The Effect of Totalizer Bound Tightening on MaxSAT Solving

Ana Tatabitovska

Delft University of Technology

TU Delft

The Effect of Totalizer Bound Tightening on MaxSAT Solving

by

Ana Tatabitovska

Thesis Committee:	Dr. E. Demirović	TU Delft
	Dr. S. Picek	TU Delft
	M. Flippo	TU Delft

The thesis is part of the project "Towards a Unification of AI-Based Solving Paradigms for Combinatorial Optimisation" (OCENW.M.21.078) of the research programme "Open Competition Domain Science - M" which is financed by the Dutch Research Council (NWO).

TU Delft

Preface

The journey of writing this thesis has been one of the most joyful and challenging experiences I have encountered thus far. However, as a lifelong lover of mathematical puzzles and riddles, the opportunity to work on a topic as complex as MaxSAT was an honor.
A heartfelt thank you to Professor Emir Demorović, whose years of experience with MaxSAT guided my way through this thesis. I am also very grateful for Maarten Flippo, for his insight during our numerous discussions. And to the final member of my research group, I would like to thank Imko Marijnissen, for the friendship and support. To my family and friends, I would first like to apologize for talking about my thesis for the better part of 9 months, and would like to thank you for listening. A special thank you to my father, who gifted me my first computer, and to my mother, who has offered only the kindest words when I would lament about the difficulty of research.

Ana Tatabitovska
Delft, October 2023

Abstract

Maximum Satisfiability (MaxSAT) is a known problem within the optimization field which has led many different solving approaches to be devised in the last several decades. From Linear Search to unsatisfiable core-based solvers, many MaxSAT algorithms rely on cardinality constraints to express how many soft clauses can be violated at most. However, as MaxSAT is expressed in the Conjunctive Normal Form, there is a need to translate, or encode, these cardinality constraints into CNF. A popular encoding algorithm, the Totalizer Encoding, is used within these solvers - a system of encoding that builds a binary tree to express the cardinality constraint. This paper aims to introduce an alternate construction for the Totalizer Encoding, referred to as the Layered Totalizer Encoding, which interleaves the mechanics of encoding and solving as to cut down on runtime as well as potentially solve previously unsolved instances. The research shows that the Layered Totalizer Encoding outperforms Linear Search on average, solves more instances than Linear Search, and can be tuned through heuristics to show even more favorable results. Moreover, the Layered Totalizer Encoding is shown to not only work as a standalone algorithm, but can also boost the performance of the OLL algorithm.

Contents

1 Introduction

The field of optimization in mathematics and computer science has long been a subject of extensive research and continues to hold significance due to its far-reaching implications in everyday life. Notable examples of optimization problems are Network Routing [24], Resource Allocation [26], and different scheduling demands in different industries [8]. A noteworthy problem in the field of optimization is MaxSAT due to its widespread usage [22]. Currently, most of the research on MaxSAT solvers is focused on using unsatisfiable cores to create cardinality constraints and find the optimal solution. The construction and encoding of these cardinality constraints significantly affect the complexity of these algorithms. Therefore, the objective of this study is to enhance the Totalizer Encoding technique [20] by exploiting intermediate cardinality constraints, and perform a comparative analysis between this new manner of encoding and the established Linear Search algorithm [17].

The work presented in this paper aims to answer three research questions.

- **To what extent does lower boundary tightening on individual nodes in the Totalizer Encoding improve the performance of the Linear Search algorithm?** This is the main research question which sets the tone for the rest of the queries. Namely, the goal is to discover whether this new approach to encoding could lead to better runtimes and potentially solve new instances.
- **How can heuristic-tuning further improve the performance of the Layered Totalizer Encoding?** Due to the many different factors at play when solving a MaxSAT instance and encoding cardinality constraints, it is important to look into how the Layered Totalizer can further be improved. These factors range everywhere from MaxSAT instance size, to hardware specifications, to SAT Solver choices, however the specific performance factor of interest for this study is the variable ordering in the Totalizer tree.
- **How does the OLL algorithm perform when the Layered Totalizer Encoding is used to encode its cardinality constraints rather than the Generalized Totalizer Encoding?** The final major research question this work aims to analyze is how an unsatisfiable core-based algorithm benefits from this new encoding. The OLL algorithm [21] was chosen for this question because it serves as the motivation behind the Layered Totalizer Encoding.

The primary contribution of this paper is the new Layered Totalizer Encoding, which is built in a left-to-right, upward fashion and exploits intermediate cardinality constraints in order to raise the lower bounds imposed on nodes. This Layered Totalizer is extended to the Weighted MaxSAT problem. Furthermore, the Layered Totalizer Encoding is used within the OLL algorithm itself as the encoding function, and this Layered OLL is measured against the regular OLL to determine if the new encoding could improve a core-based algorithm.

Chapter 2 introduces and explains concepts related to MaxSAT solving relevant to understanding Layered Totalizer Encoding. Chapter 3 presents previous work done in the field of MaxSAT solving, as well as certain works related to more general concepts that are important to the Layered Totalizer Encoding. Chapter 4 contains the central aspect of the study, namely the theoretical and practical overview of the Layered Totalizer Encoding. Moreover, this chapter contains explanations of the different heuristics that can be tuned in order to alter the performance of the Layered Totalizer Encoding.

Chapter 5 outlines the manner in which testing of the Layered Totalizer was performed, how results are interpreted and why certain decisions are made regarding the analysis of the results. Chapter 6 demonstrates the results of the aforementioned tests and discusses what the results mean in the context of the research questions. Finally, Chapter 7 concludes the research presented in this paper, summarizes the results, and gives an overview of possible future work.

2

Preliminaries

This section aims to present the concepts of SAT, MaxSAT, Linear Search, the OLL algorithm, and the Totalizer Encoding.

2.1. Notation and Theoretical Background

The Boolean Satisfiability Problem, commonly referred to as **SAT**, has many real-life applications as it is used in scheduling, verification, security, and other schema-based applications [28] [9].

Formally, let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of boolean variables, and let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of clauses, where each clause c_i is a disjunction of literals. A literal is either a variable x_i or its negation $\neg x_i$, denoted as the set $L = \{\neg l_1, l_1, \neg l_2, l_2, \ldots, \neg l_n, l_n\}$. The Conjunctive Normal Form is the conjunction of all these clauses, known as **CNF**. The SAT problem can then be defined as finding a truth assignment $A : L \to \{0, 1\}$ such that for every clause $c_i \in C$ at least one literal in c_i evaluates to true. A clause c_i is said to be satisfied if $\exists l_j \in c_i : A(l_j) = 1$. A clause c_i is said to be unsatisfied if $\forall l_j \in c_i : A(l_j) = 0$.

In other words, the SAT problem is to determine whether there exists a truth assignment that satisfies every clause simultaneously. If such an assignment exists, the formula is satisfiable; if no such assignment exists, the formula is unsatisfiable.

The example in Figure 2.1 shows a SAT problem with 5 variables and 7 clauses. As can be seen, clauses are not limited in size or number, and potentially many solutions exist for a given problem. One such solution to this problem is $x_1 = 0, x_4 = 0, x_5 = 0$, where the values of x_2, x_3 can take either of the values without violating the clauses. Though small problems can easily be solved by humans, as the problems grow in both the number of variables and clauses, the need for so-called SAT solvers or SAT oracles is evident.

Example 1: SAT Instance

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$

$$C = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$

Figure 2.1: Example of a SAT problem instance with 5 variables and 7 clauses, as originally shown in [15]

SAT solvers are algorithms or tools created to solve the SAT problem. These solvers take as input a set of clauses and attempt to satisfy all of them, with the output being the assigned values to each variable if all clauses are satisfied, or an unsatisfied result. Moreover, solvers can also be invoked with an additional input parameter, namely an assumption. An **assumption** is simply a variable fixed to a certain value. Specifically, when invoking Oracles with assumptions, the question the Oracle is answering is not only whether there is a solution to the instance, but whether a solution exists where

a certain variable is set explicitly to 0 or 1. Over the years, many solvers have been developed on the basis of different techniques. Two of the most advanced implementations are Glucose4 [2] and CaDiCal [11], both of which are variations of a Conflict Driven Clause Learning Solver (CDCL) SAT solver [30]. A CDCL Solver is a SAT solver that learns from conflicts between clauses and creates new clauses to represent this conflict. This allows the solver to avoid the same conflicts and explore the search space of possible solutions more efficiently [30]. This paper only uses SAT solvers as a black box tool, thus further discussion of solver intricacies is not included. However, this study may still allow for further research into how different types of solvers may affect the solving time for a given instance.

MaxSAT is an extension of the standard SAT problem, which adds an additional layer of complexity to the problem, namely the concept of optimization. **Optimization** in Computer Science is the process of minimizing or maximizing a given mathematical inequality equation, known as the objective function. What constitutes an optimal solution depends on the problem at hand. MaxSAT encodes these concepts using the building blocks provided by the SAT problem. MaxSAT is defined as an optimization problem that has the goal of satisfying the maximum number of clauses. To be precise, MaxSAT has a set of variables, X, and a set of clauses, C, just like the SAT problem, where its set C is composed of two subsets, the set of hard clauses, H, and the set of soft clauses, S. For the length of this paper, all soft clauses are considered to be unit soft clauses, meaning composed of a single literal, as any MaxSAT instance can be presented through unit soft clauses [3]. MaxSAT instances can be separated into two categories: unweighted and weighted. In the unweighted case all soft clauses have a weight of 1, and in the weighted case each soft clause can have any integer value as its weight. In a given MaxSAT instance, the goal is to satisfy all hard clauses and as many soft clauses as possible or to maximize the weight of the satisfied soft clauses. Each MaxSAT instance can be viewed as having the objective function:

$$\max \sum w_i \cdot \text{is\_SAT}(s_i)$$

where w_i is the weight of a given soft clause and is_SAT(s_i) is either 0 or 1 depending on whether the current literal assignment can satisfy the given clause.

Figure 2.2 builds on top of the example given in Figure 2.1 by adding five new soft clauses. For simplicity, the examples shown in this Chapter all utilize the same unweighted example.

Example 2: MaxSAT Instance

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$
$$S = \{(x_1), (x_2), (x_3), (x_4), (x_5)\}$$

Figure 2.2: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses and 5 soft clauses, as originally shown in [15]

Unlike the SAT version of the problem presented in Figure 2.2, there is no straightforward solution that will satisfy all hard and all soft clauses. If the assignment devised for the SAT instance, namely $x_1 = 0, x_4 = 0, x_5 = 0$, is applied, at least 3 soft clauses are violated, resulting in a minimum cost of 3 for this instance. However, this solution represents just one of several potential solutions to the problem. It is not possible to determine if this solution is optimal by solely focusing on solving the hard clauses. As will be shown throughout this chapter, 3 is in fact the optimal solution for this problem, however this is a coincidence. Consequently, an alternative perspective of the problem is required, in which the soft clauses are taken into consideration.

2.2. Algorithms and the Totalizer Encoding

This subsection provides an overview of two MaxSAT algorithms that served as the basis for the new encoding presented in this research: Linear Search and the OLL algorithm. Additionally, it examines the Totalizer Encoding in detail and lays out the groundwork for the Layered Totalizer Encoding.

2.2.1. Linear Search

The **Linear Search** algorithm is an iterative MaxSAT algorithm that uses calls to a SAT oracle to find an optimal solution. In this instance, iterativity entails the incremental testing of potential values for the optimal cost of the solution, while the SAT Oracle denotes a SAT solver treated as a black box, in which the input is a formula, yielding a corresponding outcome. This algorithm encompasses two variants, namely, Linear Search UNSAT-SAT and Linear Search SAT-UNSAT. In the context of these two versions of Linear Search, "UNSAT" denotes unsatisfiability, whereas "SAT" signifies satisfiability. Both instances of the Linear Search algorithms utilize a new concept, namely **relaxation**. Relaxation in terms of MaxSAT refers to the process of introducing new variables, the set of I, and using these variables to relax the soft clauses. Relaxing a clause is to add a fresh variable to a soft clause. The concept of relaxation is intuitive - introduce new variables which can be set to true if a certain soft clause is violated. To illustrate the concept of relaxation, see Figure 2.3

Example 3: MaxSAT Instance With Relaxation

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$I = \{y_1, y_2, y_3, y_4, y_5\}$$
$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$
$$S = \{(x_1 \vee y_1), (x_2 \vee y_2), (x_3 \vee y_3), (x_4 \vee y_4), (x_5 \vee y_5)\}$$

Figure 2.3: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses and 5 soft clauses with their respective relaxation variables and the set of relaxation variables, as originally shown in [15]

As seen in Figure 2.3, a new set of variables is introduced, I, and each variable in I corresponds to exactly one soft clause. To ensure the amount of violated soft clauses is minimal, a so-called cardinality constraint is imposed over the relaxation variables. In the scope of MaxSAT, a cardinality constraint is a mathematical expression that limits the number of elements that can be set to TRUE, more specifically, the number of clauses that can be violated at most. The cardinality constraint is in a Pseudo-Boolean form, wherein a certain set of variables is presented as a summation on the left-hand side, and the right-hand side is an integer. The left and right-hand sides are separated by one of the five possible relation operators, namely $\leq, \leq, =, \geq, \geq$. An example of a cardinality constraint can be seen in Figure 2.4, depicting a cardinality constraint consisting of five boolean variables that can take on a maximal total amount of 3 or less.

Example 4: Cardinality Constraint

$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 3$$

Figure 2.4: Example of a cardinality constraint with five variables and a right-hand side of 3

However, this cardinality constraint is not in CNF, like the hard and soft clauses in any MaxSAT instance are. This presents an issue for any SAT solver, as SAT solvers can only work with CNF formulas. To ameliorate this, an encoding can be performed - a transformation from one form to another. This transformation can have a significant impact on the performance of any MaxSAT algorithm, which is the central topic of this research. The specific encoding of interest is discussed further in this Chapter.

The **Linear Search UNSAT-SAT** algorithm, henceforth referred to as LUS, is the variation of Linear Search that determines the optimal solution by testing all possible values that the cost of the solution can take on, starting from the smallest up to the largest. This concept is where the name for the algorithm comes from: first select the smallest possible cost that the instance could have, call the SAT oracle with this value, and if the problem is unsatisfiable with that cost, increase the cost by 1. To illustrate this algorithm, the same MaxSAT example presented throughout this chapter is used.

Example 5: MaxSAT instance with LUS

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$I = \{y_1, y_2, y_3, y_4, y_5\}$$
$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$
$$S = \{(x_1 \vee y_1), (x_2 \vee y_2), (x_3 \vee y_3), (x_4 \vee y_4), (x_5 \vee y_5)\}$$
$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 0$$

Figure 2.5: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses and 5 soft clauses along with the cardinality constraint created at the start of the Linear Search UNSAT-SAT, as originally shown in [15]

Figure 2.5 shows the cardinality constraint set at the lowest possible value, 0, with all soft clauses relaxed. This modified version of the problem is then passed to the SAT oracle. For this instance, the oracle reports UNSAT, meaning there is no solution in which no soft clause is violated. As such, the right-hand side increases by 1, leading to the cardinality constraint below.

$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 1$$

Once more, the problem is passed to the SAT oracle, and once more the oracle determines whether there is a solution with a cost of 1. It is easy to see why the algorithm is called Linear Search UNSAT-SAT, due to its progression from unsatisfiable to satisfiable solutions. The algorithm is stopped when the first satisfiable assignment is reached. It can be deduced that the algorithm finds the optimal solution because it tests the possible costs of the problem systematically, one by one. Thus there is no cost lower that leads to a satisfiable solution the first satisfiable solution encountered.

Algorithm 1 Linear Search UNSAT-SAT

Input H, S
Output $cost$

1: $cost \leftarrow 0$
2: $S_{relaxed} \leftarrow Relax(S)$
3: **while** True **do**
4: $\quad result_{SAT}, result_{cost} \leftarrow SAT(H, S_{relaxed}, CNF(\sum y_i \leq cost))$
5: $\quad$ **if** $result_{SAT}$ is TRUE **then**
6: $\quad\quad return\ cost$
7: $\quad$ **else**
8: $\quad\quad cost \leftarrow cost + 1$
9: $\quad$ **end if**
10: **end while**

The pseudocode for LUS is presented in Algorithm 1. The algorithm takes two sets of clauses, the hard and soft clauses denoted as H and S respectively, as input and outputs the minimum cost, denoted as $cost$, for the given MaxSAT instance. In the algorithm, the initial value of $cost$ is set to 0, and the soft clauses are relaxed as previously described. The relax function is responsible for introducing a new relaxation variable for each soft clause and appending it to the clause using the $\vee$ operator. Subsequently, the algorithm explores all possible values ranging from 0 to the maximum potential cost, which is equal to the number of soft clauses in the instance. In each iteration, the instance, along with the cardinality constraint in CNF form, is passed to the SAT solver. If the SAT oracle returns TRUE, the value of $result_{SAT}$ is set to TRUE and the algorithm identifies the lowest cost, indicating the optimal solution. Otherwise, the algorithm continues its execution by incrementing the right-hand side of the cardinality constraint by 1. As discussed during the introduction of SAT solvers, the SAT Oracle can output not only the SAT or UNSAT result, but also the potential cost it has calculated. Although this potential cost, known as $result_{cost}$, is not taken into account for the Linear Search UNSAT-SAT

algorithm, it is of importance in the following algorithm and has been introduced here for the sake of clarity.

It should be noted that converting the cardinality constraint from the regular Pseudo-Boolean form does not impact the correctness of the algorithm - any encoding can give the desired results, hence the encoding into CNF form is left ambiguous in this algorithm.

Conversely, the **Linear Search SAT-UNSAT** algorithm, henceforth referred to as LSU, works in the opposite manner of LUS. With Linear Search SAT-UNSAT, iterative calls are made to the SAT oracle and the algorithm progresses from satisfiable solutions until the first unsatisfiable solution is reached, at which point the algorithm halts. Everything else regarding the algorithm remains the same, including the cardinality constraint, the encoding of said constraint, and so on. If the MaxSAT example is considered again, it can be observed that the only difference is the value of the right-hand side as shown in Figure 2.6. To determine the starting value of the right-hand side of the cardinality constraint in Linear Search SAT-UNSAT, an initial call is made to the SAT oracle with all soft clauses relaxed but no cardinality constraint imposed. Whatever cost returned by this SAT Oracle is the highest value taken into consideration for the final result. In the case of the instance below, an assumption will be made that the SAT Oracle returned a cost of 5 on this first call as to be able to demonstrate how the algorithm works.

Example 6: MaxSAT Instance with LSU

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$I = \{y_1, y_2, y_3, y_4, y_5\}$$
$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$
$$S = \{(x_1 \vee y_1), (x_2 \vee y_2), (x_3 \vee y_3), (x_4 \vee y_4), (x_5 \vee y_5)\}$$
$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 5$$

Figure 2.6: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses and 5 soft clauses along with the cardinality constraint created at the start of the Linear Search SAT-UNSAT, as originally shown in [15]

If the formulation of the problem is passed to the SAT Oracle, a satisfiable answer will be obtained, indicating that the optimal cost of this problem could be 5 or less, but not more. To test for a lower optimal cost, the right-hand side is decremented by 1, and the same problem formulation with the new cardinality constraint is passed to the SAT Oracle, as shown below.

$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 4$$

The iterative process continues until the first unsatisfiable answer is obtained from the SAT oracle, indicating that the last satisfiable cost is the optimal cost for the given instance. It is worth noting that instead of starting the process at the highest cost, such as 5 in this case, a more efficient approach exists to obtain the solution and reduce the number of calls to the SAT oracle. Specifically, the relaxed soft clauses and hard clauses can be passed to the SAT oracle without the cardinality constraint. The cost of the solution returned by the SAT oracle can then be extracted and used as the starting point for the optimal cost. This practical modification in the algorithm makes LSU more commonly used in practice.

Algorithm 2 Linear Search SAT-UNSAT

Input H, S
Output $cost$

1: $S_{relaxed} \leftarrow Relax(S)$
2: $result_{SAT}, result_{cost} \leftarrow SAT(H, S_{relaxed}))$
3: $cost \leftarrow result_{cost}$
4: **while** True **do**
5: $result_{SAT}, result_{cost} \leftarrow SAT(H, S_{relaxed}, CNF(\sum y_i \leq cost))$
6: **if** $result_{SAT}$ is TRUE **then**
7: $cost \leftarrow result_{cost} - 1$
8: **else**
9: $return\ cost + 1$
10: **end if**
11: **end while**

As seen in Algorithm 2, the structure of the algorithm is similar to that of Algorithm 1, which is to be expected as they share many similarities. A notable difference here occurs in terms of how the change of $cost$ is managed. Namely, every time the SAT Oracle returns a satisfiable answer, the new lowest cost becomes equal to the $result_{cost}$ reported by the solver. Then, the question becomes whether there is a value for $cost$ lower than that of $result_{cost}$, which is why in line 6 of the pseudocode the final value of $cost$ for a given iteration becomes $result_{cost} - 1$. If this value of $cost$ results in an UNSAT result from the Oracle, then the algorithm simply amends the value of $cost$ with $+1$, and this is the optimal solution.

2.2.2. OLL Algorithm

The **OLL** algorithm represents a significant departure from the Linear Search algorithm, as it goes beyond simply trying each cost naively for the problem. Instead, OLL solvers learn from unsatisfiable solutions, understanding what factors led to their occurrence. Additionally, the OLL algorithm dynamically constructs cardinality constraints based on specific information, rather than generating a single top-most constraint that encompasses all relaxation variables. The OLL algorithm, particularly its RC2 implementation, is a core-guided MaxSAT algorithm that iteratively calls a SAT oracle, extracts unsatisfiable cores from it, and generates cardinality constraints based on these cores [15]. To comprehend the functioning of OLL, it is necessary to first introduce the concept of an unsatisfiable core.

An **unsatisfiable core** refers to a subset of soft clauses, denoted as $core \subseteq S$, such that the combination of this core and the hard clauses leads to an unsatisfiable result, i.e., $core \wedge H$ is UNSAT [23]. In a given problem, there can exist multiple unsatisfiable cores of varying sizes. These unsatisfiable cores can be extracted by SAT Oracles if the instance is unsatisfiable in its current state. For example, in Figure 2.2, a potential core subset can be represented by

$$\{(x_1), (x_2), (x_3), (x_4)\}$$

Each soft clause within the core is then relaxed in the same manner as the one used in the Linear Search algorithms, and the augmented soft clauses are transferred to the set of hard clauses, denoted as H. A cardinality constraint is formulated as the summation of these relaxation variables, taking the form $\sum y_i \leq 1$, and is added to the soft clauses S which makes it eligible to be a part of an unsatisfiable core in a later iteration, whilst the soft clauses participating in the cardinality constraint are moved to the set of hard clauses. In cases where the core already contains a cardinality constraint from a previous iteration, instead of relaxation, the right-hand side variable is incremented by 1. The RC2 algorithm subsequently follows a similar approach to the LUS algorithm by invoking a SAT oracle. If the oracle returns an unsatisfiable result along with a new unsatisfiable core, the aforementioned procedure is repeated. Conversely, if the oracle returns a satisfiable result, it signifies that the optimal solution has been reached.

Example 7: MaxSAT Instance with OLL

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$I = \{y_1, y_2, y_3, y_4\}$$
$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5),$$
$$(\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5), (x_1 \vee y_1), (x_2 \vee y_2), (x_3 \vee y_3), (x_4 \vee y_4)\}$$
$$S = \{(x_5), (y_1 + y_2 + y_3 + y_4 \leq 1)\}$$

Figure 2.7: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses and 5 soft clauses at the beginning of the execution of the OLL algorithm, showcasing the first cardinality constraint created based on an unsatisfiable core, as originally shown in [15]

To illustrate the procedure, the same MaxSAT instance as before is used and modified to reflect OLL execution, as shown in Figure 2.7. It should also be noted that much like in Linear Search, the cardinality constraint is present in Pseudo-Boolean form. For ease of reading, the cardinality constraint has been left as is and is not encoded in its CNF form.

As described, after the modification of the problem, the SAT oracle is called once again with the new adjustments. Specifically, the next unsatisfiable core found is $(y_1 + y_2 + y_3 + y_4 \leq 1)$, meaning a simple adjustment is needed according to the RC2 algorithm - the right-hand side is increased by 1. Then the new cardinality constraint in the soft clause set is:

$$(y_1 + y_2 + y_3 + y_4 \leq 2)$$

As the formula is not yet satisfiable, a third unsatisfiable core is found comprising of:

$$\{(y_1 + y_2 + y_3 + y_4 \leq 2), (x_5)\}$$

Per the algorithm two new soft clauses are formed, and one clause is moved to the hard clauses. Specifically, the set of hard clauses now contains $(x_5 \vee y_5)$. As with the previous iteration of the algorithm, the cardinality constraint that has found itself in the unsatisfiable core has its right-hand side increased by one, leading to the new cardinality constraint of:

$$(y_1 + y_2 + y_3 + y_4 \leq 3)$$

A new cardinality constraint is created to express the relation between the elements of the unsatisfiable core; a soft clause and a cardinality constraint in this case. The new cardinality constraint is a so-called composite cardinality constraint, which means that it is a cardinality constraint that contains a cardinality constraint within itself. In this particular case, the composite cardinality constraint takes on the form of:

$$(y_5 + \neg(y_1 + y_2 + y_3 + y_4 \leq 2) \leq 1)$$

At this point of the algorithm, a satisfiable result is obtained from the SAT Oracle, and thus the RC2 algorithm concludes.

Algorithm 3 shows the pseudocode for the RC2 algorithm as originally showcased in [15].

Algorithm 3 RC2

Input: H, S
Output: $cost$

1: $cost \leftarrow 0$
2: **while** True **do**
3: $I = \emptyset$
4: $(result, core) \leftarrow SAT(H, S)$
5: **if** result == True **then**
6: **return** $cost$
7: **else**
8: $cost \leftarrow cost + 1$
9: **for** $clause_i \in core$ **do**
10: **if** $clause_i$ is not a cardinality constraint **then**
11: $S \leftarrow S \setminus clause_i$
12: $H \leftarrow H \cup (clause_i \cup y_i)$
13: $I \leftarrow I \cup \{y_i\}$
14: **else**
15: $I \leftarrow I \cup \{clause_i\}$
16: $rhs(clause_i) \leftarrow rhs(clause_i) + 1$
17: **end if**
18: **end for**
19: $S \leftarrow S \cup CNF(\sum_{y_i \in I} y_i \leq 1)$
20: **end if**
21: **end while**

Algorithm 3 demonstrates the functionality of the RC2 implementation of the OLL algorithm, as presented in [15]. Similarly to LUS, the cost is initialized to 0, and an iterative process is carried out involving calls to the SAT oracle. However, in the RC2 implementation, the relaxation of soft clauses is not performed upfront. Instead, soft clauses are only relaxed if they appear in the unsatisfiable core identified by the SAT oracle, as depicted in lines 10 through 13. Conversely, if an unsatisfiable core contains a previously established cardinality constraint, the right-hand side is incremented by 1. Once these steps are executed for all elements of the unsatisfiable core, the new cardinality constraint representing the unsatisfiable core is added to the set of soft clauses, as presented in line 19, and the process is repeated.

2.3. Totalizer Encoding

As previously discussed in the overview of both Linear Search and the OLL algorithm, cardinality constraints play a crucial role in determining the optimal solution for a given MaxSAT instance. These constraints define an upper limit on the number of soft clauses that can be violated when invoking the SAT oracle. The focus is on specifying the number of violated clauses rather than explicitly identifying which clauses can be violated. This allows the SAT oracle to explore various combinations of violated clauses as long as the total count remains within or below the specified limit.

However, cardinality constraints are typically expressed in the Pseudo-Boolean form, which is not compatible with SAT solvers. Therefore, an encoding process is required to transform these Pseudo-Boolean expressions into the appropriate CNF form. While several encoding techniques exist, this research focuses on the Totalizer Encoding, and more specifically, the Generalized Totalizer Encoding, which lies at the core of the investigation.

The **Generalized Totalizer Encoding** [16] utilizes a Binary Tree structure to devise all possible clauses and values required for representing a given cardinality constraint. A **Binary Tree**, which is a special type of tree data structure, permits each node to have a maximum of two children. Within the context of the Totalizer Encoding, individual relaxation variables are represented by leaf nodes, while the upper-level nodes depict distinct cardinality constraints associated with different variable combinations. It is important to note that the term ”cardinality constraint” has been previously used to refer solely to the sum of relaxation variables in the case of Linear Search or the sum of variables representing unsatisfiable cores in OLL. However, within the Totalizer Encoding, any non-leaf node signifies a

cardinality constraint. As an illustration of the structure, Figure 2.8 provides an example.

Example 8: Totalizer Encoding Tree

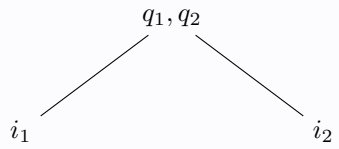

Figure 2.8: Totalizer Encoding of the possible values the sum $i_1 + i_2$ can take

The two leaf nodes depicted in Figure 2.8 each represent relaxation variables for a MaxSAT instance. To construct the subsequent layer of the Totalizer Encoding Binary Tree, the value of $i_1 + i_2 \leq k$ needs to be expressed. In this scenario, k is restricted to three possible values: 0, 1, and 2. These values stem from the fact that both i_1 and i_2 are limited to 0 or 1. Consequently, the following three cardinality constraints arise:

$$i_1 + i_2 \leq 0$$
$$i_1 + i_2 \leq 1$$
$$i_1 + i_2 \leq 2$$

These cardinality constraints must be expressed within the parent node. To appropriately represent these constraints, the representation utilized is known as the Unary Numeral System or the "Unary Representation." The Unary Representation is a straightforward method of expressing a number using only 0s and 1s. It represents a number b by constructing a sequence of 1s with a length of b, followed by as many 0s as needed. To illustrate how this would appear in the context of the three cardinality constraints concealed in the upper node of Figure 2.8, the variables q_1 and q_2 are considered. Then, the value 0 is represented as 00, the value 1 is represented as 10, and the value 2 is represented as 11.

For ease of understanding, in all following figures, including this one, the subscript of a given variable indicates what value it represents. In this case, q_1 is variable representing the cardinality constraint denoting that at most one soft clause can be violated, and q_2 is the variable representing the cardinality constraint violating at most 2 soft clauses.

Using the technique of representing different right-hand side values for a cardinality constraint, the Totalizer Encoding Tree can be constructed for any given problem. However, the tree itself is not in CNF form either, necessitating the creation of clauses to establish the relations between the variables in the children nodes and the parent nodes. These clauses denote the constraints and rules to be imposed. Referring back to the example in Figure 2.8, there are four possible outcomes the values i_1 and i_2 can take on:

$$i_1 = 0, i_2 = 0$$
$$i_1 = 1, i_2 = 0$$
$$i_1 = 0, i_2 = 1$$
$$i_1 = 1, i_2 = 1$$

Because the value of 0 is represented by the lack of any variable being set to 1 rather than its own variable, the sum of $i_1 + i_2 = 0$ does not require encoding. When one variable is set to 0 and another to a non-zero value, a clause needs to be expressed to allow only one child variable to be set to true. In this case, two combinations lead to $i_1 + i_2 = 1$, and thus the following clauses are generated:

$$(\neg i_1 \vee q_1) \wedge (\neg i_2 \vee q_1)$$

Finally, the clause that accounts for the case of $i_1 + i_2 = 2$ is generated as follows:

$$(\neg i_1 \vee \neg i_2 \vee q_2)$$

The generalization of this procedure can be found in Figure **??**. In the figure, Q and R represent the child nodes, while P represents the parent node. The variables within each node that correspond to a certain number in the Unary Representation are denoted by lowercase letters. The subscripts of each variable represent weights, which are relevant in the case of weighted MaxSAT. In the unweighted case, each soft clause has a weight of 1, hence each variable withing a given node represents the full range of integers beggining at the lower bound, namely 0, and its upper bound.

Formula 1: Totalizer Encoding

$$\left(\bigwedge \begin{array}{l} q_{w_1} \in Q.\text{node\_vars} \\ r_{w_2} \in R.\text{node\_vars} \\ w_3 = w_1 + w_2 \\ p_{w_3} \in P.\text{node\_vars} \end{array} (\neg q_{w_1} \vee \neg r_{w_2} \vee p_{w_3}) \right) \wedge \left(\bigwedge \begin{array}{l} s_w \in (Q.\text{node\_vars} \cup R.\text{node\_vars}) \\ w = w' \\ p_{w'} \in P.\text{node\_vars} \end{array} (\neg s_w \vee p_{w'}) \right)$$

Figure 2.9: Totalizer Encoding formulation as originally presented in [16]

As observed in the formula above, the rules for clause creation are straightforward. A clause is formed to represent the sum of two non-zero variables, involving three variables (one from each participating node). Additionally, a clause is formed when one child is ”set” to zero. In the case of the sum clause, the variable from the parent node must be the sum of the variables from the child nodes, as demonstrated in the example depicted in Figure 2.9. Conversely, when creating the clause with one child set to 0, the variables or weights of the child and parent must be equal.

This encoding is suited for both the unweighted and weighted MaxSAT problem, with the only difference being in the number of variables produced in the Binary Tree. In the unweighted case, there is a linear increase of variables - namely, each layer will produce at most $(v + w) - 1$ new variables, where v and w represent the amount of variables in the child nodes respectivelly. On the other hand, when dealing with the weighted case, in the worst-case scenario each level can introduce up to $2^(v + w - 1)$ variables due to the variation of weights [16].

Example 9: Totalizer Encoding of MaxSAT Instance

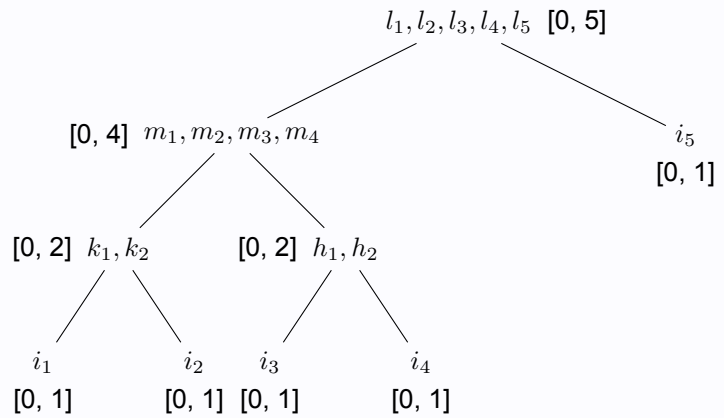

Figure 2.10: Full Totalizer Encoding Tree for the running example as shown in Figure 2.5

Figure 2.10 displays the complete Totalizer Encoding Tree for the running example of Linear Search UNSAT-SAT. The tree was constructed using the procedure described earlier and consists of 9 nodes and a total of 16 variables, including 5 relaxation variables as leaf nodes. The figure illustrates the upper and lower boundaries for each node, indicating that every number within the range, including the upper bound, can be a valid value for the right-hand side of a cardinality constraint.

Below is the pseudo code for the encoding showed in **??**.

Algorithm 4 Totalizer Encoding

Input: $child_{left}, child_{right}, parent$
Output: $clauses$

1: $clauses \leftarrow \emptyset$
2: **for** $x \in child_{left}$ **do**
3: **for** $z \in parent$ **do**
4: **if** $x_{value} == z_{value}$ **then**
5: $clauses \leftarrow clauses + (\neg x \vee z)$
6: **end if**
7: **end for**
8: **end for**
9: **for** $y \in child_{right}$ **do**
10: **for** $z \in parent$ **do**
11: **if** $y_{value} == z_{value}$ **then**
12: $clauses \leftarrow clauses + (\neg y \vee z)$
13: **end if**
14: **end for**
15: **end for**
16: **for** $x \in child_{left}$ **do**
17: **for** $y \in child_{right}$ **do**
18: **for** $z \in parent$ **do**
19: **if** $x_{value} + y_{value} == z_{value}$ **then**
20: $clauses \leftarrow clauses + (\neg x \vee \neg y \vee z)$
21: **end if**
22: **end for**
23: **end for**
24: **end for**
25: **return** $clauses$

The encoding takes as input the child nodes along with the parent node, and outputs the clauses that should be added to the SAT Oracle in order to ensure the cardinality constraint is properly represented for the MaxSAT instance. From line 2 to 8, and from line 9 to 15, the encoding shown on the right-hand side of **??** is performed, whilst from line 16 to 25 the encode of the the left-hand side of the encoding in **??** is carried out. It should be noted that this pesudocode utilizes the term value rather than weight. This is to keep in line with the example demonstrated throughout the chapter. Thus an example of x_{value} as shown in line 4, is the value of 3 of variable m_3 as shown in Figure 2.10

3

Related work

To understand how the different fields of MaxSAT solving came to be, it is pertinent to look at the history of MaxSAT solving. [22] is a literature survey that presents the state-of-the-art algorithms for MaxSAT in 2013. Its extensive overview of the history of MaxSAT shows that first algorithms were focused on stochastic local search, in which a random assignment was initially set after which variables were flipped from TRUE to FALSE and the solution was approximate rather than exact [13]. [22] continues by introducing two of the main families of interest for MaxSAT solving, the iterative approach and the core-guided approach. A more recent overview of the MaxSAT landscape can be found in [18], which discusses not only algorithms for MaxSAT solving, but heuristics can can be applied to different algorithms to improve performance. One such heuristic presented in [18] is variable ordering, which is a point of interest for the Layered Totalizer Encoding.

Linear Search belongs to the aforementioned iterative algorithms [22]. [17] is an example of an implementation of Linear Search UNSAT-SAT, which focuses on increasing the lower bound, whilst [14] contains an implementation of Linear Search SAT-UNSAT which focuses on lowering the upper bound. Though new MaxSAT algorithms have been introduced throughout the past two decades, Linear Search algorithms are still of interest in the research field. [7] introduces a Linear Search algorithm that is preceded by a core-based phase, showcasing how an iterative algorithm could benefit from information provided by a core-guided algorithm. Moreover, [7] demonstrates promising results that show that this core-boosted Linear Search performs well against other state-of-the-art solvers.

[12] introduces two approaches, namely Diagnosis Based and Encoding Based, The Diagnosis Based approach is the first core-based approach introduced for MaxSAT solving. Different versions of this core-extraction process have spawned since, such as PM1 MSU1, WPM1, [12] and further developments of these algorithms such as MSU4 [19]. The version of interest in this paper is the OLL algorithm [21], which belongs in the family of core-guided algorithms. The OLL algorithm uses the cores extracted from the SAT solver to define and create cardinality constraints and allows for these newly introduced variables to be part of the core found in the next iteration. For the purpose of this study, the implementation of OLL used from this point on is known as RC2 [15].

Cardinality constraints play an important role in MaxSAT solving, as both iterative and core-guided algorithms make use of them. An example of their significance can be found in [1], in which research shows that limiting the size of a cardinality constraint impacts the performance of the OLL algorithm. Thus, it is easy to see that smaller cardinality constraints are one of the factors that lead to a faster solving time. A secondary point of note when dealing with cardinality constraints is how to encode them such that they fit the CNF format. [27] presents two encoding principles based on sequential counters. This encoding predates the Totalizer Encoding, however, it serves as a good example of how encodings impact the number of clauses and variables. A recent overview on encoding practices for cardinality constraints is provided by [29], which compares tree-based encodings to other encodings. This survey provides a good basis to understanding why tree-based appraoches, such as the Totalizer Encoding, have gained popularity in recent years - namely, they are constantly among the top 2 performers for a given benchmark set.

The Totalizer encoding was first introduced by [5], as a way to encode Boolean cardinality constraints. This encoding builds a Totalizer tree, such that each node of the tree represents the possible

sum of its children nodes. Each node represents the possible sums with unary variables, which means there is one variable to represent each possible value of the sum being encoded. The original Totalizer Tree needed to be re-encoded at each pass over the linear search algorithm, however, [16] and [20] have improved upon the use of the Totalizer by making it more flexible and reusable by introducing unit clauses to not re-encode the full Totalizer tree.

4

Layered Totalizer Encoding

The main contribution of this paper is the Layered Totalizer, an adjustment to the construction and encoding of the Totalizer Encoding, aimed at potentially reducing execution time. The chapter begins by discussing the concept and motivation behind this new Totalizer approach, followed by a discussion of correctness, nd example, and finally the pseudocode. Additionally, the paper explores various scenarios in which the Totalizer is tested and heuristic learning is applied.

4.1. The Layered Totalizer

Chapter 2 provides an overview of the Totalizer Encoding technique, which converts a given Pseudo-Boolean constraint into CNF clauses suitable for SAT solvers. However, the resulting Totalizer Encoding, whether weighted or unweighted, often generates a large number of clauses and variables, which can potentially lead to a subpar runtime of the SAT Oracle. Therefore, the focus of the research is to explore to what extent raising the lower bound of nodes in the Totalizer tree can reduce the solution space by removing variables and clauses and fixing assignment of variables to a value.

To address the question, the systematic approach of OLL is analysed. In contrast to Linear Search, which employs a single cardinality constraint encompassing all relaxation variables at the top node of the Totalizer Tree, the OLL algorithm utilizes multiple smaller Totalizer Trees. Each of these trees represents a distinct cardinality constraint that denotes an unsatisfiable core. The improved performance of OLL, compared to Linear Search, stems from this distinction. By focusing solely on cardinality constraints that are guranteed to increase their right-hand value and generating only the necessary clauses and variables, irrelevant information is eliminated. Furthermore, the cardinality clauses produced by the OLL algorithm tend to be smaller than those generated by Linear Search, resulting in a reduced workload for the SAT sovler partially due to the smaller number of clauses and variables generated.

While the ability of OLL to construct these smaller and more concise Totalizer Trees is contingent on its knowledge of unsatisfiable cores, it is possible to borrow certain aspects of its functionality. More precisely speaking, the OLL algorithm creates smaller cardinality constraints that can potentially go on to be part of composite cardinality constraint. However, even before becoming part of a composite cardinality constraint, the original cardinality constraint is tightened. This idea of tightening cardinality constraints at various iterations is the inspiration behind the Layered Totalizer Encoding.

As presented in Chapter 2, the Totalizer Tree is built bottom-up, node by node, wherein each node represents different cardinality constraints as expressed by a set of variables. Thus it follows that the fewer variables present in a pair of child nodes, the fewer possible values the parent has to represent. Figure 2.10 displays the lower and upper boundaries for each node, and the lower boundary can be leveraged to reduce the number of variables. The objective is to tighten the lower bound, increasing it as much as possible until it reaches its maximum limit. With each raise of the lower boundary, there is one less value present in the node, and thus fewer values for any nodes in the upper layers that have the tightened node in its subtree. If the bounds on one layer are tightened, the subsequent layer in the Totalizer Tree can exclude the values that have been identified as unnecessary. As a result, the number of variables in that layer is reduced.

The process begins by relaxing the soft clauses with new relaxation variables and representing

these variables as leaf nodes in the Totalizer Tree. The leaf nodes do not require any additional clauses since they are not yet connected by cardinality constraints. Next, the pruning stage involves sequentially assuming the negation of each relaxation variable and querying the SAT Oracle. If the Oracle responds with an unsatisfiable answer, then this variable has to be set to True for the solution to be True.

To properly observe the workings of the Layered Totalizer, the example is presented once again, and its corresponding Layered Totalizer step-by-step.

Example 10: MaxSAT Instance

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$

$$I = \{y_1, y_2, y_3, y_4, y_5\}$$

$$H = \{(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_2), (\neg x_3 \vee \neg x_4), (\neg x_1 \vee \neg x_5), (\neg x_2 \vee \neg x_5), (\neg x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)\}$$

$$S = \{(x_1 \vee y_1), (x_2 \vee y_2), (x_3 \vee y_3), (x_4 \vee y_4), (x_5 \vee y_5)\}$$

$$y_1 + y_2 + y_3 + y_4 + y_5 \leq 0$$

Figure 4.1: Example of a MaxSAT problem instance with 5 variables, 7 hard clauses, and 5 soft clauses, as originally shown in [15], with relaxed soft clauses at the start of the Layered Totalizer Encoding process

As depicted in Figure 4.1, five soft clauses exist, corresponding to five relaxation variables requiring encoding in the Totalizer Tree. Figure 4.2 displays the encoding of these variables, along with their respective lower and upper boundaries. The initial steps of this process align with the description provided in Chapter 2, involving the creation of nodes to encompass the variables. However, this marks the divergence between the two encodings. Subsequently, the bound tightening process is initiated by assuming the negation of each variable individually and querying the SAT Oracle. In practice, this is done by invoking the SAT Oracle by providing not only the clauses to it, but a negative assumption of a single variable at a time.

Example 10.1: Layer 1 of Layered Totalizer Encoding

y_1	y_2	y_3	y_4	y_5
[0, 1]	[0, 1]	[0, 1]	[0, 1]	[0, 1]

Figure 4.2: The leaf-node layer, Layer 1, of Figure 4.1 as built by the Layered Totalizer Encoding procedure

The first SAT Oracle call made includes all of the hard clauses, the relaxed soft clauses, and the assumption of $y_1 = 0$, with all other variables left to take on either value. The Oracle then returns satisfiable for this particular example, which means that the boundary cannot be tightened further - that is, there is a solution that includes $y_1 = 0$. Had unsatisfiable been the answer from the Oracle, the deduction would be that there is no solution where $\neg y_1$ is set to 1, meaning y_1 has to always be set to 1 to satisfy this particular instance. The more high-level meaning of this assumption and result can be simply explained as a query to the SAT Oracle which aims to answer whether there is a solution to the MaxSAT instance in which a particular clause or a set of clauses are not violated.

This process is repeated for all of the leaf nodes present in the tree, one by one. At the end of the bound tightening procedure, for this particular problem, no bound has been tightened, thus the process of creating the second layer can begin.

Example 10.2: Layer 2 of Layered Totalizer Encoding

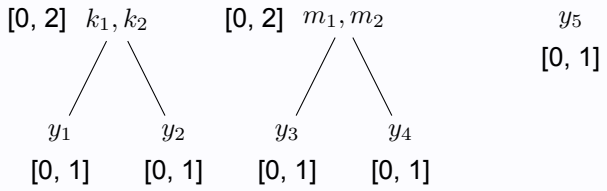

Figure 4.3: Layer 2, of Figure 4.1 as built by the Layered Totalizer Encoding procedure, before any SAT Oracle calls

Once again, the Layered Totalizer Tree currently looks like the regular Totalizer Tree because there were no nodes for which the lower boundary could be increased.

The necessary clauses between parent and child nodes are created as outlined in Chapter 2, and the first SAT Oracle call is made with the new variables shown in Figure 4.3, along with the soft clauses, hard clauses, and Totalizer clauses. Additionally, the assumption of $\neg k_1$ is provided. In this particular case, the SAT Oracle returns an unsatisfiable result, indicating that at least either y_1 or y_2 must be set to true for the MaxSAT instance to be satisfied. Consequently, the lower boundary of node k_1, k_2 is raised to 1, signifying that k_1 is always set to true. Next, a SAT Oracle call is made with the assumption of $\neg k_2$ to determine if the lower boundary can be further increased. The SAT Oracle responds with a satisfiable result, indicating that the lower bound remains at 1. The same process is repeated for the other node in the layer, node m_1, m_2, resulting in a similar increase in its lower boundary by 1. Figure 4.4 depicts the updated state of the Layered Totalizer Tree, where redundant variables have been removed and the lower bounds have been raised.

Example 10.3: Layer 2 of Layered Totalizer Encoding - Result

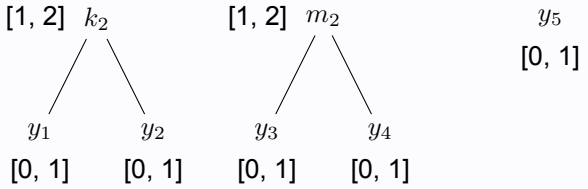

Figure 4.4: Layer 2, of Figure 4.1 as built by the Layered Totalizer Encoding procedure, after SAT Oracle calls, with tightened lower bounds

Based on the SAT Oracle's response, the lower boundary is adjusted or the next node is processed. The tree construction process continues by creating a layer of nodes and performing SAT calls with assumptions of a specific variable being set to false. Figure 4.5 illustrates that the upper-most node with variables h_3 and h_4 initially has a lower boundary of 2 and consists of only two variables. This is a result of the previous layer, where variables were removed and the lower bound was increased. This example highlights the effectiveness of variable pruning and its influence on the tree by eliminating unnecessary encoding of information.

Example 10.4: Layer 3 of Layered Totalizer Encoding

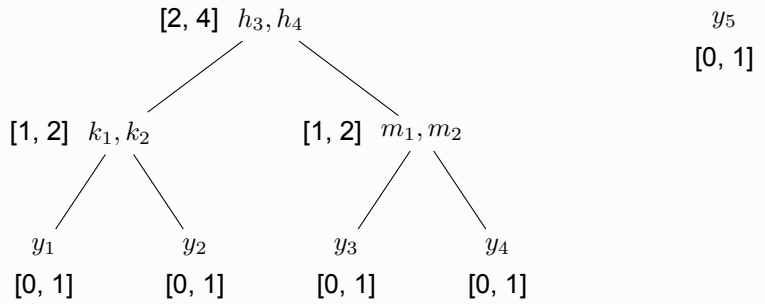

Figure 4.5: Layer 3, of Figure 4.1 as built by the Layered Totalizer Encoding procedure, before any SAT Oracle calls, demonstrating the ripple-effect of a parent node's starting lower bound when tightening is possible in the children nodes

The procedure of generating clauses for the new node and its variables is repeated, followed by a SAT Oracle call. The call results in a satisfiable answer, indicating that the lower boundary cannot be raised.

Subsequently, the final node, which serves as the root of the tree, is created as depicted in Figure 4.6. From this point onward, the process resembles that of Linear Search. Specifically, a main cardinality constraint is established to express the relationship among all relaxation variables, and each possible value on the right-hand side of the cardinality constraint is iteratively tested. Notably, two out of the six possible constraint values have already been eliminated, leading to a reduced number of SAT calls required at the top level.

Example 10.5: Layer 4 of Layered Totalizer Encoding

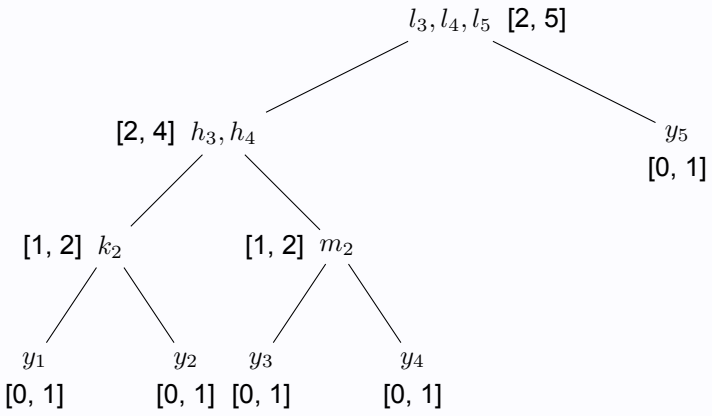

Figure 4.6: Layer 4, of Figure 4.1 as built by the Layered Totalizer Encoding procedure, before any SAT Oracle calls

A call is made to the SAT Oracle assuming $\neg l_3$, resulting in an unsatisfiable answer. As a result, the lower bound of the root node is increased by one. Subsequently, another SAT Oracle call is made assuming $\neg l_4$, which yields a satisfiable solution, representing the optimal solution. As these calls follow the regular flow of Linear Search, the final result is not displayed in Figure 4.6, instead it is left with a lower boundary of 2 at the topmost node to show that its lower boundary is increased strictly due to the bounding in lower layers.

This encoding approach involves a trade-off. It requires more SAT Oracle calls, but it leads to potentially faster results from the Oracle due to a reduced number of clauses and fixed variables.

It is important to note that it is theoretically sound to remove any variable or clause that is fixed to the value of 1 or TRUE in a given MaxSAT instance. This is because clauses containing variables set

to true will always evaluate to TRUE. Removing these variables and clauses is equivalent to fixing all the removed variables to always be TRUE and including them in the solver.

Algorithm 5 Layered Totalizer Encoding

```
   Input: H, S
   Output: cost
1: S_relaxed, I ← relax'(S)
2: node_layers ← ∅
3: while true do
4:     if node_layers is ∅ then
5:         layer ← initial_layer(I)
6:     else
7:         layer ← create_layer(node_layers.last)
8:     end if
9:     for node in layer do
10:        for variable in node do
11:            result ← SAT(H, S, variable = 0)
12:            if result is UNSAT then
13:                node.lower_bound ← node.lower_bound + 1
14:            else
15:                break
16:            end if
17:        end for
18:    end for
19:    node_layers ← node_layers ∪ layer
20: end while
21: return node_layers.last.lower_bound
```

Algorithm 5 shows the pseudocode for the Layered Totalizer Encoding. The first notable difference between this algorithm and those previously presented is the change in the functionality of the relax function. The only change is that the current version of relax, named relax', returns the set of relaxation variables as well. As seen in line 2, a structure is initiated to hold all of the layers of the Layered Totalizer as an easy means to access them. If the first layer of the Layered Totazlizer is being created, then the set of relaxation variables, I, are used to create the bottom-most nodes. Conversely, if at least one layer already exists then the last layer created is used as the basis for the creation of the next layer. The method create_layer itself simply follows the formula shown in Figure 2.9. After a new layer has been created, the boundary-raising procedure begins as seen in lines 9 through 18. As previously explained, for each node in a layer, and each variable in said node, a SAT call is invoked with the variable of question passed as an assumption with a value of 0. If the answer from the Oracle is UNSAT then the lower boundary of the given node can be raised, otherwise the algorithm can move on to testing the variables of another node in the layer. At the end of the boundary-raising section, the newly created (and bounded if possible) layer is added to the set of all layers. Finally, the algorithm returns the lower boundary found at the top-most node.

4.2. Variable Ordering for Better Intermediate Constraints

Variable Ordering can impact how well an algorithm performs due to several reasons. In the terms of the Layered Totalizer Encoding, Variable Ordering can help find good intermediate constraints at lower levels and the sooner a lower-level constraint is tightened, the fewer variables, and by extent clauses, are created at upper levels.

As was shown in the previous section through the running example, the removal of a variable at a node in level i leads to a decrease of its parent's variables in level $i + 1$ by 1. Clause-wise, however, because a variable has been eliminated, there is a decrease of $n+1$ clauses, where n is the size of the set of variables in its sibling node and 1 for the encoding of itself along with the variable in the parent node that represents the same sum. The formula can be further generalized as

$$v * (n - 1)$$

where v is the total number of variables removed from the child node.

Thus it is easy to see that the bigger v is, the more variables and clauses are removed at a given level, thus positively impacting the upper levels by having fewer variables to encode. However, how many variables are removed at a given layer is partially linked to how the variables are ordered in the leaf nodes.

In the example depicted in Figure 4.6 and upon rearranging the leaf nodes, different lower bounds can potentially be obtained. This phenomenon is attributed to the concept of unsatisfiable cores introduced in Chapter 2. When a set of soft clauses forms an unsatisfiable core, the corresponding cardinality constraint's right-hand side must be increased to at least 1, and potentially more depending on the problem's structure. On the other hand, if two soft clauses do not form an unsatisfiable core, the right-hand side of their cardinality constraint remains at 0 in the conswcutive layer of the Layered Totalizer. The size of the unsatisfiable core directly affects how quickly said core is encountered and bounded in the Totalizer Tree, leading to a ripple effect in the upper layers of the tree.

Unfortunately, determining an unsatisfiable core is not an easy task, nor is ordering the variables in an approximately optimal manner even if the cores are given to us. To that extent, this section introduces heuristic approaches to ordering the leaf nodes in the Layered Totalizer Encoding tree in order to observe how a different variable ordering can impact the runtime of the algorithm, and to potentially explore what a ”good” variable ordering is.

4.2.1. Most Frequently Occurring Variable Ordering

The first Variable Ordering heuristic of interest is the Most Frequently Occurring Variable. As the name suggests, the variables are ordered from most frequently occurring to least, as observed in the hard clauses of a given MaxSAT instance.

Firstly, all variables are ordered from most frequently occurring to least, based on their frequency in the hard clauses. The variable order serves as the foundation for the arrangement of the leaves in the Layered Totalizer Tree. The most frequent variable is considered first, which means the soft clause containing the most frequently occurring variable has its relaxation variable placed in the left-most position in the Layered Totalizer tree. This ordering follows for each soft clause.

The intuition behind this order is straightforward. It is assumed that a variable which appears frequently in hard clauses is more likely to be involved in an unsatisfiable core along with another variable that also appears frequently in hard clauses. While this assumption may not always hold true, it has the potential to be valid in certain instances of MaxSAT.

Unfortunately, this heuristic has a downside as well, because if it holds that frequently occurring variables form an unsatisfiable core, then it leads that infrequently occurring variables do not form a core. In fact, by the outlined logic, it is more probable that a frequently occurring variable forms a core with an infrequently occurring variable, rather than two infrequent variables. Then, if this line of reasoning holds up for certain MaxSAT instances, a possible outcome is a Layered Totalizer Tree which has a left subtree with tightly bound nodes, whilst the right subtree is left with loosely bound nodes, if bounded at all. This is not necessarily true - there is no guarantee that the most frequently occurring variable is part of any unsatisfiable core - however, it is pertinent to determine whether this heuristic Variable Ordering outperforms not ordering the leaf nodes in any manner.

4.2.2. Most Frequently Occurring Variable Pair Ordering

A closely related Variable Ordering to that of the Most Frequently Occurring Variable Ordering is to look at the Most Frequently Occurring Variable Pair Ordering [10]. This approach has the same benefits and reasoning behind it as the single-variable version, as well as the same pitfalls. The goal behind this alternate frequency-based approach is to determine whether an interaction of variables in the hard clauses, and its consequent frequency, could be more significant in variable ordering than considering each variable's frequency in isolation.

The procedure for finding the most frequently occurring pair of variables is much the same as the single variable case - the pair frequencies in the hard clauses is determined and these frequencies are used to order the relaxation variables in the lowermost level of the encoding tree.

4.2.3. Core-Based Variable Ordering

As discussed at the beginning of this section, an intermediate constraint in the Layered Totalizer Encoding Tree can only have its lower boundary tightened if it represents an unsatisfiable core. By ordering all leaf nodes in a specific manner that guarantees their interaction with other relaxation variables higher up in the tree, it becomes possible to successfully tighten lower bounds. This especially holds if the unsatisfiable cores used to order the relaxation variables are small, meaning that nodes in lower levels of the encoding trees would be tightened, which is a desired outcome as previously discussed.

However, several adjustments need to be made before ordering the leaf nodes based on unsatisfiable cores. Firstly, obtaining multiple cores is necessary to establish the proper order. Simply obtaining one core through a single call to the OLL algorithm is insufficient. Multiple calls are needed to gather as many variable relations as possible, ensuring the accuracy of the ordering. This requires making multiple calls to OLL or another core-based algorithm. However, one challenge arises from the fact that OLL may potentially solve the problem before enough cores are generated for creating the variable order.

Secondly, while OLL has been beneficial in improving the Totalizer Encoding, it is not the only algorithm for core generation. Another core-based MaxSAT algorithm called the Implicit Hitting Set Algorithm (IHS) comes into play. Despite the possibility of IHS also solving the instance prematurely, its approach to core generation aligns better with the objective [6]. The primary reason behind the use of IHS over OLL is simple - a core generated by OLL could contain a cardinality constraint or a variable not present in the original formulation of the problem. On the other hand, IHS generates cores only containing the variables found in the original problem. Though IHS is an impressive and popular algorithm in its own right, as it is being used as a black-box variable generator for this study, it will not be discussed in detail beyond its ability to generate unsatisfiable cores.

The third and final challenge in ordering the relaxation variables to form cores quickly in the encoding tree is the consideration that a variable can exist in multiple cores. This raises the question of determining which core is more desirable and how to make that determination.

For the first obstacle, specifically the issue of core generation potentially solving the problem for the core-based algorithm itself, the decision has been made to exclude instances solved by IHS during the core-generation phase from being considered as solved by the Layered Totalizer. If the problem is solved by IHS within the given time frame, it is considered an invalid instance for the Core-Based Variable Ordering. Although this approach results in the rejection of many solved instances, the focus is not solely on the speed of performance of a core-ordered Layered Totalizer, but rather on determining whether any instances not solved by another Variable Ordering method can be solved.

The final identified obstacle involves a single variable being present in multiple unsatisfiable cores simultaneously. In this research, the approach to variable ordering is chronological. Specifically, if a variable x_i appears in a core for the first time, its position in the Variable Ordering is determined by that core. If the same variable x_i appears in a subsequent core, it is disregarded, and its original position in the Variable Ordering is maintained.

Lastly, the generation process of the Core-Based Variable Ordering is discussed. The procedure begins with the execution of the IHS algorithm, which is limited by a specified time duration. Termination occurs either when the time limit is reached or when an optimal solution is found, which is disregarded. The first produced core is the starting point for the ordering of the relaxation variables, meaning that if a soft clause appears in the core, its relaxation variable is placed in the ordering. For any next core, if it contains a soft clause that has not been encountered so far, its relaxation variable is placed in the variable order. Thus, if a potential core trace is $[\{(x_1), (x_2), (x_5)\}, \{(x_3), (x_4), (x_5)\}]$, the relaxation variable ordering would amount to $\{y_1, y_2, y_5, y_3, y_4\}$ where each y_i relaxation variable responds to the soft clause of (x_i).

4.3. Layered-Linear Hybrid

The main goal of this research is to explore how the Layered Totalizer Encoding performs in comparison to Linear Search - however, it is important to also look at how they work in tandem. The assumption is that the Layered Totalizer Encoding outperforms Linear Search for certain instances, though it is also expected for the inverse to hold. Moreover, there might be some instances only solved by only one of the algorithms but not both.

A hybrid approach would present a new perspective - namely, what would happen if certain layers of the Totalizer Tree are built using the layered approach and others by the regular Totalizer Encoding. As

established, raising the lower bound in the lower layers of the Totalizer has a ripple-effect for the upper layers, however, raising the lower bound in the upper parts of the Totalizer Tree would only impact the small number of layers above it and might not play a role as important as bounding at the start.

This theory can be tested by introducing the Hybrid Totalizer Encoding, in which a certain amount of layers are built through the Layered Totalizer Encoding and the rest with the regular Generalized Totalizer Encoding, followed by Linear Search. The implementation of this idea is as straight-forward as it appears; several layers are set to be built and have their lower bounds increased, whilst the rest are built as described in Chapter 2. After the full Totalizer Tree is built the standard Linear Search UNSAT-SAT algorithm is used to find the optimal solution.

4.4. Layered Totalizer in OLL

As noted throughout this work, the OLL algorithm serves as a theoretical inspiration for the Layered Totalizer Encoding. As such, it is pertinent to explore how the Layered Totalizer performs in the RC2 implementation of the OLL algorithm. When using the newly proposed encoding in Linear Search the improvement can be seen through the lower overall solve time at the price of more SAT calls.

Going back to the discussion in Chapter 2, an unsatisfiable core can be of any size, and a minimal core implies the existence of a core for which no proper subset of the elements forms a core itself. Core minimization is an area of research of its own within core-based algorithms that aims to find these minimal cores and use them as the building blocks for MaxSAT solving. However, finding a minimal core is an expensive task, which grows more complex the bigger the discrepancy is between the core returned by the SAT Oracle and the minimal core itself.

Though several different core minimization techniques exist and are used, a potential contribution the Layered Totalizer Encoding could have to the OLL algorithm is identifying minimal cores. To be more precise, given an unsatisfiable core, as the Layered Totalizer Encoding builds the Totalizer Tree layer by layer, it could potentially raise the bounds of the inner nodes as it does in Linear Search. As has been discussed, only a node representing an unsatisfiable core can have its bound raised. Hence, given an unsatisfiable core and its Totalizer Tree, if a given inner node can have its bound raised, it implies that this proper subset of variables is a core within itself. This does not necessarily mean that this proper subset is a minimal core itself, rather that the core represented at the top-most node is not minimal.

To explore this possibility, the Layered Totalizer Encoding has been given to the RC2 algorithm instead of the Generalized Totalizer Encoding. What this means is that for each cardinality constraint devised during the main phase of the RC2 algorithm the encoding used is the Layered Totalizer Encoding. The Totalizer Tree is built following the procedure displayed in Algorithm 5. In practice, this leads to more SAT calls during the RC2 algorithm, with a significant portion of those SAT calls occurring during the Totalizer Tree encoding. This in turn means that the OLL algorithm will not be able to process the unsatisfiable cores from those SAT calls. Instead, the purpose of these SAT calls is to tighten the lower bounds on the nodes of the Totalizer Tree.

As implied, this procedure can be used as an alternative to existing core-minimization procedures. Comparison among different core-minimization algorithms is left for future work, as this paper only aims to determine if an OLL algorithm can benefit from the Layered Totalizer Encoding in any way. The work from [7] is in close relation to this idea, and seeing as there is precedent that unsatisfiable cores and Linear Search can work in a hybrid manner, a Layered Totalizer Encoding OLL algorithm is a promising premise.

5 Methodology

Due to the nature of the research topic of this paper, some statistical testing can be applied to deduce how well the Layered Totalizer Encodingperforms. However, much of the data gathered is better examined through a descriptive approach. To that extent this section presents the data gathering process, the data processing procedure, as well as the meaning of the data analyzed and how each characteristic is examined in the Chapter 6 is presented.

5.1. Tools and Libraries

In order to have a consistent result and remove as many confounding variables as possible, the Generalized Totalizer Encoding as well as the Layered Totalizer Encoding were both implemented in Python. Both Totalizer Encodings share the same interface, meaning that methods and data structures were kept as similar as possible as to avoid potential noise in the data produced by using different tools.

Moreover, for each algorithm, the SAT Oracle in use is Glucose-4, as provided by PySAT [14]. As PySAT is a Python library, it is worth mentioning that the implementation of any of the Totalizers is dependent on the speed of Python's functionality itself. An example of Python's impact on the runtime is its slow read-in time when compared to languages such as C and Rust.

The Variable Ordering heuristics were implemented from scratch as well except for the Core-Based Variable Ordering. Because PySAT offers its own IHS implementation [14], its interface is used to generate the needed cores.

5.2. Data

The data of interest for this research is the performance of the Layered Totalizer Encoding in comparison to Linear Search, as well as the different heuristic approaches presented in Chapter 4. To collect this data, all algorithms are given the same set of MaxSAT instances. The MaxSAT instances given to these Totalizer Encodings are all from the MaxSAT 2022 competition [4]. Then, the data collected from each instance is the time it took for each instance to be solved in seconds.

The next piece of information of interest is the number of variables and clauses removed per instance in the Layered Totalizer Encoding. This data is gathered at the end of execution by looking at only the variables and clauses generated by the Layered Totalizer Encoding and Linear Search.

Finally, the correlation or pattern between the times it takes to create the layers for the Layered Totalizer Encoding is explored.

5.3. Tests

5.3.1. Devised Calculations

The goal is to compare all relevant Totalizer Encodings to gain a broader understanding of how to employ and improve the Layered Totalizer Encoding. To that extent, different tests were performed on the gathered data, and the results are interpreted.

The heart of this research is the potential speedup in seconds provided by the Layered Totalizer Encoding over Linear Search. To that extent, the first point of interest is the speedup factor obtained

by the Layered Totalizer Encoding. The speedup factor is defined as the percentage of time by which the Layered Totalizer is faster than Linear Search. The deduced formula is shown below.

$$SF = \begin{cases} (1 - \frac{t_{lay}}{t_{lin}}) * 100 & \text{if } t_{lay} \leq t_{lin}, \\ -(1 - \frac{t_{lin}}{t_{lay}}) * 100 & \text{if } t_{lay} > t_{lin} \end{cases}$$

The piecewise function defines the behaviour of the Speedup Factor such that the values obtained denote a positive value between 0 and 100 if the Layered Totalizer Encoding produces the faster runtime, and a negative value between 0 and -100 if the runtime of Linear Search is faster. This factor can be averaged over all instances in the benchmark set to determine the overall faster algorithm and can be used in combination with other metrics to find a correlation.

However, it is not only pertinent to discover if the Layered Totalizer outperforms Linear Search, but to also determine why it outperforms Linear Search and under what conditions. As discussed in Chapter 4, the Layered Totalizer has the potential to eliminate certain variables from being considered, which leads to the elimination of clauses as well. The question of interest, then, is whether this variable and clause elimination is correlated to the performance of the Layered Totalizer, and if yes, what can be concluded based on the result in terms of performance.

The number of variables created with a given Generalized Totalizer Encoding is a fixed number depending on the number of soft clauses. Conversely, the number of variables in the number of variables in the Layered Totalizer Encoding depends not only on the number of soft clauses, but on how much the lower boundary can be tightened, meaning there is no formula or generalization as to how many variables are within a given Layered Totalizer Encoding. The difference in variables per Totalizer Encoding can be expressed as the Variable Decrease Factor, which looks at the percentage of variables that are not present in the Layered Totalizer Encoding versus the Generalized Totalizer Encoding. The calculation for the Variable Decrease Factor can be defined as:

$$VDF = (1 - \frac{v_{lay}}{v_{lin}}) * 100$$

The factor is similar to that of the Speedup Factor as the goal is to find a correlation between these factors. Similarly to the SF calculation notation, v_{lay} denotes the amount of variables created by using the Layered Totalizer Encoding and v_{lin} the amount of variables generated by the Linear Search algorithm which relies on the Generalized Totalizer Encoding.

As noted, a decrease in variables indicates a decrease in clauses. To that extent, the Clause Decrease Factor is introduced, and defined as:

$$CDF = (1 - \frac{c_{lay}}{c_{lin}}) * 100$$

In order to determine whether there is a correlation between the aforementioned factors, a statistical analysis in the form of Linear Regression is performed in Chapter 6. The correlation of interest is the one between the Speedup Factor and VDF and CDF respectively. The aim of this Linear Regression analysis is to determine the strength of the correlation and to extract a possible recommendation on when to use the Layered Totalizer Encoding based on the regression formula. As can be observed, both the VDF and CDF function follow the same structure as the SF calculation, bar the condition that the layered factor is bigger than the linear factor, as the number of both variables and clauses can only decrease when using the Layered Totalizer approach.

The third and final aspect of interest with respect to the performance of the Layered Totalizer Encoding is the number of SAT calls per layer as well as the time spent to encode and tighten each layer. This metric provides insight into how the layers impact each other in terms of potential information gain. Moreover, on the basis of the SAT calls made per layer and time spent on said layer, a potential baseline could be formed to determine when to switch from the Layered Totalizer Encoding to the Generalized Totalizer Encoding followed by Linear Search. The assumption regarding this metric is intuitive and goes as follows: if the layer l_i has made a significant amount of SAT Oracle calls and tightened its nodes' bounds, then its successor, layer l_{i+1}, would potentially need less time to be encoded, due to the amount of values removed by the previous layer. The reverse notion also holds - if layer l_i makes few SAT calls per node, meaning the nodes' lower boundaries are not significantly tightened if at all, the successor layer l_{i+1} will potentially take longer to perform its bounding.

To be able to analyze any potential connection between two consecutive layers in a Layered Totalizer Encoding, it is pertinent to define a relation between a given layer, the number of SAT calls, and the time spent in encoding said layer. Though it might be intuitive to simply average over the time spent on layers l_i and l_{i+1} and then simply find the ratio between the two averages, due to the very different structure of the benchmark instances, this calculation would not provide any deeper knowledge or understanding for a randomly selected MaxSAT instance. Instead, the SAT Call Cost ratio is introduced, which looks at the average cost of a SAT call, in seconds, of a given layer.

$$SCC = \frac{t_l}{n_s}$$

The SCC ratio will then be used to plot different instances and to determine how the cost of a SAT call changes for a given instance or family of instances. The formula above speicifies how the average cost per layer is calculated for a given layer. Namely, t_l is the total time in seconds it took for a given layer to go through all SAT calls instantiated at that level, whilst n_s is the number of SAT calls performed on said level.

6 Results

This chapter of the paper focuses on analysing the results as described in Section Chapter 5. The results can be summarized as overall positive for the Layered Totalizer Encoding approach. The section details the different tests and comparisons conducted between the presented versions of the Layered Totalizer versus Linear Search, the analysis of the heuristic approaches, and the combination of the Layered Totalizer Encoding and the OLL algorithm. In brief, for all tests, the Layered Totalizer Encoding outperforms Linear Search in terms of both speed, amount of instances solved, and amount of unique instances solved. The heuristic approaches show some deeper insight into the structure of the problems and the behaviour of the Layered Totalizer under different circumstances, though they do not show a significant increase in instances solved - rather they provide certain unique solutions and different runtimes. Finally, incorporating the Layered Totalizer Encoding into the OLL algorithm has shown itself as the overall leader of any other algorithm presented, solely in the number of instances solved.

6.1. Unweighted Layered Totalizer Encoding and Linear Search

The first point of interest in this research is whether the Layered Totalizer Encoding provides an improvement over Linear Search UNSAT-SAT. To this extent, as explained in Chapter 5, 581 benchmark instances are tested, limited to 3600 seconds of execution time, for both algorithms with the results in Figure 6.1.

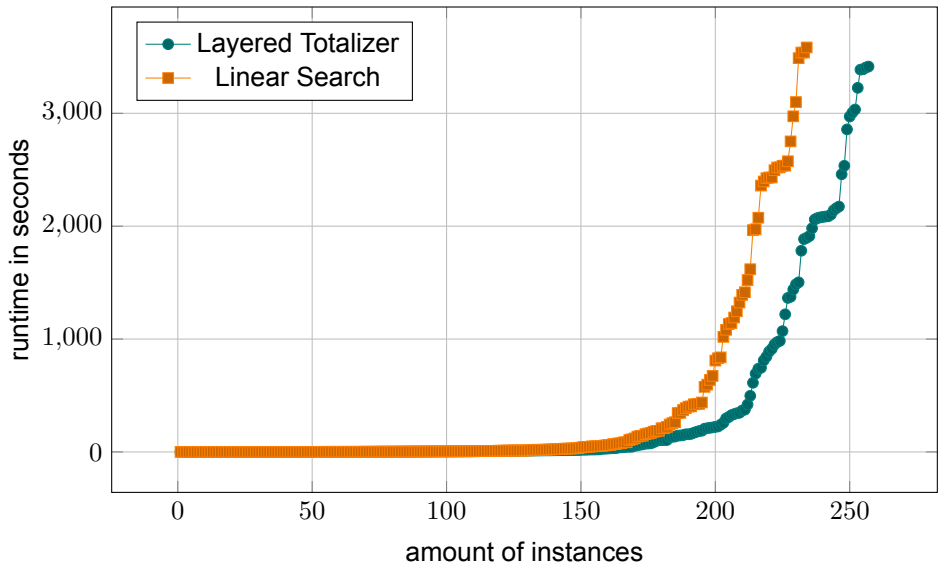

Figure 6.1: Cactus plot showing the performance of the Layered Totalizer Encoding against Linear Search

As can be seen in Figure Figure 6.1, the Layered Totalizer first and foremost solves more instances than LUS. Overall, the Layered Totalizer Encoding algorithm solved 257 instances, while Linear Search solved 234. Out of these solved instances, 231 instances were solved by both algorithms, with Layered Totalizer Encoding solving an additional 25 unique instances, and Linear Search solving 3 unique instances. Out of these 231 shared instances, the Layered Totalizer Encoding solved 167 instances faster, with an average speedup of 41%. Conversely, 63 problems were solved faster by Linear Search in comparison to the Layered Totalizer, with an average gain of 46%over the Layered Totalizer. The 3 unique instances solved by Linear Search provide examples of situations in which the Layered Totalizer Encoding could run into problems. Namely, in instances where little bounding takes place in lower layers and SAT calls grow more expensive as each additional layer is built, by the time the Layered Totalizer Encoding reaches the last layer the time might have run out. On the other hand, the Linear Search algorithm directly begins these expensive calls on the final layer and as such can solve the problems in the 3600 second time frame.

Based on the collected and analyzed data, it is evident that the Layered Totalizer demonstrates superior performance compared to Linear Search in over 70% of the tested problems. On average over all instances, the Layered Totalizer Encoding showed a 17% faster runtime. Additionally, the Layered Totalizer exhibits a higher capability to solve unique problems compared to Linear Search. These findings indicate that the Layered Totalizer algorithm is a strong contender against the Linear Search algorithm. Furthermore, there is potential for combining these two algorithms to tackle problems that neither algorithm can solve individually, offering even more promising prospects.

6.2. Weighted Layered Totalizer Encoding and Linear Search

Similarly to the Unweighted case, the Weighted Layered Totalizer Encoding and Weighted Linear Search UNSAT-SAT were tested against 563 benchmark instances provided by the MaxSAT competition. Figure Figure 6.2 shows the cactus plot that shows how both algorithms performed with respect to the above-mentioned benchmarks.

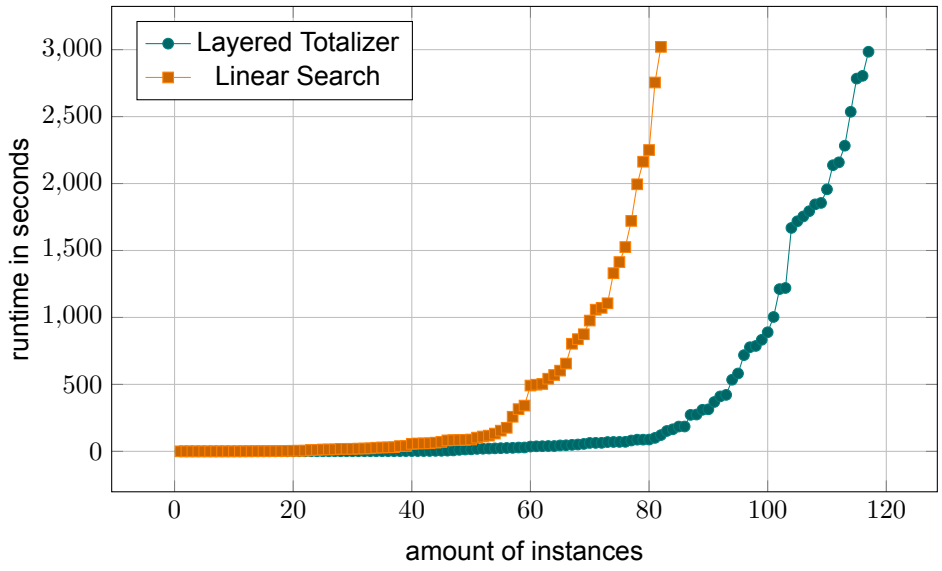

Figure 6.2: Cactus plot showing the performance of the Weighted Layered Totalizer Encoding against Weighted Linear Search

Figure Figure 6.2 displays the cactus plot illustrating the performance comparison between the Weighted Layered Totalizer and Weighted Linear Search algorithms. Once again, it is evident that the Layered Encoding surpasses the Linear Search algorithm.

Among the provided problem instances, the Weighted Layered Totalizer successfully solved 116, while Weighted Linear Search solved 82, with an overlap of 80 instances. This indicates that the Weighted Layered Totalizer encoding solved 37 unique instances, whereas Weighted Linear Search only managed to solve 2. In the 80 overlapping instances, the Weighted Layered Totalizer Encoding outperformed Weighted Linear Search in 49 problems, achieving an average runtime speedup of over

50%. Conversely, in cases where Weighted Linear Search solved the problems faster, its average runtime was 29% faster than that of the Weighted Layered Totalizer.

Although the weighted and unweighted instances exhibit differences that affect the speed of the algorithms, it is worth noting that the Weighted Layered Totalizer demonstrates a larger speedup compared to the unweighted version. Additionally, when considering the problems that were solved faster by Linear Search, it can be observed that in the weighted instances, Weighted Linear Search achieved a speedup of approximately 30%, while unweighted Linear Search achieved a speedup of 50%. This discrepancy indicates that in weighted instances, even when the Weighted Layered Totalizer Encoding is slower than Weighted Linear Search, the negative impact on performance is less pronounced compared to the unweighted case. This outcome was anticipated due to the exponential expansion of the Totalizer Encoding, which results in a greater number of variables in the weighted instances encoding. Consequently, the Weighted Layered Totalizer has a more substantial positive impact and a comparatively smaller negative impact on performance. It should also be noted that the weighted algorithms solved less than the unweighted ones, meaning that some of the discrepancy could be due to this difference.

6.3. Relation Between Unweighted Variables and Clauses, and Time

To grasp why the Layered Totalizer Encoding performs better on a majority of instances as well as how the number of variables and clauses impacts the solving time, this section delves into the aforementioned ratios that relate speedup and variable and clause decrease, respectively.

First, a look into variable decrease and speedup as shown in Figure 6.3.

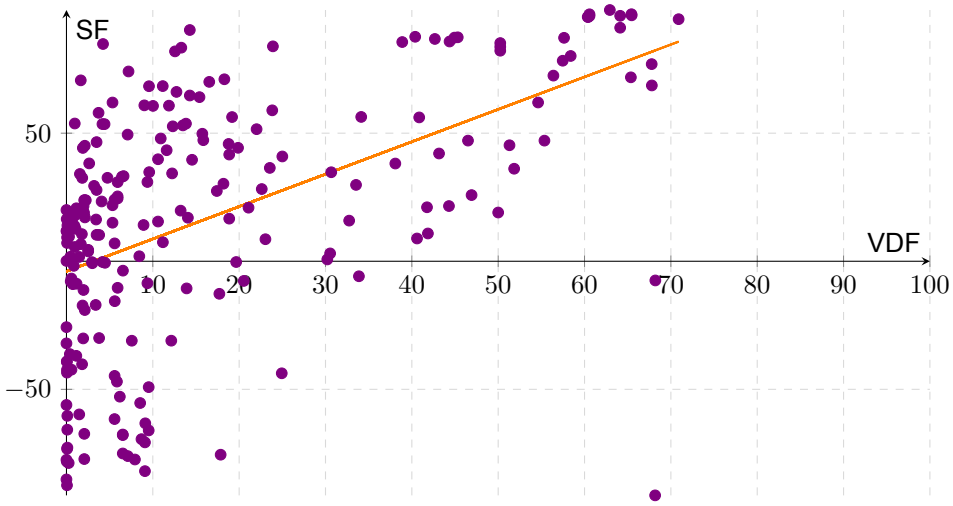

Figure 6.3: Linear Regression Analysis and scatter plot showing the positive linear relationship between the Variable Decrease Factor and Speedup Factor, named VDF and SF respectively, where each dot represents a singular instance solved by both Linear Search and the Layered Totalizer Encdoing

The graph above shows the scatter plot of the relation between the Variable Decrease Factor and Speedup Factor as described in Chapter 4.

The relation between these two factors shows whether variable decrease and solving time are correlated. More precisely, the goal is to discover by how much, and if at all, the bounding of inner nodes in the Totalizer Tree is responsible for the faster solving time of the Layered Totalizer.

To that extent, the scatter plot in Figure 6.3 underwent a Linear Regression Analysis, which determines the correlation between two variables and its strength. This analysis provides several parameters, such as the p-value, sum of squares, as well as the regression line equation, which all describe the strength of the relation between the two variables.

In this specific case, the linear regression equation explains how much speedup can be obtained depending on how many variables have been eliminated. The Linear Regression analysis can be seen in Table 6.1.

	Equation	P-value	X-intercept	R^2
Linear Regression	$\hat{(y)} = 1.2651x - 3.956$	0.0001775	3.127	0.2757

Table 6.1: Breakdown of statistics obtained through Linear Regression

Table 6.1 depicts several metrics produced by the Linear Regression analysis. The first entry in the table is the equation which describes the relation between the two variables. Namely, what the equation conveys is that for each 1% increase in the Variable Decrease Factor, there is a 1.2651% increase in the Speedup Factor. In essence, the equation helps predict and explain how eliminating variables through the Layered Totalizer Encoding positively impacts the runtime. The p-value shown in the table represents the significance of the results, meaning this value determines whether the results are statistically significant enough to draw conclusions from. Per convention, a value smaller than 0.05 indicates a statistically significant result, and as the p-value in Table 6.1 is less than 0.05, the correlation between decrease in variables and decrease in solving time is statistically significant. The X-intercept represents the predicted value of x when $\hat{y}$ is set to 0. This value is 3.127, which implies that if the decrease of variables is less than 3%, there is no guarantee that the Layered Totalizer Encoding will outperform Linear Search. This is in line with the original theoretical reasoning presented in Chapter 4 - that in a worst-case scenario where none or almost none of the bounds can be raised, the Layered Totalizer Encoding could perform worse due to the extra SAT calls with no benefit. Finally, the R^2 value describes the strength of the causative relationship between the two variables. The linear regression model yielded an R-squared value of 0.28, suggesting that approximately 28% of the variance in Y can be explained by changes in X. While this value indicates a relatively weak relationship between the two variables, it is still significant given the right-tailed distribution of the data as well as the large sample size. The moderate R^2 result suggests that there are other variables at play that impact the runtime of the Layered Totalizer, which is to be expected. These other factors are significant aspects of the solving process, such as the original size of the MaxSAT instance, but also include more peripheral factors, such as processor speed and coding language.

Aside from the regression results demonstrating the correlation and moderate causation between the Variable Decrease Factor and Speedup Factor, Figure 6.3 also provides an insight into the likelihood that the Layered Totalizer Encoding will outperform Linear Search after a certain point of variable elimination. Namely, for $x > 20$, meaning 20% of variables have been eliminated, there are only five shown instances for which Linear Search performs better, versus the majority of instances for which the Layered Totalizer has outperformed. This information could lend itself to potentially devising a hybrid Layered Totalizer to Linear Search algorithm which changes algorithm depending on how many variables have been eliminated.

Figure 6.4 shows the relation between the Clause Decrease Factor and Speedup Factor. A first observation on this graph is its similarity to Figure 6.3, which is to be expected seeing as the number of clauses created depends entirely on the variables present in a given child node. In fact, the impact of removing variables is potentially better showcased in this figure, seeing as there are multiple data points mapped close to the x-axis value of 100%, indicating that with enough variables removed a very significant decrease in clause number is observed.

Similarly to Figure 6.3, the more clauses are removed by using the Layered Totalizer Encoding approach, the more likely it is that the Layered Totalizer will outperform Linear Search.

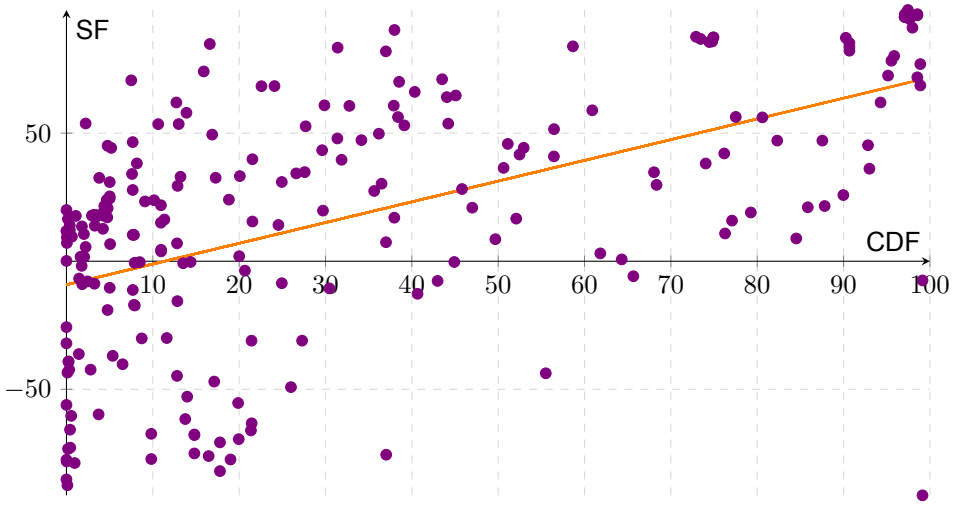

Figure 6.4: Linear Regression Analysis and scatter plot showing the positive linear relationship between the Clause Decrease Factor and Speedup Factor, named CDF and SF respectively,]where each dot represents a singular instance solved by both Linear Search and the Layered Totalizer Encdoing

As was done for Figure 6.3, the Linear Regression analysis was carried out leading to the following table:

	Equation	P-value	X-intercept	R^2
Linear Regression	$\hat{(y)} = 0.8094X - 9.1929$	0.00005551	11.3579	0.3029

Table 6.2: Breakdown of statistics obtained through Linear Regression

The table presents results similar to those of Table 6.1, which is to be expected as the number of clauses is directly impacted by the number of variables in the Totalizer Tree. The linear equation shows that for every 1% increase of the Clause Decrease Factor means a 0.8094% increase in the Speedup Factor. The p-value is less than the conventional factor of 0.005, meaning that the relation between the clause decrease and solving time decrease is statistically significantly correlated. The X-intercept shows that the Layered Totalizer is more likely to outperform Linear Search if more than 11.3579% of the clauses have been eliminated. The R^2 is slightly higher than that of the relation between the Variable Decrease Factor and Speedup Factor. There is a possible indication here that the amount of clauses is more significant to the runtime than the amount of variables, however, to decrease the number of clauses, the number of variables needs to be decreased; hence this conclusion does not give new insight into how to improve the algorithm.

Once more, it is observable that after a certain X-axis value, namely $x > 30$, most data points are above $y = 0$, meaning that if 30% or more clauses are eliminated by the Layered Totalizer Encoding, it has a high likelihood of solving a given instance faster than Linear Search.

6.4. Relation Between Weighted Variables and Clauses, and Time

A Linear Regression analysis was also carried out on the weighted instances of MaxSAT with its results displayed below. As the weighted instances were fewer in number with regard to how many were solved by either Linear Search or the Layered Totalizer Encoding, the results are not as reliable. Namely, due to the amount of data points, this data set does not pass the Shapiro-Wilks test of normality [25]. This test checks how close the data is to having a Normal Distribution, and the results of this test indicate that the data cannot be considered normally distributed. To that extent, all of these results can and should be re-examined to determine whether the conclusion are as reliable as the dataset provided in the previous section.

Figure 6.6 displays the scatter plot of the gathered data with respect to the Variable Decrease Factor and Speedup Factor. The figure also shows the linear equation obtained by the Linear Regression analysis.

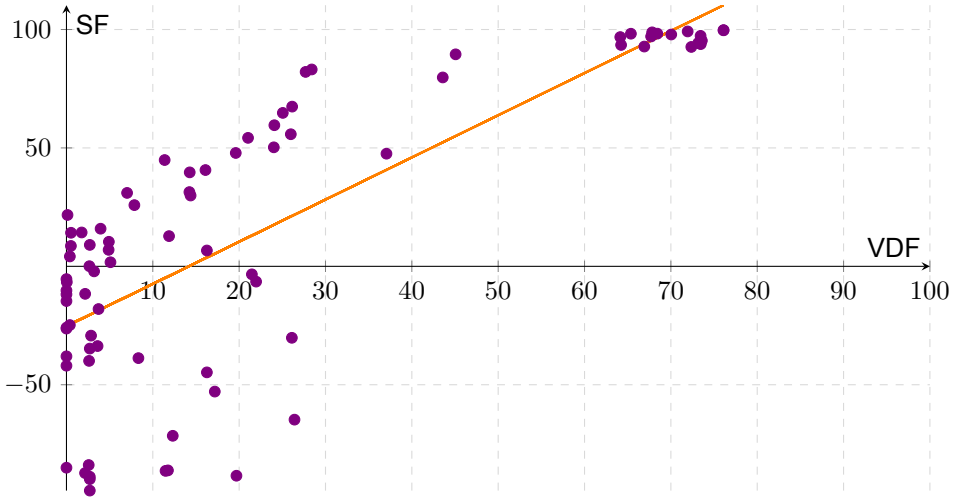

Figure 6.5: Linear Regression Analysis and scatter plot showing the positive linear relationship between the Variable Decrease Factor and Speedup Factor, named VDF and SF respectively for the weighted instances, where each dot represents a singular instance solved by both Linear Search and the Layered Totalizer Encdoing

Much like the unweighted case, these results indicate a positive linear correlation which indicates that the variable decrease obtained by the Layered Totalzer Encoding has a positive impact on the runtime of the algorithm. In fact, partially due to the smaller sample size, this figure might visually indicate a stronger linear relation. Table 6.3 breaks down the important observations of the data.

	Equation	P-value	X-intercept	R^2
Linear Regression	$\hat{(y)} = 1.7807X - 25.2444$	0.00002269	14.1769	0.6047

Table 6.3: Breakdown of statistics obtained through Linear Regression

The linear equation demonstrates that for every increase of 1% of the Variable Decrease Factor there is a 1.7807% increase in the Speedup Factor for the weighted MaxSAT instances. The data obtained is statistically significant according to convention. The Layered Totalizer Encoding more definitely outperforms Linear Search if more than 14% of variables have been eliminated and the causative relationship between variable decrease and runtime is stronger with an R^2 of 0.6047. This discrepancy between the weighted and unweighted cases could be due to the difference in variable growth per node, however, due to the smaller sample size of solved instances, a definitive answer cannot be given.

Once again, after a certain variable decrease%age, the Layered Totalizer Encoding is almost guaranteed to overtake Linear Search in solving time, with this value being approximately 25%.

Though the results are more promising with the weighted instances, it should once again be taken into account that the data set is smaller and not normally distributed, meaning further tests with other instances could lead to counter indications.

6.5. Relation Between Weighted Clauses and Time

This section presents the results of the Linear Regression analysis between the Clause Decrease Factor and Speedup Factor. Figure 6.6 presents the scatter plot and linear equation of the aforementioned analysis.

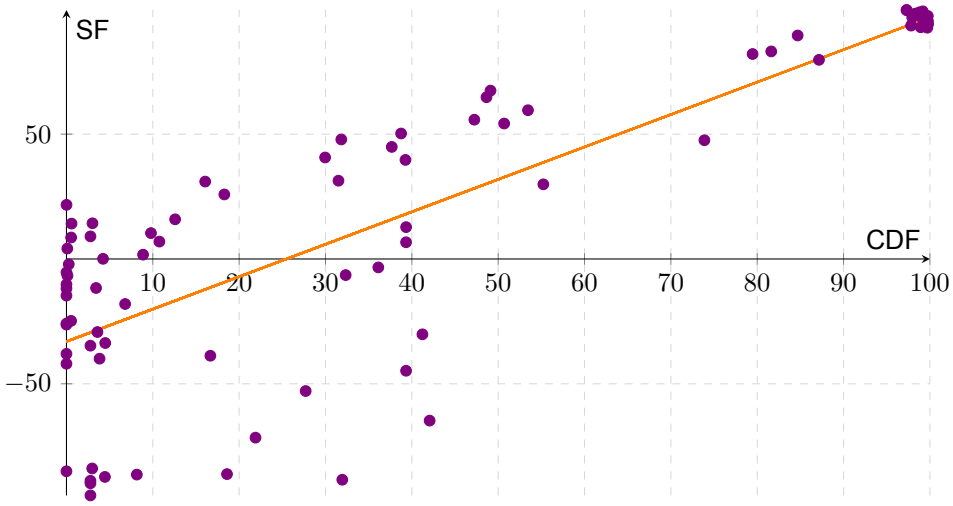

Figure 6.6: Linear Regression Analysis and scatter plot showing the positive linear relationship between the Clause Decrease Factor and Speedup Factor, named CDF and SF respectively, for the weighted instances, where each dot represents a singular instance solved by both Linear Search and the Layered Totalizer Encdoing

Much like in the unweighted case, the data for the weighted MaxSAT instances indicates a stronger relationship between solving time and clause quantity. However, as previously observed, the amount of clauses depends directly on the amount of variables and their weights, hence this impact of clauses is only secondary to the impact of the variables within the encoding. Nonetheless, the figure above demonstrates the power of the Layered Totalizer Encoding for certain instances where the lower bound can be tightened. Specifically, looking at the cluster of points located on the rightmost side of the x-axis shows how big of an impact this bounding technique can have on the runtime.

Table 6.4 displays the important observations from the Linear Regression analysis.

	Equation	P-value	X-intercept	R^2
Linear Regression	$\hat{(y)} = 1.2989x - 33.0777$	0.00001964	25.4661	0.674

Table 6.4: Breakdown of statistics obtained through Linear Regression

The linear equation shows that for each 1% increase of the Clause Decrease Factor there is a 1.2989% increase in the Speedup Factor, meaning the more clauses eliminated the faster the solving time. The p-value is once again low enough to be statistically significant. The x-intercept demonstrates the value at which there is a higher certainty that the Layered Totalizer Encoding will overtake Linear Search with regard to runtime. Finally, the R^2 value indicates a strong direct relationship between the two variables.

As with the other results up until this point, the value at which the Layered Totalizer Encoding is almost guaranteed to overtake Linear Search is at the 40% point of the x-axis.

As with the Variable Decrease Factor, a disclaimer with regard to the results should be given. Due to the inconsistency with a normal distribution these data points have, the Shapiro-Wilks test does not pass its p-value threshold and as there are not enough data points, the conclusion drawn from this analysis is not guaranteed to be correct. As such a repeat test of this nature can be done with different benchmarks that can confirm or deny the strength of the linear relationship between these two variables.

6.6. Unweighted Variable Ordering Heuristics

By examining the various Variable Ordering heuristics discussed in earlier chapters, the objective is to assess the influence of an effective or ineffective ordering of leaf nodes in the Totalizer Tree and its potential impact on performance. The performance of the implemented Variable Ordering heuristics, along with the standard Layered Totalizer Ordering, can be observed in Figure 6.7 and Figure 6.8. The results are split in two figures due to the different data processing steps. As noted in Chapter 4, the core generation process, IHS, can lead to a solution of a given instance before the core-generation

time comes to an end. To that extent, many instances were removed from the set of solved instances as they were not solved by the method discussed in this study, namely Layered Totalizer Encoding.

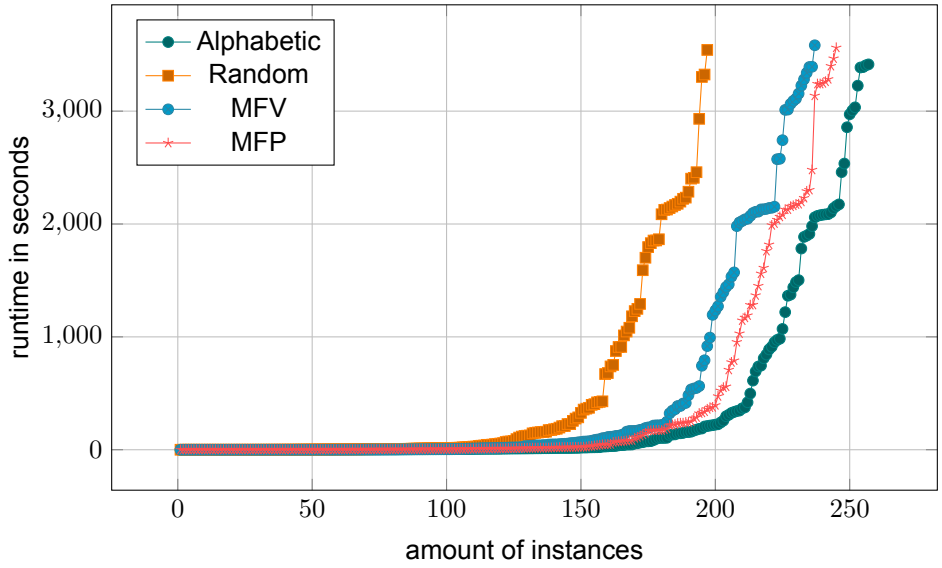

Figure 6.7: Cactus plot showing the performance of the different Non-Core Variable Ordering Heuristics

Figure 6.7 displays the non-core Variable Ordering heuristics and their performance in comparison to the standard Alphabetic Layered Totalizer as well as the baseline of a randomly ordered Layered Totalizer. As expected, the randomly ordered Totalizer Encoding performed the worst, whilst the Alphabetic Totalizer proved to be the best. As presented in Chapter 3, the two frequency based variable orderings are Most Frequently Occurring Variable, denoted as MFV in the table, and Most Frequently Occurring Pair of Variables, denoted as MFP in the table.

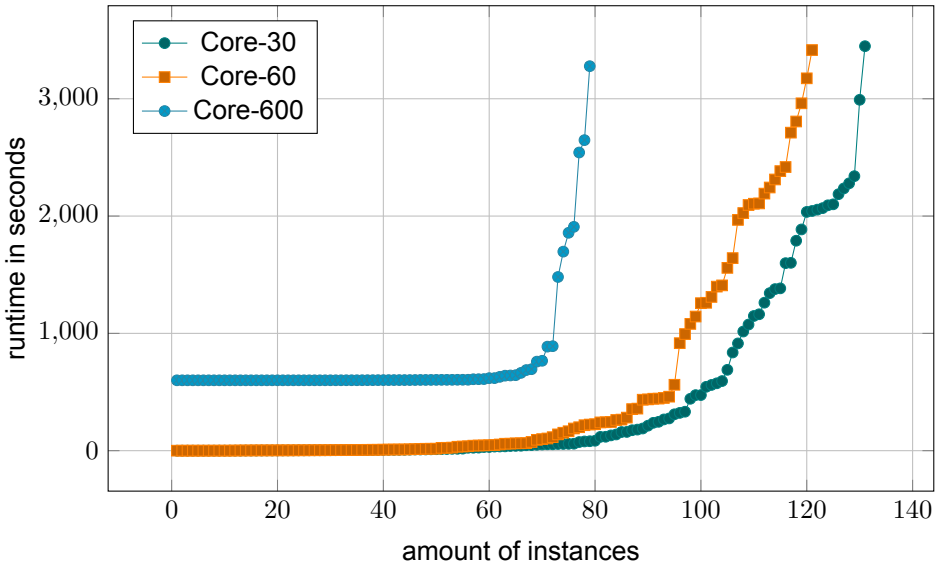

Figure 6.8: Cactus plot showing the performance of the different Core Variable Ordering Heuristics

Figure 6.8 shows the results of the core-based ordering heuristics. Each heuristic is labeled as Core-X, where X denotes the number of seconds for which the core-generation process was ran, thus Core-30 means the core generation algorithm ran for 30 seconds. Though the table shows a significantly less impressive runtime and a lower number of overall instances solved, this result especially shows the power of a good ordering. As can be seen most prominently in the performance of the Core-600

heuristic, which runs the core-generation phase for 600 seconds, has an almost flat line for 70 instances - implying that with a good variable ordering the bounding process is a lot more successful in binding nodes in lower levels of the Totalizer Tree.

Nonetheless, it is not only vital to determine which Variable Ordering solves the most instances, but which heuristic is the fastest, and which heuristic solved what amount of instances that no other heuristic solved. Table Figure 6.9 shows the breakdown of relevant information per heuristic used. For each Variable Ordering presented are the unique instances solved, followed by how many instances the given heuristic dominated given that at least one other Variable Ordering also solved that instance, and finally the total instances solved by each Variable Ordering heuristic.

	Alphabetic	MFV	MFP	Core-30	Core-60	Core-600
Unique Solution	2	30	0	1	1	0
Fastest Instances Total	120	49	34	44	12	0
Total Instances	257	237	245	131	121	79

Figure 6.9: Numeric breakdown of the Variable Ordering performances

As observed in the table above, it is not as straightforward to determine which Variable Ordering Heuristic to choose when solving a MaxSAT instance with the Layered Totalizer Encoding. First, a look into the worst-performing heuristic, Core-600. Core-600 uses IHS to generate cores for 600 seconds before stopping and ordering the variables according to the cores found so far. With Core-600, more than 200 instances had to be removed from the pool of solved instances because IHS managed to solve them before the 600 seconds were up. Though it is not surprising that IHS solved these instances in such a short time, it is surprising that there is not even one unique instance solved when using this heuristic. However, this is easily justified by the simple fact that IHS is a well-established algorithm that has proven to be good at solving many different types of MaxSAT instances in short periods of time. It is then no wonder that if IHS did not manage to solve an instance in 10 minutes, then perhaps the Layered Totalizer Encoding might not be able to solve it in the remaining 50 minutes. Moreover, there is an overall decrease in the total number of instances solved in comparison to its Core-Based counterparts, which can easily be explained by the time change. To be precise, for a given instance that was solved in Core-60 and not solved by Core-600, it is simply due to the fact that the instance was solved by IHS and then removed from the set of solved instances. Despite the non-dominant performance of the Core-600 heuristic, it should be noted that measure was kept of the time it took to solve the instances after the ordering was done - meaning the actual time it takes for the Layered Totalizer to be built without the 10 minutes spent on core-generation was measured. The achieved speedup when not taking into account core-generation is an impressive 82% for the 62 instances in which Core-600 outperformed the no-heuristic approach. Although Core-600 does not provide unique solutions or the fastest solving time for any instance, it shows that with a proper Variable Ordering, suited for the Totalizer Tree, a significant speedup can be achieved.

The same analysis can be extended to Core-30 and Core-60. Both suffer from the same pitfall - a race against IHS and its core-generation, which ultimately solves the problem itself. It can also be seen that even though both Core-30 and Core-60 solved a unique instance, these instances in particular are more of a measure regarding which instances get solved with IHS before the time is up, rather than an actual metric that speaks about the usefulness of the heuristic with regard to the Layered Totalizer Tree leaf ordering. Nonetheless, Core-30 and Core-60 each proved to be the fastest for a non-negligible portion of the overlapping instances. With regard to Core-30, there was a 34% increase in speed in comparison to the non-heuristic approach, whilst Core-60 displayed a 40% speed gain when compared to the non-heuristic Totalizer. Unlike Core-600, the 30 and 60-second overhead from the ordering procedure is insignificant in the speedup.

Although the Core-Based heuristics proved that a good Variable Ordering tailored to the Totalizer's functionality can lead to new and faster solutions, the focus now shifts to the Variable Ordering heuristics that use properties of the MaxSAT instance and their effects.

The Most Frequent Variable Ordering heuristic is in the lead out of all heuristics with regard to how many unique solutions it provides, with 30 unique instances. Moreover, the second fastest approach

is the Aplabetic approach. Although certain shortcomings of this approach were outlined in Chapter Chapter 4, it seems that the positives outweighed the negatives in practice. Due to the nature of this ordering, the time it took to reorder the variables is negligible, making it a great candidate for any use case. Regarding the speedup, on average for each instance, there was a 30% speedup compared to the Alphabetic approach, which is not as impressive as the introduced Core-Based heuristics. However, with the presence of the 30 unique solutions, this heuristic ordering should not be ignored.

Finally, let's move onto the Most Frequent Variable Pair Heuristic, which is the heuristic with the most instances solved, only preceded by the Alphabetic approach. With the same drawbacks as the Most Frequent Variable Heuristic, it once again demonstrates that in practice these drawbacks might not be as deterring. Although it did not solve any unique solutions, it was the fastest heuristic for 34 of the instances. Unfortunately, there was only an increase of about 10% in runtime compared to the Alphabetic approach, meaning that this heuristic pales in comparison to its counterpart.

6.7. Unweighted Hybrid Approach

Figure Figure 6.10 displays the performance of each Hybrid Totalizer Encoding in comparison to a fully layered encoding and Linear Search.

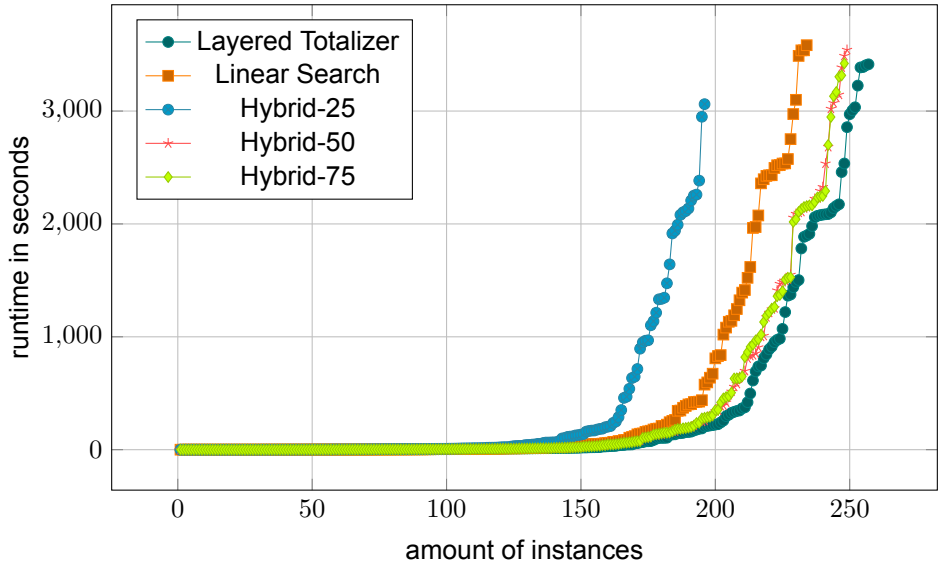

Figure 6.10: Cactus plot showing the performance of the different hybrid Totalizer Encodings and their performance compared to the pure Layered Totalizer Encoding and pure Linear Search

This table presents the results from the several different hybrid models, where each model is denoted by Hybrid-X, where X denotes the percentage of the Totalizer Encoding build by the Layered Totalizer Encoding. As shown in the cactus plot, the Layered Totalizer Encoding outperforms any of the hybrid approaches in terms of total instances solved. Surprisingly, the Hybrid-25 performs the worst in terms of total instances solved. Though the intuition from the previous section suggests that even small lower-boundary increments can have a big impact, it seems that only bounding the lower quarter of the layers is an ineffective strategy. On the other hand, Hybrid-50 and Hybrid-75 do solve more instances than Linear Search, however, there is not a big gap between the performance of the two Hybrids.

Table Figure 6.11 displays the breakdown of results more precisely. Hybrid-50 and Hybrid-75 only differ by 1 instance, however, both have instances for which they were the fastest. Hybrid-25 has the most fastest instances out of any hybrid approach, which shows that even though it solves the least amount of instances, its speed is non-negligible.

	Layered	Linear	25	50	75
Unique Solution	8	1	0	0	0
Fastest Instances Total	132	54	31	28	16
Total Instances	257	234	196	249	248

Figure 6.11: Numeric breakdown of the hybrid Totalizers' performace

For the instances in which Hybrid-25 outperformed the rest, there was an almost 30% increase in runtime. Hybrid-50 and Hybrid-75 also showed an only 30% increase, which is not surprising.

Given the results, it seems that a hybrid approach does not offer either a significant speedup, nor were new instances solved by any hybrid option. The hybrid approach has proven to not outperform the Layered Totalizer Encoding on its own, and the reasoning behind this can be seen through the difference in the amount of instances solved by Hybrid-25 and Hybrid-50. Namely, if an impactful bounding step is not encountered before switching to Linear Search, the time spent tightening lower bounds is in essence ”wasted” time, as the cost of the SAT calls at the topmost level remain largely unchanged. This can further be seen in the following section, where different instances are analysed and show how the cost of a SAT call per layer changes during the execution of the Layered Totalizer Encoding.

6.8. Unweighed Instance Family Overview

This section aims to summarize the results obtained for the Layered Totalizer with respect to the different families of instances. Below, Table 6.5 displays how each variation of the Layered Totalizer Encoding, as well as Linear Search, performed.

As the results appear daunting and not every instance family is of importance, selected families will be discussed more in depth below.

File Name	Lay	Lin	25	50	75	Rand	Core-600	Core-60	Core-30	MFV	MFP
aes (7)	0	0	0	0	0	0	0	1	1	0	0
aes-key-recovery (16)	16	15	14	16	16	7	1	1	1	10	16
atcoss (13)	6	5	5	5	5	6	0	2	4	6	6
bcp (16)	16	15	13	15	15	14	3	3	4	15	16
causal (16)	16	15	13	15	15	16	11	14	15	16	16
close_solutions (16)	2	3	2	3	3	2	1	1	1	2	2
decision-tree (13)	1	1	0	0	0	1	1	1	0	1	1
des (16)	2	1	0	2	1	0	0	1	2	2	2
drmx-atmost (16)	16	16	14	16	16	0	14	16	16	16	16
drmx-cryptogen (16)	0	0	0	0	0	0	0	0	0	8	0
exploits-synthesis_changjian_zhang (3)	2	1	1	2	2	2	2	2	2	2	0
extension-enforcement (16)	0	0	0	0	0	0	0	0	0	0	0
fault-diagnosis (16)	9	9	7	9	9	9	1	4	4	9	9
frb (16)	16	16	15	16	16	3	4	6	7	4	16
gen-hyper-tw (16)	3	3	2	3	3	3	0	1	1	3	3
HaplotypeAssembly (6)	0	0	0	0	0	0	0	0	0	0	0
hs-timetabling (1)	0	0	0	0	0	0	0	0	0	0	0
kbtree (14)	1	1	1	1	1	1	0	0	0	13	1
large-graph-community_detection_jabbour (8)	5	5	5	5	5	4	0	0	0	4	5
logic-synthesis (16)	6	6	5	6	6	1	1	1	2	5	6
maxclique (16)	12	13	13	13	13	13	6	7	8	13	12
maxcut (14)	0	0	0	0	0	0	0	0	0	0	0
MaximumCommonSub-GraphExtraction (16)	16	14	11	14	14	15	2	11	12	14	15
MaxSATQueriesInterpretableClassifiers (16)	9	8	7	9	9	9	2	3	3	9	9
mbd (16)	2	2	1	2	2	1	2	2	2	2	2
min-fill (16)	4	2	1	2	2	4	0	2	3	3	4
optic (16)	0	0	0	0	0	0	0	0	0	1	0
phylogenetic-trees_berg (15)	4	3	3	4	4	8	2	2	3	9	0
planning-bnn (15)	10	8	5	10	10	10	5	7	8	11	11
program_disambiguation-Ramos (10)	9	8	3	9	9	9	2	3	3	3	9
protein_ins (12)	12	12	8	12	12	12	0	0	0	0	12
railroad_reisch (11)	7	0	4	7	7	7	3	7	7	8	2
railway-transport (6)	1	1	1	1	1	1	0	1	1	1	1
ramsey (2)	0	0	0	0	0	0	0	0	0	0	0
RBAC_marco.mori (16)	6	6	5	6	6	4	3	3	3	6	6
reversi (5)	2	2	1	2	2	2	0	0	0	2	2
scheduling (5)	1	1	1	1	1	0	1	2	1	1	1
scheduling_xiaojuan (16)	10	7	7	8	8	0	1	3	2	5	9
SeanSafarpour (16)	0	0	0	0	0	0	0	0	0	0	0
security-witness_paxian (16)	0	0	0	0	0	0	0	0	0	0	0
set-covering (16)	0	0	0	0	0	0	0	0	0	0	0
setcover-rail_zhendong (4)	0	0	0	0	0	0	0	0	0	0	0
spinglass (2)	1	0	0	1	1	0	0	0	0	0	1
treewidth-computation (16)	13	14	11	13	13	14	0	1	1	13	13
uaq (16)	7	7	7	7	7	7	6	6	7	7	7
uaq_gazzarata (11)	9	9	8	9	9	9	3	5	5	9	9
xai-mindset2 (17)	5	5	2	5	5	3	2	2	2	4	5

Table 6.5: Tabular representation of the different Instances of Families and how each Layered Totalizer Encoding variation performed

Firstly, to acknowledge the families for which no instances were solved by any of the solvers, such as the set-cover family or hs-timetabling. The precise reason as to why these families proved difficult for the Layered Totalizer Encoding is not necessarily the same - for certain families the instances were so long that during the 3600 seconds time limit the read-in process was not finished. For others, due to the

complexity of the problem, the structure, or even the implementation of the Layered Totalizer Encodings, a solution was not found. Conversely, there are instance families for which every solver performed more or less the same and managed to solve every instance. These families are drmx-atmost as well as MaximumCommonSub-GraphExtraction. The frb family can also fall under this category, as the frb instances got eliminated from the set of solved solutions for the core-based heuristics. These families have the same commonality - all solutions of these instances have an optimal result closer to the upper bound than the lower bound of the top-most node. As established, the lower bound of any node is at least 0, whilst the top-most node has the upper bound of the number of soft clauses in the instance. It is then intuitive as to why optimal solutions relatively close to the maximum - many nodes get bound early.

However, a more interesting result to analyse is the families where there is an over or underperformers so that the power of the Layered Totalizer can be highlighted. An example of this occurrence is the drmx-cryptogen family. As observed, with the exception of the Most Frequently Occurring Variable, no solver manages to crack even one instance. However, the MFV heuristic managed 8. These 8 instances are proof of how much the variable order in the leaf nodes can have an impact on the overall performance of the Layered Totalizer Encoding. The takeaway from this family and the families which had all instances solved is that a preprocessing step determining the structure of the problem can greatly improve the performance.

6.9. Weighted Variable Ordering Heuristics

Now let's analyze the performance of the Weighted Layered Totalizer with different Variable Ordering. Figure 6.12 and Figure 6.13 display the performance of the previously defined ordering heuristics as well as the Alphabetic approach.

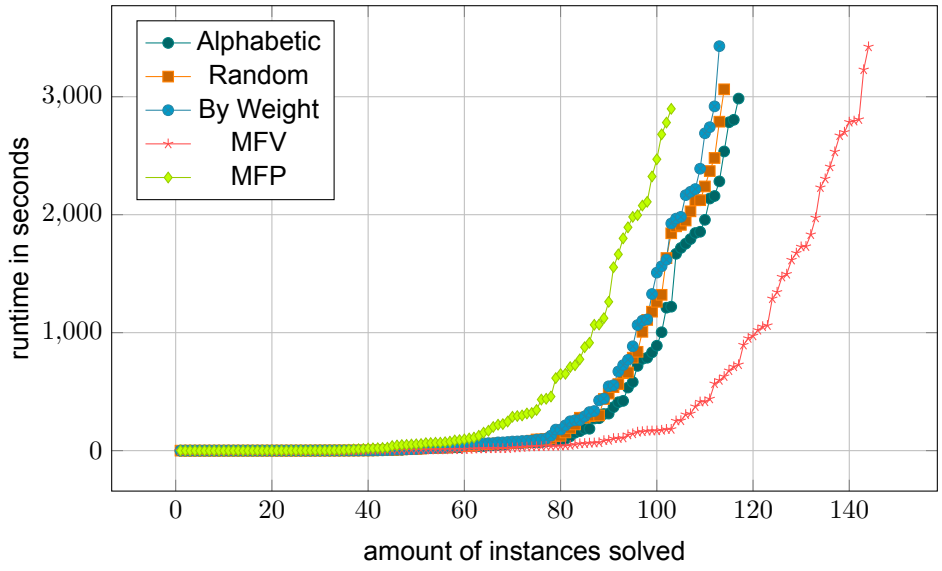

Figure 6.12: Cactus plot of all Non-Core Variable Ordering Heuristics for weighted instances

The cactus plot above displays the results obtained by the non-core Variable Ordering heuristics. Aside from having the Alphabetic ordering and Random orderings as baselines, a new order was added in the form of weight-based ordering. As the name suggests, the soft clauses with the highest costs took precedence. The results are notably different in the weighted case as the random ordering and weighted ordering are similar to the Alphabetic approach, with the Most Frequently Occurring Variable taking the lead.

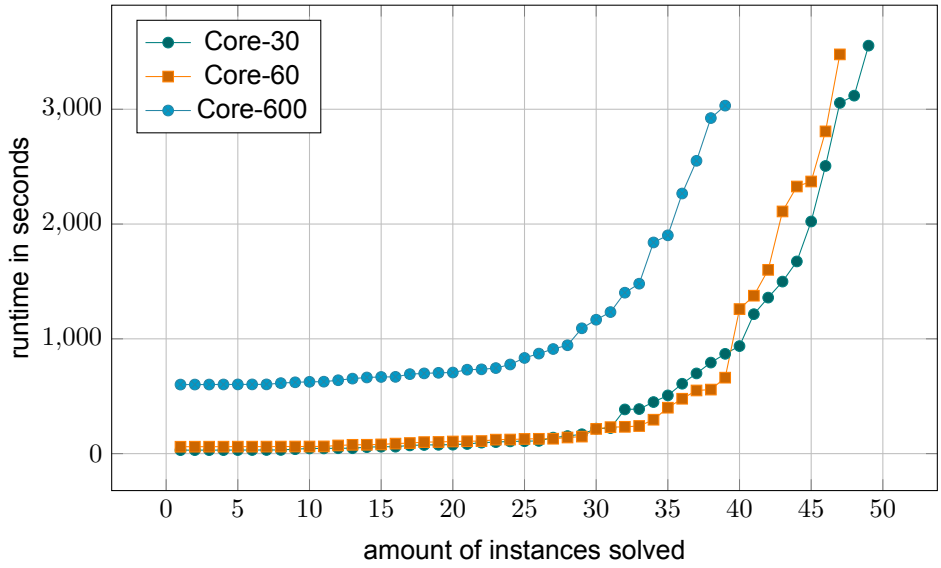

Figure 6.13: Cactus plot of all Core-Based Variable Ordering Heuristics for weighted instances

Similar to the unweighted case, the Core-600 heuristic performs the worst in terms of instances solved. It, along with Core-30 and Core-60, faces the same challenges in the weighted and unweighted instances, which involve a race with IHS and core-generation resulting in many solutions being deemed invalid. Table Figure 6.14 reveals that none of the Core-Based heuristics led to any unique instances being solved, and they solved less than half of the instances solved by the Alphabetic approach. While each Core-Based heuristic did exhibit better performance in certain cases, the difference was not significant.

	Alphabetic	MFV	MFP	Core-30	Core-60	Core-600
Unique Solution	0	55	3	0	0	0
Fastest Instances Total	15	102	33	10	5	1
Total Instances	95	144	103	48	47	30

Figure 6.14: Numeric breakdown of the different Variable Orderings for the weighted instances

On the other hand, the Most Frequently Occurring Variable and Most Frequently Occurring Variable Pair heuristics showed an increase in the number of instances solved and outperformed other heuristics in most cases. This is particularly apparent for the Most Frequent Occurring Variable ordering, which solved 50% more instances than the Alphabetic approach. In fact, for the instances solved faster by the Most Frequently Occurring Variable and the Alphabetic approach, a speedup of 55% was observed for the 80 shared instances.

A similar trend can be seen with the Most Frequently Occurring Variable Pair, although to a lesser extent.

In general, the speedup gained from the Most Frequently Occurring heuristics is instance dependent rather than a deep exploitation of known problem properties. When using Core-Based heuristics to order the variables in the leaf nodes of the totalizer tree, a known property is exploited - specifically, that unsatisfiable cores can be tightened in lower parts of the tree by ordering the variables to mimic parts of the Totalizer Encodings observed in OLL. The order of the leaf variables determines the level at which a certain constraint is encountered, making it desirable to use unsatisfiable cores to encounter these constraints earlier. On the other hand, the heuristics related to Most Frequently Occurring orders are less concerned with what is specifically beneficial to the Totalizer Tree, but rather rely on observations of the instance and a loose intuition that variables occurring frequently may form a core.

In summary, a generalized conclusion cannot be drawn regarding the preferable variable ordering heuristic for any particular problem. Instead, these heuristics are used to demonstrate the impact of

different leaf orderings on the Totalizer Tree, whether positive or negative.

6.10. Weighted Hybrid Approach

Now moving onto the results of the Hybrid Totalizer Encodings for the weighted MaxSAT instances. Figure Figure 6.15 displays the performance of the different encodings in terms of the number of instances solved.

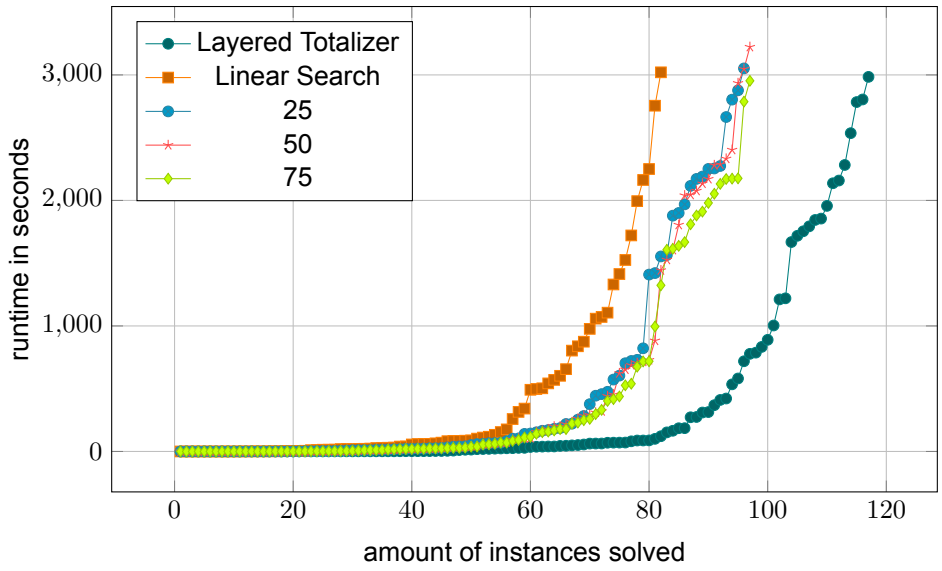

Figure 6.15: Cactus plot showing the performance of the different hybrid Totalizer Encodings along with the pure Layered Totalizer Encoding and pure Linear Search

As with the unweighted case, it is apparent that the hybrid approaches do not offer unique solutions, but do offer better runtimes. The issue of unintentionally omitting an important bounding step by switching to Linear Search can even more obviously seen here.

	Layered	Linear	25	50	75
Unique Solution	21	2	0	0	0
Fastest Instances Total	13	25	22	5	35
Total Instances	116	82	96	97	97

Figure 6.16: Numeric breakdown of the hybrid Totalizers' performace for the weighted instances

Though there are no new instances solved, it is interesting to observe the amount of instances for which the hybrid approaches were faster. The point of interest is the discrepancy between the amount of instances for which the three hybrid approaches had the fastest runtime, specifically the observation that Hybrid-25 and Hybrid-75 have significantly more best runtimes than Hybrid-50. The conclusion here is that it is better to choose either a fully Layered Totalizer Encoding approach or a fully Linear Search approach. This could be tied to the smaller sample size solved, or the amount of variables generated.

6.11. Weighted Instance Family Overview

As done with the unweighted case, this section presents the instance family overview and how each version of the Layered Totalizer Encoding performed.

File Name	Lay	Lin	25	50	75	Rand	Core-600	Core-60	Core-30	MFV	MFP
abstraction-refinement (10)	0	0	0	0	0	0	0	0	0	0	0
af-synthesis (16)	2	0	1	1	1	2	1	2	2	3	2
auctions (16)	9	0	0	0	0	9	0	0	0	8	0
binaryNN (16)	0	0	0	0	0	0	0	0	0	10	0
BTBNSL (32)	0	0	0	0	0	0	0	0	0	0	0
causal-discovery (16)	0	0	0	0	0	0	0	0	0	0	0
CSG (10)	0	0	0	0	0	0	0	0	0	0	1
csg-xiaojun (16)	0	0	0	0	0	0	0	0	0	0	0
css-refactoring (11)	0	0	0	0	0	0	0	0	0	0	0
dalculus (16)	4	0	4	4	4	4	0	0	0	5	4
decision-tree (16)	0	0	0	0	0	0	0	0	0	0	0
drmx-atmostk (16)	4	0	5	4	5	4	0	0	0	0	4
haplotyping-pedigrees (15)	0	0	0	0	0	0	0	0	0	0	0
hs-timetabling (9)	0	0	0	0	0	0	0	0	0	1	0
industrial (11)	0	0	0	0	0	0	0	0	0	0	0
lisbon-wedding (16)	0	0	0	0	0	0	0	0	0	1	0
maxcut (16)	2	1	1	2	2	2	0	0	0	0	2
MaxSATQueriesInterpretableClassifiers (15)	8	8	8	8	8	8	3	3	3	9	8
metro (16)	15	15	14	14	14	15	14	15	15	15	14
MinimumWeightDominatingSetProblem (7)	0	0	0	0	0	0	0	0	0	0	0
min-width (16)	0	0	0	0	0	0	0	0	0	0	0
mpe	0	0	0	0	0	0	0	0	0	4	0
OptimalQuantumCircuitMapping (2)	6	7	6	6	6	6	2	3	3	11	6
ParametricRBACMaintenance (16)	0	0	0	0	0	0	0	0	0	4	0
planning-bnn (6)	6	6	6	6	6	6	0	3	3	6	6
power-distribution (16)	11	11	10	11	11	11	0	0	0	16	11
preference-planning (12)	10	10	10	10	10	11	0	0	0	5	10
railroad-sc (6)	0	0	0	0	0	0	0	0	0	0	0
railroad-scheduling (8)	5	0	2	2	2	2	2	3	3	6	2
railway-transport (5)	1	0	1	1	1	1	1	1	1	1	1
ramsey (4)	0	0	0	0	0	0	0	0	0	0	0
RBACMacro (15)	0	0	0	0	0	0	0	0	0	2	0
relational_inference (10)	1	0	0	0	0	0	0	0	0	2	0
rna-alignment (16)	16	14	15	15	15	16	11	13	13	10	15
scSequencing (16)	0	0	0	0	0	0	0	0	0	0	0
Security-CriticalCyber-PhysicalComponents (15)	0	0	0	0	0	0	0	0	0	0	0
set-covering (16)	0	0	0	0	0	0	0	0	0	0	0
shiftdesign (16)	0	0	0	0	0	0	0	0	0	0	0
spot5 (16)	13	6	9	9	9	13	4	4	4	13	13
staff-scheduling (12)	1	1	1	1	0	1	0	0	0	1	1
switching-activity-maximization (1)	0	0	0	0	0	0	0	0	0	0	0
tcp (16)	0	0	0	0	0	0	0	0	0	0	0
timetabling (14)	2	2	2	2	2	2	0	0	0	7	2
warehouses (8)	1	1	1	1	1	1	0	0	0	1	1

Table 6.6: Tabular breakdown of the performance of the different Weighted Layered Totalizer Encodings

There is little to be said that was not visible in the unweighted case with regard to the positives - namely that a good or bad ordering can make or break the runtime of a given Layered Totalizer Encoding. An example of a Variable Ordering not performing too well can be seen in the drmx-atmost family, wherein the Random Variable Ordering did not manage to solve a single instance whilst the other Encodings managed to solve almost all instances.

The important takeaway from the weighted case is simply to acknowledge the added complexity onto the Layered Totalizer Encoding due to the exponential growth of variables. As can be observed in the table above, there are many families for which a couple of instances were solved by different

versions of the Totalizer Encoding, however not a significant number.

6.12. Layer Analysis

So far, the results section has presented every defined goal in Chapter 5 with the exception of the observations conducted on the layers of the Layered Totalizer Encoding. The assumption made is that there is a certain relation between a given layer l_i and its successor l_{i+1}) in relation to the time its takes to encode said layers. More precisely, there was an assumption that given the raw time and number of SAT Oracle calls made in a certain layer, there would be a way to predict somewhat accurately the time it would take to build the next layer.

After gathering the aforementioned data and performing a simple calculation to determine the average cost of each SAT Oracle call within a level, it can be concluded that an overreaching general conclusion is not achievable. To illustrate why defining such a relationship is difficult, several instances, representative of their instance families, have been selected as examples and shown in Figure 6.17

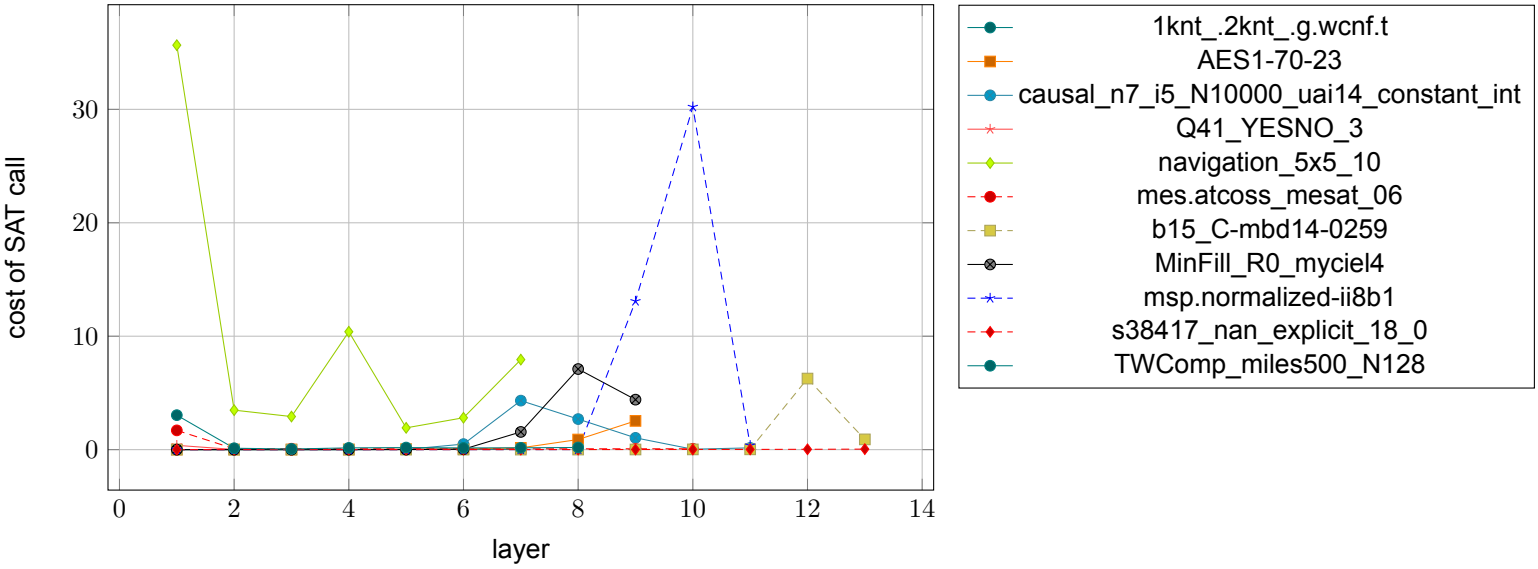

Figure 6.17: Plot demonstrating different instances from different families

The table above demonstrates 11 instances, all from different instance families, and their average SAT Oracle call time per layer. The x-axis represents the layer of the Totalizer Tree in question, with layer 1 denoting the lowermost level consisting only of leaf nodes. As not every Totalizer Tree is of the same height, certain instances only go up to 3 layers with others moving all the way to 13. The y-axis shows the average time a SAT call costs, in seconds, for each layer. There are several instances which have a peak in time spent per layer before reaching the topmost node. On ther other hand, there are also instances wherein the most costly level is the lowermost level. There are instances with changing costs of layers with no peak, or layers which simply stagnate in cost per layer. There is a slight pattern of the initial layer costing more than the following couple of layers, however this is not a general rule accross all instance families.

Then, perhaps instead of stating a global rule with regard to how much each successive layer costs with relation to its predeccesor, a look into different families' instances could be taken.

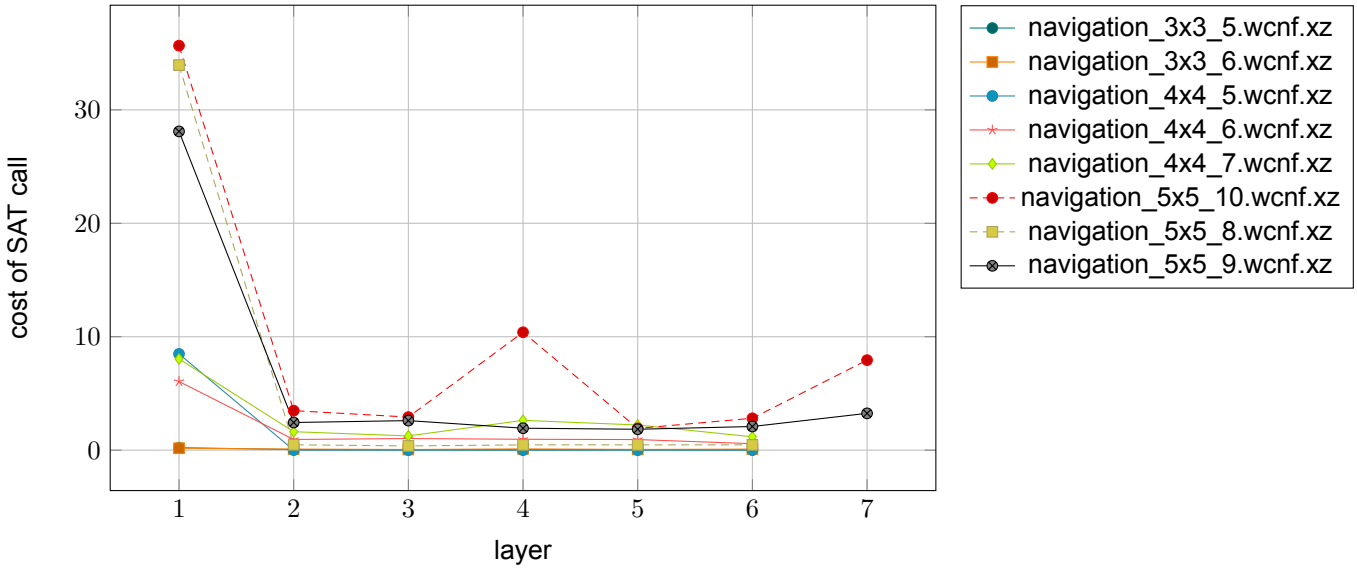

Figure 6.18: Plot demonstrating the cost of SAT call per layer across all layers for the subset of navigation problems in the planning-bnn family

Figure 6.18 shows a subset of the planning-bnn family of instances, specifically all instances dealing with navigation. As can be seen in the figure above there is a certain repeating element in this family - that is, a very costly first layer followed by comparably cheaper layers. A potential reason behind this behaviour is the structure of the problem leading to the SAT Oracle running into conflicts late in its execution, partly due to the low presence and relation between the relaxation variable of interest and the original clauses. More elaborately, the more levels are built, the stronger the relationships between relaxation variables become. At layer 1, no relaxation variable is related to any other relaxation variable. This means there is a low presence of this relaxation variable in the clauses, leading any SAT Oracle to potentially spend a lot of time before running into a conflict or solution. Then, for the instances at hand, adding more clauses helps with establishing each variable more firmly, meaning a solution of conflict is encountered faster.

However, this is all instance-specific behaviour that does not necessarily relate to any other family of instances.

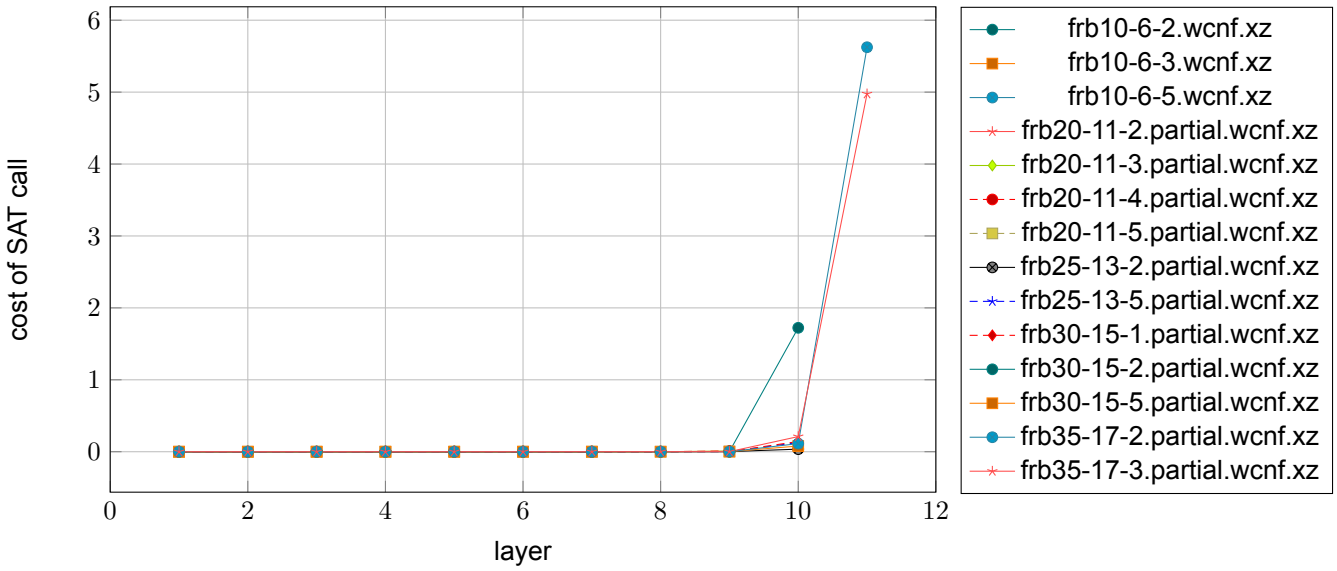

Figure 6.19: Plot demonstrating the cost of SAT call per layer across all layers for the ferb family

Figure 6.19 demonstrates the plot of the frb family of instances and their behaviour with regard to layer expenses. As shown in the figure, this family also has a pattern of behaviour, bar a couple of instances. Namely, each level costs more or less the same. However, the reasoning behind this family's behaviour is unrelated to the structure of the problem - rather to the final optimal solution. Every instance within this family has an optimal value close to the upper bound. Logically it follows that the closer the optimal value to the upper bound, the more nodes can be bound at lower levels thus leaving little work to upper levels. Take frb30-15-2.partial as an example problem of this occurrence - each of this instance's nodes in Layer 2 have their bounds raised to 1, with their upper bound staying at 2. Almost every node at Layer 3 has its lower bound raised to 3, with its upper bound staying at 4. This family is the perfect example of how the Layered Totalizer propagates this rippling effect throughout the layers, as well as being the example of which problems benefit the most when solved with the Layered Totalizer Encoding over Linear Search.

6.13. Limited Unweighted Layered Totalizer Encoding

Up until this point in the testing process, there has been no time limit imposed on individual segments of the Layered Totalizer Encoding, rather a general maximum of 3600 seconds. However, much like a hybrid option cuts off the Layered Totalizer Encoding from proceeding with its execution based on the height of the Totalizer Tree, it is possible to cutoff the bounding process in the Layered Totalizer encoding with a time limit. More specifically, the point of interest is to determine whether halting tightening of a given node after 300 seconds would positively impact the runtime of the algorithm. The intuition behind this testing scenario is to use the benefits that bound tightening provides even if not every node has been tightened fully.

Figure 6.20 demonstrates the cactus plot of the aforementioned test, with both Linear Search and the unlimited Layered Totalizer presented as baselines.

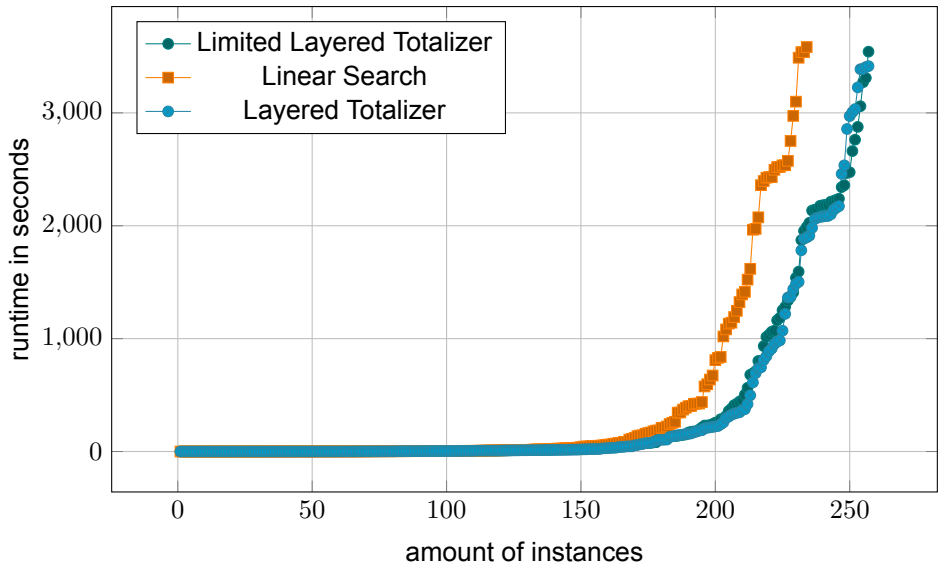

Figure 6.20: Cactus plot showing the performance of the Layered Totalizer Encoding in which every node has a time limit of 300 seconds maximum

As observed, the Limited and regular Layered Totalizer Encodings solve the same amount of problems. However, a look under the hood shows that they each have 8 unique instances that the other did not solve. The 8 instances solved by the Limited Layered Totalizer Encoding in fact are also not shared with Linear Search and are instances with lengthy runtimes. In fact, several of the 8 unique instances solved by the Limited Layered Totalizer are instances that were not solved by any other solver in the 2022 MaxSAT competition. These results are not surprising - as discussed in Chapter 4, the weaknesses and strengths of the Layered Totalizer are two sides of the same coin. Where bounding each node on the lowest level in the tree is almost always cheap, getting to upper levels of the Totalizer Tree in which few if any nodes have been bound can result in a SAT Oracle call as expensive as that

of the top-most node. Then, certain instances are sure to benefit from having a mix of a little bound tightening but a cutoff to proceed to the upper levels.

It should be noted that majority of instances solved by the Layered Totalizer have Totalizer Trees in which a full layer can be encoded in less than a second, bar the top-most node. This means that the cut-off of 300 seconds was very generous with the instances as many of them did not get impacted.

Though the Limited Layered Totalizer provided more insight into how the Layered Totalizer can be used, it is worth noting that the Layered Totalizer Encoding did outperform the Limited Layered Totalizer by 12%. This is also not surprising, as the cutoff could have potentially inhibited certain important bounds to be raised for a given instance leading to more time spent in the top-most Tree node instead.

6.14. Limited Weighted Layered Totalizer Encoding

Unlike its Unweighted counterpart, the Weighted Limited Layered Totalizer Encoding performance is more similar to Linear Search than the regular Layered Totalizer. Figure 6.21 shows how each of the three algorithms performed on the benchmarks.

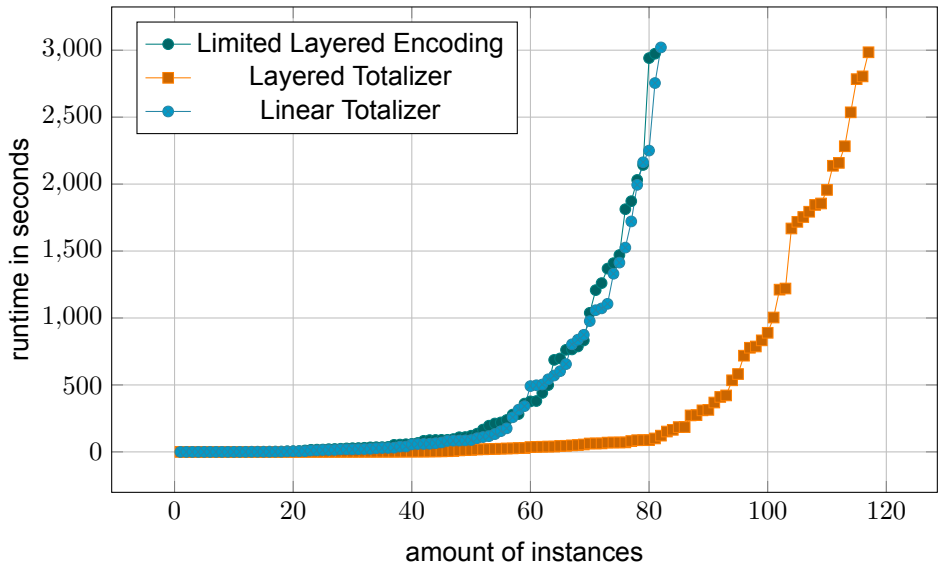

Figure 6.21: Cactus plot showing the performance of the Layered Totalizer Encoding in which every node has a time limit of 300 seconds maximum for the weighted instances

The Limited Layered Totalizer Encoding has only two unique benchmarks that the other algorithms did not manage to solve, though unlike in the unweighted case, these instances did not take a lengthy time or were unsolvable by any other 2022 MaxSAT solver.

The observations made in the previous section still ring true here, though it should also be acknowledged that as weighted MaxSAT instances produce more clauses and variables in the Totalizer Encoding leading to each SAT call having to deal with a potentially bigger load of information. Then, a repeat of this experiment with a different cutoff limit might lead to more similar results to those observed in the unweighted case.

With regard to the speed of the Limited versus unlimited Layered Totalizer Encoding, the Limited Encoding under performed by 33%. Once again, this is to be expected, as cutting off an important bound-raising SAT Oracle call too early in the lower layers pushed the burden onto the upper layers, thus slowing down the runtime.

6.15. Layered Totalizer Encoding OLL

The preceding sections presented the results for the several scenarios that the Layered Totalizer Encoding was put through with Linear Search as its direct competitor. However, the Layered Totalizer Encoding can not only work as a standalone algorithm, but can be inserted into any other algorithm that uses the Totalizer Encoding. To this extent, as outlined in Chapter 4, an experiment was conducted in which the RC2 implementation of the OLL algorithm was adjusted so that it uses the Layered

Totalizer Encoding as its Totalizer Encoding of choice. Figure 6.22 visualized the results of running RC2 with the Layered Totalizer and the Generalized Totalizer Encoding. As can be seen in the graph, the Layered Totalizer RC2 solves an additional 50 instances that were not solved by the regular RC2 implementation with no core-minimization techniques.

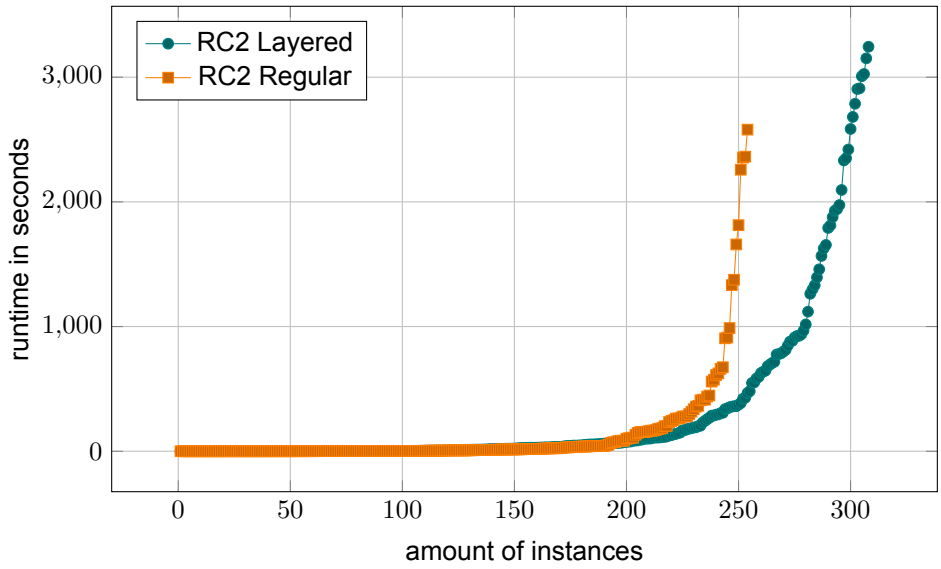

Figure 6.22: Cactus plot showing the performance of the RC2 algorithm when using the Layered Totalizer Encoding versus the Generalized Totalizer Encoding

As with the other results shown in this chapter, the full picture cannot be conveyed solely through a scatter plot. To that extent, Figure 6.23 showcases some observation with regard to the two RC2 algorithms. As can be seen, though the RC2 algorithm that makes use of the Layered Totalizer solved more instances overall and more unique instances, it rarely performed the fastest. In fact, the average runtime of the Layered RC2 algorithm was 31.5% slower than that of the regular RC2.

	Layered-RC2	Regular-RC2
Unique Instances	59	5
Number of solved the fastest	77	171
Total Instances	308	255

Figure 6.23: Numeric breakdown of the performance of the RC2 algorithms

However, when talking about the Layered Totalizer's speed in the context of RC2, there is a new factor to consider. As explained in Chapter 2, the OLL algorithm generates an unsatisfiable core at each iteration. Though only one such core is returned from the SAT Oracle, there may be several unsatisfiable cores, and based on which cores are returned by the SAT Oracle different Totalizer Trees will be built. More importantly, when running RC2 with the Layered Totalizer, when building said Totalizer Tree, the nodes are immediately bound with the top-most node of the Totalizer Tree also being bound until a satisfactory result is returned by the SAT Oracle. Due to making SAT calls without retrieving cores in order to tighten lower bounds and tightening said bounds, the cores encountered in the algorithm can and will form the baseline RC2 which uses the Generalized Totalizer Encoding. Then, following this line of reasoning it is understandable why the Layered Totalizer might slow down RC2, however it also explains its performance in terms of solved instances - that is, different cores are encountered which for these benchmarks have proven more useful.

As with the Layered Totalizer Encoding as a stand-alone algorithm, there is potential for different heuristics to be applied to the Layered Totalizer Encoding. Beyond understanding that using the Layered Totalizer Encoding leads to different core traces and hence different runtimes, it is also important to

look at the instance families solved by the Layered Encoding RC2 when compared to a regular Layered Totalizer Encoding.

The table below showcases the amalgamation of all Layered Totalizer Encoding algorithms such that from each heuristic approach only the best result is extracted and used. In comparison is the Layered Totalizer Encoding OLL.

Family Name	LTE OLL	GTE OLL	LTE algorithm
aes (7)	1	1	1
aes-key-recovery (16)	15	15	16
atcoss (13)	5	5	6
bcp (16)	16	15	16
causal (16)	16	11	16
close_solutions (16)	9	8	3
decision-tree (13)	0	0	1
des (16)	13	15	2
drmx-atmost (16)	16	5	16
drmx-cryptogen (16)	0	0	8
exploits-synthesis_changjian_zhang (3)	0	0	2
extension-enforcement (16)	4	6	0
fault-diagnosis (16)	13	12	9
frb (16)	8	3	16
gen-hyper-tw (16)	3	3	3
HaplotypeAssembly (6)	5	5	0
hs-timetabling (1)	0	0	0
kbtree (14)	1	1	13
large-graph-community-detection_jabbour (8)	5	5	5
logic-synthesis (16)	11	9	8
maxclique (16)	13	5	13
maxcut (14)	0	0	0
MaximumCommonSub-GraphExtraction (16)	15	14	16
MaxSATQueriesInterpretableClassifiers (16)	9	9	9
mbd (16)	16	16	2
min-fill (16)	5	2	4
optic (16)	4	4	1
phylogenetic-trees_berg (15)	1	1	9
planning-bnn (15)	7	6	11
program_disambiguation-Ramos (10)	8	7	9
protein_ins (12)	12	3	12
railroad_reisch (11)	0	0	8
railway-transport (6)	2	2	1
ramsey (2)	0	0	0
RBAC_marco.mori (16)	15	16	6
reversi (5)	2	2	2
scheduling (5)	1	1	2
scheduling_xiaojuan (16)	4	4	10
SeanSafarpour (16)	0	0	0
security-witness_paxian (16)	9	7	0
set-covering (16)	0	0	0
setcover-rail_zhendong (4)	0	0	0
spinglass (2)	1	0	1
treewidth-computation (16)	13	13	14
uaq (16)	12	9	7
uaq_gazzarata (11)	9	7	9
xai-mindset2 (17)	9	7	5

Table 6.7: Tabular breakdown of the performance of the OLL algorithm with the Layered Totalizer Encoding against the standard OLL algorithm and the best performances from any standard Layered Totalizer Encoding implementation

Table 6.7 displays some interesting results regarding the performance of all algorithms discussed so far. Firstly, a second look into the drmx-cryptogen family. The standard Layered Totalizer approach is the only algorithm that managed to solve any instance in this family as seen in the LTE algorithm column. On the other hand when looking at the HaplotypeAssembly family, either version of the OLL algorithm

outperformed the regular Layered Totalizer Encoding. Another interesting entry is the drmx-atmost, for which the regular OLL algorithm has the weakest performance.

What all these family instances convey is that both a linear approach or a core-based approach have their advantages. However, more importantly, is that the Layered Totalizer Encoding can be used in both manners to solve a larger amount of instances that a Generalized Totalizer Encoding might not solve. In fact, when choosing only the best performing heuristic per family, a Layered Totalizer Encoding approach solves 42 more instances than the regular OLL algorithm.

While these results are promising there are a few key drawbacks that need to be acknowledged. Firstly, there are certain instances for which the regular OLL algorithm experienced an out-of-memory error, whilst the Layered Totalizer OLL algorithm did not. These are only a handful of instances, though still noteworthy. Secondly, the implementation of every Totalizer Encoding was built for the purpose of this study - this means that the OLL algorithm presented in this section is not the most optimized OLL algorithm, nor are the encodings. Specifically, the algorithms implemented for this study might not use the most efficient data structures or have the most optimized for and while loops, as the primary goal was the ability to store and access information for the sake of analysis. All of these factors can skew the results, however, every precaution has been taken to lower the impact any confounding variables.

7

Conclusion

The goal of this work was to explore a possible improvement of the MaxSAT solving process by interleaving the Totalizer Encoding phase and the calls to the SAT Oracle in order to reduce the unnecessary information encoded in the Totalizer Tree. As shown by the results in Chapter 6 this approach provides a speedup for both the weighted and unweighted MaxSAT instances. Moreover, this approach has shown that the newly introduced Layered Totalizer Encoding can solve certain instances not solved by Linear Search.

Beside constructing the standard Layered Totalizer Encoding, it was of the essence to determine how this new Encoding could be improved upon. To that extent, this paper provided several different parameter tuning measures, such as variable ordering and a hybrid approach of a mixture of the Layered Totalizer Encoding and Linear Search. Different approaches yielded different results, with a possibility into a deeper look into different kinds of variable ordering or hybrid approaches. A stand-out among these approaches seems to be the Most Frequently Occurring Pair Variable Ordering for the weighted MaxSAT instances.

A final point of interest was the performance of the Layered Totalizer in an alternative algorithm such as the OLL implementation of RC2. This aspect would give an even more in-depth answer as to how the Layered Totalizer Encoding can be used to obtain results to previously unsolved instances as the involvement of unsatisfiable cores impacts which Totalizer Trees are seen throughout the solving process. This experiment led to the best overall results in terms of number of instances solved, with over 300 instances being solved in under an hour.

Tightening lower bounds within the Totalizer Tree in the process of encoding the problem has proven to have a positive impact on the field of MaxSAT solving both in terms of instances solved and the solving time of said instances. However, the Layered Totalizer Encoding is far from being perfect as demonstrated throughout this work. To fully explore the impact the Layered Totalizer Encoding can have on MaxSAT solving there are several questions that furutre work can explore.

The first and easiest adjustment to be done to the Layered Totalizer is a more efficient implementation. As it currently exists it is coded using python, which is not the fastest language when it comes to certain methods. Then a better implementation in C or C++ could majorly benefit the Encoding and show its true power.

Another point of interest are the results presented in the Limited Layered Totalizer Encoding experiment, which suggests a statistically significant relation between a decrease in variables and a decrease in solving time. This knowledge can be used during the encoding time to keep track of how many variables have been eliminated up to a certain point and if, say, after half of the Totalizer has been encoded and there is not a significant decrease in variables one could stop the execution and switch to Linear Search.

A more theoretical observation can be made on the implementation of the Layered Totalizer Encoding in the context of RC2 and core minimization. As the Layered Totalizer Encoding builds its tree from a given core, it is possible to encounter tightening of inner-bounds which implies that the provided core is not minimal. Then the question becomes which cores are most suitable for the Layered Totalizer Encoding and how different core sizes and complexities could impact the runtime. To answer this

question several different core traces for the same instance would need to be looked at and judged to determine which has the best outcome.

Finally, there are many different variable ordering heuristics that can be tried with the Layered Totalizer Encoding, such as neural network generated orderings or graph generated orderings.

In conclusion, the Layered Totalizer Encoding provides an alternative to Linear Search and the Generalized Totalizer Encoding. It has proven as a worthy alternative in all of the tested scenarios and has even showed useful when implemented in an OLL algorithm. There are many more areas left to be explored and tweaked with respect to the Layered Totalizer and the first step in exploring this new idea has shown that interleaving the encoding and solving process can solve previously unsolved MaxSAT instances.

References

[1] Mario Alviano, Carmine Dodaro, and Francesco Ricca. “A MaxSAT Algorithm Using Cardinality Constraints of Bounded Size”. In: *Department of Mathematics and Computer Science, University of Calabria, Italy* (2016).

[2] Gilles Audemard and Laurent Simon. “GLUCOSE: a solver that predicts learnt clauses quality”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2009.

[3] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. “Maximum Satisfiability”. English. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. 2nd ed. Frontiers in Artificial Intelligence and Applications. Netherlands: IOS PRESS, 2021, pp. 929–991. ISBN: 978-1-64368-160-3. DOI: 10.3233/FAIA201008.

[4] Fahiem Bacchus et al. *MaxSAT Evaluation 2022 : Solver and Benchmark Descriptions*. eng. 2022. URL: http://hdl.handle.net/10138/347396.

[5] Olivier Bailleux and Yacine Boufkhad. “Efficient CNF Encoding of Boolean Cardinality Constraints”. In: *Principles and Practice of Constraint Programming – CP 2003*. Ed. by Francesca Rossi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 108–122. ISBN: 978-3-540-45193-8.

[6] Jeremias Berg, Fahiem Bacchus, and Alex Poole. “Abstract Cores in Implicit Hitting Set MaxSat Solving”. In: *Theory and Applications of Satisfiability Testing – SAT 2020*. Ed. by Luca Pulina and Martina Seidl. Cham: Springer International Publishing, 2020, pp. 277–294. ISBN: 978-3-030-51825-7.

[7] Jeremias Berg, Emir Demirović, and Peter J. Stuckey. “Core-Boosted Linear Search for Incomplete MaxSAT”. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Ed. by Louis-Martin Rousseau and Kostas Stergiou. Cham: Springer International Publishing, 2019, pp. 39–56. ISBN: 978-3-030-19212-9.

[8] Jeremias Berg, Antti Hyttinen, and Matti J\”arvisalo. “Applications of MaxSAT in Data Analysis”. In: *Proceedings of Pragmatics of SAT 2015 and 2018*. Ed. by Daniel Le Berre and Matti J\”arvisalo. Vol. 59. EPiC Series in Computing. EasyChair, 2019, pp. 50–64. DOI: 10.29007/3qkh. URL: https://easychair.org/publications/paper/6HpF.

[9] E. K. Burke and S. Petrovic. “Recent research directions in automated timetabling”. In: *European Journal of Operational Research* 140.2 (2002), pp. 266–280. DOI: 10.1016/S0377-2217(02)00004-6.

[10] Vijay Durairaj and Priyank Kalla. “Variable Ordering for Efficient SAT Search by Analyzing Constraint-Variable Dependencies”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 415–422. ISBN: 978-3-540-31679-4.

[11] B. K. F. M. Fleury and M. Heisinger. “Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020”. In: *SAT COMPETITION* 50 (2020), p. 2020.

[12] Zhaohui Fu and Sharad Malik. “On Solving the Partial MAX-SAT Problem”. In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 252–265. ISBN: 978-3-540-37207-3.

[13] Holger H. Hoos. “An adaptive noise mechanism for walkSAT”. In: *AAAI/IAAI*. 2002.

[14] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. “PySAT: A Python Toolkit for Prototyping with SAT Oracles”. In: *SAT*. 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26. URL: https://doi.org/10.1007/978-3-319-94144-8_26.

[15] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. “RC2: an Efficient MaxSAT Solver”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11 (Sept. 2019), pp. 53–64. DOI: 10.3233/SAT190116.

[16] Saurabh Joshi, Ruben Martins, and Vasco Manquinho. “Generalized Totalizer Encoding for Pseudo-Boolean Constraints”. In: *Principles and Practice of Constraint Programming*. Ed. by Gilles Pesant. Cham: Springer International Publishing, 2015, pp. 200–209. ISBN: 978-3-319-23219-5.

[17] Daniel Le Berre and Anne Parrain. “The Sat4j library, release 2.2”. In: *JSAT* 7 (Jan. 2010), pp. 59–6.

[18] Chu Min Li and Felip Manyà. “MaxSAT, Hard and Soft Constraints”. In: *Handbook of Satisfiability*. 2021. URL: https://api.semanticscholar.org/CorpusID:28884712.

[19] J. Marques-Silva and J. Planes. “Algorithms for Maximum Satisfiability Using Unsatisfiable Cores”. In: *Conference on Design, Automation, and Testing in Europe*. 2008, pp. 408–413.

[20] Ruben Martins et al. “On Using Incremental Encodings in Unsatisfiability-based MaxSAT Solving”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (Dec. 2015), pp. 59–81. DOI: 10.3233/SAT190102.

[21] Antonio Morgado, Carmine Dodaro, and Joao Marques-Silva. “Core-Guided MaxSAT with Soft Cardinality Constraints”. In: *Principles and Practice of Constraint Programming*. Ed. by Barry O’Sullivan. Cham: Springer International Publishing, 2014, pp. 564–573. ISBN: 978-3-319-10428-7.

[22] Antonio Morgado et al. “Iterative and core-guided MaxSAT solving: A survey and assessment”. In: *Constraints* 18 (Oct. 2013). DOI: 10.1007/s10601-013-9146-2.

[23] Nina Narodytska and Nikolaj Bjørner. “Analysis of Core-Guided MaxSat Using Cores and Correction Sets”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 26:1–26:20. ISBN: 978-3-95977-242-6. DOI: 10.4230/LIPIcs.SAT.2022.26. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16700.

[24] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. 5th edition. San Francisco, CA: Morgan Kaufmann, 2011. ISBN: 978-0123850591.

[25] S. S. SHAPIRO and M. B. WILK. “An analysis of variance test for normality (complete samples)”. In: *Biometrika* 52.3-4 (Dec. 1965), pp. 591–611. DOI: 10.1093/biomet/52.3-4.591. URL: https://doi.org/10.1093/biomet/52.3-4.591.

[26] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. 8th edition. Hoboken, NJ: Wiley, 2009. ISBN: 978-0470128725.

[27] Carsten Sinz. “Towards an optimal CNF encoding of boolean cardinality constraints”. In: *Principles and Practice of Constraint Programming - CP 2005* (2005), pp. 827–831. DOI: 10.1007/11564751_73.

[28] T. A. Toffolo and J. Vanschoren. “Optimizing resource allocation with soft constraints in project scheduling”. In: *European Journal of Operational Research* 224.2 (2013), pp. 340–349. DOI: 10.1016/j.ejor.2012.08.010.

[29] Ed Wynn. “A comparison of encodings for cardinality constraints in a SAT solver”. In: *ArXiv* abs/1810.12975 (2018). URL: https://api.semanticscholar.org/CorpusID:53108040.

[30] Lintao Zhang and Sharad Malik. “Conflict driven learning in a Boolean satisfiability solver”. In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*. IEEE. 2002, pp. 160–164.