Karlsruher Institut für Technologie

Bachelor thesis

Compression of Propositional Models

Valentin Schenk

Date: 23. September 2024

Supervisors: Prof. Dr. Peter Sanders
Dr. rer. nat. Dominik Schreiber
Dr. rer. nat. Markus Iser

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 23.09.2024

Abstract

As SAT solvers become more powerful, the instances that they solve get bigger and bigger. With the increasing size of the SAT instances, the propositional models, that are outputted by the SAT solvers, also get bigger and take up more space. In order to decrease the size of the propositional models that are permanently stored, we propose an algorithm to compress propositional models. We also describe the corresponding decompression algorithm . The compression algorithm uses unit propagation to find variables for which the assignment can be derived from different variable assignments. The assignments for these variables do not have to be stored explicitly, as it can be derived from elsewhere. Our algorithm finds a reduced set of variable assignments from which all other assignments can be derived. This set is the core of the compressed model. We use different methods to further minimize the size of the compressed models such as ordering heuristics and off-the-shelf compression algorithms. During the experimental evaluation, our algorithm achieved compression ratios up to over 18 compared to storing the model as a bitvector with an average execution time of 119 seconds per model.

Zusammenfassung

Da SAT-Solver immer leistungsfähiger werden, werden auch die Instanzen, die sie lösen, immer größer. Mit den wachsenden SAT-Instanzen, werden auch die Modelle, die von den SAT-Solvern ausgegeben werden, immer größer und brauchen mehr Speicherplatz. Um den Speicherplatzverbrauch von permanent gespeicherten Modellen zu verringern, stellen wir einen Algorithmus zur Kompression von aussagenlogischen Modellen vor. Außerdem stellen wir den dazugehörigen Algorithmus zur Dekompression vor. Der Kompressionsalgorithmus verwendet Unit-Propagation, um Variablen zu finden, deren Belegung sich durch andere Variablenbelegungen herleiten lässt. Die Belegungen dieser Variablen müssen nicht explizit gespeichert werden, da sie hergeleitet werden können. Unser Algorithmus findet eine kleinere Menge an Variablenbelegungen, aus denen sich alle anderen Belegungen herleiten lassen. Diese kleinere Menge an Belegungen ist der Kern des komprimierten Modells. Wir verwenden verschiedene Methoden und Techniken, um die Größe der Menge an Variablenbelegungen weiter zu reduzieren. Unter anderem verwenden wir Heuristiken, um die Variablen zu sortieren, und bereits vorhandene Kompressionsalgorithmen. Während der experimentellen Evaluation haben wir Kompressionsraten von bis zu über 18, im Vergleich zu einem Modell, das als Bitvektor gespeichert ist, erreicht und die durchschnittliche Ausführungszeit liegt bei 119 Sekunden pro Modell.

Contents

1 Introduction

In this chapter, we explain the motivation behind this thesis and describe our contributions.

1.1 Motivation

The Boolean satisfiability problem, also known as SAT, describes the problem of determining if it is possible to satisfy a given set of propositional clauses. Programs that try to solve the SAT problem are called SAT solvers, and they play an important role in formal verification [24] and the automation of electronic designs [21]. If a SAT solver finds that a set of propositional clauses is satisfiable, it outputs the assignment of the variables to either $TRUE$ or $FALSE$ that satisfies all clauses. This assignment is also called a propositional model. As SAT solvers become more powerful, the instances that they can solve continuously become bigger and bigger [10]. As a result of that, the models that are found by the SAT solvers also get bigger and contain more variables. Modern SAT solvers can solve instances that contain hundreds of thousands of variables. If the instance is satisfiable and a model is found, the assignment for every variable has to be stored explicitly, which takes up more space, with increasingly bigger models. This becomes relevant, when the models are stored permanently, as for example in the Global Benchmark Database (GBD) [16].

1.2 Contribution

In order to reduce the space that is needed to store propositional models, we propose an algorithm that compresses these models and significantly reduces the space needed to permanently store them. The algorithm works by using unit propagation to find variable assignments that can be derived from other assignments. So the algorithm finds a reduced set of variable assignments, from which all other assignments of the model can be derived. This means that it is sufficient to only store this reduced set of assignments, as the rest of the model can be reconstructed with this set. Multiple additional techniques are used to increase the efficiency of the compression algorithm and decrease the size of the compressed model. For example, ordering heuristics, which are also used in many SAT solvers, are used to minimize the number of assignments needed for the compressed models and additional compression algorithms are used to further compress the reduced set of assignments. In addition to the compression algorithm, we propose a decompression algorithm, that converts compressed models back into the original models. This is done by also using unit propagation and the assignments from the compressed model to derive the remaining variable assignments that were not part of the compressed model. During the decompression, there are also multiple techniques needed in order to fully restore the original model, which are described in detail in the following chapters.

In the experimental evaluation, we tested different ordering heuristics and generic compression algorithms. For the different combinations, we compared the compression ratios and execution times. During the test, we found out that the choice if a static or a dynamic ordering heuristic is used, has a big impact on these values. With static heuristics we achieved average compression ratios over 6.5 compared to storing the original model as bitvector and with dynamic heuristic compression ratios over 18 were achieved. So, dynamic heuristics achieve better compression ratios than static heuristics, but they take more time to compute, which impacts the execution time. The average execution time for static heuristics was measured at around 45 seconds, whereas for dynamic heuristics, the average execution time was around 119 seconds.

1.3 Structure of Thesis

The thesis is structured into multiple chapters. In Chapter 2, the main concepts are introduced that are needed to understand the rest of the thesis. In Chapter 3 the algorithms are explained in detail. First, the algorithmic details of the compression algorithm are described and then the same is done with the decompression algorithm. After that, in Chapter 4, some interesting details about the implementation of the algorithms are given. In the last chapter, chapter 5, the results of the experimental evaluation are presented and the effectiveness and performance of the algorithms are proven.

2 Preliminaries

In this chapter we introduce the concepts and algorithms that are used in the thesis and that are necessary to understand the rest of the work.

2.1 The SAT Problem

The Boolean satisfiability problem (SAT) is the problem of determining if a given set of propositional formulas is satisfiable. It is the first problem that was proven to be NP-complete by Levin and Cook [5]. A SAT instance consists of a set of variables and a set of boolean clauses that are constructed from these variables. A boolean clause is built from literals and boolean operators like OR ($\vee$) or AND ($\wedge$). A literal is either a variable or a negated variable. The clauses are normally given in conjuctive normal form (CNF). A clause is in conjunctive normal form if it consists of the logical OR of one or more literals. For example, x is a positive literal, $\neg y$ is a negative literal and $x \vee \neg y$ is a clause in CNF.
A SAT-instance is satisfiable if there is an assignment to the variables to either $TRUE$ or $FALSE$, so that all clauses are satisfied. In this case, the variable assignment is called a model. If there is no possible assignment that satisfies all clauses, the instance is unsatisfiable.
An example for a satisfiable SAT instance are the clauses $x \vee y$ and $\neg x \vee \neg y$, because the assignment $x = TRUE, y = FALSE$ satisfies both clauses and therefore the whole instance is satisfiable.
Programs that try to find a model for a given SAT instance are called SAT solvers. As SAT is NP-complete, there theoretically is no real efficient algorithm and the worst case runtime is always exponential relative to the input size. However, there are multiple techniques to make the search more efficient. Most modern SAT solvers use conflict-driven clause learning (CDCL) [22]. The CDCL algorithm works by choosing a variable and assigning a value to it. Then unit propagation, which is explained in more detail below, is used to simplify the clauses. These two steps are repeated until either all clauses are satisfied or a conflict is found. In case of a conflict, a new clause is derived from this conflict which ensures that this variable assignment is not tried again. The search is then restarted from an earlier state and the process continues until the SAT instance is either satisfied or found to be unsatisfiable. The algorithm that is presented in this thesis takes many concepts and techniques from CDCL SAT solvers and the most important of them are described in more detail below.

The most common use cases for SAT solvers are formal verification [24] and the automation of electronic designs [21]. Also, because SAT is NP-complete, many interesting problems can be converted into a SAT instance, such as the vertex cover problem or scheduling problems [12] which are important for many different areas. So, the solving of SAT instances is also relevant for the solving of many other hard problems.

2.2 Unit Propagation

Unit Propagation is an algorithmic concept used in most modern SAT solvers. It was first introduced in the Davis-Putnam-Logemann-Loveland (DPLL) algorithm in 1961 [7] which is the base for modern CDCL SAT solvers. Unit Propagation is used to simplify a set of propositional clauses by using implications between the different clauses.
The technique bases on so called unit clauses which are clauses that only contain one literal. For example, x or $\neg y$ are unit clauses. For a SAT instance to be satisfied, all the clauses have to be satisfied, so for unit clauses there is only one possible assignment to achieve that goal. In the previous example that would be $x = TRUE$ and $y = FALSE$. Because the assignment of the variable in the unit clause is fixed, it is possible to simplify other clauses by applying one of two rules. The first rule states that all clauses that contain a literal assigned to $TRUE$ can be deleted. This is because the whole clause is satisfied if one literal evaluates to $TRUE$. The second rule states that in every clause that contains a literal assigned to $FALSE$, that literal is deleted from the clause because it cannot be satisfied if the unit clause is satisfied. By applying these rules, it is possible that new unit clauses are created which can be used to simplify the clauses further.
For instance, the set of clauses $\{x \vee y, \neg x \vee \neg y \vee z, x\}$ can be simplified to $\{\neg y \vee z, x\}$ using unit propagation.
Most SAT solvers work by assigning a value to a variable and then using unit propagation to simplify the clauses to minimize the number of assignments that have to be made to satisfy all clauses. Unit propagation also plays a very important role in the algorithm that is described in this thesis.

2.3 Ordering Heuristics

Branching heuristics are another technique used in most modern SAT solvers to make them more efficient. As mentioned above CDCL SAT solvers work by assigning a value to a variable and then using unit propagation to simplify the clauses. Instead of assigning a value to a random variable, branching heuristics try to find the variable and the assignment to this variable that maximizes the number of clauses that get satisfied in the next propagation step. This makes the rest of the problem easier to solve, as there are less clauses that still need to be satisfied which increases the efficiency of the SAT solver. In the algorithms that are described here, the heuristics are only used to establish an order between the variables

and not to assign the variables. Because of that, in the following, the heuristics will be calles ordering heuristics instead of branching heuristics. The heuristics work by calculating and assigning a value to each variable which is depending on different factors for different heuristics. The variables are then sorted by their values and get assigned in that order. Heuristics can either be static or dynamic. If a heuristic is static, the heuristic values are calculated once and are not changed after that. With dynamic heuristics, the values get updated as the propagation steps are made.
Ordering heuristics also play a major role in this thesis and in the following the heuristics that were used are explained in more detail.

2.3.1 MOMS heuristic

The term ”Maximum Occurrences - Minimum Size” (MOMS) does not describe a single heuristic, but a whole group of ordering heuristics that all base on the same general idea. However, in this thesis the term "MOMS heuristic" will refer to the heuristic that was proposed by Jon William Freeman in his thesis in 1995 [11]. As the name already suggests, this heuristic focuses on the variables that have the most occurrences in clauses of minimum size. The size of a clause is defined by the number of literals that occur in the clause.
The heuristic value of a variable v is calculated with the following formula:

$$h(v) = f^*(v) \cdot f^*(\neg v) \cdot 2^n + f^*(v) + f^*(\neg v) \tag{2.3.1}$$

Where $f^*(v)$ is the number of occurrences of v in unsatisfied clauses of minimal length and n is a constant integer value.
The values get calculated for each variable and then they are ordered, so that the variable with the highest heuristic value gets assigned first. This ordering heuristic can either be used as a static heuristic or as a dynamic, if the values are constantly updated as clauses are satisfied.

2.3.2 Jeroslow-Wang

The Jeroslow-Wang branching heuristic was introduced in 1990 and is named after its authors Robert G. Jeroslow and Jinchang Wang [17]. Similar to the MOMS heuristic, this heuristic is also based on the length of the clauses, but instead of only focusing on the clauses of minimum size the Jeroslow-Wang heuristic considers clauses of all sizes in the calculation.
For this heuristic, the values for each variable v are calculated using the following formula:

$$h(v) = \sum_{v \in c, c \in F} 2^{-|c|} \tag{2.3.2}$$

Where F is the set of all clauses in the SAT instance, c is a clause from that set and $|c|$ is the number of variables that are contained in the clause.

As seen in the formula, the heuristic includes all clauses in which the variable is contained and weights them by their size. So, occurrences in clauses with less variables have a bigger impact on the heuristic value that occurrences in bigger clauses. The variables are also ordered so that the variable with the highest heuristic values gets assigned first. The Jerowlow-Wang heuristic can also either be used as a static or a dynamic heuristic.

2.4 Compression Algorithms

Compression algorithms are algorithms that take data as input and encode them into a representation that takes up less space than the original. There are two main categories of data compression, lossless and lossy compression. Lossless compression encodes the data in a way, so that it is possible to retrieve the original data without any loss of information. In most cases, the algorithms use statistical redundancies in the input data to achieve this form of compression. Two popular examples for lossless compression methods are the Huffman code [15] and the Lempel-Ziv [25] algorithms that are explained in more detail below. On the other hand, lossy compression encodes the data in a way where some information is dropped and not recoverable. This method is mostly used in image and audio processing, as it is acceptable there to lose some information to save space. Many popular file formats make use of lossy compression, such as the JPEG format [1] for images or the MP3 format for audio [2]. But in this thesis, the focus lies on lossless compression, as these are the only ones used during model compression, because the compressed model should be reconstructed perfectly without any loss of data.

2.4.1 Generic compression algorithms

There are two generic compression algorithms used on this work. These algorithms are only used from external libraries and are not implemented as part of the thesis. In the following, we will briefly explain the main concepts of the two algorithms.

The first algorithm is LZ4 [19]. It is a lossless compression algorithm that was developed by Yann Collet and first introduced in 2011. LZ4 is based on the concept of the LZ77 compression algorithm [25]. LZ77 compressed data by replacing sequences that occur multiple times in the input data with a reference to the first occurrence of that sequence. So only one instance of each sequence and the references to that sequence have to be stored. LZ4 builds on that approach and uses its own format to store the sequences and references.

The second compression method that is used is ZIP. ZIP is a file format that supports lossless data compression. It allows multiple compression algorithms, but the most popular is the DEFLATE algorithm [9] which is a combination of the previously mentioned LZ77 algorithm and Huffman coding. A Huffman code is a variable-length code that encodes characters by their probabilities of occurrence in the input data. Characters that occur more often in the input data are represented with fewer bits than characters with less occurrences in the input data. Using these different representations of the characters, the data gets compressed losslessly. The DEFLATE algorithm compresses data in two stages. During the first stage, the data gets compressed using the LZ77 algorithm and after that the resulting representation is compressed again using a Huffman code. Zip compression is mostly used to compress one or multiple files but in this work it is also used to compress raw data.

2.4.2 Golomb-Rice codes

Golomb-Rice coding is another lossless compression method [14]. In contrast to the previously mentioned methods, it is only possible to encode integer values with this code. It is a variable-length code where the length of the encoding is depending on the probabilities of the different values. Smaller values need less space than bigger values, so the Golomb-Rice code is efficient in applications, where smaller values are a lot more common than bigger ones.

For the encoding a tuneable parameter r, also called the divisor, is used which is a power of two. The value that should be encoded is then divided by the divisor and the quotient and the remainder of the division is saved. The quotient is encoded with a unary encoding which either consists of only zeros or ones. This unary encoded value is the first part of the whole encoded value. After that, a separation bit follows to mark the end of the unary encoding. The separation bit must have the opposite value of the unary encoded value. The remainder then gets encoded using a binary encoding, consisting of $log_2(r)$ bits. This value is then appended to the other encoded values and completes the Golomb-Rice encoded value.

For example, if the divisor $r = 4$ and the value 6 should be encoded, the division results in $6/4 = 1 \cdot 4 + 2$. So, the quotient is 1 and the remainder 2. The unary encoding of 1 with ones would be 1 and the binary encoding of 2 would be 10. Together with the separation bit, the final Golomb-Rice encoded binary value is 1010.

The encoded values can be decoded again by counting the bits until the separation bit to get the quotient. Because the length of the remainder can be derived from the divisor, the remainder can just be retrieved by reading the correct number of bits after the separation bit. The quotient is then multiplied with the divisor and after the remainder is added, the original values is restored.

2.5 Related Work

To our knowledge, there are currently no publications that deal directly with the topic of propositional model compression. However, there are other areas in the context of SAT solving where compression is used.

One of these areas is the compression of formal proofs. As the SAT problem is NP-hard, it is not possible to solve in polynomial time, but it is possible to verify a solution efficiently. If a SAT instance is satisfiable and a model was found, it can easily be verified by assigning the variables and checking if all clauses are satisfied. If a SAT instance is not satisfiable, most modern SAT solvers generate a formal proof to show why there exists no model for this instance. Many of the solvers use clausal proofs [8] to represent these formal proofs which will not be explained in detail here. One problem with clausal proofs is that they can get exponentially big compared to the initial instance [3]. To reduce this problem, there are multiple techniques and algorithms that aim to compress these proofs to reduce their size and make them easier to be verified. Two methods of compression were introduced in 2010 by Scott Cotton [6]. The first method uses structural hashing to extract shared structure from a proof that can be merged to decrease the size of the proof. The second method uses a sequence of so called splitting operation to re-arrange the proof in a way that is takes up less space. Another algorithm is called RecycleUnits that was first proposed by Bar-Ilan et al. in 2009 [4]. This method uses unit clauses in the proof to achieve the compression, which also will not be described in detail here. In the same paper, another algorithm called Recycle-Pivots is presented, which uses so called pivot variables to compress proofs. This method is also not described in detail here.

Another area where compression was tried is the descriptions of logic formulas and clauses. This was done in a bachelor thesis by Jens Manig in 2018 [20], that tests multiple approaches to compress propositional clauses that are stored in the Dimacs CNF file format.

3 The Compression and Decompression Algorithms

In this chapter the main approaches and algorithmic concepts of the compression and decompression algorithms are explained.

3.1 General Overview

The approach consists of two separate algorithms, the compression and the decompression algorithm. The compression algorithm is responsible for encoding and, as the name suggest, compressing the model. On the other hand, the decompression algorithm is responsible for decoding the compressed data and reconstructing the original model.

The general pipeline of the compression algorithm is shown in Figure 3.1. The algorithm takes a propositional model and a set of clauses as input and outputs the compressed model. The main part consists of a loop that contains the ordering heuristic, the prediction model and unit propagation. First, a variable is chosen by using the ordering heuristic. This variable is then assigned with the value from the model and passed to the prediction model which is used to increase the efficiency of the algorithm and is explained in more detail below. After that, unit propagation is performed to simplify the clauses. If there are still unsatisfied clauses after the unit propagation, the loop continues and a new variable is chosen. But if all clauses are satisfied, the loop stops and an additional compression algorithm is used to further compress the result. The output of this compression model is the compressed model that is returned by the algorithm. The whole compression process is described in detail in Section 3.2.

The compressed models are converted back into the original model by the decompression algorithm. An overview of the pipeline is shown in Figure 3.2. The first step of the decompression process is to reverse the compression that was done as the last step of the model compression. This is done by passing the compressed model into the decompression algorithm that matches the one with which it was compressed. After that, the decompression algorithm consists of a loop that is very similar to the loop used in the compression algorithm. The only difference is the prediction model which is used to restore the assignment of the chosen variables. The loop is also executed until all clauses are satisfied and after that the original model is returned.

In the following sections, the different pipeline steps are explained in more detail. In Section 3.2, the compression process is explained and all the steps are described there. The Ordering heuristics, Prediction model, Unit propagation and the compression algorithms are described in this section. In Section 3.3, it is explained how the decompression works, which is done by describing the differences to the compression algorithm, as they are very similar. So, the decompression algorithms and the prediction model decoder are described in this section, but the ordering heuristics and unit propagation are not explained again.

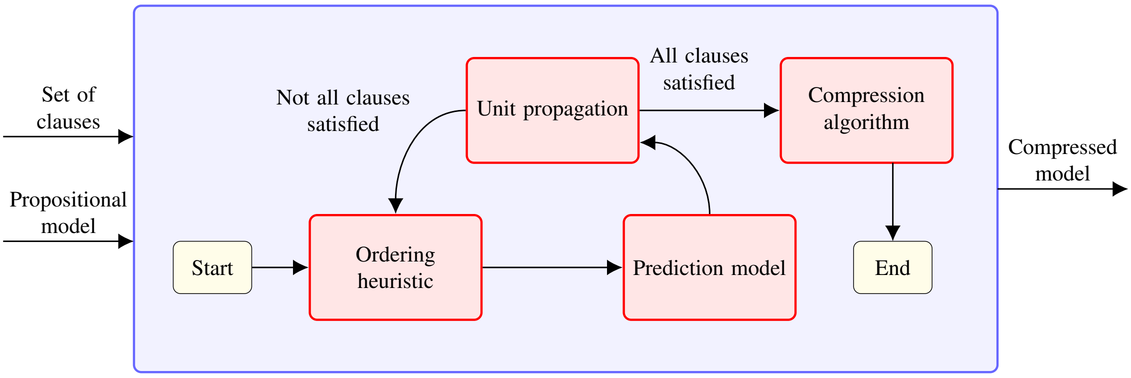

Figure 3.1: An overview of the compression algorithm.

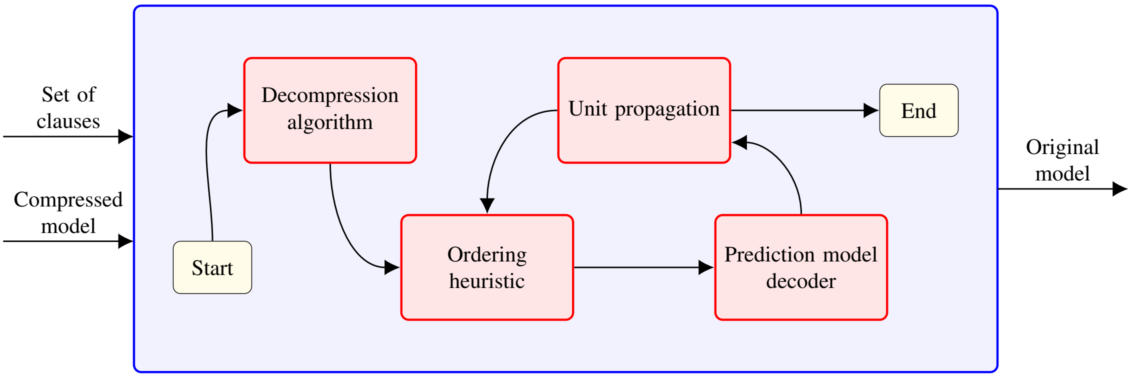

Figure 3.2: An overview of the decompression algorithm.

3.2 Compression

3.2.1 Overview

There are many different ways to compress propositional models. The easiest way is to just change the representation of the model. For instance, models are normally stored by saving an integer value for each variable that either has a ”-” in front of it or not to show the assignment of the variable. A possible representation of the model, that would take up less

space, would be to simply store the model as a bitvector, where a 0 represents a negative assignment, 1 a positive assignment and the position in the bitvector would represent the variable.
But there are also more complex techniques that can be used to compress propositional models. The approach that is followed in this thesis uses unit propagation to achieve compression. The models are compressed by using the unit clauses that were found during unit propagation. Because unit clauses only have one valid assignment in the current context, the value of the variable can be derived from the other assigned variables and does not have to be stored explicitly. For example, if the clause $x \vee y$ is given and x is assigned with $FALSE$. In this case, y is a unit clause and the only possible way to satisfy the clause is by assigning y with $TRUE$. But because the value of y can be derived from the assignment of x, it is not necessary to store the value of y and it is enough to save the assignment of x. This is the main concept used in this thesis to compress the models, so the structure of the algorithm is similar to those of CDCL SAT solvers, where a variable is chosen, assigned and the unit propagation is performed. This is repeated until all clauses are satisfied and a smaller set of variable assignments is determined, from which all other assignments can be derived.
As already shown in Section 3.1, there are multiple techniques used, like ordering heuristics and the prediction model, to increase the compression ratio and efficiency of the compression algorithm. These techniques are described in more detail below.

3.2.2 Ordering Heuristics

After the model and the clauses are parsed and before the actual compression can start, the heuristic values for each variable have to be calculated. This is necessary to establish an order in which the variables are assigned during the compression process, which aims to make the compression more efficient and achieve a higher compression ratio. In SAT solvers, ordering heuristics are used to increase the efficiency by choosing the variables in an order, so that the number of assignments that are derived during unit propagation is maximized. This also helps with model compression, because if more assignments can be derived during unit propagation, less variables have to be assigned explicitly. That means that the compression ratio is increased, because less variable assignments have to be stored explicitly. The variables are sorted by the heuristic values and stored in a priority queue, because this data structure allows efficient ordered access of the variables. The heuristic values are calculated once for every variable before the main compression starts. If the heuristic, that is used, is used as a static heuristic, the heuristic values and the order of the variables are not changed after the initial calculations. If the heuristic is used as a dynamic heuristic, the values are updated during the compression process and the order of the variables can change dynamically. We consider three different heuristic methods.
The first possible method to order the variables is by not using a heuristic at all. In this case the variables are assigned in the order in which they were parsed by the program.

This method requires no additional computations, but it also does not improve the final compression ratio, as shown in Section 5.2.

The second method for variable ordering is using the Jeroslow-Wang heuristic [17]. The heuristic values for each variable are calculated using the Formula 2.3.2 which was already described in more detail in Section 2.3.2. When the heuristic is used statically, the values are calculated once for every variable. Then the order is established using these values, so that the variable with the highest heuristic value is assigned first and then this order is not changed during the compression. On the other side, if the heuristic is used dynamically the values are updated during the propagation. Every time a clause is satisfied, the heuristic value of each variable in this clause changes, because the clause is no longer relevant for the heuristic. Because the values are a sum of terms for every clause, it is possible to update the values by simply subtracting the term that belongs to the satisfied clause. With this method, the values can be updated for every variable v in a satisfied clause using the formula:

$$h_{new}(v) = h_{old}(v) - 2^{-|c_{sat}|} \tag{3.2.1}$$

where c_{sat} is the clause that was satisfied. Making these updates needs additional computation time, but because the values are always adjusted to the current state, the heuristic gets more precise which can lead to better compression ratios.

The third method for establishing an order between the variables, is the MOMS heuristic, as it was described in Section 2.3.1. This heuristic follows a similar approach to Jeroslow-Wang, but instead of considering all clauses, the MOMS heuristic only uses the clauses of minimum size. So, before the values for each variable can be calculated, the minimum size of all clauses has to be determined. To do this, the algorithm iterates over all clauses and compares the size of the current clause with the minimum size of the previous clauses. In addition to the minimum size, the number of clauses that have minimum size is also stored and is relevant for the dynamic use of the heuristic. After these values are determined, the heuristic value for each variable can be calculated using the Formula 2.3.1 which is also further described in Section 2.3.1. If the heuristic is used statically, these values are then used to order the variables and no further changes to the heuristic values are made during the compression. But if the MOMS heuristic is used dynamically the values change during the compression process. Similar to Jeroslow-Wang, the value of a variable has to be updated, if the variable is contained in a satisfied clause of minimum size. Which calculations have to be made to update the variable is dependent on whether the variable occurs as a positive or negative literal in the clause. For example, if the variable occurs as a positive literal in the satisfied clause, the formula for the heuristic value can be displayed as

$$h_{new}(v) = ((f^*(v) - 1) + f^*(\neg v)) \cdot 2^n + (f^*(v) - 1) \cdot f^*(\neg v) \tag{3.2.2}$$

where $f(v)$ takes the occurrences from before the clause was satisfied. This formula can then be simplified to:

$$h_{new}(v) = (f^*(v) + f^*(\neg v)) \cdot 2^n + f^*(v) \cdot f^*(\neg v) - 2^n - f^*(\neg v) \tag{3.2.3}$$

which can also be displayed as:

$$h_{new}(v) = h_{old}(v) - 2^n - f^*(\neg v) \tag{3.2.4}$$

On the other hand, if the variable occurs as negative literal the resulting formula would be:

$$h_{new}(v) = h_{old}(v) - 2^n - f^*(v) \tag{3.2.5}$$

Because 2^n is a constant, it can easily be subtracted and for the rest, the number of occurrences of the inverted literal has to be determined and then this number is also subtracted from the initial heuristic value. If all clauses of the current minimum size have been satisfied, the minimum clause size has to be newly determined and the heuristic values for all variables, that are not assigned yet, have to be recalculated. This procedure is repeated until all clauses are satisfied. In the same way as with the Jeroslow-Wang, the dynamic use of the MOMS heuristic adds computational complexity, but in return it can increase the compression ratio.

3.2.3 Prediction Model

The prediction model sits between the ordering heuristic and unit propagation and aims to improve the compression ratio by predicting the assignment of the variables. If the prediction is correct, the assignment of the variable can be inferred by the decompression algorithm by using the same prediction model. So, in this case, it is not necessary to explicitly store the assignment, which reduces the size of the compressed model. The model works by the assumption that a variable is more likely to be assigned with the value with which it occurs in more unsatisfied clauses. So, if a variable occurs in more unsatisfied clauses as positive literal than it occurs as negative literal the variable is more likely to be assigned with $TRUE$ than with $FALSE$. This prediction is then compared to the actual value, that the variable gets assigned with by the model. If the value is identical to the guess of the prediction model, the prediction is right and if the values are different, the prediction is wrong. If the variable in the previous example would be assigned with $TRUE$, the prediction would be correct.

Every time a new variable is chosen by the heuristic, the prediction model is applied to it and a boolean value is saved that indicates whether the prediction is correct or not. So, after all clauses are satisfied, there is a set of boolean values which shows for every assigned variable if the prediction was correct. These boolean values could be used as a bitvector that represents the compressed model, because the information about the prediction would be enough to know how each variable has to be assigned. However, this representation would take the same space as storing the assignment directly as a bitvector. It is possible to compress the prediction model values further. This is done by using a delta encoding. The delta encoding looks at the values in the order in which they were created and only stores the distances between the two nearest $FALSE$ values. For example, if the set of boolean values $\{0, 1, 1, 0, 1, 0\}$ would be encoded to $\{0, 2, 1\}$. So, if the prediction model is accurate and the predictions are correct for a majority of the variables, the size of the delta encoded values becomes smaller in comparison to the initial boolean values which then increases the compression ratio.

To increase the accuracy of the prediction model, the general assumption can be inverted throughout the compression. This happens, if the predictions are wrong multiple times in a row. We propose to set a fixed number of times which the prediction model can repeatedly be wrong. If this number is exceeded, the prediction model in inverted. This means that from this point on the prediction is correct if the assigned value is the one with which the variable occurs in less unsatisfied clauses, which also means that all Boolean values are inverted. The prediction model can flip multiple times during the compression if the inverted model is also wrong for more times in a row than the fixed parameter allows. This technique needs no additional space, because the inversion can be detected implicitly by the decompression algorithm.

3.2.4 Unit Propagation

Unit propagation is the key concept to compress the models. It is executed after a variable was chosen by the heuristic and assigned with the value from the original model. Unit propagation tries to simplify the remaining clauses and tries to find variables whose assignment can be derived from the previous assignments. The general algorithm is explained in Section 2.2. In the following, we outline how unit propagation is used in the model compression algorithm.

Every time a variable gets chosen and assigned with a value, it is inserted into a queue before unit propagation is executed. The propagation consists of a loop that is executed until there are no more variables in the queue. In this loop, always the first variable of the queue is taken as the variable that will be propagated. The main propagation is similar to the procedure described in Section 2.2 and the two rules are applied to the clauses in which the variable, that is currently propagated, occurs. But there are some things that are different from the standard unit propagation.

The first difference is, that the values needed for the prediction model need to be updated. For every variable that occurs in a clause that is satisfied in the current propagation step, the number of unsatisfied clauses in which it occurs has to be updated. These values are needed to make the prediction as described in Section 3.2.3.

The second difference is that the heuristic values have to be updated. This also has to be done for all variables that occur in a clause that is satisfied during the current propagation step. For each of these variables, the heuristic value is updated as described in Section 2.3. So, for example, if given the clause $x \vee y \vee z$, the variable x is assigned with $TRUE$ and propagated, the clause gets satisfied. So, the values for the prediction model and the heuristic values have to be updated for the variables y and z.

The rest of the unit propagation is similar to the procedure described above. If a unit clause is found during unit propagation, the variable is inserted into the propagation queue. The propagation continues until the queue is empty. After that a new variable has to be picked and assigned.

In most modern CDCL SAT solvers, unit propagation is the part that requires the most computation. It is the operation that is performed the most often, because the solver has to backtrack and restart when a conflict is encountered. As a consequence of that, unit propagation often is a bottleneck in SAT solvers. However, this is not a problem in the here described algorithm, because the model is known beforehand. Therefore, there are no conflicts and restarts and the unit propagation has to be executed only once for each variable. So, a lot of performance issues that exist in SAT solvers do not pose a problem in the model compression algorithm.

3.2.5 Compression Algorithms

After all propagation steps are made and all clauses are satisfied, the resulting product is a set of boolean values from the prediction model, which represent the assignments that were taken from the original model, as described in Section 3.2.3. These values are then compressed further using the ZIP compression, the LZ4 algorithm and Golomb-Rice codes which are described in more detail in Section 2.4. The algorithms for ZIP and LZ4 compression are not implemented as part of this thesis and are used with external libraries. However, the Golomb-Rice encoding is implemented in this work.

Some of the algorithms need a different representation of the input data than others.

The ZIP and LZ4 compression methods need a string value as input, so the values from the delta encoding have to be converted. This is done by appending all values to one another and separating them with a whitespace character. The resulting string is used as input for the compression library and then compressed with the respective algorithm. The output of the compression is also a string value that contains the compressed data.

The Golomb-Rice encoding can only compress integer values, so the values from the delta encoding do not have to be converted and can just be passed to the compression algorithm. Each integer value is compressed separately. The values are considered in the order in which the values were passed. The bit representations of the compressed integers are then appended to each other and converted into a string. This string is then returned by the compression algorithm.

After the compression is done, the resulting string is written to the output file and the model was successfully compressed.

3.3 Decompression

3.3.1 Overview

The goal of the decompression algorithm is, to convert the compressed model back into the original model. In order to achieve this, variables have to be assigned with values from the compressed model and unit propagation has to be performed to restore the assignments of the variables, that were not part of the compressed model. So, the general decompression process is very similar to the compression algorithm, but some steps have to be adjusted, as already shown in Section 3.1. The first change is at the beginning of the pipeline, where the compressed model is input into a decompression algorithm which decodes the data that was compressed during the last step of the compression. These decompression algorithms are explained in more detail below. The main loop of the decompression is the same as in the compression algorithm where a variable is chosen, assigned with a value, and the unit propagation is performed until all clauses are satisfied. The main difference in this loop between the compression and decompression algorithm is the prediction model. During the compression, the prediction model is used to make some assignments implicit. During the decompression, this information, that comes from the compression prediction model is decoded and used to assign the chosen variables with the correct values. This is explained in more detail below. After the main loop has finished, the assignments of all variables are successfully restored and the original model can be returned.

3.3.2 Decompression Algorithm

As seen in Section 3.2, the compressed model consists of the prediction model values that were encoded using an additional generic compression algorithm. So, in order to use the prediction model values to decompress the model, they first have to be decoded.
If the ZIP or LZ4 compression algorithms (see Section 2.4) are used, the compressed model is simply passed as string into the corresponding library. The library then decodes the string and returns another string which then contains the prediction model values separated by whitespace characters. This string is then converted into a set of integer values which contains the original prediction model values.
If the Golomb-Rice code (Section 2.4.2) is used, the decoding is not handled by an external library and is implemented as part of the decompression algorithm. The compressed model is also submitted as a string. From the binary data of the string, the prediction model values can be decoded one by one, in the order in which they appear in the compressed model. The values are decoded by using the method already described in Section 2.4.2. When all prediction model values are decoded successfully, they are returned and can now be used to decompress the actual model.

3.3.3 Prediction Model

In the compression algorithm, the value that is propagated is assigned with the value from the original model. For obvious reasons, this is not possible for the decompression algorithm. So, the decompression has to use the information provided by the compressed model which consists of the prediction model values to assign the variables.

The values are considered in the order in which they occurred in the compressed model. After the first variable is chosen, the first prediction model value is loaded. As described in Section 3.2.3, the prediction model values are delta encoded, so each value describes the distance between two false predictions. These distances are then used to determine the assignments of the variables. They are considered one by one and are used as counter to determine when the next prediction is false. If the prediction for the current variable is correct, it is assigned according to assumption described in Section 3.2.3. For example, if the variable x occurs in three clauses with the literal x and in five clauses with the literal $\neg x$, the variable would be assigned with $FALSE$. After every assignment with correct prediction, the current distance value is decremented by one. If the value gets to zero, this indicates that the prediction for the current variable is false. In this case, the variable is assigned with the inverted value from the prediction model. So, in the previous example, x would be assigned with $TRUE$. Then the next distance is loaded and the process is repeated. The procedure is repeated until all the distances from the prediction model are used. If there are still unassigned variables, the predictions for them are correct and they are assigned as previously described.

The automatic inversions of the prediction model also need to be considered. As described above in Section 3.2.3, the prediction model is flipped if there are too many false predictions in a row. There are no marks in the compressed model that indicate when the prediction model is flipped, but it is possible to get that information implicitly from the prediction model values. If two consecutive predictions are false, the distance between them is zero, so a zero is written to the compressed model. So, the number of consecutive false predictions can be determined by looking at the number of consecutive zeros that appear in the values of the compressed model. To determine if the prediction model must be flipped, a counter can be used that always holds the current number of consecutively read zeros. So, if this counter reaches a value that equals the threshold used during the compression, minus one, the prediction model needs to be flipped. The subtraction of one is necessary, because the prediction model values describe the distances between the false predictions. For example, if there are ten false predictions in a row, there would be nine consecutive zeros in the values of the compressed model. When that threshold is reached, all further assignments are inverted compared to how it was described above. After the inversion the counter is reset and if it reaches the threshold again, the prediction model is flipped again.

4 Implementation Details

In this chapter, we go into detail on the implementation of the algorithms, especially the implementation of unit propagation and the Golomb-Rice codes.

4.1 General Details

This section goes more into detail on the implementation of the algorithms. The application is written in C++ and the full source code can be found here: `https://github.com/Captainval99/propositional-model-compression`. There are two separate programs, but they use the same source code for most of the key concepts, like the ordering heuristics, unit propagation and the compression algorithms.
There are also some parts of the program that were taken from other projects. For instance, the parser used to convert the user input into the right format is taken from the "Global Benchmark Database, C++ Extension Module (GBDC)" [13]. The ordering heuristics use a priority queue for efficient access and ordering of the variables. Because the C++ standard library does not contain a priority queue implementation that allows the updating of priorities, an implementation was taken from the "Minisat" SAT solver [23]. In addition to that, as already described in Section 2.4, the LZ4 and ZIP compression algorithms are not implemented as part of the thesis and instead the lz4 [19] and zlib [26] libraries are used to compress the data.
In the following, the implementation of unit propagation and the Golomb-Rice coding are explained in more detail, because they are both key components of the program whose implementations are crucial for a good performance.

4.2 Unit Propagation

Unit propagation is one of the key concepts of the algorithms and a lot of computation time is spent executing unit propagation. So, an efficient implementation of unit propagation is crucial to ensure a good performance of the overall program. To accomplish this, three main values and data structures are used during unit propagation. The first is the trail which is a vector that holds all variables that are propagated. It acts as propagation queue that is used to determine which variable is propagated next. The next important value is the head which is an integer value that holds the number of variables that have already been propagated. This is needed to determine when the unit propagation is finished. The third important data structure is the values array that holds the assignment of every variable. Before the propagation is started, the variable, that should be propagated, must be inserted into the trail and its assignment must be set in the values array. The propagation then runs as long as the head value is smaller than the number of elements in the trail. The unit propagation is implemented using full occurrence lists. That means that for every variable two lists are stored that hold pointers to the clauses in which the variable occurs as positive or as negative literal. This allows for an efficient propagation, because, in order to apply the rules described in Section 2.2, it is enough to iterate over the occurrence lists. In addition to that, a counter is stored for every clause that holds the number of unassigned variables in the clause. This is used to efficiently determine if a clause is a unit clause, because if the counter reaches a value of one and the clause is still not satisfied, the clause has only one unassigned variable left which then can be assigned and inserted into the trail. So, these methods allow for a fast and efficient unit propagation.

4.3 Golomb-Rice codes

Golomb-Rice coding is the only generic compression method used in this thesis, that was implemented from scratch.

The compression is implemented as a function that takes a vector of integer values as input. It returns a vector of chars which holds the appended byte representations of the compressed integer values. The integers are compressed one by one. First, the quotient and remainder have to determined, as described in Section 2.4.2. Because the divisor must be a power of two, the division and modulo operations can be implemented efficiently by using bit shifting and masking. After that, the quotient is unary encoded with ones. The algorithm works with one byte at a time, which is represented by a char, so a value called ”offset” is used that holds the number of bits in the current byte that are already filled with data. This value is then used to determine if the unary encoding fits into the current byte or if additional bytes are needed. If additional bytes are needed, the current byte is filled and then inserted into the output vector. The rest of the data is then written into a new char. Then, a zero is inserted as separation bit. This is implemented by just adding one to the offset value. After that, the remainder is binary encoded and written in the same way as the quotient. This procedure is then repeated for all integer values and the compressed bits are appended to another by using the offset value. After all integers are compressed, the remaining unused bits in the last byte have to be flipped to ones in order to ensure a correct decompression.

The decompression is also implemented as a function that takes a string, that contains the binary representation of the compressed integers, as input. It also outputs a string that contains the decompressed integers separated by whitespaces. This is done instead of outputting the integer values directly, to make the output consistent with the other compression methods, which makes switching between them easier. The decompression also reads the input one byte at a time and also has a ”offset” value. The algorithm works as described in Section 2.4.2. First the unary encoded value is read until the separation bit and then a fixed number of bits is read for the remainder. In the same way as during the compression it could be that the binary representation of a value spans over multiple bytes which had to be considered during the implementation. The final multiplication can again be implemented efficiently with a bit shift. This procedure is repeated until the last byte was read. After that the values are converted and then returned.

5 Experimental Evaluation

In this chapter, we test different configurations of the algorithms and compare them by the compression ratios they achieve and their execution times.

5.1 Experimental Setup

The algorithms were tested on a machine equipped with an AMD Epyc 7551P 32 core CPU and 256 GB of RAM. The operating system running on this machine is Ubuntu 20.04.
The data set on which the evaluation was performed consists of 5376 models for 3030 formulas. These models were taken from a bigger set of 38776 models, but, in order to reduce the computation time of the benchmarks, the number of models were capped to three per set of clauses. For the sets of clauses, for which more than three models exist, the three models were chosen randomly. All tests in this evaluation are performed on this test set.
There are three parameters that must be set before the benchmarks can be performed.
The first parameter is the n parameter that is used in the calculation of the heuristic values for the MOMS heuristic, as explained in Section 2.3.1. Due to previous tests, the value is set to 10 for this evaluation.
The second parameter is the divisor r used in the Golomb-Rice coding, as already described in Section 2.4.2. This parameter is also chosen based on previous test results and because the value must always be a power of two, it is set to 2 for this evaluation.
The third parameter is the value that determines when the prediction model is inverted, as described in Section 3.2.3. To determine the value of the parameter, an 80:20 split is made on the test set. The 80% are used to test different parameter values and decide which value performs the best. The performance of the compression is measured by the geometric means of the compression ratio and the hit rate of the prediction model. The compression ratio compares the size of the compressed model to the size that the complete model would have if it would be stored as a bitvector, and the prediction model hit rate indicated how many predictions were correct. The other 20% of the test set are used as a validation set to confirm the decision of the parameter value. The results of the tests are shown in Table 5.1. As seen in the table, a parameter value of 5 achieves the best compression ratios and the highest prediction model hit rates. This is also confirmed by the validation set, as this value also achieves the best results there. So, this is the value that is used in all further evaluation benchmarks.

Prediction model flip value	80% training set		20% validation set	
	Compression ratio	Prediction model hit rate	Compression ratio	Prediction model hit rate
No inversions	5.867	0.606	6.426	0.615
5	6.397	0.701	6.914	0.707
10	6.313	0.684	6.771	0.689
30	6.199	0.648	5.987	0.640
50	6.174	0.638	5.903	0.623
100	6.159	0.629	5.880	0.614
200	6.110	0.621	5.853	0.611

Table 5.1: Test results for different prediction model parameter values.

5.2 Different Ordering Heuristics

The different ordering heuristics, as described in Section 2.3, have a big in impact on the compression ratio and the performance of the compression algorithm. The heuristics were tested on the full test set and the bitvector compression ratio was measured for every model. For the compression algorithm, the Golomb-Rice coding was used during these tests. The results of the tests are shown below in Table 5.2. From this data it can be seen, that using an ordering heuristic increases the compression ratio compared to using no heuristic. It can be seen that using the Jeroslow-Wang and MOMS heuristics as dynamic heuristics leads to an significant increase of the compression ratio.

Branching heuristic	Geometric mean of compression ratios	Median of compression ratios
No heuristic	5.440	3.895
Jeroslow-Wang static	6.507	3.943
Jeroslow-Wang dynamic	18.722	15.085
MOMS static	6.084	3.960
MOMS dynamic	17.827	14.320

Table 5.2: Comparison of the compression ratios with different branching heuristics.

This can also be observed below in Figure 5.1, where the static and dynamic Jeroslow-Wang heuristics are compared with the compression ratios of every model. In this figure it can be seen that for the majority of the models a higher compression ratio is achieved by using the dynamic heuristic. But there are also models for which both heuristics achieve similar compression ratios.

Because the geometric means of the compression ratios for the dynamic Jeroslow-Wang and the dynamic MOMS heuristic are close to each other, a more detailed comparison of the compression ratios is shown in Figure 5.2. In this comparison, it can be seen that for most models the results are close to each other, but there are some models which achieve a higher compression ratio with the Jeroslow-Wang heuristic instead of the MOMS heuristic.

So overall, the dynamic heuristics achieve significantly higher compression ratios than static heuristics with the dynamic Jeroslow-Wang heuristic performing the best. But the dynamic heuristics need more computational work to be calculated which impacts the performance of the compression algorithm, which is shown in more detail in Section 5.5.

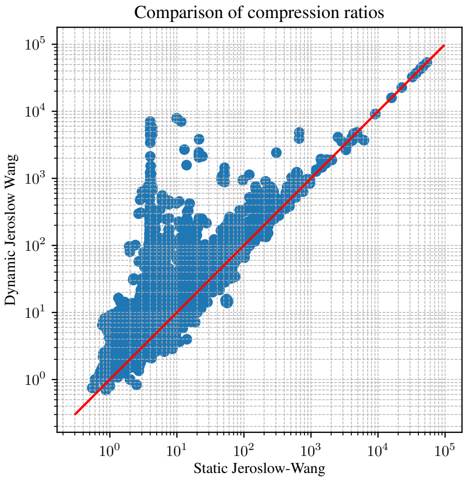

Figure 5.1: Static Jeroslow-Wang vs. Dynamic Jeroslow-Wang

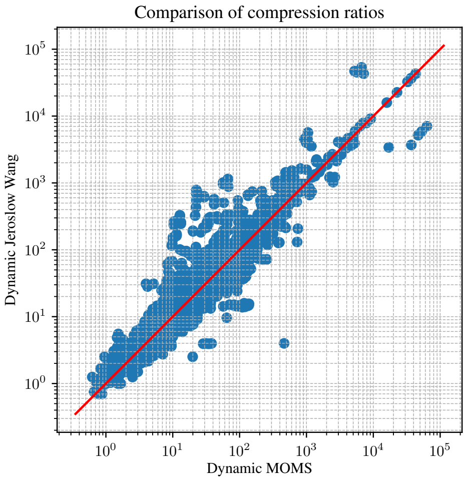

Figure 5.2: Dynamic MOMS vs. Dynamic Jeroslow-Wang

Figure 5.3: Comparisons of the compression ratios for different ordering heuristics.

5.3 Prediction model

The prediction model was added to the algorithm to increase the compression ratio. In order to show that this is true, we compared the sizes of the compressed models with and without the prediction model. For this benchmark, the dynamic Jeroslow-Wang heuristic was used in combination with the Golomb-Rice code. The prediction model flip value is set to 5, as described in Section 5.1. With the prediction model and the Golomb-Rice encoding, the average size of the compressed models is 665 bytes. Without the prediction model, the reduced set of variables, which is the core of the compressed model, contains on average 9488 variables. If the assignments of these variables would be stored in a bitvector, they would take up 1186 bytes on average, which is significantly bigger than with the prediction model. So, the prediction model decreases the size of the compressed models and therefor increases the compression ratios.

Another important value, that shows the effectiveness of the prediction model, is the prediction model hit rate. This value describes the percentage of predictions that were correct.

In order to have a positive effect on the compression ratio, the value should be above 50%. In Table 5.3 the geometric means of the prediction model hit rates are shown for the different ordering heuristics. It can be seen that the value is above 50% for all heuristics. The dynamic MOMS heuristic performs the best with a value of 0.743 but there is no significant difference between the other heuristics.

Branching heuristic	Geometric mean of prediction model hit rates	Median of prediction model hit rates
No heuristic	0.706	0.716
Jeroslow-Wang static	0.702	0.716
Jeroslow-Wang dynamic	0.715	0.727
MOMS static	0.727	0.738
MOMS dynamic	0.743	0.762

Table 5.3: Comparison of the prediction model hit rates with different branching heuristics.

In Figure 5.4, the distribution of the prediction model hit rates is shown. This graph also bases on a benchmark, that uses a dynamic Jeroslow-Wang heuristic. In the graph, it can be seen that for the majority of the models the prediction model hit rate lies over 50%.There are also some models for which the hit rate only is at around 20%, but these are an exception.

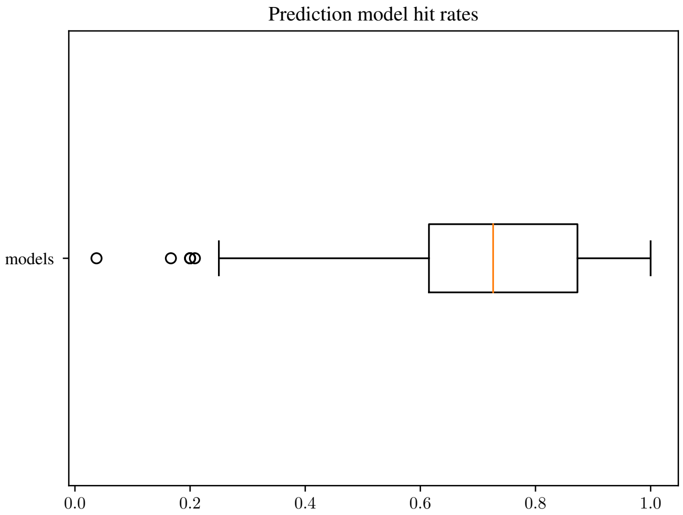

Figure 5.4: Boxplot of the prediction model hit rates.

5.4 Different Compression Algorithms

In this thesis, we propose three different compression algorithms that are used to further decrease the size of the compressed model. So, the choice of the compression algorithm also has an impact on the final compression ratio. We tested the different compression algorithms with a dynamic Jeroslow-Wang heuristic, as it achieved the best results in Section 5.2. The results of the benchmarks are shown in Table 5.4. From this data, it can be seen that the Golomb-Rice coding achieves significantly higher compression ratios than ZIP compression or the LZ4 algorithm. This could be due to the fact that the Golomb-Rice only encodes integer values, so that the values from the prediction model can be compressed directly and do not have to be converted into a string, as it is necessary with the other compression algorithms.

Compression algorithm	Geometric mean of compression ratios	Median of compression ratios
Golomb-Rice	18.722	15.085
ZIP	10.879	7.422
LZ4	7.120	5.391

Table 5.4: Comparison of the compression ratios with different compression algorithms.

In the Figures 5.5 and 5.6, direct comparisons between the Golomb-Rice coding and the other two compression algorithms are shown. In both figures, it can be seen that for the majority of the models the Golomb-Rice coding achieves a higher compression ratio than the ZIP or LZ4 compression. But in both cases, there are also exceptions, where the other compression algorithm performs better than Golomb-Rice.

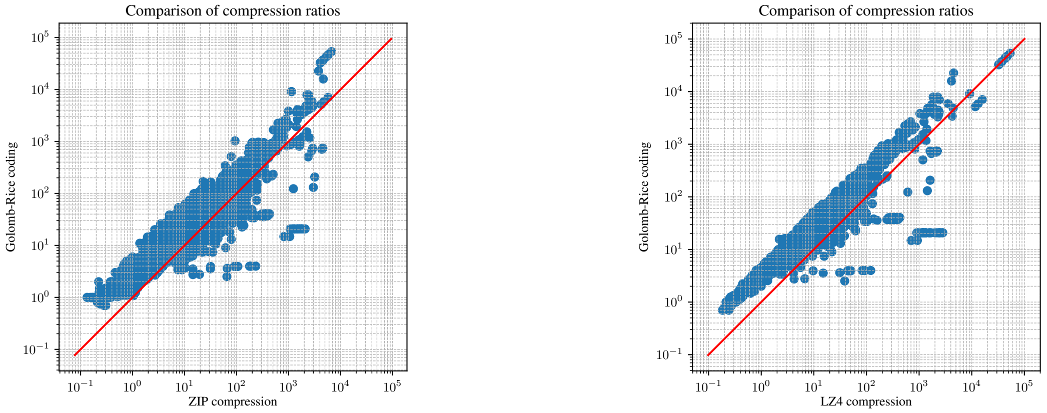

Figure 5.5: Golomb-Rice vs. ZIP

Figure 5.6: Golomb-Rice vs. LZ4

Figure 5.7: Comparisons of the compression ratios for different compression algorithms.

5.5 Compression and Decompression Times

Besides the compression ratios, the performance of the algorithm is also a very important factor. The time it takes to compress or decompress a model depends on multiple things. The ordering heuristic that is chosen has a big impact on the compression time. In Table 5.5 the average compression time for each heuristic is shown. The first thing that can be seen from this data is that using a static heuristic improves the compression time compared to using no heuristic. This is most likely the case, because when a heuristic is used, less variables are chosen and propagated which reduces the time that is spent executing unit propagation. The second thing that can be seen in the table is that the dynamic heuristics need more time than the static heuristics. The dynamic Jeroslow-Wang heuristic takes more than double the time than the static Jeroslow-Wang heuristic. This is not surprising, as the dynamic heuristics require significantly more computation to update the heuristic values. But in return, the dynamic heuristics achieve higher compression ratios, as seen in Section 5.2.

Branching heuristic	Average compression time in seconds
No heuristic	98.577
Jeroslow-Wang static	45.081
Jeroslow-Wang dynamic	118.701
MOMS static	50.702
MOMS dynamic	78.376

Table 5.5: Comparison of the compression time with different branching heuristics.

In the Table 5.6, the average compression times are compared with the different compression algorithms. The tests were also done with a dynamic Jeroslow-Wang heuristic. In the results, it can be seen that the Golomb-Rice coding needs the least amount of time out of the three algorithms. This most likely comes from the fact that the Golomb-Rice coding is easy to calculate and can be implemented efficiently by using bit shifting and masking.

Compression algorithm	Average compression time in seconds
Golomb-Rice	118.701
ZIP	152.131
LZ4	175.409

Table 5.6: Comparison of the compression times with different compression algorithms.

In addition to the compression times, the decompression times are also a relevant factor. We tested this with a dynamic and a static Jeroslow-Wang heuristic and the Golomb-Rice coding. The average decompression time for the dynamic heuristic is 157.434 seconds.

For the static heuristic, the average decompression time was 46.939 seconds. In Figure 5.8, the compression and decompression times are compared for every model with the dynamic heuristic and in Figure 5.9, the same is done for the static Jeroslow-Wang heuristic. In the figures can be seen that all the values are around the center line, which means that the compression and decompression times are similar.

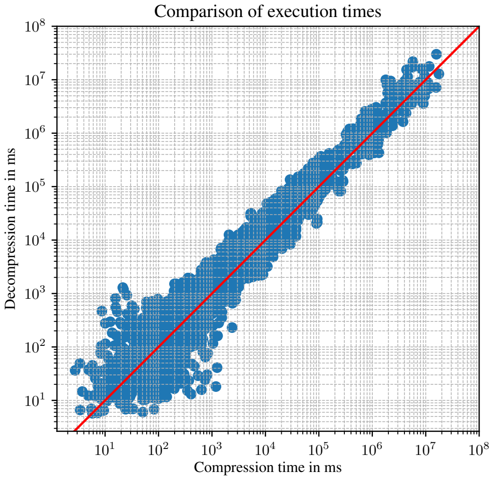

Figure 5.8: Dynamic Jeroslow-Wang

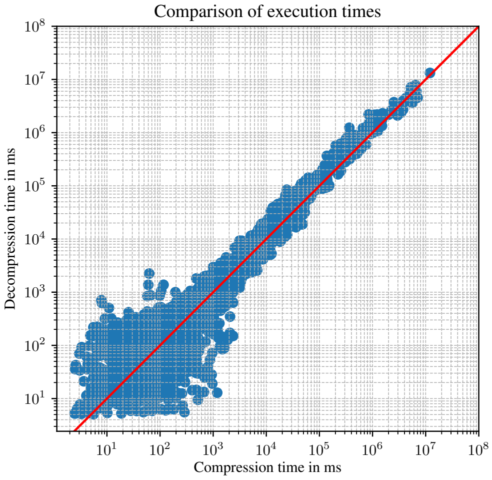

Figure 5.9: Static Jeroslow-Wang

Figure 5.10: Comparisons of the execution times for different ordering heuristics.

In Figure 5.11 a CDF-plot of the compression times. The plot was also created by using a dynamic Jeroslow-Wang heuristic and the Golomb-Rice compression algorithm. It was created by sorting all compression times in an ascending order and plotting the compression times on the x-axis and the position in the sorted list on the y-axis.

With the results from this section, Section 5.2 and Section 5.4, the configuration that achieves the highest compression ratio is the combination of a dynamic Jeroslow-Wang heuristic and the Golomb-Rice coding, with an geomertric mean of 18.722. But this combination also has a high compression time, with an average of 118.701 seconds. Another configuration that is faster and has only slightly worse compression ratios is the combination of a dynamic MOMS heuristic and the Golomb-Rice coding. This combination has a geometric mean of compression ratios of 17.827 and has an average compression time of 78.376 seconds, which makes it a good compromise between the compression time and the compression ratio.

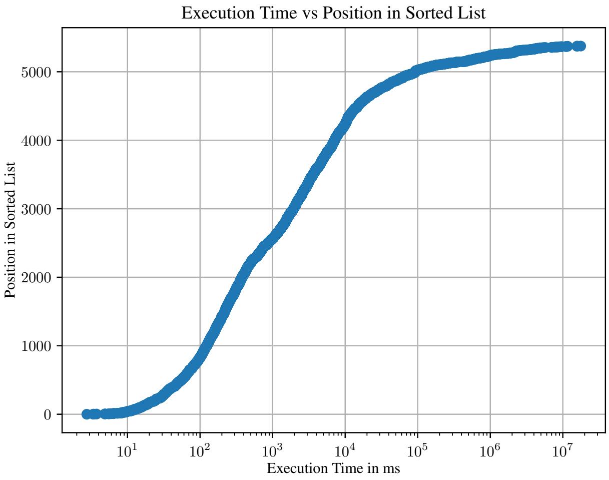

Figure 5.11: CDF-plot of the compression times.

6 Conclusion

In this chapter, we summarize the contents of this thesis and give an overview over the work that could be done in the future.

6.1 Discussion

In this thesis, we first explained our motivation and described, why it is useful to compress propositional models. After that, we introduced the concepts of unit propagation, ordering heuristics and described the additional compression algorithms that are used in the algorithms. Then the two main algorithms were described in detail. First, we touched on the compression algorithm, explained the different steps of the pipeline and how they work together to compress the models. After that, we explained the decompression algorithm and described, how the pipeline steps differ from the compression algorithm and how the original models are restored from the compressed models. In the end, we presented the results of our evaluation benchmarks and showed different configurations of the compression algorithm. We tested the algorithm with these different configurations and determined, which configuration configuration achieves the highest compression ratio.

6.2 Future Work

There are multiple things that can be improved or added to the algorithms in the future. For instance, different compression algorithms could be included, in order to further improve the compression ratios. Especially, arithmetic coding [18] could be interesting, as it allows for an efficient way to compress the prediction model values. Furthermore, the explicit use of don't care values could be added, because in the current state the is no explicit handling of don't care values and variables with that value are not included in the decompressed models. Also, the inversion of the prediction model could be improved. Instead of just having a static parameter, that determines when the prediction model is inverted, it could be useful to take a more dynamic approach. Such an approach could consider different factors, like the number of clauses and variables and determine the parameter value dynamically based on these factors. The same could also be done with the ordering heuristics. The heuristic could also be chosen dynamically by considering different factors, with the goal of choosing the ordering heuristic that would achieve the highest compression ratio for the specific model. But a consequence of this approach would be that the heuristic that was chosen would have to be encoded into the compressed model, in order to correctly decompress the model, which would increase the size of the compressed model.

Bibliography

[1] ISO/IEC 10918-1:1994. Information technology digital compression and coding of continuous-tone still images: Requirements and guidelines. Standard, International Organization for Standardization, 1994.

[2] ISO/IEC 11172-3:1993. Information technology coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s. Technical report, International Organization for Standardization, 1993.

[3] Albert Atserias and Moritz Müller. Automating Resolution is NP-Hard. *J. ACM*, 67(5):31:1–31:17, September 2020.

[4] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-Time Reductions of Resolution Proofs. In Hana Chockler and Alan J. Hu, editors, *Hardware and Software: Verification and Testing*, pages 114–128. Springer, 2009.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, May 1971. Association for Computing Machinery.

[6] Scott Cotton. Two Techniques for Minimizing Resolution Proofs. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing SAT 2010*, pages 306–312. Springer, 2010.

[7] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394397, jul 1962.

[8] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[9] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996.

[10] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of sat. *Commun. ACM*, 66(6):6472, may 2023.

[11] Jon William Freeman. Improvements to propositional satisfiability search algorithms, 1995.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., October 1990.

[13] gbdc. Gbdc: Global benchmark database, c++ extension module, `https://github.com/Udopia/gbdc`.

[14] S. Golomb. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[15] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[16] Markus Iser and Christoph Jabs. Global Benchmark Database. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:10, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[17] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1(1):167–187, September 1990.

[18] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984.

[19] lz4. Lz4/lz4: Extremely fast compression algorithm, `https://github.com/lz4/lz4`.

[20] Jens Manig. Kompressionstechniken für beschreibungenvon sat formeln, 2018.

[21] João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, DAC ’00, pages 675–680. Association for Computing Machinery, June 2000.

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

[23] minisat. Minisat: A minimalistic and high-performance sat solver, `https://github.com/niklasso/minisat`.

[24] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, April 2005.

[25] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[26] zlib. Zlib data compression library, `https://github.com/madler/zlib`.