing and keeping track of extension variables and their definitions, substituting extension variables into learnt clauses, and deleting unused extension variables.

2. **ER Solver Heuristics:** We motivate a handful of natural heuristics for each component of the framework; e.g., only considering low-LBD clauses when defining extension variables, substituting variables in low-width clauses, and deleting low-activity extension variables. We also consider some heuristics for introducing extension variables: one based on randomly selecting literals from clauses with high activities, and another based on common pairs of literals that appear frequently among high-activity clauses.

3. **Empirical Evaluation:** We implement the heuristics discussed above and perform a large set of experiments to evaluate their efficacy. We additionally implement the Local Extended Resolution idea introduced in the `GlucosER` and `MiniSATER` solvers [3] to check the robustness of their results. We especially focus on the Pigeonhole Principle (PHP) formulae because short proofs are known for this class of instances [27], and Urquhart formulae, because previous ER techniques reported improved performance on these instances [3]. Both the PHP and Urquhart families of problem instances are considered to be difficult for traditional CDCL SAT solvers.

The rest of the thesis is structured as follows: Chapter 2 provides background on the SAT problem (including Boolean Constraint Propagation and Extended Resolution) and relevant information from the field of machine learning (particularly Reinforcement Learning); Chapter 3 presents our methods for Prioritized BCP, an empirical evaluation of the different techniques, and a discussion of related work; Chapter 4 presents our framework for developing Extended Resolution SAT solvers, a selection of the algorithms developed to address the problem, an empirical evaluation of our method, and a discussion of related work; and finally, Chapter 5 concludes the thesis and outlines potential areas of research for future work.

Chapter 2

Background

This chapter presents the background information necessary to understand the contributions in this thesis, including an overview of the Boolean Satisfiability (SAT) Problem, Conflict-Driven Clause-Learning (CDCL) SAT solvers, and relevant methods from machine learning. We present definitions of relevant terminology and review important concepts.

2.1 SAT Solving

SAT solvers are algorithms for automatically solving instances of the Boolean Satisfiability Problem. To understand the definition of this problem, we first need to define some key concepts about Boolean formulae.

A Boolean formula is essentially a function over Boolean variables, which are variables that can take on one of two values: either *true* (represented by the symbol $\top$) or *false* (represented by the symbol $\bot$). Boolean formulae can be represented syntactically by combining Boolean operators, or semantically by *truth tables*, which list the output values corresponding to each possible input. Since truth tables must contain separate entries for every possible input, they tend to be much larger than the smallest equivalent syntactic combination of operators. Thus, Boolean formulae are conventionally expressed by combining Boolean operators rather than by their truth tables.

2.1.1 Conjunctive Normal Form (CNF)

In general, Boolean formulae can have arbitrarily complex structures, with diverse operators nested and sequenced in many different ways. This structural complexity makes it difficult for SAT solving algorithms to efficiently reason with these formulae directly. Fortunately, all Boolean formulae can be expressed in multiple equivalent ways, and there is an efficient algorithm which can transform an arbitrary Boolean formula into Conjunctive Normal Form (CNF) [83], which is a format that is much easier to reason about. Consequently, the vast majority of modern SAT solvers expect the input formula to be in CNF. For the remainder of this thesis, Boolean formulae are assumed to be in CNF.

We define Boolean formulae in CNF syntactically as follows: A *variable* (typically represented by one of the symbols x, y, or z) is a Boolean variable. A *literal* is either a variable by itself (x), which is said to be a *positive* literal, or the negation of a variable ($\neg x$), which is said to be a *negative* literal. A *clause* is a disjunction of literals; e.g., $c = (x_1 \vee x_2 \vee \neg x_3)$. Finally, a *formula* is a conjunction of clauses; e.g., $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee \neg x_5)$. For convenience, we will sometimes refer to a clause as a set of literals, where the disjunctions between literals are implied. Similarly, we will sometimes refer to a formula as a set of clauses, where the conjunctions between clauses are implied.

Evaluating Boolean Formulae Recall that Boolean formulae represent functions over Boolean variables, where mapping variables to different truth values can result in different outputs for the formula. This mapping of variables to truth values, either $\top$ or $\bot$, is called an *assignment*, denoted α. If an assignment maps every variable in the formula to a truth value, the assignment is said to be *complete*. Otherwise, the assignment is said to be *partial*. If a variable x appears in an assignment α, x is said to be *assigned* by α. Otherwise, x is said to be *unassigned*. For convenience, we sometimes express variable assignments as a set of literals, where the sign associated with a literal stands in for the truth value in the assignment. For example, the assignment $\alpha = \{x \mapsto \top, y \mapsto \bot\}$ maps x to true and y to false. Equivalently, this could be denoted $\alpha = \{x, \neg y\}$.

Then, given a CNF Boolean formula and a variable assignment α, we can compute the truth value of the formula *under* α as follows:

- The value of a variable x is given directly by α. If α is a partial assignment that does not map x to a truth value, then the truth value of x is undefined.
- The truth value of a literal l is equal to the truth value of x if $l = x$. Otherwise, if $l = \neg x$, then the truth value of l is equal to the negation of the truth value of x. If the truth value of x is undefined, then the truth value of l is also undefined.

- A clause C evaluates to $\top$ if it contains any literal l which evaluates to $\top$. A clause C evaluates to $\bot$ if every literal in C evaluates to $\bot$. Otherwise, the truth value of C is undefined.
- A formula ϕ evaluates to $\top$ if every clause in ϕ evaluates to $\top$. A formula ϕ evaluates to $\bot$ if it contains any clause C which evaluates to $\bot$. Otherwise, the truth value of ϕ is undefined.

Clauses that evaluate to $\top$ under an assignment are said to be *satisfied*, and those that evaluate to $\bot$ are said to be *falsified*. Clauses that are neither satisfied nor falsified, and which contain only a single unassigned literal, are said to be *unit clauses*, and the unassigned literals in such clauses are called *unit literals*. Assignments which cause ϕ to evaluate to $\top$ are called *satisfying assignments*, and assignments which cause ϕ to evaluate to $\bot$ are called *falsifying assignments*. Falsifying partial assignments cannot be extended to satisfying complete assignments by assigning additional variables. In SAT solvers, variable assignments are typically stored in a list, where variables are positioned in the list according to the order in which they became assigned. This ordered list is known as the *assignment trail*. For convenience, we use the terms *variable assignment* and *assignment trail* interchangeably.

2.1.2 The Boolean Satisfiability Problem

Given an input Boolean formula ϕ, the Boolean Satisfiability (SAT) problem is the question of whether it is possible to assign values to the variables of ϕ such that ϕ evaluates to $\top$. We refer to algorithms for solving this problem as *SAT solvers*. SAT solvers determine whether ϕ is *satisfiable* (i.e., there exists a satisfying variable assignment) or *unsatisfiable* (i.e., every possible variable assignment is a falsifying assignment). When ϕ is satisfiable, the SAT solver must output "SAT" with a satisfying assignment. Otherwise, if ϕ is unsatisfiable, the SAT solver must output "UNSAT". In some applications, the SAT solver is also required to output a proof of the formula's unsatisfiability.

2.1.3 Boolean Constraint Propagation

The Boolean Constraint Propagation (BCP) algorithm, sometimes also called unit propagation, was first introduced as part of the DPLL SAT solver [29, 28]. The key observation underlying the BCP algorithm is the following: if a partial assignment α can be extended to a complete satisfying assignment α' and α falsifies all the literals in a clause C except

Algorithm 1: The traditional algorithm for performing BCP

Input: FIFO propagation queue: q
Output: A falsified clause if one exists

```
1  while q.size() > 0 do
2      p ← q.pop();
3      foreach clause c rendered unit by p, resulting in unit literal l do
4          if c is falsified then
5              q.clear();
6              return c;
7          else
8              q.push(l);
9              assign(l);
10             reason[var(l)] ← c;
11         end
12     end
13 end
14 return no conflict;
```

for one unassigned literal l, then α' must assign the variable corresponding to l such that l evaluates to $\top$. The BCP algorithm consists of repeatedly finding and assigning these unit literals until it reaches a fixed point. This fixed point can be SAT (i.e., the formula is satisfied by the current assignment), conflict (i.e., the formula is falsified under the current assignment), or UNKNOWN (i.e., BCP alone cannot determine the satisfiability of the input formula under the current partial assignment). Variables assigned through BCP are said to be *implied*. The BCP algorithm is responsible for assigning all the additional variables in the input formula implied by previously assigned variables and identifying clauses that are falsified by the new variable assignments if such clauses exist.

The pseudocode for the traditional BCP algorithm (which we refer to as Immediate BCP) is presented as Algorithm 1. For brevity, we omit the details of BCP associated with maintaining data structures for the two-watched literal scheme, which is an optimization for the BCP algorithm. When a CDCL SAT solver makes a branching decision on a variable, that variable (along with its assigned polarity) is added to the propagation queue. In Immediate BCP, the propagation queue is a First-In-First-Out (FIFO) data structure, so the algorithm processes variables in the order that they appear in the queue. This corresponds to lines 1 and 2 of Algorithm 1, where Immediate BCP continually gets the next variable p from the queue until there are no variables remaining to be processed. Line 3

iterates through all the clauses and checks whether they become unit clauses after assigning p. If the clause is falsified by the assignment of p, Immediate BCP clears the propagation queue and returns the conflicting clause in lines 4-6. Otherwise, the unit literal l is assigned the value $\top$ and is added to the propagation queue in lines 8 and 9. The value assigned to l is immediately available to assist further unit propagation. Line 10 records the clause that is responsible for the propagation – this is used to maintain an *implication graph*, which is then used for learning clauses after BCP detects a conflict. Finally, if BCP finishes propagating everything in the propagation queue without encountering a falsified clause, it returns *no conflict* at line 14.

Two-Watched Literal Scheme: The major development in BCP since its inception in DPLL was the invention of the two-watched literal scheme, introduced as part of the `Chaff` SAT solver [64]. The two-watched literal scheme enables the efficient detection of unit literals and falsified clauses in many practical contexts, thus greatly accelerating the speed of BCP. The insight behind this scheme is that clauses become unit only when there is exactly one literal unassigned under a partial assignment. Thus, if the BCP scheme keeps track of two unassigned literals for each clause, it can efficiently determine whether or not a clause is unit without examining the entire clause.

2.1.4 Conflict-Driven Clause-Learning (CDCL) SAT Solver

The DPLL class of algorithms pioneered by Davis and Putnam [29, 28] represents the first significant attempt to automatically solve instances of the SAT problem. The DPLL algorithm is a backtracking search algorithm, where the algorithm guesses assignments for variables, computes the implications of its guesses using BCP, and reverts its guesses if they do not result in a satisfying variable assignment.

After several decades of research work, the field was revolutionized by the introduction of the CDCL paradigm in the `GRASP` SAT solver [62], which now serves as the foundation of most modern SAT solvers. Whereas DPLL implicitly learned that certain variable assignments led to conflict and did not explore them again due to the backtracking nature of the search, CDCL built upon DPLL by explicitly learning clauses to block conflicting variable assignments. This change resulted in improvement over the DPLL algorithm because learnt clauses could block the underlying reason for the conflict rather than the entire variable assignment that led to the conflict. Another reason for the performance improvement of CDCL over DPLL is that learnt clauses can be used as part of the unit propagation procedure. Pseudocode for the CDCL procedure is presented as Algorithm 2.

Algorithm 2: Pseudocode for the CDCL Procedure

```
   Input: Boolean formula φ
   Output: SAT or UNSAT
 1 while true do
 2 |   propagate();
 3 |   if conflict() then
 4 |   |   analyze();
 5 |   |   if rootConflict() then return UNSAT ;
 6 |   |   backjump();
 7 |   else
 8 |   |   if allVariablesAssigned() then return SAT ;
 9 |   |   maybe forget();
10 |   |   maybe restart();
11 |   |   decide();
12 |   end
13 end
```

Branching In CDCL solvers, variables can become assigned in two different ways: either by BCP as described in Section 2.1.3, or as a *decision* when no clauses have been falsified and there are no unit clauses remaining to be propagated (line 11 of Algorithm 2). In this second case, the solver essentially guesses the value corresponding to a variable. Variables assigned this way are said to be *decision variables* or *branch variables*, and the combination of a decision variable and its corresponding value is called a *decision literal* or *branch literal*. The *decision level* for a variable x is the number of decision variables already in the assignment trail when x is assigned, excluding x itself.

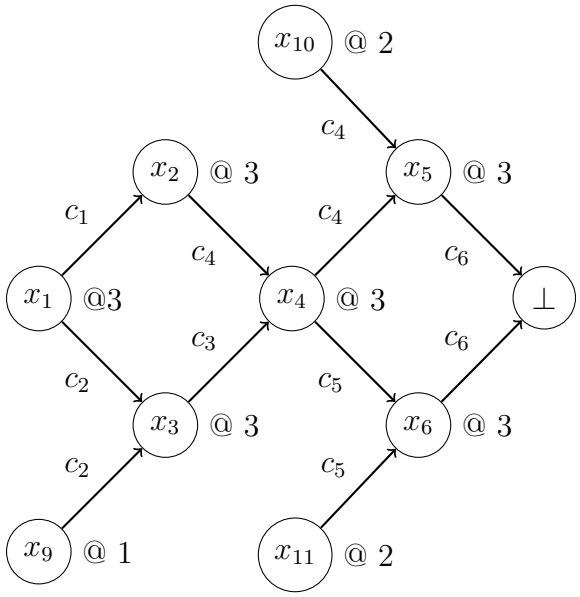
As evidenced by results from empirical work [64, 57], the order and polarity in which the solver branches upon variables is a critical factor in the overall running time of the algorithm. Thus, the design of the *decision heuristic* (a.k.a. *branching heuristic*) in CDCL solvers is the subject of a large body of research in the SAT research community. In practice, the branching heuristic is often divided into a *variable selection heuristic*, which chooses the order of decision variables, and a *polarity heuristic*, which chooses the truth value to assign to the decision variable. Some common variable selection heuristics are VSIDS [64, 8], CHB [56], and LRB [59]. We note that the polarity heuristic in most SAT solvers is the phase saving heuristic [68], and as a result, the SAT literature often refers to the variable selection heuristic alone as the branching heuristic, assuming that that the polarity heuristic is phase saving. In this thesis, we will usually also refer to the

Previous assignment:
$\{x_9@1, \neg x_{10}@2, \neg x_{11}@2\}$

Decision:
$x_1@3$

$c_1 = (\neg x_1 \vee x_2)$
$c_2 = (\neg x_1 \vee x_3 \vee x_9)$
$c_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 = (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 = (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 = (\neg x_5 \vee \neg x_6)$
$c_7 = (x_1 \vee x_8)$

(a) Clause Database



(b) Implication graph

Figure 2.1: A clause database and a partial implication graph, focusing on decision level 3. Variable assignments are labeled with their decision levels, and edges are labeled with their reason clauses. This example was adapted from the original GRASP paper [62].

variable selection heuristic as the branching heuristic. However, where it is relevant, we will distinguish between the variable selection heuristic and the polarity heuristic.

Implication Graphs The CDCL paradigm improves upon the DPLL algorithm by deductively learning additional clauses when clauses in the formula become falsified. This is done by maintaining a data structure known as an *implication graph*, which records the reasons and levels at which variables are assigned, either via a decision or through BCP. When a clause is falsified, the solver examines the implication graph to determine the reason for the conflict, and learns a new clause to prevent that conflict from reoccurring [62].

An implication graph is a directed acyclic graph. Every assigned variable is a node in the graph, where the values of variables come immediately from decisions made by the SAT solver, or are *implied* by unit clauses. Directed edges correspond to unit clauses, where there is an edge from x to y if both x and y appear in a unit clause C, the literal for x is falsified in C, and y is the unit literal in C. By convention, nodes with no incoming edges (*source nodes*) correspond either to unit literals in the input formula or to decisions

made by the SAT solver. When the solver is in a conflicting state and a clause is falsified, the corresponding implication graph is called the *conflict graph*. In this case, there is also an extra *conflict node* which corresponds to the falsified clause. An example of an implication graph is shown in Figure 2.1. In illustrations of implication graphs in this thesis – particularly, for conflict graphs – to focus on the part of the graph responsible for the conflict, we typically omit many parts of the implication graph which are set at decision levels before the conflicting decision level. We will also omit the decision level labels and edge labels in the remainder of this thesis.

Clause Learning When a conflict arises and a clause is falsified during a solver's search, the conflict is typically only caused by a small subset of the decision variables. The major difference between the DPLL and CDCL algorithms is that where DPLL implicitly blocks the combination of all the decision variables at the time of conflict, CDCL instead explicitly learns clauses encoding the underlying reason for the conflict to prevent it from happening again. In order to do this, the CDCL algorithm performs *conflict analysis* by examining the implication graph after BCP discovers a conflict (line 4 of Algorithm 2).

Clauses can be learned using the conflict graph by starting from the conflict node and exploring backward along the edges, keeping track of a frontier of edges that have not yet been explored [62]. At any time along this backward exploration, the vertices connected to the origins of the unexplored edges imply the conflict by unit propagation. Then the partial variable assignments corresponding to those vertices will always be a falsifying assignment, so the solver can block that assignment by negating the conjunction of those variables. A more graphical way to understand the algorithm is to look at the frontier as a "cut" of the conflict graph, partitioning the vertices into two sets: one which contains vertices on the same side as the conflict, and another which contains vertices on the same side as the decision variables. Then for any cut where all the decision variables are on the "left" and the conflict is on the "right", the vertices immediately to the left of the cut correspond to a possible learnt clause.

For any given conflict graph, there may be many different ways to construct a cut of the graph, and consequently, there are many possible clauses which can be learnt when the solver encounters a conflict. Hence, there are many different possible clause learning schemes, but the one which is most popular in current SAT solvers – and which is the only learning scheme used in this thesis – is the *First Unique Implication Point* (1UIP) clause learning scheme [64, 62]. This scheme never explores edges corresponding to variables set at decision levels before the conflict, so the only remaining question is where to place the cut at the conflicting decision level. Let d be the decision variable which was set immediately

before the conflict. The 1UIP scheme works by considering nodes v in the graph where every path from d to the conflict node passes through v (i.e., *unique implication points*). Since there may be multiple such nodes, the scheme selects the one closest to the conflict node (i.e., the *first* UIP encountered when exploring backward from the conflict node). Then the scheme places the cut immediately before this node (i.e., intersecting the incoming edges), which yields the 1UIP learnt clause.

Backtracking and Backjumping Once the solver has detected a conflict and generated a learnt clause, it must backtrack and unassign some of the assigned variables until it is no longer in a conflicting state (line 6 of Algorithm 2). In DPLL, it is only the most recent decision that leads to the conflict, so it is sufficient to only unassign the variables set at the conflicting decision level, up to and including the decision variable x which led to the conflict. In this situation, it is guaranteed that no clauses will be falsified after unassigning those variables. This policy of only undoing the variable assignments at the most recent decision level (going from decision level n to decision level $n-1$) is known as *chronological backtracking*, or simply *backtracking*. After backtracking, the solver knows that x must be set to the opposite polarity under its current partial assignment, so it "decides" to set x to the other truth value. Once the solver has encountered conflicts with both polarities for x, it backtracks by an additional decision level.

Although the idea of backtracking is similar for CDCL, the implementation is slightly more complicated: if the learnt clause C only contains a single literal l from the conflicting decision level n (which is the case for 1UIP learnt clauses), then after backtracking, C becomes a unit clause, which should propagate l at the next highest decision level n' in C, which could then result in additional implications at level n'. Since it is often the case that $n' \neq n-1$, it is not sufficient to backtrack by a single decision level – rather, the solver must undo all the variable assignments at made after decision level n'. This policy of reverting the variable assignments from multiple decision levels at once is known as *non-chronological backtracking*, or *backjumping*, and is the standard backtracking/backjumping technique for modern CDCL solvers [62].

Solver Restarts A *restart* is essentially a backjump to decision level 0: it discards the existing implication graph and unassigns all the variables set after decision level 0 [33]. However, unlike backjumping, restarts are not triggered by conflicts and learnt clauses. Instead, restarts are typically triggered by a heuristic condition when the solver is in a *non-conflicting* state after BCP (line 10 of Algorithm 2). At first glance, restarts appear counter-productive, discarding the search effort that had been made by the solver. How-

ever, restarts only change the variable assignment – they do not affect the other parts of the solver state; restarts preserve the solver's learnt clauses and also the data associated with heuristics, such the ordering prescribed by the branching heuristic. In practice, restarts allow SAT solvers to quickly solve instances which otherwise require enormous amounts of computation time [32, 58], and solvers with restarts have been theoretically proven to be more effective[1] than solvers without restarts [54].

Clause Deletion Each clause learnt by a CDCL solver blocks a (partial) assignment, thereby reducing the size of the solver's search space. Unfortunately, storing and using learnt clauses is not free – there is a memory cost required to store learnt clauses, and even ignoring the details of real-world computer architectures, there is a run time cost associated with checking each clause for propagation during BCP. In the worst case, CDCL solvers may have to learn *exponentially* many clauses [25, 27], which quickly becomes impractical for solver implementations. Fortunately, many learnt clauses are only useful in a "local" section of the solver's search, and can be forgotten after the solver no longer needs them, thereby freeing up computational resources for additional learnt clauses. In some cases, it may be productive to forget learnt clauses even if they are useful in multiple parts of the search, as long as they can be quickly re-learnt.

For these reasons, modern CDCL SAT solvers frequently perform *clause deletion*, discarding large numbers of learnt clauses according to *clause deletion heuristics* (line 9 of Algorithm 2). Clause deletion heuristics must be designed carefully: if they delete the clauses the solver needs in order to progress, the solver may enter a loop of deriving useful clauses, deleting them, and re-deriving the useful clauses. Unfortunately, it can be difficult to determine *a priori* which learnt clauses are only useful in the short term, and which will be useful in the solver's future search. The approach taken for this problem in most modern CDCL solvers is to define some measure of clause "quality" and then use it to determine which learnt clauses can be deleted, where "high quality" clauses are kept and "low quality" clauses are discarded. We note that only learnt clauses are deleted – removing the original input clauses changes the meaning of satisfiability, and it is non-trivial to preserve the original formula's satisfiability or unsatisfiability if its clauses are deleted.

Measures of Clause Quality There are two different approaches to defining clause quality: the first is to use theoretical arguments for situations where clauses are useful, and the second is to collect data during solver execution to predict whether clauses will

[1] The propositional proof systems corresponding to solvers with restarts are exponentially stronger than those without restarts.

be useful in the future. A natural property to use as a measurement of clause quality is the size of the clause (i.e., the number of literals in the clause). Intuitively, shorter clauses prune away a larger section of the search space, so shorter clauses are "higher quality" clauses. A metric with similar intuition to clause size is the concept of *Literal Block Distance* (LBD), which is defined as the number of distinct decision levels appearing in a clause at the time that it is learnt by the solver [4]. Intuitively, clauses which can be falsified by a smaller number of decisions are more useful than clauses which can only be falsified by a large number of decisions, so clauses with small LBD values are "higher quality" clauses. Another way to select "high quality" clauses is to assume that clauses which have proven to be useful recently in a solver's search will continue to be useful in the future, using the solver's actual behaviour to determine the quality of clauses. One possible approach to this is to extend the idea of VSIDS [64] from variables to clauses resulting in the idea of clause activity [31].

2.1.5 Propositional Proof Systems

Directly studying SAT solvers from a theoretical perspective can be quite difficult: SAT solvers tend to be very complex algorithms with many moving parts, and this makes them difficult to model mathematically. As an abstraction, theorists study SAT solvers from the lens of *proof complexity*: the execution of SAT solving algorithms can be modelled using *proof systems*, and then proving formal theorems about the proof systems allows for statements about the behaviour of the corresponding algorithms. In general, proof systems consist of axioms and inference rules for deriving new facts from old ones, but in this thesis, it is sufficient to think of proof systems as rules for deriving additional clauses.

Then a *proof* for a clause is the series of inference rule statements required to derive it. We will primarily be concerned with proofs for the empty clause, which corresponds to an unsatisfiable formula ("refutation" proofs). Some concepts which are often used to discuss proof systems are "strength" and "automatability". The idea of strength is used to compare different proof systems: if two proof systems are able to prove that a formula is unsatisfiable, the one which requires fewer proof steps – or equivalently, the one with the smallest proof size – is said to be "stronger" than the other. The idea of automatability refers to the difficulty of searching for proofs in a proof system, or the time complexity required for a deterministic algorithm to find a proof that is at most polynomially larger than the shortest possible proof [7]. If any proof in proof system A can be converted into a proof in proof system B in at most polynomial time, A is said to "p-simulate" B. If A p-simulates B and B p-simulates A, then A and B are said to be "p-equivalent."

Resolution In CDCL, the clause learning procedure can be modelled by applications of the *resolution* inference rule, which is a rule for deriving a new clause (known as the *resolvent*) from two existing clauses. In particular, consider two clauses C_A and C_B where C_A contains a literal l and C_B contains the negation of the literal, $\neg l$). Whenever such a pair of clauses exists, resolution permits the derivation of the clause containing the literals $(C_A \cup C_B) \setminus \{l, \neg l\}$.

$$\frac{(a_1 \vee a_2 \vee ... \vee a_n \vee l) \quad (\neg l \vee b_1 \vee a_2 \vee ... \vee b_m)}{(a_1 \vee a_2 \vee ... \vee a_n \vee b_1 \vee a_2 \vee ... \vee b_m)} \; \textit{Resolution}$$

The *general resolution* proof system (Res) is the proof system which contains resolution as the only inference rule. Res is a *sound* and *complete* refutation proof system. Here, *sound* means that the resolvent is always a logical consequence of the two input clauses, and *complete* means that if a formula is unsatisfiable, Res is able to derive the empty clause after finitely many applications of the resolution proof rule) [71].

The clause learning scheme described in Section 2.1.4 can be viewed as a series of applications of the resolution proof rule: performing resolution on all the clauses corresponding to edges on the conflict side of the implication graph results in the same clause as the learnt clause generated by the clause learning scheme. Since all the clauses learnt by CDCL can be modelled by an equivalent series of resolution steps, the time complexity of the CDCL algorithm on an input of size n is bounded by the size of the smallest possible Res proof for that same input. In fact, under some assumptions on branching and restarts, CDCL is known to p-simulate general resolution [69]. Unfortunately, despite this promising result, it turns out that unless $P = NP$, resolution is not automatable [2].

Extended Resolution Another proof system which is closely related to Res is the Extended Resolution (ER) proof system discovered by Tseitin [83], which introduces one additional inference rule (known as the *extension rule*) in addition to resolution: it allows for *extension variables* (also known as *new variables*) to be introduced into the formula by the addition of clauses encoding their definitions, where new variables x are defined as being equivalent to a disjunction of two literals a and b. This is done by converting the formula $x \leftrightarrow (a \vee b)$ to CNF, which results in the addition of the clauses $(\neg x \vee a \vee b) \wedge (x \vee \neg a) \wedge (x \vee \neg b)$. Some work in the literature uses an alternative definition of the extension rule, where new variables can be defined to be equal to any formula over existing variables. However, this is not necessary and does not affect the theoretical strength of ER: a definition over a large formula can be broken down into smaller definitions by introducing additional extension variables. These two variants of ER are p-equivalent.

Since the inference rules in ER comprise a strict superset of the inference rules in RES, it is obvious that ER p-simulates RES: every RES proof is already a ER proof. However, not every ER proof is a RES proof, and RES does not p-simulate ER. In fact, ER is exponentially stronger than RES; i.e., there are formulas where the smallest RES proof is exponentially larger than the smallest ER proof [34, 27]. Despite its conceptual simplicity, ER is as strong as the strongest known propositional proof systems [52]. Unfortunately for SAT solver developers, ER p-simulates RES and RES is not automatable, so ER is also not automatable.

2.1.6 Empirical Evaluation

The performance of SAT solvers is typically compared whenever changes are made to their source code, heuristics, or parameter values. In this thesis, as well as in much of the work in the SAT community, solvers are compared using *cactus plots* and *PAR-2 scores*.

Cactus Plot Results of experiments are typically visualized using cactus plots. To generate a cactus plot for a solver, the solver is run on every instance in a benchmark, and the computation time required to solve each instance is recorded. If a solver runs out of time, the recorded time is set equal to the time limit. Then, the instances are sorted in ascending order according to the computation time, and each instance is assigned an index corresponding to its position in the sorted order. A point is plotted for each instance, where the x-axis corresponds to the solving time, and the y-axis corresponds to the index. Then, the points in the plot can be interpreted as the number of instances that were solved for a specific amount of computation time. Finally, although the data is discrete, data points with consecutive indices are often joined by straight lines. An example of a cactus plot is presented as Figure 2.2. Curves which are higher up and further to the left indicate stronger solvers.

PAR-2 Score Penalized Average Runtime (PAR) scores [12] are the standard metric used to compare SAT solver performance, both in the empirical SAT literature and at the SAT competition. In both these contexts, SAT solvers are not permitted to run indefinitely – there is a CPU time limit which is enforced for each problem instance. A PAR score is essentially a weighted average of a SAT solver's running time across an entire benchmark; it tries to reward instances where the SAT solver successfully solved the problem and penalize instances where the SAT solver ran out of time. For example, using a time limit of 5000 seconds, the penalty is $2 \times 5000 = 10000$. Therefore, the PAR-2 score for each solver is

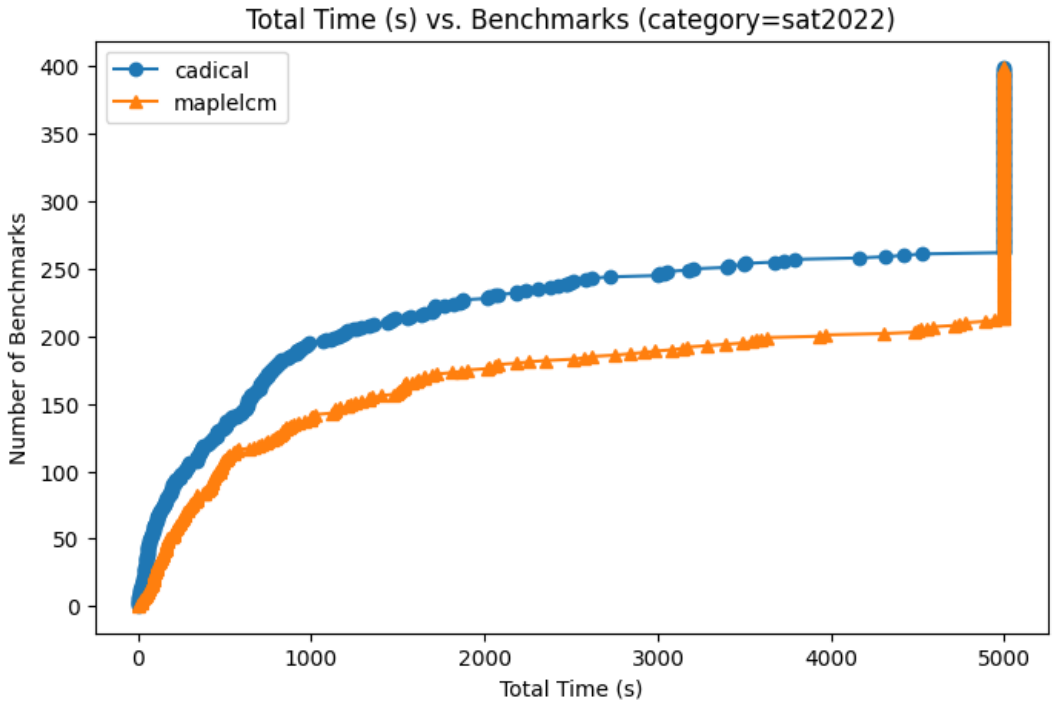


Figure 2.2: A cactus plot for the `CaDiCaL` and `MapleLCM` SAT solvers on the 400 instances from the main track of the SAT 2022 competition using a 5000 second time limit.

computed as follows, where n is the total number of instances and t_i is the time taken by the solver to solve instance i:

$$PAR\text{-}2 = \frac{1}{n}\sum_{i}^{n}\begin{cases} t_i & \text{if instance } i \text{ was solved} \\ 10000 & otherwise \end{cases} \tag{2.1}$$

Computational Environment All the experiments in this thesis were conducted using a time limit of 5000 seconds of CPU time on Intel E5-2683 v4 Broadwell @ 2.1GHz CPUs on the Graham computing cluster [67] for every solver on each instance. The time limit was chosen according to the standard for the SAT competition [21].

2.2 Machine Learning

Machine learning is a field of computer science which is broadly concerned with learning patterns from data, typically for either classifying data or making predictions about future data. Machine learning techniques have been wildly successful in tackling difficult problems in many different domains, including formal logic and the development of SAT solvers [59]. Although there are many approaches to machine learning, we will focus solely on *reinforcement learning* in this thesis.

2.2.1 Reinforcement Learning

At its core, reinforcement learning is a tool for solving optimization problems, where there is some notion of an "optimal" or "best" solution. Under the reinforcement learning paradigm, these optimization problems are modelled as a series of interactions between an *agent* and its *environment*, where the agent chooses between *actions* depending on the *state* of its environment, and the environment responds to these actions by some modification in its state. As suggested by its name, the key idea in reinforcement learning (RL) is analogous to the idea of reinforcement in psychology [82]. When the agent chooses an action that causes a "desirable" change in its environment, it receives *positive reinforcement* as feedback, rewarding its good behaviour. Similarly, when the agent chooses an action that causes an "undesirable" change in its environment, it receives *negative reinforcement*, punishing its bad behaviour. In reinforcement learning, the positive and negative reinforcement are often collectively called the *reward*.

The Multi-Armed Bandit Problem Although there are many sophisticated problems and techniques in RL, we only consider a very simple formulation of RL problems in this thesis. In particular, we consider the Multi-Armed Bandit (MAB) problem, and introduce it with the classical motivating example [79]. Consider a gambler (a "bandit") with access to n slot machines ("arms"), each of which rewards different amounts of money according to different probability distributions. When the gambler plays a slot machine, they *sample* from that machine's *reward probability distribution* and receive a sum of money corresponding to the sampled value. The goal of the gambler is to maximize their total winnings by playing the slot machines with the best expected outcomes.

However, the rewards and probability distributions for each slot machine are not known to the gambler ahead of time, so the gambler must choose between *exploring* new options and *exploiting* options that have already been explored. Specifically, the gambler needs to

decide on which slot machines to play, how many times to play each of them, and the order in which they are played. As a further complication (and variation of the MAB problem), the reward distributions for each slot machine may also change over time.

2.2.2 Thompson Sampling

There are many approaches to the MAB problem, and some variations of the problem are known to have *optimal* strategies (i.e., strategies which maximize the expected value of the total long-term rewards). One strategy is the Thompson sampling method [80, 81], where the intuition is to choose between the arms proportionally to the probability that each arm is optimal [75]. This is done by maintaining a continuous probability distribution between 0 and 1 for each arm, where 1 indicates that the arm is optimal, and 0 indicates that the arm is not optimal. Specifically, the Thompson sampling method works by randomly sampling values from *beta distributions* (discussed below), choosing the arm with the largest sampled value, and updating the probability distribution for the chosen arm based on the rewards.

Beta distribution Beta distributions constitute a family of continuous probability distributions between 0 and 1, parameterized by two "shape parameters" α and β, which can take any positive real value. The probability density function of the beta distribution is given by the following equation, where $\Gamma(z)$ is the gamma function [46]:

$$f(x;\alpha,\beta) := \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \cdot x^{\alpha-1}(1-x)^{\beta-1} \tag{2.2}$$

Note that the shape of this function is given by $x^{\alpha-1}(1-x)^{\beta-1}$, and the coefficient involving the gamma function is just a constant normalization factor.

One property of this distribution which is crucial for Thompson sampling is that if the *prior* probability distribution is a beta distribution, then the *posterior* distribution after making additional observations from new data is also a beta distribution, just with different parameter values. This means that updating the priority distribution for Thompson sampling is very convenient – it can be done simply by updating the parameter values. In particular, one way of interpreting the shape parameters is to let α denote the total number of "successes", and to let β denote the total number of "failures".

Some other important properties of this distribution which are relevant to this thesis are that the mean of the distribution is given by $\frac{\alpha}{\alpha+\beta}$ [46], and the variance of the distribution is given by $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$ [46]. Observe that as the number of successes increases, the mean

of the distribution moves toward 1, and as the number of failures increases, the mean of the distribution moves toward 0. Furthermore, as the number of trials increases, the variance in the probability distribution decreases, approaching 0 as the number of trials grows large.

2.2.3 Exponential Moving Average

Exponential moving averages (EMA) are a type of weighted moving average in which the weights decrease exponentially. In this thesis, we only consider EMA in the context of time-series data, where EMA is used to place greater emphasis on recent data. The intuition for using EMA is that old data might be misleading when considering systems that change over time, so the influence of each data point in our analysis should decrease as the data grows older. Consider a list of data: $d_0, d_1, ..., d_n$, where d_t is the datum for time-step t. There are two different formulations of EMA, which can be simply stated in terms of recurrences. Using a decay value r, these recurrences are as follows:

$$\begin{aligned} \overline{EMA}_0 &= d_0 \\ \overline{EMA}_{t+1} &= r \cdot \overline{EMA}_t + (1-r) \cdot d_{t+1} \end{aligned} \tag{2.3}$$

$$\begin{aligned} EMA_0 &= d_0 \\ EMA_{t+1} &= r \cdot EMA_{t+1} + d_{t+1} \end{aligned} \tag{2.4}$$

In the special case where every d_i takes on an equal value d, the final values of $\overline{EMA}_n$ and EMA_n can each be computed using the equation for a sum of a geometric series. This yields the following expressions for $\overline{EMA}_n$ and EMA_n:

$$\begin{aligned} EMA_n &= \frac{d \cdot (1-r^n)}{1-r} \\ \overline{EMA}_n &= d \cdot (1 + r^n - r^{n-1}) \end{aligned} \tag{2.5}$$

Then, considering the largest value of d_i and taking the limit of these expressions as n tends to infinity yields upper bounds on the maximum possible values of the exponential moving averages, denoted here as $\overline{EMA}_{sup}$ and EMA_{sup}.

$$\begin{aligned} EMA_{sup} &= \frac{d}{1-r} \\ \overline{EMA}_{sup} &= d \end{aligned} \tag{2.6}$$

Chapter 3

Priority-Based Algorithms for Boolean Constraint Propagation

The design and implementation of efficient Boolean Constraint Propagation (BCP) engines is one of the key reasons responsible for the remarkable success of modern Conflict-Driven Clause-Learning (CDCL) SAT solvers in solving practical problem instances obtained from real-world applications. It is well known that a significant portion of a modern SAT solver's time is spent performing BCP – often close to 80% of the total solving time [64] – so it is essential for solvers to perform BCP as efficiently as possible. Consequently, considerable effort has been invested into improving the performance of BCP. Perhaps the most important innovation in this context is the development of the two-watched literal scheme [64], which greatly increased the rate at which variables are propagated without sacrificing the correctness of the propagation algorithm.

Despite the substantial research that has already gone into the development of BCP, there is still a lot of room left to be explored in terms of the design of practical BCP algorithms. Notably, modern SAT solvers improve upon the heuristics used in the scheme and the implementation of the scheme itself, but the underlying algorithm remains the same. We observe that there are two features that all previously proposed BCP algorithms share: namely, they perform assignments *immediately* and propagate variables *in-order*. More precisely, by *immediately*, we mean that variables are assigned as soon as the corresponding unit clause is discovered, and by *in-order*, we mean that variables are propagated in the order in which they are inserted on the assignment trail (viewed as a FIFO queue) by the solver during its execution. We call such algorithms "Immediate BCP". As we show in this work, there are scenarios where such policies may not be the optimal way to propagate variables in a CDCL SAT solver. In certain settings, it may be prudent to delay assigning

a variable or perform propagation based on an order different from the policy favoured by current BCP engines.

Before describing Delayed BCP, we first formally characterize the concept of the *variable assignment lifecycle*, which is essentially a state transition diagram whose states correspond to the stages a variable may be in during a solver's execution, and whose transitions correspond to solver actions (e.g., backjumping). We believe that this explicit characterization enables a clearer understanding – and consequently, exploration – of novel designs of BCP algorithms. The idea that the variables of an input Boolean formula may go through various states during a solver's execution, such as assigned or unassigned, is widely understood by the SAT community. We add to this by identifying a new state, namely, *queued* (i.e., the variable is stored in a queue and is ready to be propagated), that had previously not been explicitly recognized. We also identify the solver actions that may trigger transitions (e.g., backjumping) between the various variable states. We describe these transitions and describe the variable states more precisely in Section 3.1.

A queued variable is one whose truth value has been determined through unit propagation but has not yet been processed by BCP. In particular, it is possible to design a BCP algorithm such that a queued variable does not contribute to setting additional variables' values during BCP. Put differently, depending on the order in which the queued variables may be subsequently assigned or processed, one can get very different kinds of BCP algorithms, such as *delayed* and *out-of-order* as described below.

3.1 Definitions

3.1.1 Variable Assignment Lifecycle

Based on our understanding of Immediate BCP, we can identify at least four major states which variables transition through in a CDCL solver: **dormant**, **queued**, **processing**, and **processed**. Further, we observe that the variables transition between these states whenever the solver executes a major action such as backjumping or BCP. The resultant transition diagrams, or *variable assignment lifecycles*, are helpful in understanding how the Immediate BCP and Delayed BCP algorithms differ from each other. Below, we provide more formal definitions for the four major states that variables can be in.

Definition 3.1. Queued: A variable x_i is **queued** if x_i is stored in the propagation queue. Depending on the BCP algorithm, queued variables may be either assigned or unassigned; i.e., it is possible to have queued variables which do not assist in unit propagation.

The propagation queue may be either a FIFO or priority queue. Note that the propagation queue holding queued variables does not have to be an explicit data structure when implemented in practice – in particular, FIFO propagation queues are commonly stored implicitly as part of the assignment trail.

Definition 3.2. Processing: A variable x_i is **processing** if x_i is currently being unit propagated. When x_i is processing, there may be literals that remain to be queued or assigned by unit propagation from x_i.

Definition 3.3. Processed: A variable x_i is **processed** iff x_i has been fully unit propagated. This means that either

(a) BCP has discovered a falsified clause containing x_i, or
(b) All the literals unit propagated by x_i are either queued or assigned.

Definition 3.4. Dormant: A variable x_i is **dormant** if it does not fall in any of the above states. In particular, a dormant variable x_i does not have any truth value at all, and it is not stored in the propagation queue.

3.1.2 Priority, Propagation, and Assignment Orders

We formally define various terms such as **immediate**, **delayed**, **in-order**, and **out-of-order** to more precisely characterize the behaviour of BCP and explain our proposed methods. To distinguish between the internal state of a SAT solver and the intuitive idea of prioritization, we also define three different orderings of the variables: **priority order**, **propagation order**, and **assignment order**.

Definition 3.5. Immediate and **delayed** variable assignment: We say that a BCP algorithm performs **immediate** variable assignment during its execution if it assigns a variable immediately after discovering a corresponding unit clause[1]. Otherwise, we say that the BCP algorithm performs **delayed** variable assignment.

Definition 3.6. In-order and **out-of-order** variable propagation: We say that a BCP algorithm performs **in-order** variable propagation if it processes and propagates variables in the same order that variables appear on the assignment trail. Otherwise, we say that the BCP algorithm performs **out-of-order** variable propagation.

[1]This is sometimes called ***greedy*** variable assignment in informal contexts.

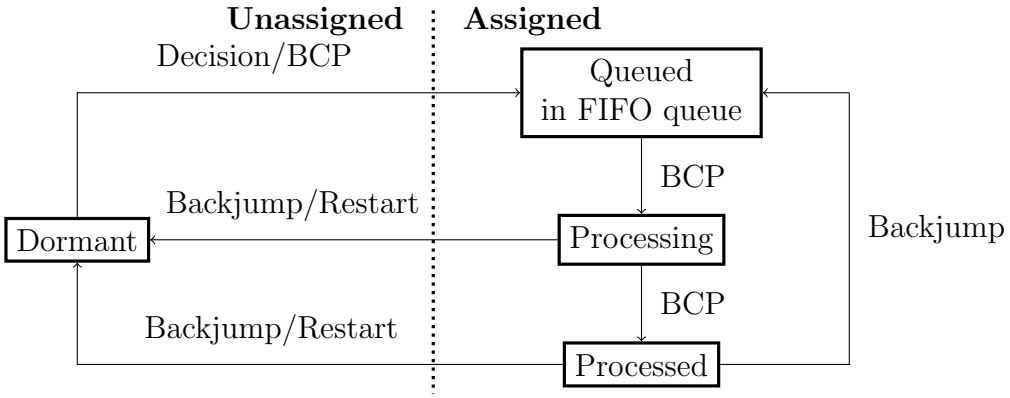


Figure 3.1: The traditional variable assignment lifecycle under Immediate BCP, showing how variables move between the different states as they are assigned and unassigned during a solver's search. A variable x transitions from "processing" to "dormant" if a conflict is discovered while propagating x or $\overline{x}$. A variable y transitions from "processed" to "dormant" if it is unset during a restart or during a backjump after a conflict. A variable z transitions from "processed" to "queued in FIFO queue" if z or $\overline{z}$ is the asserted literal in a conflict.

Definition 3.7. Variable priority order: For a given call to BCP, the **variable priority order** is the desired ordering for propagating variables. This ordering can be either static or dynamic, defined explicitly by some heuristic (e.g., orders defined by variable selection heuristics such as VSIDS or LRB), or implicitly by the structure of the BCP algorithm.

Definition 3.8. Variable propagation order: For a given branch in a SAT solver's recursive search tree, the **variable propagation order** is the order in which variables are processed by the BCP algorithm. This corresponds to the order in which variables are popped off the propagation queue.

Definition 3.9. Variable assignment order: For a given branch in a SAT solver's recursive search tree, the **variable assignment order** is the order in which variables are assigned, either by BCP or the variable selection heuristics. (It turns out that for all BCP algorithms we consider, the order in which variables are inserted into the assignment trail exactly corresponds to the variable assignment order.)

We note that there is a subtle difference between the propagation order and the priority order. While it is true that the priority order affects the propagation order, setting a priority order does not completely determine the propagation order. This is because the

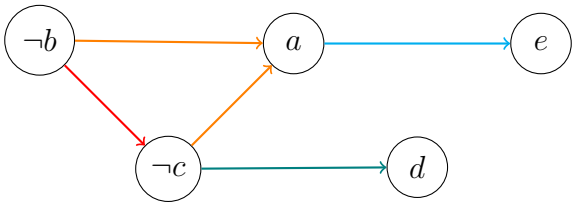


Figure 3.2: The implication graph for the formula $F = (b \vee \neg c) \wedge (a \vee b \vee c) \wedge (c \vee d) \wedge (\neg a \vee e)$ under the initial partial assignment $\alpha = \{\neg b\}$.

propagation order depends on the literals that are currently available to be propagated. For example, in Figure 3.2, if we choose the priority order (e, a, b, c, d) and use immediate variable assignment, the first variable to be propagated must be b regardless of the priority order, since it is initially the only assigned variable. Since c and a are assigned as a result of propagating b, BCP can choose to propagate either a or c. Since a occurs before c in the chosen priority order, a is propagated next, assigning e. Similarly, e is propagated before c, which is propagated before d. Then the propagation order for this example is (b, a, e, c, d). If we had used delayed variable assignment instead, the propagation order would be (b, c, a, e, d). Observe that both these propagation orders are different from the chosen priority order.

Observe also that after propagating the variables b and c from the clauses in the shown formula, the solver has two choices: it can either check for variables implied by a, or check for variables implied by d. The existence of this choice presents us with an opportunity to design novel BCP algorithms, and is the focus of this chapter.

3.2 New BCP Algorithms

In this section, we describe the Delayed BCP algorithm in detail, starting by motivating the idea of prioritization. We note that that one could envision a variety of BCP methods by using different priority orders, or by further modifying the variable lifecycle diagram.

3.2.1 Motivation for Priority Orders in BCP

To motivate our technique, we examine the role of BCP in CDCL SAT solvers. There are at least two purposes that BCP serves: first, it seeks to discover conflicts via unit propagation, and second, it constructs the conflict graph to aid clause learning. For example, consider

the Immediate BCP method, which immediately assigns variable values upon discovering corresponding unit clauses (instead of merely queuing values for assignment at a future point in time). It goes without saying that this greedy assignment of values to variables may not always be the fastest way of discovering conflicts or necessarily result in "optimal" ones. Having said that, this approach does work well in many settings, and in the absence of any other heuristic, it has remained the mainstay in BCP design.

However, irrespective of how one defines clause quality, it is fair to say that the way Immediate BCP constructs conflict graphs may not always result in learning optimal clauses. Therefore, it behooves us to explore different ways of constructing conflict graphs than the default one used by modern CDCL solvers.

One way to change the way conflict graphs are constructed by solvers is to change the order in which they assign and propagate variables, relative to the current Immediate BCP approach. A very natural order to consider here is the priority order as defined by branching heuristics such as VSIDS [64], LRB [59], CHB [56], or VMTF [72]. We already have considerable empirical evidence that branching in the order defined by these branching heuristics works well for many practical applications. Hence, it is natural to consider using them for priority BCP as well.

The design of our Delayed BCP algorithm takes advantage of existing priority orders as defined by these branching heuristics, and our empirical results bear out the hypothesis that priority orders can be more effective relative to the default propagation order used by the Immediate BCP method. Further, our algorithms are designed to take advantage of any priority order that a solver developer or user may choose to define.

3.2.2 Design Considerations

A subtle factor which one needs to consider when designing priority-based BCP algorithms is that simply selecting a variable priority order does not fully prescribe a variable assignment order or a variable propagation order. The clause learning algorithm used by modern CDCL SAT solvers imposes an invariant on the assignment trail: namely, variables must appear on the assignment trail in topological order; i.e., if a variable x implies y, x must appear before y on the assignment trail. BCP determines the order in which variables are assigned, and thus, the order in which they appear on the assignment trail. Modifying the BCP algorithm to perform variable propagation according to a given priority order, while respecting this assignment order invariant, presents us with at least two choices, each of which results in a different priority-based BCP algorithm. The first choice is to propagate variables in an order different from the assignment order, and the second choice is to assign

variables in an order different from the order in which the corresponding unit clauses are discovered by BCP.

We further note that our proposed algorithms can be used in conjunction with any priority order; i.e., the desired priority ordering of the variables can be selected independently of the BCP algorithm. Thus, the concept of priority-based BCP introduces two different design spaces: the selection of the priority order, and the architecture of the BCP algorithm itself. In this work, we choose to focus on the design on the BCP algorithm, while leveraging the priority orders as defined by branching heuristics such as VSIDS [64] or LRB [59].

3.2.3 Delayed BCP

One way of modifying the Immediate BCP algorithm to respect a variable priority order is to delay the assignment of a variable until the point at which BCP processes that variable (delayed variable assignment), but propagate variables in the order that they are assigned (in-order variable propagation). This choice delays the detection of conflicts that would otherwise have been found under immediate variable assignment, but ensures that the propagation order is still a topological order with respect to the implication graph. It is only after a variable has been assigned a value that it can contribute to unit propagation.

Therefore, this choice means that detected conflicts are caused by variables at the conflicting decision level that appear earlier in the variable priority order, since only assigned variables can participate in conflicts. We note that variables set at earlier decision levels may participate in conflicts regardless of their priority.

We refer to the algorithm resulting from these choices as **Delayed BCP**, which we present as Algorithm 3. This algorithm replaces the traditional FIFO propagation queue with a priority queue, and decision and asserted literals are implicitly given maximal priority. Where this algorithm differs significantly from Immediate BCP is in how literals are assigned. This difference can be understood from Delayed BCP's variable lifecycle diagram (Figure 3.3), where the "queued" state has been moved to the "unassigned" category as compared with Immediate BCP's lifecycle diagram. Since variables are queued but not assigned, this means that they do not assist in unit propagations until they are popped off the propagation queue and assigned. This means that Delayed BCP requires more computational effort to detect conflicts than Immediate BCP.

Delayed BCP introduces an additional complication as compared with Immediate BCP: since variable assignment is postponed, Delayed BCP must check the propagation queue when searching for conflicts and before adding a variable to the queue. In practice, instead

Algorithm 3: Delayed BCP

Input: Propagation queue (priority queue) pq.
Output: A falsified clause if one exists.

```
1  while pq.size() > 0 do
2      p ← pq.pop();
3      assign(p);
4      foreach clause c rendered unit by p, resulting in unit literal l do
5          if c is falsified then
6              pq.clear();
7              return c;
8          else if pq.contains(−l) then
9              assign(−l);
10             pq.clear();
11             return c;
12         else if not pq.contains(l) then
13             reason[var(l)] ← c;
14             pq.push(l);
15         end
16     end
17 end
18 return no conflict;
```

of performing a search on the queue, it is more effective to keep a separate data structure to check (in constant time) whether a variable has been queued, and the value with which it was queued. Additionally, when a conflict occurs as a result of a variable in the propagation queue, the conflicting literal must be assigned and placed on the assignment trail.

We summarize this algorithm with the following invariant, where v_1 and v_2 are variables:

Invariant 3.10. if v_1 occurs earlier in the priority order than v_2, and both v_1 and v_2 are available for assignment, it must be true that v_1 is assigned and processed before v_2 is assigned.

3.2.4 Other Possible BCP Variants

Another choice for modifying the traditional BCP algorithm to respect a variable priority order is to process variables in an order different than they are assigned (out-of-order

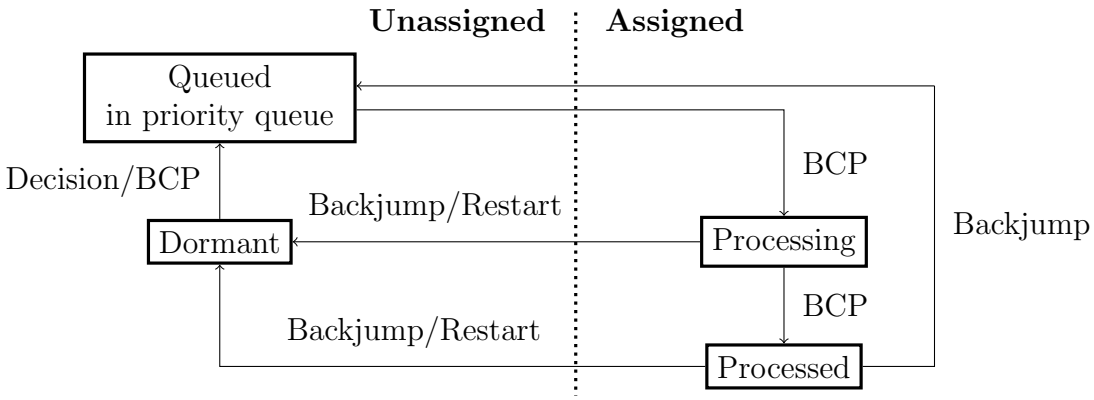


Figure 3.3: The lifecycle of a variable under **Delayed BCP**. The only conceptual differences from the variable lifecycle under Immediate BCP are that the "queued" state has moved from being assigned to being unassigned, and that the propagation queue is now a priority queue.

variable propagation), while assigning variables immediately once suitable unit clauses have been discovered (greedy variable assignment). This has the benefit that conflicts can be detected as soon as possible, but the detected conflicts might involve variables that occur very late in the priority order. Essentially, the only difference between Immediate BCP and Out-of-Order BCP is that the latter uses a priority queue instead of the traditional FIFO queue. This enables Out-of-Order BCP to propagate variables with respect to a priority order, whereas Immediate BCP cannot. One subtlety to consider here is that decision literals and asserted literals should always be assigned and propagated before any of the literals they imply, and thus they should always be considered to have maximal priority.

We summarize this algorithm with the following invariant, where v_1 and v_2 are variables:

Invariant 3.11. if v_1 occurs earlier in the priority order than v_2, and both v_1 and v_2 are available for propagation, it must be true that v_1 is processed before v_2 is processed.

Although we have characterized the Out-of-Order BCP algorithm, we do not explore it in detail in our empirical work because preliminary experiments using an implementation on the `MapleLCM` [60] solver indicated a severe degradation of solver performance across all considered benchmarks. Moreover, out-of-order propagation vastly complicates the implementation of the technique in the `CaDiCaL` [10] solver, which relies on in-order propagation as an invariant for its specialized conflict analysis procedure.

One could also envision a variant of BCP with a further modification of the variable lifecycle diagram to add a transition from "Processing" to "Queued", where the BCP

algorithm switches between propagating different variables, and variables can go back and forth between "Processing" and "Queued" multiple times before all their implications have been propagated. However, we do not explore this idea at all due to its conceptual complexity – it is not clear how this idea would be beneficial, and there is no obvious mechanism for deciding when to partially process a variable and when to fully process it.

3.3 Priority-Based BCP with RL

In our empirical work, we observe that the various algorithms for BCP do not perform equally well on all instance classes. In particular, Immediate BCP appears to perform the best on SAT competition instances, whereas Delayed BCP appears to perform the best on SATcoin instances. This motivates a natural question: can we switch between the different BCP variants to improve the performance of the solver in general? We approach this problem using Reinforcement Learning.

We would like to learn a policy for choosing when to choose Immediate BCP over Delayed BCP, or vice versa. Ideally, this would be a function which takes the current state of the solver as input, and outputs the best version of BCP to use for the next call to unit propagation (i.e., the version of BCP which results in the highest-quality learnt clauses). Unfortunately, learning and using this ideal function is not practical: considering the entire state of the solver is unfeasible in practice, and it is difficult to separate the effects of different versions of BCP when the current state of the implication graph is affected by all the versions of BCP used previously. Therefore, we consider a significantly simplified approximation of the ideal function: whenever the solver restarts (i.e., when the decision level is 0), taking the recent performance of each version of BCP as input, output the version of BCP which will result in the highest-quality learnt clauses until the next time the solver restarts.

Assuming that SAT solvers perform local search (which is encouraged by branching heuristics like VSIDS [64]), we know that recent solver behaviour is a good predictor for future solver behaviour. Therefore, we can predict the best version of BCP to use by switching between the different BCP algorithms and keeping track of solver performance using each algorithm. There are multiple ways to measure solver performance, but we consider two heuristics, both of which resulted in approximately the same performance in our preliminary testing. The first heuristic is to use the average LBD [4] of learnt clauses during each period between restarts as a proxy for measuring solver performance. LBD was chosen here because it is often used to measure the quality of learnt clauses in the SAT literature. The second heuristic we tested is the average number of propagations resulting

Algorithm 4: Decayed Thompson Sampling Algorithm

Input: The BCP variant $action_{prev}$ used in the previous period, and a Boolean value *success* indicating whether $action_{prev}$ resulted in a success.

Output: The variant of BCP $action_{next}$ to use in the next period.

```
// Decay all shape parameters
for each action_i ∈ actions do
    alphas[action_i] ← alphas[action_i] × decay;
    betas[action_i] ← betas[action_i] × decay;
end
// Update the beta distribution of the previous action
if success then alphas[action_prev] ← alphas[action_prev] + 1 ;
else betas[action_prev] ← betas[action_prev] + 1 ;
// Select the action with the largest sampled value
sample_next ← 0;
action_next ← null;
for each action_i ∈ actions do
    sample_i ← samplebetadistribution(1 + alphas[action_i], 1 + betas[action_i]);
    if sample_next ≤ sample_i then
        sample_next ← sample_i;
        action_next ← action_i;
    end
end
return action_next
```

from each decision. The intuition for this metric is that learnt clauses are useful if they constrain the search space (i.e., by forcing additional propagations). For brevity, we choose to use LBD for all the results presented in this thesis.[2]

This selection problem is an optimization problem – we want to find a sequence of BCP variants which maximizes the performance of the solver as measured by some proxy metric M. We model this optimization problem as an instance of the Multi-Armed Bandit problem with two arms, where the decision agent chooses between two actions, each of which corresponds to a choice of BCP algorithm. We approach this problem using the Thompson sampling method [80, 81] to balance the exploration of the different options with the exploitation of options which are known to perform well. Upon each solver

[2]Propagations per decision (PPD) is still a good metric: a variant of our solver using this idea, named `MapleCaDiCaL_PPD_500_500`, achieved third place in the 2023 SAT Competition [23].

restart, we use the Thompson sampling method to select the variant of BCP to use until the next solver restart. During this period, we measure the average value of the metric M, and at the end of the period, we compare it with the average historical value of M, which is represented using a exponential moving average (Equation 2.3). If the most recent average value of M is larger than the historical value of M, we say that the most recent decision was a success, and give the decision agent a reward for that decision by incrementing the α parameter for the corresponding beta distribution. Otherwise, the most recent decision was a failure, and we increment the β parameter of the beta distribution instead.

Finally, we modify the Thompson sampling method by applying a multiplicative decay to the α and β shape parameters for each beta distribution after each restart (Equation 2.4). This modification is made out of the concern that over a long solver run, the decision agent might settle on a single option and never explore other options, which is problematic because different variants of BCP could be useful at different stages of the search, especially as the search space evolves as a result of clause learning. Without this decay, the shape parameters could grow without bound, and the decision agent could eventually get stuck picking a single option. The inclusion of the decay means that the shape parameters are bounded, which means that there is always a chance for the decision agent to explore other options. We present the modified Thompson sampling algorithm as Algorithm 4.

Consider the scenario where the solver always chooses the same variant of BCP, and where every decision is a success. The value of α resulting in the limit of this process is clearly an upper bound on all the possible values of α. Similarly, this gives an upper bound on the value of β when we assume that every decision is a failure. Let $0 < r < 1$ be the decay value for the shape parameters. By Equation 2.6, the upper bound is $\frac{1}{1-r}$, which is a constant. Therefore, the shape parameters α and β are bounded between 0 and $\frac{1}{1-r}$.

3.4 Empirical Evaluation

In this section, we show that Delayed BCP performs better than Immediate BCP on the `SATcoin` benchmark, and that the propagations performed by Delayed BCP are of "higher quality" than the propagations performed by Immediate BCP. We further demonstrate that our chosen priority order performs better than chance, and that our reinforcement learning method is able to learn a policy for switching between the BCP variants which performs better than chance.

To evaluate the efficacy of Delayed BCP as compared with Immediate BCP, we conduct a large number of experiments and compare the computation times required to solve a

Solver	Description
`CaDiCaL`	Unmodified `CaDiCaL` solver
`CaDiCaL_dbcp_branch_order`	Delayed BCP using the branching priority order
`CaDiCaL_dbcp_rnd_order`	Delayed BCP using a random priority order
`CaDiCaL_dbcp_rl_switch`	RL switching between BCP variants
`CaDiCaL_dbcp_rnd_switch`	Random switching between BCP variants

Table 3.1: Summary of solvers used for the empirical evaluation of Delayed BCP.

broad variety of benchmark instances. Since the focus of our method alters the behaviour of BCP, we also measure the number of propagations per second to estimate the overhead associated with maintaining the additional data structures of each technique.

3.4.1 Experimental Setup

Solvers To determine whether changes in solver performance due to our methods generalize across different CDCL SAT solvers, we chose to implement our methods on three different solvers: `MapleSAT` [55], `MapleLCM` [60], and `CaDiCaL` [10]. `CaDiCaL` is currently one of the best SAT solvers across all classes of problem benchmarks, so improvement against this solver represents an improvement in the state of the art. Although the `MapleSAT` and `MapleLCM` solvers are no longer considered state-of-the-art, we include them in our comparison because their source code differs significantly from `CaDiCaL`. In our experimentation, we observed similar changes in performance across all solvers with and without our proposed priority BCP methods, which gives us greater confidence in the robustness of our technique. However, for clarity, we only present the results for the `CaDiCaL`-based solvers in the plots and tables in the main body of the thesis. In our experiments, we implement several variations of our Delayed BCP method[3]. We present a summary of our solvers in Table 3.1. We also include the Virtual Best Solver (VBS) in our comparison as a reference for the best possible improvement using our techniques.

As discussed in Section 3.2.2, we choose to use the solvers' existing branching heuristics to define the variable priority ordering during BCP. This means that `CaDiCaL` uses eVSIDS [8] whereas `MapleSAT` and `MapleLCM` use VSIDS [64], CHB [56], and LRB [59]. We considered every combination of the `MapleSAT` and `MapleLCM` solvers with the VSIDS, CHB, and LRB variable prioritization heuristics, with each of the different variants of BCP (Immediate BCP and Delayed BCP). We note that the `MapleLCM` solver dynamically

[3]Source code for the solvers is available at https://github.com/chjon/cadical-priority-bcp

Benchmark	# of Instances	Source / Description
SATcoin	236	Restricted-nonce bitcoin-mining [39, 40]
UNSATcoin	300	Unsatisfiable variants of SATcoin [39, 40]
SAT2022	400	SAT Competition 2022 (Main Track) [22]
SAT2021	400	SAT Competition 2021 (Main Track) [20]
SAT2020	400	SAT Competition 2020 (Main Track) [19]
SAT2019	400	SAT Competition 2019 (Main Track) [24]
Total	2136	

Table 3.2: Summary of benchmark instances used for the empirical evaluation of Delayed BCP. For brevity in our other tables and cactus plots, the SAT competition instances are combined into a single SATcomp benchmark containing 1600 instances.

switches between its branching heuristics – we implemented our method such that the priority order used by BCP also switches appropriately. We also experimented with setting MapleLCM to use a fixed branching heuristic, which produced analogous results.

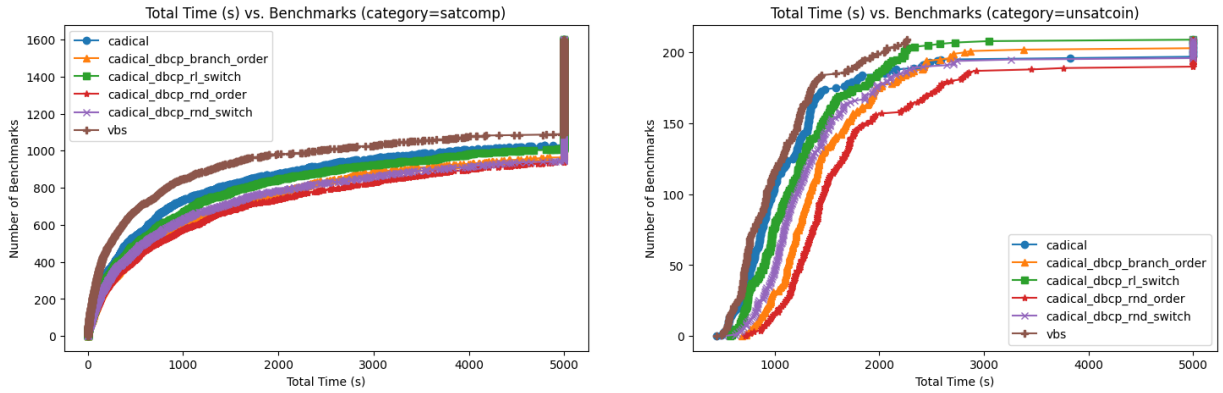
Benchmarks To evaluate the efficacy of our approach, we considered six different classes of instances, for a total of 2136 considered benchmark instances. Our testing primarily focused on cryptographic applications of SAT and instances from previous years of the SAT competition (2019-2022), including both satisfiable and unsatisfiable instances. A summary of these benchmarks is presented in Table 3.2.

3.4.2 Experiments and Results

To isolate the change in performance due to each component of our method, we perform a series of experiments, incrementally augmenting the solver with our proposed changes. The overall performance results of all our experiments are presented in the cactus plots in Figure 3.4. The PAR-2 scores for each solver are summarized in Table 3.3.

Experiment 1: Priority BCP vs. Traditional BCP

We begin by comparing the performance of the base solver using the traditional Immediate BCP method (CaDiCaL) with the same solver using the Delayed BCP method instead (CaDiCaL_dbcp_branch_order). For both CaDiCaL and MapleLCM, we observe mixed results



(a) `SATcomp` (b) `UNSATcoin`

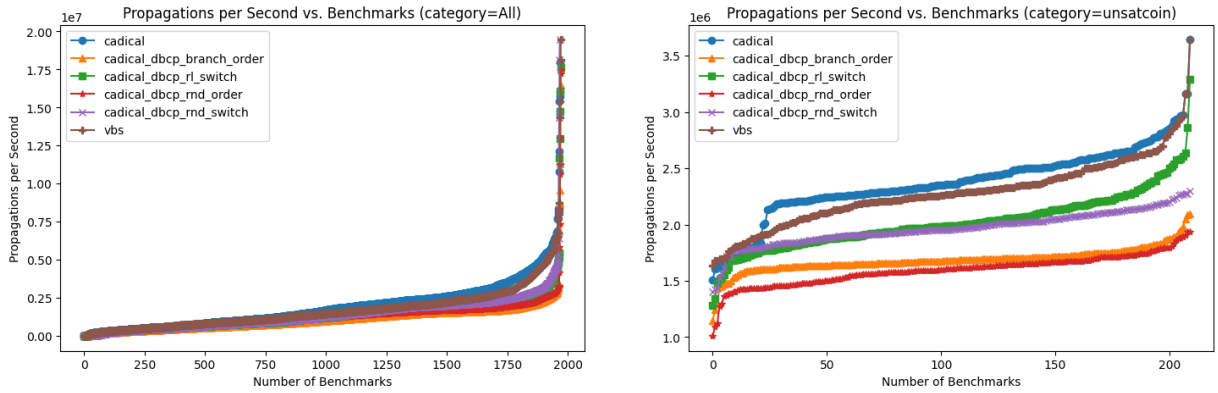
Figure 3.4: Cactus plots comparing the performance of our method with the baseline method using the `CaDiCaL` solver on the `SATcomp` and `UNSATcoin` benchmarks.

Solver	`SATcomp`	`SATcoin`	`UNSATcoin`
`VBS`	3661.15	438.16	1045.35
`CaDiCaL`	4131.66	753.87	1642.17
`CaDiCaL_dbcp_branch_order`	4605.08	882.11	1741.59
`CaDiCaL_dbcp_rnd_order`	4803.99	1356.38	2387.13
`CaDiCaL_dbcp_rl_switch`	4306.26	634.07	1299.58
`CaDiCaL_dbcp_rnd_switch`	4672.13	1464.11	1888.08

Table 3.3: PAR-2 scores summarizing the performance of each solver for each benchmark.

when using the Delayed BCP method. From Table 3.3, we observe that in terms of PAR-2 scores, `CaDiCaL_dbcp_branch_order` performs slightly worse than `CaDiCaL` on all the considered benchmarks. However, this solver also solves more instances on the `UNSATcoin` (see Figure 3.4b) and `SATcoin` instances (not shown). Therefore, we conclude that overall, the Delayed BCP technique is still competitive with the traditional method.

The competitive performance of the `CaDiCaL_dbcp_branch_order` solver is especially notable when considering the propagation rates of each solver: examining the total number of propagations and the propagation rate for each solver reveals that our proposed methods propagate variables at a lower rate than the traditional technique, resulting in fewer total propagations overall (see Figure 3.5). The decreased propagation rate can be attributed to the computation time associated with maintaining priority queues, which are computationally more expensive to use than FIFO queues. However, despite the lower propagation



(a) `SATcomp` (b) `UNSATcoin`

Figure 3.5: Plots comparing the propagation rate of our method with the baseline method using the `CaDiCaL` solver on the `SATcomp` and `UNSATcoin` benchmarks.

rate, our solvers still perform competitively with the baseline, which suggests that the Delayed BCP technique performs "smarter" propagations than the traditional method. In particular, we observe that the VBS has a slightly lower propagation rate than the base solver, indicating that greedily maximizing the propagation rate without consideration for the assignment and propagation orders is not an optimal policy when considering overall solver behaviour.

Experiment 2: Random Priority Order vs. Branching Heuristic Order

The results of Experiment 1 seem to agree with our intuition that the assignment and propagation orders are important, and that there is still room for improvement in the BCP algorithm. In Section 3.2.1, we proposed using the ordering defined by the branching heuristic as the priority ordering for BCP. This is the ordering which was used in Experiment 1. To evaluate the importance of the chosen priority order, we compare `CaDiCaL_dbcp_branch_order` with a randomized priority order, which we implement as `CaDiCaL_dbcp_rnd_order`. Since the ordering defined by the branching heuristic changes over time, we also dynamically modify the randomized priority order for a fairer comparison. Rather than modifying the priority order after every conflict like what is done for the order based on the branching heuristics, we chose to shuffle the priority order only upon restarts to reduce the effect of the overhead associated with shuffling the priority order.

From this experiment, we observe that the Delayed BCP solver using the branching

heuristic for the priority order significantly outperforms the Delayed BCP solver with the random priority order (see Figure 3.4). Again, this affirms that the priority order is important, and it agrees with our expectation that a "smart" version of BCP should perform better than random chance. Therefore, for our remaining experiments, we choose to use each solver's branching heuristic to define the priority order.

Experiment 3: Random Switching between BCP Variants vs. RL

From Experiment 1, we know that Immediate BCP performs better on some benchmarks, and Delayed BCP performs better on others. Extrapolating this result, we conclude that for each call to BCP during a solver's search, one of the BCP variants must lead to better overall solver performance. Therefore, intelligently choosing which version of BCP to use at each point in a solver's search should result in better overall performance. As described in Section 3.3, we choose to approach this problem using reinforcement learning, which we implement in the `CaDiCaL_dbcp_rl_switch` solver.

As a control, we implement the `CaDiCaL_dbcp_rnd_switch` solver which switches randomly between the different BCP variants with uniform probability upon each restart. This allows us to determine whether the RL method performs better than chance. As seen in Figure 3.4, the `CaDiCaL_dbcp_rl_switch` solver performs significantly better than `CaDiCaL_dbcp_rnd_switch`. Disappointingly, the `CaDiCaL_dbcp_rl_switch` solver only performs slightly better than the `CaDiCaL_dbcp_branch_order` solver (which doesn't use RL) on the `SATcomp` benchmarks, and overall, is still slightly worse than the unmodified base solver, `CaDiCaL`. However, on the `SATcoin` and `UNSATcoin` instances, the RL method performs much better than both the solvers which only use a single version of BCP. This indicates that the RL agent is indeed capable of learning a policy which allows it to choose the best variant of BCP for the problem.

3.5 Related Work

Ever since the idea of BCP was introduced in DPLL [28, 29], most work on BCP in the literature has been on the efficient implementation of the algorithm. The two-watched-literal scheme introduced in `Chaff` was the first major advancement in this area. Prior to the invention of the two-watched-literal scheme, solvers detected unit clauses by maintaining a counter of falsified literals for each clause, and performing propagation once every literal except one became falsified [64]. Maintaining this counter required visiting every clause containing a literal upon making a variable assignment, even when those clauses contained

many unassigned literals. The two-watched literal scheme made it possible to visit significantly fewer clauses by introducing a new data structure – known as a *watcher* – to keep track of only two unassigned literals per clause – known as *watched literals*. When either one of the two watched literals in the watcher becomes falsified, the two-watched-literal scheme attempts to replace the falsified literal with another unassigned literal in the clause. If it fails to do so, the remaining unassigned literal is implied.

Subsequent to `Chaff`, work on BCP moved to the efficient implementation of the two-watched-literal scheme, and to heuristics for selecting watched literals. The `Lingeling` solver [9] implemented special handling of the watchers in binary clauses to avoid some of the overhead associated with the general case – in particular, binary clauses are stored completely in the watches instead of maintaining an additional clause data structure. The `Kissat` solver further improved on the implementation of the watchers and watcher stacks to optimize them for modern hardware by significantly reducing the size of the watcher data structures and carefully managing memory [11].

3.5.1 Selecting Watched Literals

Prior work in the literature also explores the problem of selecting the literals to watch in each clause. From the definition of the two-watched-literal scheme, one can observe that whenever one of the two watched literals in a clause is falsified, a new watcher needs to be computed for that clause. Therefore, if the solver chooses a new watched literal which is quickly falsified during the search, it will have to perform additional computation to choose another watched literal. This additional work could have been avoided by choosing the watched literal to be the literal in the clause which is the last to be falsified. Since it is difficult to know ahead of time which literal will be falsified last, solver developers approach this problem using heuristics to select watched literals. For example, the `Lingeling` solver implements the literal-move-to-front (LMTF) scheme [9], and recent work by Iser and Balyo chooses to select "stable" literals [45].

Since the choice of watched literals affects the order in which clauses are examined during BCP, this problem initially appears to be related to prioritizing literals during propagation. However, further inspection reveals that this connection is nebulous – watcher selection does not directly affect the order in which literals are propagated during BCP, but it can affect the order in which clauses are examined for each watched literal. This, in turn, can ultimately have indirect effects on the topology of the implication graph that is constructed during BCP and on the order in which the implication graph is explored. Since these effects are very indirect, we consider the techniques developed for the problem

of watcher selection to be largely orthogonal to the methods proposed in this thesis. The major consequence of intelligent watcher selection is not prioritization during BCP, but rather, the reduction of the number of times that each clause is examined.

3.5.2 Clause Prioritization

Since binary clauses only contain two literals, they must result in the propagation of some variable if one of the literals in a binary clause is falsified. Hence, to perform propagations as quickly as possible, a common optimization in implementations of BCP is to check the watchers for binary clauses before checking the watchers for other clauses. Although this optimization is not typically framed as prioritization and there is no explicit priority ordering over the literals, the literals in binary clauses are indeed prioritized over the literals in non-binary clauses.

There have been some recent papers further developing this type of prioritization in BCP, such as the "core-first unit propagation" technique proposed by Chen [16], and the prioritized propagation heuristic implemented by `CaDiCaL_PriPro` [5].[4] Rather than prioritizing literals to propagate, they focus instead on prioritizing sets of clauses to check for propagation. For example, in the core-first unit propagation technique, the BCP algorithm examines the watchers for core clauses before checking the watchers for all other clauses. Therefore, unit literals will be discovered and propagated in a different order than the order which would result from all clauses having equal priority during BCP. In cases where a literal is asserted by multiple clauses, this class of prioritization techniques has a direct effect on the topology of the implication graph, which makes these techniques much more closely related to our method, conceptually. However, we note that this class of clause prioritization techniques is still orthogonal to our prioritization method – delaying the assignment of variables or propagating variables out of order with respect to the assignment trail would still result in different BCP algorithms.

[4] We thank the reviewers of our SAT conference paper submission for bringing these techniques to our attention.

Chapter 4

A Framework for Extended Resolution SAT Solvers

All search algorithms can be viewed as algorithms which explore and prune a search tree, systematically learning which portions of the search space do not contain the solution until either a solution is found, or the entirety of the search space has been pruned away and no solutions exist. Modern CDCL SAT solvers perform this learning very explicitly: after encountering a conflicting state at some point along the search, they generate a clause encoding the reason for the conflict in order to avoid exploring that conflicting portion of the search space. To ensure that the learnt clauses generated by a solver do not erroneously affect the algorithm's final output, the SAT community uses proof systems to model the behaviour of their solvers, and requires that each learnt clause can be derived by the proof rules in their corresponding proof systems. This means that a SAT solver's computational power is limited by the strength of its underlying proof system: if small proofs do not exist for a certain problem instance in a given proof system, then any SAT solver based on that proof system must necessarily take a long time to solve that problem instance.

Traditional CDCL SAT solvers derive learnt clauses in a process that be modelled by the Resolution proof system (Res) [6]. Unfortunately, there are many instance classes which are known to require at least *exponential-sized* proofs for Res. One such class of instances is the family of Pigeonhole Principle formulae [27]. Therefore, if we hope to develop SAT solvers which are capable of solving these problems quickly, we must explore techniques with more powerful underlying proof systems. Some approaches to this problem from the literature include symmetry-breaking techniques [63, 73] which exploit some of the structure present in real-world instances and combinatorial problems, and Satisfiability-Driven Clause-Learning (SDCL) SAT solvers [37, 36] which can learn clauses even when

the solver is not in a conflicting part of the search tree. In our approach, we utilize the Extended Resolution (ER) proof system [83], which is very closely related to Res, and which is known to be exponentially stronger [27, 49].

Unfortunately, it is not simple to automate the generation of proofs for strong proof systems. Given a set of axioms and proof rules, there is a combinatorial explosion in the choices of how rules should be applied and the order in which they should be applied, which means that there are an enormous number of different ways to discover a final proof. For this reason, every SAT solver relies on a set of heuristics to restrict the set of possible actions to guide its search for a small proof. In particular, for ER, existing work in the literature only attempted to automate a very restricted version of the proof system. We address this shortcoming by developing a solver framework which is capable of generating ER proofs in much greater generality, and identify design spaces for heuristics to restrict the types of proofs generated rather than starting from a restricted proof system.

In this chapter, we describe our ER SAT solver framework, implement it over an existing SAT solver, and demonstrate that it is capable of implementing existing methods from the literature. We additionally identify a new set of heuristics for our framework and empirically demonstrate that it improves the performance of the solver on a class of problem instances.

4.1 Major Components

The power of the ER proof system comes from the ability to compactly represent facts about the input formula as extension variables, and to efficiently reason over those facts by reasoning over the extension variables. In general, extension variables can be defined over arbitrary Boolean formulae. However, in keeping with the original definition of ER [83], we only consider extension variables x of the form $x \leftrightarrow a \vee b$, where a and b are literals. As noted in Section 2.1.5, this limitation does not significantly affect the power of the ER proof system, and requiring the definitions of extension variables to take this form means that every extension variable can be treated the same way by the solver, which vastly simplifies the development of a general set of techniques for ER-based SAT solvers.

Our framework for ER-based SAT solvers is a modification of the traditional CDCL algorithm, and consists of three additional major components. A flowchart depicting the modified CDCL algorithm is presented as Figure 4.1, and a summary of each of the new components is as follows:

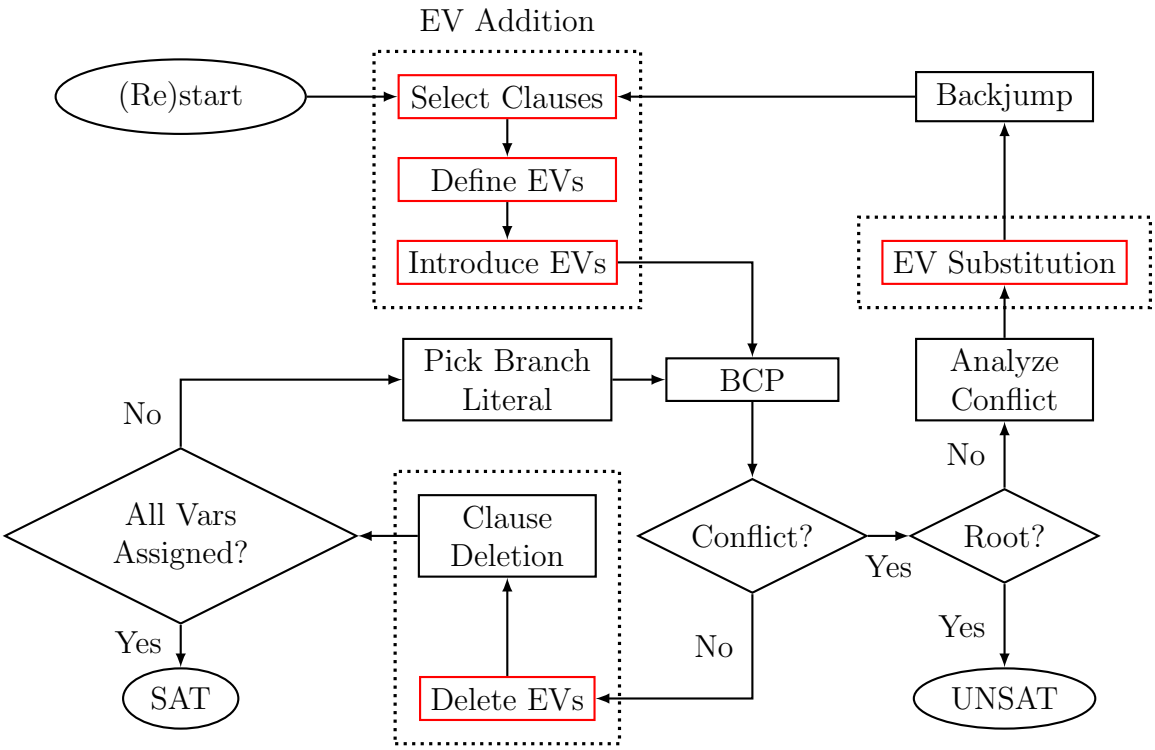


Figure 4.1: A high-level overview of the CDCL algorithm augmented with our ER framework. The components highlighted in red represent new steps introduced by our framework for managing extension variables (EVs). Components in dotted regions are optional steps during the run of the solver, which are triggered by user-defined heuristics.

1. **Extension variable addition**: This component is responsible for determining the conditions for adding extension variables, generating new variable definitions, and adding them to the solver. We observe that the procedure for introducing new variables is essentially composed of the same three steps for all the ER heuristics in the literature: clause filtering/selection (i.e., choosing an "interesting" set of clauses which can be used to define a set of variables), definition generation (i.e., taking a set of clauses and defining new variables with respect to the literals in the clauses), and clause introduction (i.e., adding clauses to the solver to encode the definitions of the new variables). Therefore, we implement our *extension variable addition* component as three subcomponents, each of which is responsible for one of these three steps.

2. **Extension variable substitution**: This component is responsible for decreasing the size of learnt clauses by replacing pairs of literals with their corresponding exten-

sion variables. For example, if the solver learns the clause $(a \vee b \vee c)$ and the extension variable $x \leftrightarrow a \vee b$ has been added to the solver, the learnt clause is changed to $(x \vee c)$. As discovered in the work by Audemard et al. [3], this substitution of literals encourages the solver to learn further clauses involving extension variables. Unfortunately, the implementation of this idea requires a quadratic-time algorithm in the worst case. We introduce a modified algorithm which improves upon the average running time of variable substitution. We note that there are some complications with this basic idea due to interactions with the invariant expected by the two-watched-literal scheme.

3. **Extension variable deletion**: Not every definition results in a useful extension variable, and even if an extension variable is useful, it may only be useful for a small portion of the solver's search. Unnecessarily storing the definitions of unused extension variables wastes memory resources and needlessly increases computational overhead when performing operations like variable substitution. Consequently, despite that the ER proof system does not prescribe variable deletion, the deletion of extension variables must be considered for practical reasons. Since the operation of deleting variables is the same regardless of how one chooses to select variables for deletion, our framework provides a common algorithm for removing extension variables from our data structures, which can then be used whenever other heuristics decide that extension variables should be deleted.

The following sections present data structures and algorithms for the implementation of our framework, including algorithms for efficiently substituting extension variables into learnt clauses and deleting extension variables.

4.1.1 Data Structures

Strictly speaking, to augment a CDCL SAT solver with Extended Resolution, it is sufficient to add a set of clauses encoding the definitions of each extension variable. However, it is helpful to keep track of the definition of each extension variable more explicitly – this enables the implementation of several practical techniques, and facilitates the development of heuristics where this information is required. Therefore, we choose to store the definitions of extension variables redundantly: in addition to a database of definition clauses (implemented as a map from extension variables to a list of definition clauses), we also store the relationship between each extension variable and the pair of literals used to define it.

We propose the `ExtDefMap` data structure for keeping track of extension variable definitions and the dependencies between them. The design of this data structure revolves around efficiently identifying *basis literals* and *basis variables*, which we define as follows:

Algorithm 5: `ExtDefMap` insertion

Input: The extension definition $x \leftrightarrow a \vee b$ to insert into the `ExtDefMap`.

```
1 if not varToPairMap.contains(x) and not pairToVarMap.contains({a, b}) then
2     varToPairMap.insert(x, {a, b})
3     pairToVarMap.insert({a, b}, x)
4     basisCount[a] ← basisCount[a] + 1
5     basisCount[b] ← basisCount[b] + 1
6     nonbasisVars.insert(x)
7     if basisCount[a] + basisCount[¬a] = 1 then nonbasisVars.remove(var(a))
8     if basisCount[b] + basisCount[¬b] = 1 then nonbasisVars.remove(var(b))
9 end
```

Definition 4.1 (Basis literal / basis variable)**.** A **basis literal** is any literal that participates in the definition of an extension variable. Similarly, a **basis variable** is any variable that participates in the definition of an extension variable.

For example, given the extension variable definitions $x \leftrightarrow a \vee \neg b$ and $y \leftrightarrow \neg a \vee c$, there are three basis variables (a, b, and c) and four basis literals (a, $\neg b$, $\neg a$, and c).

The first part of our data structure is quite obvious – we need to know the pair of basis literals corresponding to each extension variable, so we store a map data structure which maps extension variables to corresponding pairs of basis literals. We refer to this map as the `varToPairMap`. To avoid wasted computational resources, it is important not to introduce multiple extension variables with identical definitions. Thus, we also would like to be able to efficiently determine whether there is already an extension variable corresponding to a pair of literals. Unfortunately, the standard map data structure only provides efficient lookups in a single direction, so to implement this, we use a second map data structure which maps pairs of basis literals[1] to their corresponding extension variables. We refer to this second map as the `pairToVarMap`. We note that this implementation duplicates the storage of the extension variables and their basis literal pairs – what we really want is a single *bidirectional map* data structure. However, for clarity of exposition, we use a pair of maps, maintained in parallel.

[1] To avoid issues involving two different representations of the same pair of literals (e.g. $a \vee b$ vs. $b \vee a$), we assume a canonical representation of literal pairs. In our implementation, we use a total ordering over the literals and ensure that literals appear in sorted order when the pair is inserted into the map.

Algorithm 6: `ExtDefMap` deletion

Input: The extension variable x to delete from the `ExtDefMap`.

```
1  if varToPairMap.contains(x) and basisCount[x] = 0 then
2      {a, b} ← varToPairMap[x]
3      varToPairMap.remove(x)
4      pairToVarMap.remove({a, b})
5      basisCount[var(a)] ← basisCount[var(a)] − 1
6      basisCount[var(b)] ← basisCount[var(b)] − 1
7      nonbasisVars.remove(x)
8      if basisCount[a] + basisCount[¬a] = 0 then nonbasisVars.insert(var(a))
9      if basisCount[b] + basisCount[¬b] = 0 then nonbasisVars.insert(var(b))
10 end
```

Observe that new extension variables can be defined over any literals present in the formula, including previously introduced extension variables. Hence, when deleting extension variables, we must be careful not to delete extension variables which are *basis variables*. Consider the operation of checking whether an extension variable x is a basis variable: if no additional information is stored apart from the definitions of extension variables, one would have to iterate over every extension variable and check each definition to see if it contains either x or $\neg x$. This is a linear-time operation in the total number of extension variables, so simply searching for extension variables to delete can become very expensive.

Fortunately, by using the idea of reference counting, this check can be performed in constant time without affecting the asymptotic time complexity of other important operations. For each literal l, we maintain the counter `basisCount`$[l]$ (initialized to zero), which tracks the number of times each literal appears as part of a definition. The `basisCount`$[l]$ counter is incremented whenever a new extension variable whose definition contains l is added, and it is decremented whenever an extension variable whose definition contains l is deleted. Then, whenever the `basisCount`$[l]$ counter is greater than zero, l must be a basis literal. It follows that whenever `basisCount`$[l]+$`basisCount`$[\neg l]$ is greater than zero, `var`(l) must be a basis variable. As a further optional optimization, we can store a set called `nonbasisVars` which contains every extension variable that is not a basis variable. This set can be updated in constant time whenever the state of the corresponding `basisCount` counters become zero or non-zero, which allows us to efficiently access the set of variables which can be deleted.

Then the `ExtDefMap` data structure is comprised of four internal components (three maps and one set): `varToPairMap`, `pairToVarMap`, `basisCount`, and `nonbasisVars`. All

four of these components can be implemented using hash tables for constant-time modification and search. We define two modification operations for the `ExtDefMap` data structure: `insert(`x`, `$\{a, b\}$`)`, which inserts extension variables defined as $x \leftrightarrow a \vee b$, and `remove(`$x$`)`, which removes the extension variable x. We present the algorithms corresponding to `insert` and `remove` as Algorithms 5 and 6, respectively. We can also define some helper functions: `containsVar(`x`)`, which checks whether x is an extension variable; `varToPair(`x`)`, which gets the pair of basis literals corresponding to the extension variable x; `containsPair(`$a \vee b$`)`, which checks whether there is an extension variable corresponding to the pair of basis literals $a \vee b$; `pairToVar(`$a \vee b$`)`, which returns the extension variable corresponding $a \vee b$; and `isBasisLit(`l`)`, which returns true if and only if `basisCount[`l`]` is greater than zero. Note that all these functions are essentially just applications of the traditional map lookup operations, so we do not present them here.

4.1.2 Extension Variable Definition

Recall that the ER proof system consists of only two proof rules which can be sequenced in any order, with the sole restriction being that extension variables can only be used for resolution if they have already been defined. Then, in theory, we can restrict ER by dividing it into two phases: first, define every extension variable required for the proof using only the new variable rule; and second, complete the proof using only the resolution rule. This suggests that we can simply preprocess the formula to add the necessary extension variables, and then pass the preprocessed formula to a traditional CDCL SAT solver. Unfortunately, it is very difficult to determine the "optimal" set of extension variables ahead of time – the search space for a set of extension variables is enormous, and although there has been some work in the literature demonstrating situations where extension variables are useful [15, 76, 74], it is not known what properties extension variables should have to be useful in general.

Consider the problem of selecting an extension variable definition given a formula containing n variables. This is simply the problem of selecting two distinct variables, each of which can occur with one of two polarities, for a total number of $\binom{n}{2} \cdot 2 \cdot 2 = 2n(n-1)$ possibilities. Hence, we see that as additional extension variables are introduced, the number of possibilities for each new definition grows quadratically. As a result, when considering combinations of multiple extension variables, the number of combinations of possible extension variables grows super-exponentially[2].

[2]When adding $n-1$ extension variables to a formula with n variables, there are at least $4^{n-1} \cdot (2n-3)!!$ possible sets of extension variable definitions. For more details, see Appendix A.

Clearly, it is infeasible to exhaustively search for the best set of extension variable definitions. Therefore, to approach the problem of choosing definitions for extension variables, we choose not to define all the extension variables at once, but rather, to define variables intermittently, using heuristics guided by the solver's search. It is obvious that the "optimal" definitions for extension variables depend heavily on the given problem instance, so any practical algorithm for introducing useful extension variables will need to examine the formula to identify some "interesting" part of it, take the "interesting" clauses and define extension variables over some of their literals, and finally, add the new variables and their definitions into the solver. Our framework implements these steps as three subcomponents, which we call clause selection, definition generation, and variable introduction.

Clause Selection The clause selection subcomponent is responsible for identifying parts of the formula where extension variables would be useful. Our framework does not prescribe a method for performing this task; rather, it makes use of a user-defined heuristic function which takes the entire state of the SAT solver as input, and which outputs a subset of the solver's clauses. Since it can be computationally expensive to consider every clause in the formula, we include some practical design choices in the implementation of our framework – we further split clause selection into two smaller steps: first, use an inexpensive approximation to filter out clauses which are not expected to be helpful; and second, use a more refined heuristic to select from the remaining clauses. It is not strictly necessary to perform any approximate clause filtering, but we include this in our framework as an opportunity for optimization.

A general approach to clause selection would examine every clause in the formula each time the solver decides to introduce an extension variable. This is necessary for some clause selection heuristics, but for other heuristics where the set of interesting clauses does not change a lot over time, this may involve unnecessary recomputation. Thus, as a further practical optimization for selecting clauses, our framework supports performing clause selection incrementally for each learnt clause.

Definition Generation This subcomponent takes the set of clauses generated by clause selection and uses them to define a set of extension variables. Similarly to clause selection, definition generation uses a user-defined heuristic function. In this case, the heuristic function is used to choose definitions for extension variables. As an implementation note, rather than requiring extension definitions to be represented by the clauses encoding the equivalence of an extension variable with its definition, we represent new variable definitions as a mapping from extension variables to a `VarDef` data structure. We make this design

choice to move the responsibility of encoding definitions into clauses away from the user heuristic. The task of encoding definitions as clauses is common to all general ER-based techniques, so by making this task part of our framework, we simplify the process of defining extension variables and reduce the possibility of errors due to maintaining duplicated code.

The `VarDef` data structure contains the pair of literals a and b comprising an extension variable $x \leftrightarrow a \vee b$. Additionally, the data structure contains a (potentially empty) set of auxiliary clauses – some of the ER techniques proposed by prior work introduce auxiliary clauses at the same time as new extension variables [13], so we design our framework to support this by treating auxiliary clauses as part of each extension variable's definition.

Variable Introduction The variable introduction subcomponent takes the mapping of extension variables to `VarDef`s generated by the definition generation subcomponent as input and appropriately updates the solver's data structures. In particular, it registers the variable definitions in the solver's `ExtDefMap` and adds the clauses encoding the definition to the database of definition clauses.

The definitions of extension variables are encoded by Boolean formulae which assert that each extension variable should be set to true if and only if its corresponding definition is satisfied. In particular, extension variables of the form $x \leftrightarrow a \vee b$ are encoded by Boolean formulae of the form $x \leftrightarrow a \vee b$, which is represented in CNF by the following three clauses: $(x \vee \neg a) \wedge (x \vee \neg b) \wedge (\neg x \vee a \vee b)$. The task of introducing these clauses into the solver is essentially identical for all possible extension variable definitions, so we treat this as a modular subcomponent which can be implemented independently of the heuristics for variable definition.

4.1.3 Extension Variable Substitution

As Audemard et al. [3] note in their work, substituting extension variables in place of their definitions in newly learnt clauses encourages the participation of extension variables in future conflicts. Similarly to these authors, we only consider newly learnt clauses when performing variable substitution in order to reduce its associated computational overhead.

To this end, we further restrict the types of variable substitution which we perform. Regardless of how extension variables are defined, they can only appear in one of two polarities. Since the design of our framework forces all extension variables to be of the form $x = l_i \vee l_j$, this means we only need to consider two cases. In the positive case, $x \equiv (a \vee b)$, we can scan each learnt clause to identify pairs of literals that define extension

Algorithm 7: Extension Variable Substitution

Input: A clause $C = [l_1, l_2, ..., l_k]$, and `ExtDefMap` $\mathcal{X}$

Output: A clause C' equivalent to C, where literal pairs defining extension variables have been replaced by their corresponding extension variable

```
// Get indices of all basis literals in C
basisLitIdxs ← [];
for i ← 0; i < |C|; i ← i + 1 do
    if X.isBasisLit(l) then basisLitIdxs.append(i) ;
end
// Substitute basis literal pairs with extension variables
substituted ← {};
for i ← 0; i < |basisLitIdxs|; i ← i + 1 do
    p_i ← basisLitIdxs[i];
    if C[p_i] = ⊥ then continue;
    for j ← i + 1; j < |basisLitIdxs|; j ← j + 1 do
        p_j ← basisLitIdxs[j];
        if C[p_j] = ⊥ then continue;
        if X.containsPair(C[p_i] ∨ C[p_j]) then
            x ← X.pairToVar(C[p_i] ∨ C[p_j]);
            C[p_i] ← x;
            C[p_j] ← ⊥;
            substituted ← substituted ∪ {x}
        end
    end
end
// Remove ⊥
C' ← simplify(C);
// Fix implication graph
for each x_i ∈ substituted do
    if x_i is unassigned then
        (a_i, b_i) ← X.varToPair(x_i);
        Propagate x̄_i from the definition clause (x̄_i ∨ a_i ∨ b_i);
    end
end
return C'
```

variables. In the negative case, $\neg x \equiv (\neg a \wedge \neg b)$, we need to replace conjunctions of two clauses of the form $(\neg a \vee c_1 \vee c_2 \vee ... \vee c_k) \wedge (\neg b \vee c_1 \vee c_2 \vee ... \vee c_k)$ with a single clause $(\neg x \vee c_1 \vee c_2 \vee ... \vee c_k)$. For example, given a candidate learnt clause $(a \vee b \vee c \vee d)$, we can instead learn the clause $(x \vee c \vee d)$. In the negative case, $\neg x = (\neg a \wedge \neg b)$, we need to replace conjunctions of two clauses of the form $(\neg a \vee c_1 \vee c_2 \vee ... \vee c_k) \wedge (\neg b \vee c_1 \vee c_2 \vee ... \vee c_k)$ with a single clause $(\neg x \vee c_1 \vee c_2 \vee ... \vee c_k)$. Unfortunately, we found that identifying eligible pairs of clauses for this type of substitution requires high computational overhead, so we only consider substituting literal disjunctions within a single clause.

Since variable substitution requires finding an extension variable corresponding to a pair of literals, a naive approach would simply be to check every pair of literals in a clause for a corresponding extension variable. Since clauses of size k contain $\binom{k}{2}$ possible literal pairs, this approach is a $\Theta(k^2)$-time algorithm. However, from our experimentation, we observe that many literals do not participate in extension variable definitions. This suggests an optimization of the algorithm: if a clause contains k' basis literals, then only considering basis literals reduces the running time of our algorithm to $\Theta(k + k'^2)$ using the `ExtDefMap` data structure. Although the worst case performance of the algorithm is still $\Theta(k^2)$, the modified algorithm is much faster in practice. In their solver, Audemard et al. [3] prefer adding extension variables with higher VSIDS activities when there are multiple options for substitution. This is a heuristic which encourages resolution on extended variables, but does not affect the theoretical properties of the solver [3]. We found this to be computationally expensive, and instead employ a greedy algorithm to replace the first detected literal pairs. Our method for performing variable substitution is presented as Algorithm 7.

Observe that after substituting literals into C to get C', Algorithm 7 amends the implication graph in lines 21 to 26. This is necessary to ensure that C' is an asserting clause. Since C is an asserting clause, every literal in C must be falsified at the point of clause learning. However, BCP exits immediately upon detecting a conflict and does not perform all possible propagations, so C' may contain literals that are unassigned.

Correctness of Substitution Our implementation of the extended resolution solver framework performs extension variable substitution after learning a clause, but before performing backtracking. This raises the concern of whether the solver behaviour remains correct in the event that the asserting literal is substituted out of the clause. The original learnt clause must have been an asserting clause, but it is unclear whether the clause is still asserting after extension variable substitution. This concern consists of two questions: whether an asserting literal always exists after extension variable substitution, and whether the computed backtrack level for the original learnt clause remains valid for the substituted clause if the clause is asserting.

Consider the extension variable $x \leftrightarrow a \vee b$ and an asserting clause C containing the literals a and b. Let C' be the clause resulting from C after extension variable substitution. By the definition of extension variable substitution, C' contains x, and does not contain either a or b. To address the questions above, we prove the following claims:

Lemma 4.2. Every literal in C' is falsified.

Proof. Since C is an asserting clause, every literal in C is falsified. Therefore, every literal in $C \cap C'$ is falsified. Then we only need to consider literals in $C' \setminus C$, which corresponds exactly to the set of newly-substituted extension variables. Let x be any literal in $C' \setminus C$. By the definition of the substitution algorithm, this must be an extension variable with definition $x \leftrightarrow a \vee b$ for some pair of literals $a, b \in C$. Since a and b are both in C, they must be falsified. Then the definition of x is falsified, so it is possible to propagate it and assign x to false. Therefore, after the substitution algorithm amends the implication graph, x must be falsified. Since x is an arbitrary literal in $C' \setminus C$, every literal in $C' \setminus C$ must be falsified. Therefore, every literal in C' is falsified. □

Theorem 4.3. Let C be an asserting clause, and let C' be the clause resulting from substituting extension variables into C. Let x be a newly-substituted extension variable (i.e., $x \in (C' \setminus C)$) with definition $x \leftrightarrow a \vee b$. Then all the literals in C' are falsified, and the decision level of x is equal to the decision level of the literal that is falsified later out of a and b. Without loss of generality, we will assume that a is falsified before b.

Proof. We know from Lemma 4.2 that all the literals in C' must be falsified. Since C is an asserting clause, all its literals must also be falsified. The proof that x and b have the same decision level proceeds by contradiction: suppose for the sake of contradiction that the decision level of x is not equal to the decision level of b. Let l_x and l_b be the decision levels of x and b respectively. Then either $l_b > l_x$ – which is a contradiction because the definition of x would propagate b to false immediately after x is falsified at level l_x – or $l_b < l_x$, which is a contradiction because the definition of x would propagate x to false at decision level l_b. Therefore, the decision levels of x and b must be equal. □

Corollary 4.4. C' is an asserting clause, and x is the asserting literal in C' if and only if either a or b is the asserting literal in C.

Then we have shown that the solver behaviour remains correct even with variable substitution: the new clause is always asserting, and the backtracking level remains the same regardless of whether extension variables were substituted into the clause.

4.1.4 Extension Variable Deletion

Similarly to learnt clauses, some extension variables may not be useful in the solver's proof search. These extension variables ultimately slow down the solver by causing fruitless unit propagations and taking up extra memory. The extension variable deletion stage of our ER solver framework is the last phase in an extension variable's lifetime. It is responsible for removing references to the deleted extension variable from every data structure in the solver, including clause databases and the `ExtDefMap` data structure.

To the best of our knowledge, there are no known results on the best conditions in which to delete extension variables. Thus, we take a similar approach to the problem as Audemard et al. [3] - we perform extension variable deletion at the same time as learnt clause deletion. However, unlike their implementation, we do not maintain an explicit dependency graph to track dependencies between the definitions of extension variables. Instead, we again make use of the `ExtDefMap` data structure to identify non-basis variables. The variable deletion approach taken in our ER solver framework is otherwise the same as the approach used in GlucosER [3]: we delete low-activity extension variables which do not participate in the definitions of other extension variables.

4.2 Heuristics

When choosing heuristics, it is important to account for the complexity of the algorithms implementing the heuristics. These algorithms can easily dominate the running time of the solver, negating any advantage granted by the introduced extension variables.

4.2.1 Clause Selection

The heuristic used to select the set of clauses for extension variable addition has a significant effect on the relevance of the extension variables for the solver. Additionally, the number of selected clauses will affect the running time of the algorithm used to define variables. Therefore, it is prudent to choose a well-motivated clause selection heuristic. A well-known ER proof of the pigeon-hole principle [27] proceeds by defining extension variables in a manner which suggests that knowledge of the global problem structure is essential to efficiently using extension variables. One idea for approaching this is to select long clauses as a proxy for "global" knowledge. However, in our preliminary experiments, we found that this performed significantly worse than the heuristics we describe below, which represent more "local" reasoning.

High-Activity Clauses A commonly-held belief in the SAT community is that CDCL solvers perform "local" search, where learnt clauses constrain the part of the search space where the solver is searching. Although there is no consensus yet about the formal definition of "local", one idea which has been proposed is *clause activity*, which is commonly used in clause deletion heuristics [4]. Therefore, a natural heuristic to consider is to generate new variables from the clauses with the highest activities in the hope that the new variables will encode useful information about the local search space.

In particular, we choose to select the top k clauses with the highest activities. We note that it is possible for multiple clauses to have equal activities, and that when choosing the top k clauses with the highest activities, some clauses will need to be excluded despite having an activity in the top k; e.g., consider the case where the top $k+1$. There are many possible algorithms for selecting the k largest elements in a set, but we consider two which are based on common sorting algorithms:

1. The *quickselect* algorithm [42] is an algorithm for selecting the k^{th} largest element from a list of n elements. This algorithm is closely related to the *quicksort* algorithm [41], which is an *unstable* sorting algorithm[3]. For a list containing n elements, the average time complexity of the quickselect algorithm is $O(n)$, with a worst-case complexity of $O(n^2)$. Although this algorithm is usually only used to select the k^{th} largest element, a side effect of its close relation to quicksort is that it partially sorts the data such that the selected element partitions the list into two groups, where one group contains all the elements larger than the selected element. This makes it simple to adapt quickselect to the problem of selecting the top k elements.

2. The *insertion sort* algorithm [50] is a stable sorting algorithm which builds a sorted list as it scans through the input list. We use this algorithm to build a sorted list of the top k elements, and do not sort the rest of the elements. This approach has a time complexity of $O(kn)$, which is significantly worse than the complexity of the quickselect algorithm. Therefore, it behooves us to choose a small constant k to keep the runtime of this algorithm manageable.

Despite the poor time complexity of the insertion sort algorithm, our preliminary experiments indicate that the ER solver using the insertion sort algorithm actually outperforms the version of the ER solver using the quickselect algorithm. Since all other variables were kept relatively constant, this indicates that the stability of the sorting algorithm is significant in avoiding performance deterioration in the rest of the solver. Thus, in the final

[3] A sorting algorithm is *stable* if for any two elements with the same key value, their relative order in the input list is preserved in the output list.

experimental evaluation, we only consider the variant of the solver using the insertion sort algorithm for selecting the set of k clauses with the highest activities.

Clause Filtering Since the time complexity of the insertion-sort-based clause selection algorithm depends on the number of clauses it needs to examine, one possible optimization is to first use a heuristic to filter out clauses which are unlikely to be in the top k. We consider two different heuristics for this subcomponent of clause selection, based on the width and LBD of learnt clauses. To evaluate the efficacy of our methods, we also include the trivial filtering heuristic which accepts every clause.

For the clause width heuristic, we discard clauses from further consideration if the width of the clause does not lie within a certain range $[w_{min}, w_{max}]$. Similarly, for the LBD clause filtering heuristic, we exclude clauses from further consideration if the LBD of the clause does not lie within a certain range $[\ell_{min}, \ell_{max}]$. To motivate the usage of a relatively small value for w_{max}, observe that short clauses constrain the search space more tightly than long clauses, so they are more likely to participate in conflicts, giving them relatively high clause activities. A similar argument holds for ℓ_{max}. The exact values of these parameters can be tuned, but for our experiments, we choose values of $w_{max} = 7$ and $\ell_{max} = 5$.

The choice of w_{min} and ℓ_{min} is motivated not by the activities of the clauses, but rather by the probability that an extension variable would be useful for the solver. Clause with very small widths or very low LBD are already very strong constraints on the search space. Since introducing extension variables of the form $x \leftrightarrow a \vee b$ adds the clauses $(\neg x \vee a \vee b) \wedge (x \vee \neg a) \wedge (x \vee \neg b)$, introducing extension variables always introduces clauses of width 3, which is a weaker constraint than a binary clause. Therefore, a reasonable condition for filtering clauses is to exclude clauses with width $w < 3$, so we choose $w_{min} = 3$. Since the LBD of a clause is a lower bound on the width of the clause (by the definition of LBD), choosing $\ell_{min} = 3$ also satisfies this condition.

4.2.2 Extension Variable Definition

The heuristic chosen for defining extension variables is a deciding factor for the success or failure of our method. As a control heuristic, we consider selecting pairs of literals at random from the set of selected clauses. The implementation of this heuristic is trivial and is not discussed here. We also consider another heuristic, which is outlined below:

Common Literal Pairs Part of the intuition underlying the extended resolution proof system is that extension variables allow us to efficiently reason over facts about the problem

instance. Thus, one idea for introducing extension variables is to maximize the number of times that the extension variable can be reused. We approach this by attempting to maximize the number of clauses where the extension variable can be substituted.

An outline of the algorithm is as follows: First, we count the number of occurrences of each literal pair in a clause, using a hash table to keep track of the count for each pair of literals. For m clauses of size k, this can be done with time complexity $\Theta(mk^2)$, with a worst-case space requirement of $\Theta(mk^2)$ literal pairs. Then, to find the n most frequently occurring literal pairs, we use the quickselect algorithm [42] as discussed in Section 4.2.1 with an expected overall time complexity of $\Theta(mk^2)$.

4.2.3 Extension Variable Substitution

As discussed in Section 4.1.3, the worst-case time complexity of the extension variable substitution algorithm for a clause of size k is $\Theta(k^2)$. Since the variable substitution algorithm is executed for every learnt clause, the total computational overhead associated with variable substitution is potentially very high. To mitigate this, our ER solver framework supports the use of heuristics to avoid performing substitution in irrelevant learnt clauses.

Similarly to clause selection, we perform a filtering step to avoid performing the extension variable substitution algorithm on clauses which are unlikely to be useful. Again, we consider filtering clauses based on ranges of clause width or LBD, skipping the extension variable substitution step if the width of the clause does not lie within the range $[w_{min}, w_{max}]$, or when using the LBD clause filtering heuristic, skipping substitution if the LBD of the clause does not lie within the range $[\ell_{min}, \ell_{max}]$.

In particular, because the extension variable substitution algorithm has time complexity $\Theta(k^2)$ for a clause of size k, it is beneficial to choose a relatively small value of w_{max}. Otherwise, it is possible for the solver's running time to become dominated by the computation associated with the substitution algorithm. As in the case of clause selection, it is also useful to ignore clauses containing fewer than three literals. Our extension variables are defined over pairs of clauses, so unit clauses can safely be ignored. If the solver learns a binary clause containing a pair of literals corresponding to an extension variable, that variable becomes completely redundant: the assignment of the extension variable is essentially forced, so that extension variable will never participate in any useful resolution steps. Hence, we choose $w_{min} = 3$.

4.2.4 Extension Variable Deletion

Our framework supports arbitrary user-defined heuristics for selecting extension variables to delete. Although it is not clear what makes extension variables useful, we choose to take an approach similar to the traditional activity-based learnt clause deletion heuristic, and choose to delete extension variables with low activities. In particular, we consider two different activity-based methods for selecting extension variables to delete: `delConst`, which deletes all extension variables whose VSIDS scores fall below a constant threshold; and `delFrac`, which deletes the least active fraction (e.g. 50%) of all extension variables.

Our preliminary experiments measuring the effect of enabling or disabling these deletion heuristics showed that the ER solver performs much better with extension variable deletion enabled than with it disabled. We also found that the `delFrac` heuristic significantly outperforms `delConst` on average, so only `delFrac` is considered as a deletion heuristic in our final empirical evaluation.

4.3 Empirical Evaluation

In this section, we demonstrate our framework's ability to implement a variety of heuristics and facilitate the development and evaluation of new heuristics, and show that CDCL solvers augmented with our ER framework are still competitive with the unmodified CDCL solvers. To evaluate the efficacy of our ER methods, we conduct a large number of experiments and compare the computation times required to solve a broad variety of benchmark instances. We also examine the usage of extension variables in an attempt to measure the quality of the variables introduced by our methods.

4.3.1 Experimental Setup

Solvers We augmented the `MapleSAT` and `MapleLCM` solvers with our ER framework.[4] However, we found that the results for `MapleSAT` and `MapleLCM` are very similar, so for brevity, we only present the results for the stronger `MapleLCM`-based solvers in this thesis. Since the ER framework introduces many different design spaces, each of which can use one of many different possible heuristics, there is a combinatorial explosion in the number of possible ER solvers. The components of our framework interact heavily with each other, so it is difficult to predict which combination of heuristics results in the best solver.

[4]Source code for the solvers is available at https://github.com/chjon/xMapleSAT

Solver	Description
`MapleLCM`	Unmodified `MapleLCM` solver
`xMapleLCM_common`	Common literal pairs from high-activity clauses
`xMapleLCM_random`	Random literal pairs from high-activity clauses
`xMapleLCM_ler`	LER method for introducing extension variables

Table 4.1: Summary of solvers used for the empirical evaluation of our ER framework.

Benchmark	# of Instances	Source / Description
`rand3cnf`	170	Randomly-generated 3CNF instances [53]
`urquhart`	27	Urquhart instances [18, 84]
`php`	17	Pigeonhole principle (and functional PHP) [53]
`sat2019`	400	SAT Competition 2019 (Main Track) [24]
`sat2020`	400	SAT Competition 2020 (Main Track) [19]
Total	1014	

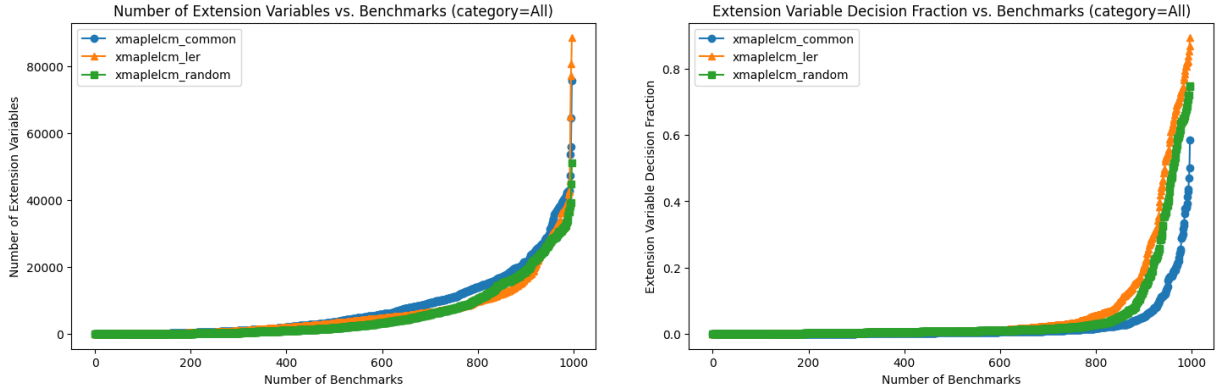
Table 4.2: Summary of instances used for the empirical evaluation of our ER-based solvers.

As a simplifying assumption to keep the number of considered solvers small, our experimentation assumes that the heuristics are independent – rather than trying to find the best combination of heuristics across every design space all at once, we look for the best heuristic locally within each design space, and take the combination of the best heuristics in our final solver. A summary of the solvers we considered is presented as Table 4.1. We also include the Virtual Best Solver (VBS) in our comparison as a reference for the best possible improvement using our techniques.

Benchmarks We selected a variety of instances for our testing. These instances are summarized in Table 4.2. The `rand3cnf` benchmark contains instances which are generally considered to be difficult for SAT solvers, and are included just to see if there is a large difference in behaviour between traditional SAT solvers and our ER-based solvers. Similarly, the `sat2019` and `sat2020` instances are included to measure whether our methods cause performance deterioration in general. The `php` family of instances is known to have short ER proofs in theory, and previous ER solvers performed well on `urquhart` instances, so if our ER techniques are successful, we should expect to see improvement on these benchmarks.

Solver	rand3cnf	urquhart	php	sat2019	sat2020
VBS	2041.47	938.15	3620.40	4641.26	5327.14
MapleLCM	2371.96	5291.61	4119.39	5058.46	5933.10
xMapleLCM_common	2502.10	3531.34	4202.47	5495.80	6054.41
xMapleLCM_random	2468.91	4003.26	4118.92	5701.78	6262.23
xMapleLCM_ler	2528.94	962.78	3668.86	5647.48	6181.24

Table 4.3: PAR-2 scores summarizing the performance of each solver for each benchmark.



(a) Maximum number of live extension variables.

(b) Extension variable decision fraction.

Figure 4.2: Plots comparing the maximum number of live extension variables and the fraction of decisions made on extension variables for each ER-based solver on each instance.

4.3.2 Experiments and Results

Experiment 1: Quantity and quality of extension variables

To compare the internal behaviour of the ER-based solvers, we measure a few statistics regarding the usage of extension variables in the implemented solvers. In particular, we measure the number of extension variables, the maximum number of live (i.e., not deleted) extension variables at any point in the solver's execution, and the number of decisions on those variables. The number of live extension variables is significant for practical reasons: since there are so many possible definitions for extension variables, we must be careful not to add too many extension variables and overwhelm the solver. Each extension variable introduced into the solver requires additional memory resources, so if a large number of variables are added, the solver's run time may become dominated by the overhead

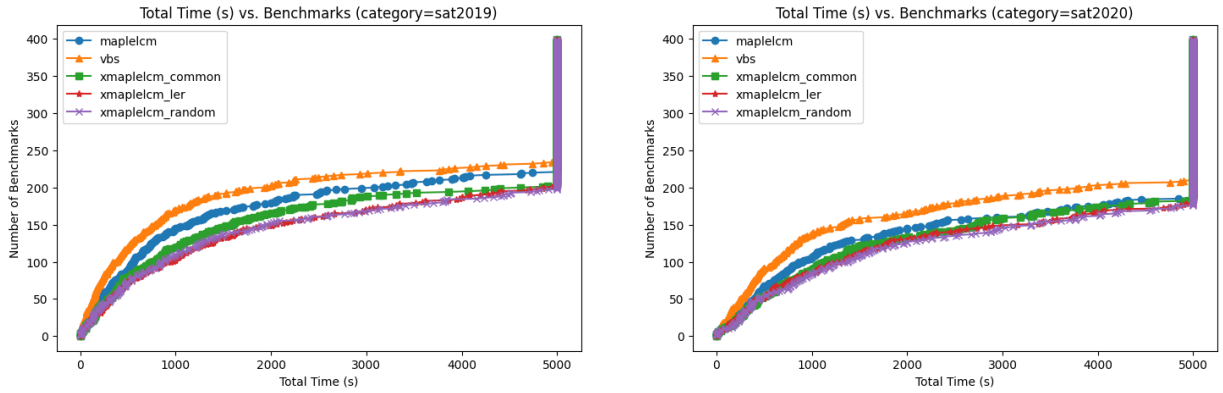
Solver	Total	Selection	Addition	Substitution	Deletion
xMapleLCM_common	5.128%	3.441%	0.377%	1.279%	0.031%
xMapleLCM_random	4.872%	3.495%	0.009%	1.252%	0.116%
xMapleLCM_ler	5.097%	3.058%	0.876%	1.138%	0.025%

Table 4.4: The average percentage of solver running time dedicated to ER-related computation for the clause selection (**Selection**), extension variable addition (**Addition**), extension variable substitution (**Substitution**), and extension variable deletion (**Deletion**) components of our ER solver framework for each solver, measured over all benchmarks.

associated with memory management. We present the data for the maximum number of live extension variables in Figure 4.2a.

From this data, we observe that for most benchmark instances, the maximum number of extension variables present in the solver at any one time is relatively small. However, in some cases, the number of extension variables present in the solver becomes significant, and even exceeds the number of variables in the original formula. Surprisingly, all the ER-based solvers introduce approximately the same number of extension variables. The LER method tries to introduce extension variables after every learnt clause, whereas the other considered methods only try to introduce extension variables upon restarts. Since all the considered ER-based SAT solvers introduce approximately the same number of extension variables, it is difficult to say whether a large number of extension variables is beneficial in ER proof search.

In the work on LER [3], the authors claim that frequent branching on extension variables indicates that the resultant proof is a proof in LER and not just Res. We extrapolate their reasoning to claim that instances where the solver frequently branches on extension variables are instances where the generated proof is an ER proof. If this is true, our experimental data shows that the considered solvers generate Res refutations and not ER proofs for the majority of problem instances. Figure 4.2b presents this data as a plot of the fraction of decisions where the solver branches on an extension variable. In the vast majority of instances, extension variables make up a minute proportion of all decisions. Examining this data in the context of Figure 4.2a, cases where the number of decisions on extension variables is large may instead be indicative of cases where the solver is overwhelmed by extension variables. Then we cannot claim with certainty that a large number of extension variable decisions implies that the resulting proof is an ER proof.



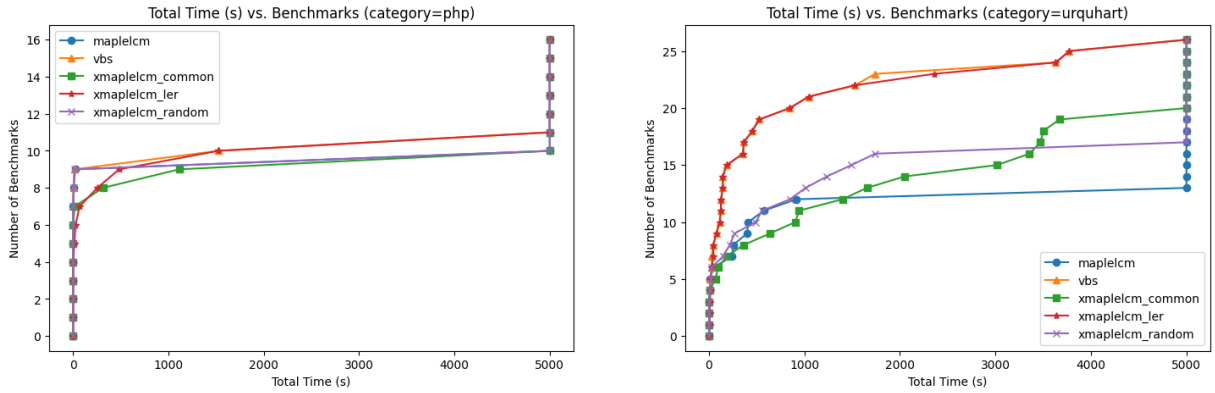
(a) `SAT2019` (b) `SAT2020`

Figure 4.3: Cactus plots comparing the performance of our method with LER and the baseline method using the `MapleLCM` solver on the `SAT2019` and `SAT2020` benchmarks.

Experiment 2: Measuring the overhead of our ER framework

A major goal of our ER framework is to facilitate the development of ER-based SAT solvers and heuristics without compromising too much of the performance of the underlying CDCL solver. When performing this development, it is often difficult to determine whether changes in overall solver performance are due to the chosen set of heuristics, or due to improvements in implementation. To aid in this process, we added timing instrumentation to our solver implementation to measure the performance overhead associated with each component in the framework. Since each of the components we introduce in our framework has the potential to dominate the running time of our solver, this allows us to identify areas to prioritize for optimization, as well as to decide whether the computational cost for implementing a certain set of heuristics outweighs the associated benefits. We present this data for each ER-based solver in Table 4.4.

We see that the average overall ER computational overhead associated with each of the ER-based solvers is fairly small, with less than 6% of total computation time going toward ER-specific computation for each solver on average. Thus, the implementation of our framework and chosen heuristics seem to be successful with respect to the goal of reducing ER-related computational overhead. We observe that the most computationally expensive component of the framework is clause selection, followed by extension variable substitution. The relatively high computational cost of extension variable substitution is expected, whereas the high cost of clause selection is somewhat surprising. Unfortunately, as seen in Figure 4.3, our ER-based methods perform slightly worse than the base solver



(a) `php` (b) `urquhart`

Figure 4.4: Cactus plots comparing the performance of our method with LER and the baseline method using the `MapleLCM` solver on the `php` and `urquhart` benchmarks.

on the diverse selection of instances from the SAT competition.

We hypothesize that the high computational cost of clause selection is a result of the streaming algorithm we use to keep track of the set of selected clauses, which is discussed in Section 4.1.2. Specifically, the cost is likely associated with learnt clause deletion and the periodic garbage collection of the clause database. The set of selected clauses keeps track of the index of each selected clause rather than duplicating and storing the entirety of each selected clause. Therefore, to prevent dangling references to clauses, we must remove deleted clauses from the set of selected clauses every time clause selection occurs, and we must additionally map each stored index to a new index upon garbage collection, where the positions of clauses in memory are altered. In light of these results, we believe that it may be worthwhile to recompute the set of selected clauses on demand every time extension variable addition is performed rather than use this streaming algorithm. We relegate further experimentation on these methods to future work.

Experiment 3: Testing the efficacy of our ER framework

As a demonstration of the versatility of our framework, we use it to implement a method from the literature and attempt to reproduce the results described in the corresponding paper. Specifically, we implement the LER method introduced by Audemard et al. [3]. They demonstrated that their ER solver outperformed the base solver on the functional pigeonhole principle (FPHP) [25, 53] and Urquhart [84] families of problems, but did not

improve on the SAT competition instances.

We implemented their method using our framework by defining appropriate heuristics for each component of the framework. Although we only observed a small improvement on the `php` benchmark, we were able to reproduce a significant improvement on the `urquhart` benchmark (see Figure 4.4). This suggests that the performance improvement of the restricted LER method over the Urquhart benchmark is robust to changes in solver implementation and combinations of heuristics, whereas the improvement over PHP is sensitive to these details. We also observe that our other ER heuristics are able to outperform the base solver on the `urquhart` benchmark. Interestingly, even the `xMapleLCM_random` solver improves performance here, which suggests that some of the success of the other methods comes from simply defining extension variables over high-activity variables.

4.4 Related Work

Tseitin's discovery of the Extended Resolution proof system [83] and Cook's proof that ER is a much stronger proof system than Resolution [27] inspired many attempts to automate ER and integrate it into SAT solvers. However, due to the difficulty of choosing useful extension variable definitions, most work has been in very restricted applications of ER, or in developing methods using alternate proof systems which can be viewed as constructing ER proofs rather than trying to automate ER directly. This body of work includes pre-processing techniques [77, 61], symmetry-breaking methods [74] and BDD-based SAT solvers [15, 76]. Comparatively, there has only been a small amount of work investigating the direct integration of ER into the heart of SAT solving algorithms.

4.4.1 Local Extended Resolution

So far, the most successful attempt to directly implement an ER-based SAT solver is based on Local Extended Resolution (LER), which is a restriction of ER that is easier to automate [3]. LER which only allows extension variables to be defined if particular pairs of clauses have already been derived; specifically, LER only allows the definition of $x \leftrightarrow a \vee b$ if the two clauses $C_1 = (\neg a \vee c_0 \vee c_1 \vee ... \vee c_k)$ and $C_2 = (\neg b \vee c_0 \vee c_1 \vee ... \vee c_k))$ are already available; i.e., C_1 and C_2 are either original or derived clauses. In their solver, Audemard et al. [3] further restrict the proof system to define extension variables only if the previous two learnt clauses have the appropriate form. They demonstrate empirical improvements on the functional pigeonhole principle and Urquhart families of instances, although the performance of the solver still scales exponentially.

4.4.2 Extended Clause Learning

Another attempt at implementing an ER-based SAT solver is the Extended Clause Learning (ECL) scheme introduced by Huang [43], which is equivalent in power to ER. Unlike LER and the original definition of ER, ECL permits the definition of new extension variables over entire clauses rather than only binary clauses. Specifically, in ECL, new extension variables $x \leftrightarrow \alpha$ can be defined when the solver reaches a conflicting assignment and α is a subset of the solver's assignment trail (i.e., every literal in α is negated by the assignment trail). At this point, it is difficult to set x with an appropriate value and decision level, so the ECL algorithm simply restarts the solver. Using their implementation of an ECL-based SAT solver, Huang shows improvements on several small families of benchmarks, as well as significant performance deterioration on some other benchmarks [43]. The ECL solver was not tested against the PHP or Urquhart instances, so it remains unclear whether the ECL approach scales well for these classes of problems.

Chapter 5

Conclusion and Future Work

In this thesis, we presented two major contributions: one for the design of BCP algorithms, and another for the design of SAT solvers based on Extended Resolution.

First, we presented Delayed BCP (as well as a design space for related BCP algorithms), a novel prioritized BCP algorithm. Delayed BCP is the first BCP algorithm to utilize a priority ordering of the literals while performing unit propagation. We demonstrated that by using an appropriate priority order, Delayed BCP is able to outperform the traditional BCP algorithm for some benchmarks, and that Delayed BCP remains competitive with the traditional algorithm despite propagating unit clauses at a slower rate. We also showed that Reinforcement Learning methods are able to learn a policy for dynamically switching between BCP algorithms during a solver's search, and that this combined approach is able to outperform both variants of BCP individually for some benchmarks.

Future empirical work in this area should study the effect of other priority orders, and investigate whether it could be advantageous to switch between priority orders rather than only switching between BCP variants. We note that the existing body of work on prioritized BCP algorithms only examines these algorithms from an empirical perspective, and it is currently not deeply understood why prioritized BCP algorithms sometimes outperforms the traditional algorithm. Thus, it would be greatly beneficial to also study the effects of prioritized BCP from a theoretical perspective, as this could lead to better choices of priority orderings and eventually, faster SAT solvers.

Second, we presented a conceptual framework for the programmatic development of SAT solvers based on Extended Resolution, and implemented it into some popular SAT solvers. We introduce the `ExtDefMap` data structure for efficiently implementing our algorithms, and explore various heuristics for each component of the framework. By examining

the computational overhead of our implementation, we observe that our design decisions are largely effective in avoiding performance deterioration. We show that our framework is flexible enough to implement a variety of heuristics, thereby demonstrating that our framework is an effective tool for quickly developing ER-based solvers and studying the effects of different heuristics. Finally, we compare ER-based solvers using various heuristics, partially reproduce some of the results in the literature, and demonstrate a class of instances where our method improves over the base solver.

As we identified in the development of our framework, there are many different design spaces associated with ER-based SAT solvers, and there are countless heuristics which could be implemented in each design space. Hence, some obvious future empirical work is to investigate the performance of additional heuristics for each component. It would also be advantageous to consider interactions between all the different components of the solver rather than optimizing the performance of each component in isolation. For example, by considering Cook's short proof of PHP [27], Audemard et al. [3] demonstrated empirically that it is insufficient just to introduce all the "best" extension variable definitions – therefore, the extension variable definition heuristic should be considered in tandem with the solver's branching heuristic.

Overall, we were successful in achieving the goals outlined at the beginning of this thesis. Our technique for prioritized BCP easily supports different heuristics for various priority orders, and our framework for developing Extended Resolution-based SAT solvers is similarly capable of implementing a wide range of heuristics for introducing and managing extension variables. For both Delayed BCP and our ER-based solvers, we identified efficiently-computable heuristics such that our solvers remain competitive with the unmodified baseline solvers over a diverse selection of benchmark instances, and we demonstrated that there are classes of instances where the solvers implementing our techniques improve over the base solvers.

References

[1] M. Alekhnovich and A.A. Razboro. Resolution is not automatizable unless w[p] is tractable. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 210–219, 2001.

[2] Albert Atserias and Moritz Müller. Automating resolution is np-hard. *J. ACM*, 67(5), sep 2020.

[3] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[5] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin (eds) Suda. Solver and Benchmark Descriptions. In *Proceedings of SAT Competition 2021*, volume B-2021-1 of *Department of Computer Science Report Series B*, Helsinki, 2021. Department of Computer Science, University of Helsinki.

[6] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Int. Res.*, 22(1):319–351, dec 2004.

[7] Paul. Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 274–282, 1996.

[8] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability*

Testing – SAT 2008, pages 28–33, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.

[10] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[11] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[12] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.

[13] Sam Buss. Suggestions for extended resolution and dual implication point learning. unpublished, 2016.

[14] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Conference on Computer and Communications Security*, 2006.

[15] Philippe Chatalic and Laurent Simon. Multiresolution for sat checking. *International Journal on Artificial Intelligence Tools*, 10:451–481, 12 2001.

[16] Jingchao Chen. Core first unit propagation. *ArXiv*, abs/1907.01192, 2019.

[17] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, jul 2001.

[18] SAT Competition. SAT 2003 Competition. https://http://satcompetition.org/. Accessed: 2023-04-14.

[19] SAT Competition. SAT Competition 2020 - Downloads. https://satcompetition.github.io/2020/downloads.html. Accessed: 2023-04-14.

[20] SAT Competition. SAT Competition 2021 - Downloads. https://satcompetition.github.io/2021/downloads.html. Accessed: 2023-04-14.

[21] SAT Competition. SAT Competition 2022 - Competition Tracks. https://satcompetition.github.io/2022/tracks.html. Accessed: 2023-04-14.

[22] SAT Competition. SAT Competition 2022 - Downloads. https://satcompetition.github.io/2022/downloads.html. Accessed: 2023-04-14.

[23] SAT Competition. SAT Competition 2023 - Downloads. https://satcompetition.github.io/2023/downloads.html. Accessed: 2023-07-12.

[24] SAT Competition. SAT Race 2019 - Downloads. https://satcompetition.github.io/2019/downloads.html. Accessed: 2023-04-14.

[25] Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 135–148, New York, NY, USA, 1974. Association for Computing Machinery.

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[27] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

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

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

[30] Jacques Dutka. The early history of the factorial function. *Archive for History of Exact Sciences*, 43(3):225–249, Sep 1991.

[31] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[32] 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):67–100, Feb 2000.

[33] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, page 431–437, USA, 1998. American Association for Artificial Intelligence.

[34] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[35] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In *AAAI*, 2008.

[36] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–58, Cham, 2019. Springer International Publishing.

[37] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 179–194, Cham, 2017. Springer International Publishing.

[38] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, 2016.

[39] Jonathan Heusser. SAT solving - An alternative to brute force bitcoin mining. https://jheusser.github.io/2013/02/03/satcoin.html, Feb 2013. Accessed: 2023-05-12.

[40] Jonathan Heusser. Satcoin. https://github.com/jheusser/satcoin, Apr 2018. Accessed: 2023-05-12.

[41] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, jul 1961.

[42] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, jul 1961.

[43] Jinbo Huang. Extended clause learning. *Artificial Intelligence*, 174(15):1277–1284, 2010.

[44] OEIS Foundation Inc. Double factorial of odd numbers, Entry A001147 in The On-Line Encyclopedia of Integer Sequences. https://oeis.org/A001147, 2023.

[45] Markus Iser and Tomáš Balyo. Unit propagation with stable watches. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Ed.: L. D. Michel*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages Art.–Nr.: 6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik GmbH (LZI), 2021.

[46] Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 2, chapter 25: Beta Distributions. Wiley, 2nd edition, 1995. ISBN 978-0-471-58494-0.

[47] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[48] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 343–356, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates drat. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 516–531, Cham, 2018. Springer International Publishing.

[50] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[51] Boris Konev and Alexei Lisitsa. A sat attack on the erdős discrepancy conjecture. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 219–226, Cham, 2014. Springer International Publishing.

[52] Jan Krajiček and Pavel Pudlák. Some consequences of cryptographical conjectures fors12and ef. *Information and Computation*, 140(1):82–94, 1998.

[53] Massimo Lauria, Marc Vinyals, Jan Elffers, Mladen Mikša, and Jakob Nordström. CNFgen, 2017. Accessed: 2021-12-18.

[54] Chunxiao Li, Noah Fleming, Marc Vinyals, Toniann Pitassi, and Vijay Ganesh. Towards a complexity-theoretic understanding of restarts in sat solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 233–249, Cham, 2020. Springer International Publishing.

[55] Jia Hui Liang and Vijay Ganesh. Solvers. https://maplesat.github.io/solvers.html#maplesat, 2016.

[56] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for sat solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 3434–3440. AAAI Press, 2016.

[57] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 225–241, Cham, 2015. Springer International Publishing.

[58] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 94–110, Cham, 2018. Springer International Publishing.

[59] Jia Hui Liang, Hari Govind V.K., Pascal Poupart, Krzysztof Czarnecki, and Vijay Ganesh. An empirical study of branching heuristics through the lens of global learning rate. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 119–135, Cham, 2017. Springer International Publishing.

[60] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for cdcl sat solvers. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 703–711. AAAI Press, 2017.

[61] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, pages 102–117, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[62] J.P. Marques Silva and K.A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[63] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in sat solving. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–114, Cham, 2018. Springer International Publishing.

[64] 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*, DAC '01, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery.

[65] Saeed Nejati. SAT-Encoding. https://github.com/saeednj/SAT-encoding, Sep 2022.

[66] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In Andrei Paskevich and Thomas Wies, editors, *Verified Software. Theories, Tools, and Experiments*, pages 120–131, Cham, 2017. Springer International Publishing.

[67] Digital Research Alliance of Canada. Graham - CC doc. https://docs.alliancecan.ca/wiki/Graham. Accessed: 2023-04-14.

[68] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[69] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 654–668, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[70] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527 – 535, 1952.

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

[72] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. Master's thesis, Simon Fraser University, 2004.

[73] Sabrine Saouli, Souheib Baarir, Claude Dutheillet, and Jo Devriendt. Cosysel: Improving sat solving using local symmetries. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 252–266, Cham, 2023. Springer Nature Switzerland.

[74] Bas Schaafsma, Marijn J. H. Heule, and Hans van Maaren. Dynamic symmetry breaking by simulating zykov contraction. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 223–236, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[75] Steven L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[76] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science – Theory and Applications*, pages 600–611, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[77] Mate Soos. CCC madness. https://www.msoos.org/2010/12/ccc-madness/. Accessed: 2023-06-25.

[78] Richard P. Stanley and Sergey Fomin. *Enumerative Combinatorics*, volume 2 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1999.

[79] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press Cambridge, 1998.

[80] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[81] William R. Thompson. On the theory of apportionment. *American Journal of Mathematics*, 57(2):450–456, 1935.

[82] Edward Thorndike. Some experiments on animal intelligence. *Science*, 7(181):818–824, 1898.

[83] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[84] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, jan 1987.

APPENDICES

Appendix A

Bounds on the Number of Sets of Extension Variable Definitions

We would like to answer the following: how many ways are there to define k distinct extension variables over n input variables? Let this number be given by the function $f(k, n)$. Unfortunately, it is difficult to answer this question as an exact function of k and n for a few reasons. However, we can compute some upper and lower bounds on this value to get an idea of its magnitude.

In this Appendix, we will make use of Stirling's asymptotic approximation for factorials and double factorials [30]:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{A.1}$$

$$n!! \approx \begin{cases} \sqrt{\pi n}\left(\frac{n}{e}\right)^{n/2} & n \text{ is even} \\ \sqrt{2n}\left(\frac{n}{e}\right)^{n/2} & n \text{ is odd} \end{cases} \tag{A.2}$$

A.1 Lower Bound

Consider the restricted problem where each variable can appear in the definition of at most one extension variable. Let $g(k, n)$ be the total number of ways to define k extension variables under this restrictions. Consider the further restriction where the basis variables only appear positively. Let $g^+(n)$ be the total number of ways to define k extension variables under the two restrictions above. Clearly, the number of ways to define a set

of extension variables while respecting these restrictions must be less than the number of ways to define a set of extension variables in general.

Observe that each time a new variable is defined, it "consumes" two previously available variables, for a net change of -1 variables each time an extension variable is defined. Then we can define at most $n-1$ extension variables, so $g^+(k,n)$ and $g(k,n)$ are not well defined for $k > n-1$. Consider the case of $k = n-1$. In this case, the relationships between variables are structured as a binary tree, where the leaf nodes correspond to the original variables and the internal nodes correspond to extension variables. Importantly, the names of the original variables are significant, whereas the names of the extension variables are arbitrary and interchangeable. Then the number of possible ways to define extension variables for this restricted case is equal to the number of full binary trees with n labeled leaf nodes.

Conveniently, this sequence is essentially given by OEIS sequence A001147 [44]: $a(n') = (2n'-1)!!$, which describes the number of full binary trees with $n'+1$ labeled leaf nodes. The proof that this is counted by this sequence is given by Example 5.2.6 in Enumerative Combinatorics [78]. We have $n = n'+1$ leaf nodes, so $g^+(n-1,n) = a(n-1) = (2n-3)!!$.

We can now compute $g(n-1,n)$ as a function of $g^+(n-1,n)$. Observe that after introducing $n-1$ extension variables, there is exactly one extension variable which is not a basis variable (i.e., the variable corresponding to the root of the binary tree). Since there are n original variables and $n-1$ extension variables, this means that there are a total of $n+(n-1)-1 = 2(n-1)$ basis literals. Since each basis variable appears exactly once in a definition, we have $2^{2(n-1)} = 4^{n-1}$ possible basis literals. Therefore, we have

$$g(n-1,n) = 4^{n-1} g^+(n-1,n) = 4^{n-1}(2n-3)!!$$

Applying Stirling's approximation (Equation A.2) gives the following formula, which grows super-exponentially in n:

$$g(n-1,n) = \Theta\left(4^{n-1}\sqrt{2(2n-3)}\left(\frac{(2n-3)}{e}\right)^{(2n-3)/2}\right)$$

Since the constraints defining $g(n-1,n)$ are unrealistically restrictive, this is a very loose lower bound on $f(n-1,n)$. It is simple to observe that the number of possible combinations of extension variable definitions must increase strictly monotonically with the number of extension variables. It follows that $f(k,n) > f(k',n)$ for all $k > k'$, and therefore, $f(k,n) \geq g(\min(k,n),n)$ for all k and n. If we consider $k = \Theta(n)$, we see that f grows super-exponentially, and so, the number of possible sets of extension variable definitions grows super-exponentially in the number of variables in the formula.

A.2 Upper Bound

We can compute an upper bound on the total possible number of sets of extension variable definitions by allowing the introduction of extension variables with identical definitions. Consider the problem of selecting a single extension variable definition given a formula containing n variables. This is simply the problem of selecting two distinct variables, each of which can occur with one of two polarities, for a total number of $\binom{n}{2} \cdot 2 \cdot 2 = 2n(n-1)$ possibilities.

After introducing this first extension variable, the formula contains $n+1$ variables. To count the number of possibilities for a second extension variable, we can pretend that we are simply defining a single extension variable over a formula containing $n+1$ variables, which gives $\binom{n+1}{2} \cdot 2 \cdot 2 = 2(n+1)(n)$ possible definitions for the second variable. Multiplying these two numbers together vastly over-counts the number of possible combinations of definitions. However, we will accept this for now as a loose upper bound. Continuing this process for k new variables, we have that the total number of combinations of extension variable definitions is upper bounded by the following product:

$$\begin{aligned} h(k,n) &= \prod_{i=0}^{k-1} 2(n+i)(n+i-1) = 2^k \cdot \frac{(n+k-1)!}{(n-1)!} \cdot \frac{(n+k-2)!}{(n-2)!} \\ &= 2^k \cdot \frac{(n-1)}{(n+k-1)} \cdot \left(\frac{(n+k-1)!}{(n-1)!} \right)^2 \end{aligned}$$

In particular, if we choose $k = n-1$, we have the following:

$$h(n-1,n) = 2^{n-1} \cdot \frac{n-1}{2n-2} \left(\frac{(2n-2)!}{(n-1)!} \right)^2 = 2^{n-2} \cdot \left(\frac{(2n-2)!}{(n-1)!} \right)^2$$

A.3 Comparison of Bounds

Since it is not immediately clear that $h(n-1,n) = o(g(n-1,n))$, we will consider the limit R of the ratio $\frac{g(n-1,n)}{h(n-1,n)}$ as $n \to \infty$ as a sanity check:

$$\begin{aligned} R &:= \lim_{n \to \infty} \frac{4^{n-1}(2n-3)!!}{2^{n-2} \cdot \left(\frac{(2n-2)!}{(n-1)!} \right)^2} \\ &= \lim_{n \to \infty} 2^n \cdot \frac{(2n-3)!! \cdot ((n-1)!)^2}{((2n-2)!)^2} \end{aligned}$$

We can evaluate this limit using Stirling's asymptotic approximations of the factorial and double factorial functions [30]:

$$
\begin{aligned}
R &= \lim_{n\to\infty} 2^n \cdot \frac{\sqrt{2(2n-3)}\left(\frac{(2n-3)}{e}\right)^{(2n-3)/2} \cdot \left(\sqrt{2\pi(n-1)}\left(\frac{(n-1)}{e}\right)^{(n-1)}\right)^2}{\left(\sqrt{2\pi(2n-2)}\left(\frac{(2n-2)}{e}\right)^{(2n-2)}\right)^2} \\
&= \lim_{n\to\infty} 2^{n-1/2} \cdot (2n-3)^{1/2} \cdot \frac{\left(\left(\frac{2n-3}{e}\right)^{(2n-3)/2}\right) \cdot \left(\frac{n-1}{e}\right)^{2(n-1)}}{\left(\frac{2(n-1)}{e}\right)^{4(n-1)}} \\
&= \lim_{n\to\infty} 2^{n-1/2} \cdot (2n-3)^{1/2} \cdot \left(\frac{2n-3}{e}\right)^{n-3/2} \cdot \frac{\left(\frac{n-1}{e}\right)^{2(n-1)}}{\left(\frac{2(n-1)}{e}\right)^{4(n-1)}} \\
&= \lim_{n\to\infty} 2^{n-(1/2)-4(n-1)} \cdot (2n-3)^{n-1} \cdot e^{-n+(3/2)+2(n-1)} \cdot (n-1)^{-2(n-1)} \\
&= \lim_{n\to\infty} 2^{-3(n-1)+1/2} \cdot (2n-3)^{n-1} \cdot e^{(n-1)+(1/2)} \cdot (n-1)^{-2(n-1)} \\
&= \lim_{n\to\infty} \left(2^{-3} \cdot (2n-3) \cdot e \cdot (n-1)^{-2}\right)^{n-1} \cdot (2e)^{1/2} \\
&= \sqrt{2e} \lim_{n\to\infty} \left(\frac{(2n-3)e}{2^3(n-1)^2}\right)^{n-1} \\
&= 0
\end{aligned}
$$

Therefore, $h(n-1,n) = o(g(n-1,n))$, so we conclude $f(n-1,n) = \Omega\left(g(n-1,n)\right)$ and $f(n-1,n) = O\left(h(n-1,n)\right)$:

$$f(n-1,n) = \Omega\left(4^n(2n-3)!!\right) \tag{A.3}$$

$$f(n-1,n) = O\left(2^n \cdot \left(\frac{(2n-2)!}{(n-1)!}\right)^2\right) \tag{A.4}$$