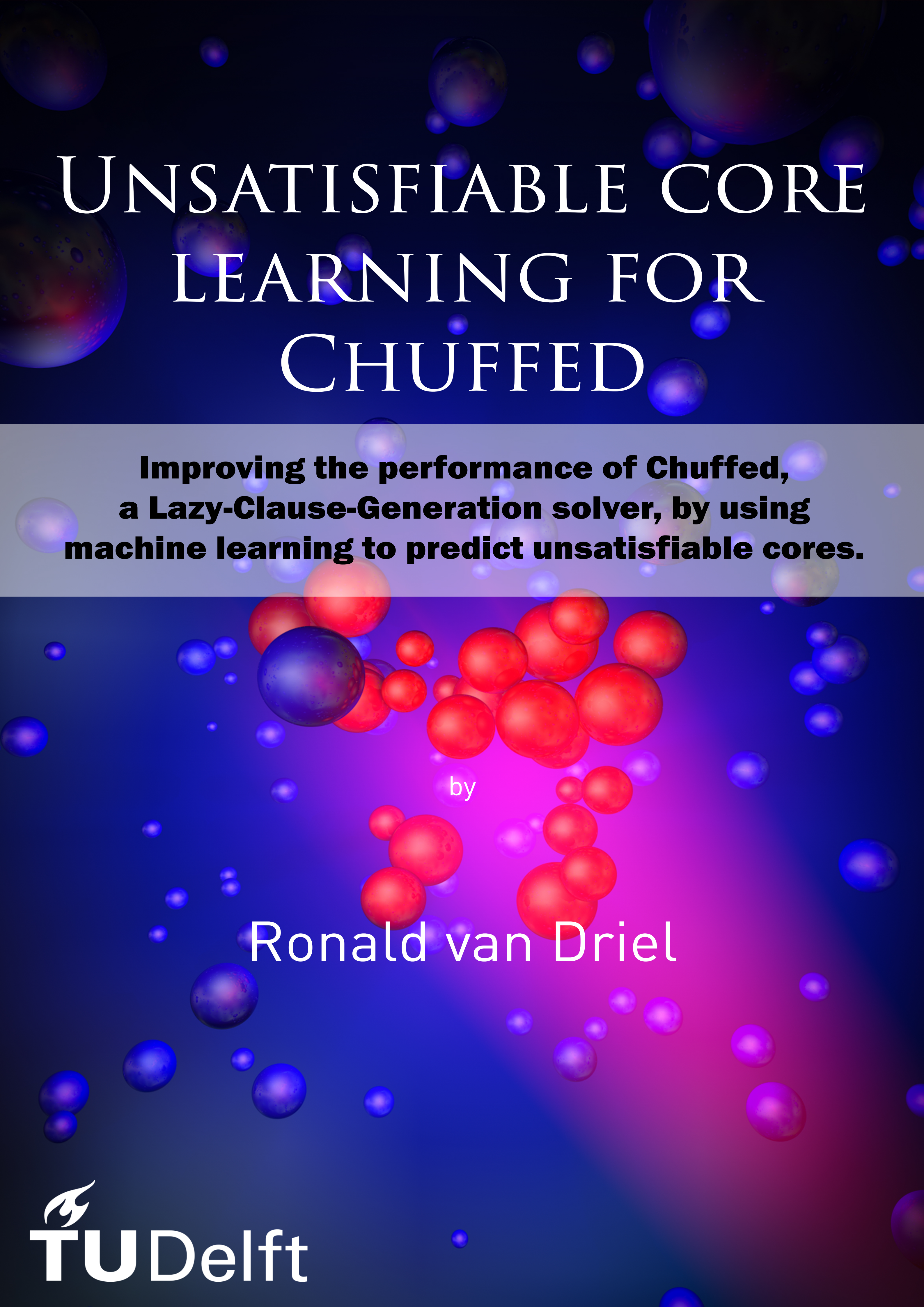


Unsatisfiable core learning for Chuffed

Improving the performance of Chuffed, a Lazy-Clause-Generation solver, by using machine learning to predict unsatisfiable cores.

by

Ronald van Driel

TU Delft

Unsatisfiable core learning for Chuffed

Improving the performance of Chuffed, a Lazy-Clause-Generation solver, by using machine learning to predict unsatisfiable cores

by

Ronald van Driel

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 24, 2020 at 9:30 AM.

Student number:	4289870	
Project duration:	September 1, 2019 – July 24, 2020	
Thesis committee:	Dr. N. Yorke-Smith,	TU Delft, Supervisor
	Dr. ir. S. Verwer,	TU Delft
	Dr. S. Roos	TU Delft

Preface

You are currently reading my Master thesis, this document marks the conclusion of my M.Sc. in Computer Science at the Delft University of Technology. A lot of time and effort went in the creation of this document, but throughout the process I have learned many valuable lessons both on personal and scientific level.

This thesis would not have been possible without the excellent guidance and support of my supervisor Dr. Neil Yorke-Smith, who I would like to thank extensively for all the meetings and guidance. Additionally, I thank Peter J. Stuckey and Kevin Leo for providing me with some insightful answers to my questions. Finally, I also would like to thank my friends and family. My parents for their unconditional support. My friends for providing welcome distractions during uninspired moments.

Ronald van Driel
Delft, July 2020

Contents

Abstract

Solving propositional satisfiability (SAT) and constraint programming (CP) instances has been a fundamental part of a wide range of modern applications. For this reason a lot of research went into improving the efficiency of modern SAT and CP solvers. Recently much of this research has gone into exploring the possibilities of integrating machine learning approaches with these solvers. However, with hybrid solvers, which combine both SAT and CP, dominating recent benchmarks it is surprising that no research has been done yet to apply those machine learning approach to improve hybrid solvers. This research proposes using a machine learning technique called unsatisfiable core learning to improve the performance of the Lazy Clause Generation solver Chuffed. The approach developed for this study uses a Graph Convolutional Network model, which is trained on a dataset containing unsatisfiable instances. This machine learning model is then used for predicting unsatisfiable cores on CP instances and the predictions are used to initialise the activity score of the Variable State Independent Sum heuristic which is incorporated in Chuffed. The resulting approach managed to consistently solve a set of Multi-mode Resource-Constrained Project Scheduling instances 2.5% faster on average. These results indicate that, while this technique was originally developed for SAT, it can also be used to improve hybrid SAT/CP solvers.

Keywords

Constraint programming (CP) • Propositional Satisfiability (SAT) • Chuffed • Graph Convolutional Network (GCN) • Variable State Independent Sum (VSIDS) • Machine Learning • Lazy Clause Generation (LCG)

1 Introduction

Both propositional satisfiability (SAT) and constraint programming (CP) are automated reasoning technologies prevalent in modern computations. They form the foundation of a wide range of real-life problems such as: scheduling, logistics and resource allocation [36]. Solving SAT or CSP instances as quickly as possible is therefore of great economical interest to many companies. In the last decade a trend has emerged where much research is being done to developing improved SAT and CP solvers by integrating them with recent machine learning techniques. However, with hybrid SAT/CP approaches dominating the recent MiniZinc challenges [30, 31], the questions arises if any of these developments can be also be useful to improve these hybrid approaches.

1.1. Motivation

The immense popularity of propositional satisfiability (SAT) and constraint-satisfaction problems (CSPs) is reflected in the vast amount of research that has been conducted over the years to make SAT and CSP solving as efficient as possible [4]. In general, SAT solvers are known for being surprisingly efficient, whereas constraint programming (CP) sacrifices some of this efficiency for more expressive power.

It is worth noting that across time the fields of SAT and CP have grown increasingly similar, this similarity has caused a raise in interest to transferring knowledge from one domain to the other. Consequently, more researchers have started exploring the possibility of directly combining SAT with CP. The intention behind these combinations is to take advantage of the efficiency of SAT solving while retaining the expressiveness of CSP [34].

Stuckey identified SAT's effective use of conflict learning as a major reason for SAT's efficiency and developed the first successful hybrid SAT/CP approach [42]. This hybrid is based on a technique called Lazy Clause Generation (LCG) [11]. LCG introduces a way to integrate SAT's conflict learning with finite domain propagation, which is a typical solving technique for CP. Recent benchmarks [30, 31] show that such hybrid SAT/CP approaches are very competitive with the state-of-the-art.

Concurrently, recent machine learning developments have inspired increasingly many researchers to explore the potency of enhancing SAT and CP solving using machine learning approaches. While research has been conducted to explore a wide range of machine learning enhanced approaches [9, 46], researchers are yet to experiment with using machine learning to improve hybrid SAT/CP methods. It is, however, interesting to try this, because these methods have shown to be very capable at beating he state-of-the-art solvers on recent benchmarks [30, 31]. Any benefit that can be gained from machine learning might push the boundaries of what is currently possible in the area of CP solving.

1.2. Background

This thesis expands on the existing research on SAT and CP. It is essential to have a basic understanding on the background of SAT and CP in order to follow the thesis statement and the reasoning process behind it. The purpose of this section is thus to briefly introduce readers who are unfamiliar with the underlying scientific background to allow them to better understand the remainder of this document. To this purpose, this section briefly introduces the definition and applications of SAT and CSP.

Additionally, the hybrid approachis which combine SAT and CSP solving techniques are of particular interest to this research and they will also briefly be touched upon. For a more complete understanding of the scientific foundations it is recommended to also read the Background & Related work chapter 2 and look into the references provided throughout this document.

1.2.1. SAT

Propositional satisfiability (SAT) is a widely studied NP-Complete problem. SAT solvers are generally known for being remarkably efficient, but they are only useful for specialised data structures.

The input for a SAT problem is called a propositional formula which comprises a set of Boolean variables and any combination of the basic logic operations AND, OR and NOT. Given such formula, the SAT problem can be viewed as finding an assignment for all variables such that the formula evaluates to true or, otherwise, determine that no such assignment exists. A simple example would be the propositional formula shown below.

$$F := (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$$

A solution to this example would be the following assignment: $x_1 = True, x_2 = False$. This means that this proposition is satisfiable. The formula comprises a total of two variables x_1 and x_2, and two clauses $(x_1 \lor x_2)$ and $(\neg x_1 \lor \neg x_2)$. When a variable appears in a clause, either accompanied with or without a negation ($\neg$), it is called a literal, which means this formula has four literals: x_1, x_2, $\neg x_1$ and $\neg x_2$.

It is worth noting that this formula is written in conjunctive normal form (CNF) which is the most common way of formulating propositional formulas within the SAT domain. A formula is considered to be in conjunctive normal form if 1) clauses only contain literals and OR operators and 2) the clauses are separated by an AND operator.

The simplicity and restrictive syntax of SAT solvers allows SAT to be used as a low-level foundation of a wide range of modern applications such as: hardware/software verification, theorem proving, cryptography, circuit design and artificial intelligence.

1.2.2. CP

Comparable to SAT, but more general, would be constraint programming (CP). Similar to SAT, CP has been a very active field of research in the past decades. While CP solvers are more expressive than SAT solver they sacrifice some of the efficiency associated with SAT.

Two variants of CSP are distinguished, being the decision problem and the optimisation problem. The decision problem for CSP amounts to finding an assignment for each variable for which all constraints are satisfied or otherwise determine that no such assignment exists. The optimisation problem, on the other hand, also tries to find the best solution with respect to some objective value. Commonly the optimisation problem is referred to as constraint optimisation problem (COP) or as a minimisation- or a maximisation problem depending on if it is desirable for the objective value to become smaller or greater. In this document CSP refers to both the decision and the optimisation variant unless otherwise specified.

A CSP is composed of the following:

1. A set of variables V = {x_0, x_1,...}
2. A universe, comprising a domain for each variable, U = {D_0, D_1,...}
3. A set of constraints C = {C_0, C_1,...}

In addition to the above, a COP also considers an objective function for which the outcome, the objective value, must be maximised or minimised.

Variables each have their own domain, being a predefined range or set of values. The constraints impose all sorts of restrictions on the values to which the variables can be assigned to. The actual types of values and the availability of constraints depend on the specific tool or library which is used. Even though there is no general consensus on the implementation of values and constraints, most CP solvers support values being Boolean, integer and float and at least implement the lesser-than, greater-than, equality and inequality constraints. Additionally, it is common for CSP solvers to expand their constraint language with as many constraints as desired. A popular constraint to include, for example, would be a constraint which requires a set of variables to be all different.

The general and versatile nature of CP solvers allows them to be widely useful to a broad range of real-world applications including: scheduling, resource allocation and logistics [36].

1.2.3. Hybrid SAT/CP

While hybridisation of CP and mathematical programming has been proposed in the early 2000s [3, 15], hybrid approaches which integrate SAT and CP have only been invented in the past decade. Combining SAT and CP has proven to be a very efficient approach to solve CP instances.

The possibility of expressing SAT problems directly as a CSP makes SAT a subset of CP, implying a strong relation between the two solving paradigms [44]. Ever since researchers realised the strong connection betwen SAT and CP, they have taken inspiration from successful approaches in one domain and tried to apply it to the other. However, it is only since the last decade that researchers have started to combine SAT and CP into a hybrid approach. The first successful SAT/CP hybrid was proposed by Stuckey [42]; this hybrid approach is based on a technique named Lazy Clause Generation (LCG). LCG is a technique which uses finite domain propagation solving from CP and combines it with SAT's conflict learning ability. Integrating SAT's conflict learning is realised by creating an inference graph which is used to prevent searching similar parts of the problem.

Both cutting-edge solvers Chuffed [6] and Google OR-Tools [1] have deployed LCG in their solving paradigm. These solvers show that LCG is very competitive compared against state-of-the-art CSP solvers. This claim is backed up by OR-Tools' performance at the MiniZinc Challenge 2017 [30] where their LCG based version of OR-Tools managed to score top three in four out of five categories. Afterwards OR-Tools has evolved towards an approach involving linear programming (LP) as well as SAT and CP. This new version of OR-Tools even managed to dominate all other competition by achieving gold on each of the five categories at the MiniZinc Challenge 2019 [31].

1.3. Research goal

Hybrid SAT/CP methods rely on SAT for representing conflicts which are encountered during the search procedure. This means these methods inherit some of SAT's characteristics and may thus benefit from SAT related research. Recently, research on combining SAT solving with machine learning has risen in popularity, therefore it makes sense to use this knowledge to apply it to improve a hybrid SAT/CP solver. For this research a modified version of Lazy Clause Generation (LCG) solver Chuffed [6] is developed and compared against the original. If this machine learning enhanced version obtains better performance, this may suggest that similar machine learning approaches could also be carried over to improve other hybrid SAT/CP solvers such as the current state-of-the-art Google OR-Tools [1].

Therefore, the primary goal of this research is to explore promising machine learning approaches used within the SAT domain and determine if their potency can be applied to improve Chuffed's performance. It is typical for CSP solvers to have a varying performance depending on different problem types, this behaviour may extend to machine learning assisted solvers where for some problem types machine learning is more helpful than others. Another important factor that could influence the impact of the machine learning integration is the available training data, sometimes they do not require training instances of the same problem types to be successful, but this need not be the case. For these reasons the secondary goals are to 1) determine if there are significant differences in performance on different problem types and 2) determine if machine learning generalises well between different training instances.

1.4. Research questions

The aforementioned research goals can be expressed as the following research questions:

1. **Is it possible to reduce the run-time of solving a CP instance by integrating machine learning in the LCG solver Chuffed?**

2. **Does the impact of using machine learning on the run-time of Chuffed differ across different problem types?**

3. **Does the impact of using machine learning depend on the type of problems in the training set?**

Answering these research questions will require obtaining data for training and testing, and developing a modified version of Chuffed which uses output from a machine learning model.

1.5. Structure of this document

This thesis follows the following structure: first a more detailed background is given alongside related work in Chapter 2. Thereafter, Chapter 3 discusses some initial experiments which were conducted prior to the actual research described in this document. Chapter 4 describes and explains which data was used and how the approach used to answer the research questions was realised. The procedure used to conduct experiments to evaluate the performance of the final approach is described in Chapter 5 which also presents the corresponding results. Statistical analysis and interpretation of these results follow in Chapter 6. Finally, a conclusion is drawn in Chapter 7 which finishes with some recommendations for future work.

2 Background & Related work

This research investigates the application of machine learning approach within the domain of propositional satisfiability (SAT) to solving constraint progamming (CP) instances. It thus finds its foundation in both the domains of SAT and CP, it is essential to know what the building blocks of this research are to understand its relevance. A more condense summary on SAT and CP is given in the introduction; this chapter, discusses relevant prior scientific advances within the SAT and CP domains. First an overview of the development within the traditional setting is given for both SAT and CP, then research is covered about combining SAT and CP solving in hybrid approaches. The last section investigates machine learning and how it has affected modern research to SAT and CP.

2.1. Traditional research

Within the context of this document traditional research refers to research which does not involve any form of machine learning. Both SAT and CP have already been popular research domains for over four decades. This has resulted in traditional research to these domains to become rather saturated, meaning that over the years a so many traditional algorithms have been proposed with each their own advantages that it has become challenging to make any significant contribution to this field. Giving an overview of all existing algorithms would require an unrealistic amount of space and time, therefore this section aims to just provide some understanding of the most important approaches and insights that are central to this research. Whereas this document focuses on research relevant to this research, a more extensive comparison between traditional SAT and CP approaches has been conducted by Bordeaux et al. [4].

2.1.1. SAT

Most of the state-of-the-art work on SAT rely on previously established techniques, one such technique originates from the work of Davis, Logemann and Loveland [7]. This work forms the foundations of arguably the most influential SAT method, which referred to as the Davis, Putnam, Logemann and Loveland (DPLL) algorithm proposed back in 1962 [8]. It is one of the first algorithms to implement a backtrack and search approach. Backtrack search is a technique which involves the construction of a search tree and then use local reasoning to prune away branches that do not contain the desired solution. To this day, the majority of SAT solvers are based on the backtrack search technique introduced with the DPLL algorithm. This can be attributed to the fact that the backtrack search approach has proven to be impeccable for complete algorithms.

An algorithm is considered to be complete when it explores the entire solution space. This property allows complete methods to always give the best solution if it exists or alternatively determine the absence of a solution if the problem is unsatisfiable. Incomplete methods, on the other hand, are typically driven by heuristics and do not necessarily perform exhaustive exploration of the search space, while typically faster at generating a good solution they lack the ability to detect unsatisfiability. Incomplete methods are typically either population-based algorithms [28], for which ant colony optimization [10] is the best known example, or algorithms based on stochastic local search [20], such as simulated annealing [43]. While some local search approaches, such as the one proposed in 2007 by Hutter et al. [22], make for a strong competitor to complete methods, complete SAT solvers remain considerably more common in both research and application. The popularity of complete methods can be accredited to their exceptional performance on real-world applications. Additionally, their ability to determine satisfiability makes them a good fit for low-level decision problems encountered in many modern applications.

Because of their prevalence, recent research on SAT solvers is primarily directed at complete methods, continuing on the DPLL backtrack search algorithm. Most of this research is aimed at improving different sorts of branching techniques and heuristics. Two major advances in this area have been the introduction of clause learning and nonchronological backtracking [27], they work especially well for the structured nature as encountered in real-world applications.

Within the context of SAT solvers, clause learning refers to the analysing of conflicts as they occur and store some information in order to prevent similar conflicts from occurring. Clause learning is typically realised by adding a clause containing relevant conflicting literals. Nonchronological backtracking, on the other hand, refers to an approach where the solver is not limited to backtrack to the most recent decision level when a conflict occurs, instead it makes use of conflict analysis to try and backtrack to a decision level that actually resolves the conflict.

Another notable advance for SAT solving has been the development of a solver called Chaff which was released in 2001 [32], at the time this solver had a major impact on the field with its remarkably efficient performance. One innovative heuristic introduced with Chaff that allowed for this performance was the Variable State Independent Decaying Sum (VSIDS) heuristic. The VSIDS heuristic can essentially be viewed as having an activity score for each literal of the original formula, this score for a literal is increased by a constant amount whenever a clause containing this literal is added during search and is decreased by division periodically over time, the solver prioritises branching on literals with the highest activity score. The nature of this procedure, having additive increments and multiplicative decay, emphasises branching on literals that appear more recently in the search history. The VSIDS heuristic manages to capture the search process of the solver due to its temporal nature, yet it is very efficient because of the property of being independent on the state of the variables. Being variable state independent allows for this efficiency because, contrary to earlier heuristics, only one value needs to be stored per variable, which is cheap to maintain. Ever since the release of Chaff, researchers have explored different variations of the VSIDS heuristic [26] and have achieved similar performance. A simplified version the of backtrack & search algorithm which uses the VSIDS heuristic is given in pseudo-code 1 below.

Further research has been done to address an issue where poor initial branching can significantly amplify the solving time of some instances. A widely adopted strategy to combat this issue is randomly restarting the solver as originally proposed by Gomes et al. [16]. With random restarts, typically the current variable assignment is discarded but additional clauses from clause learning are retained, this allows the solver to explore a different part of the search tree without repeating previous mistakes.

2.1.2. CP

Constraint programming (CP) is a broad area of research directed at solving both constraint satisfaction problems (CSP) and constraint optimisation problems (COP). The main differences between SAT and CP which sets them aside are both the fact that the variable domains in CP are not restricted to be Boolean and the fact that CP supports a wider range of constraints. Because of this, CP solvers are way more expressive and general than SAT and are more suitable for higher level applications.

Being more general, research on CP has had a different focus where, instead of focusing on decision problems like SAT, research on CP is typically more directed at the optimisation variant. This has impacted the focus of research to CP solving, where the aim was not to find a general purpose solver for all possible CP instances, instead many CP solvers rely on the tuning of heuristics or parameters in order to specialise in a specific subset of CP. This also means CP supports a more varied range of algorithms tailored to specific types of CP instances. Instead of providing a single algorithm, CP solving is usually performed with the assistance of high level modeling languages or toolkits such as MiniZinc [33] and Gecode [38].

Despite the wide range of different approaches that exist for solving CPs, a typical CP solver can be roughly divided into the two components propagation and search [36]. Propagation refers to the implementation of the constraints where values violating the constraints are systematically eliminated. Search is a higher level procedure that dictates which parts of the search tree to explore. Typically,

Algorithm 1: Backtrack & Search with VSIDS

```
input  : SAT instance
output: Satisfying assignment or UNSAT
foreach literal $i$ do $VSIDS[i] \leftarrow 0$
$assignment \leftarrow \emptyset$
while True do
    if All variables assigned then
        return SAT, assignment
    end
    $l \leftarrow$ DecideNextBranch($VSIDS$)    Choose to branch on a literal with high VSIDS score
    $assignment[v] \leftarrow l$    Assign variable to the literal branched on
    while True do
        $VSIDS[i] \leftarrow VSIDS[i] * c$    With c a constant between 0 and 1
        $status \leftarrow$ DoUnitPropagation
        if $status = conflict$ then
            if assignment = $\emptyset$ then
                return UNSAT
            end
            $literals \leftarrow$ AnalyseConflict
            for $l \in literals$ do
                $VSIDS[l] \leftarrow VSIDS[l] + k$    With k any positive constant
            end
            Backtrack    Undo any assignments that resulted in unsatisfiable state
        end
    end
end
```

search is implemented together with heuristics or branching policies to strategically traverse the search tree .

The research fields of SAT and CP have grown to become similar over time. From a high level perspective the predominant solving strategies for SAT and CP have become virtually the same. While incomplete SAT approaches are typically based on stochastic local search [20], CP methods are only different in that they offer a wider range of local search approaches integrated with constraint programming [21]. Regarding complete methods, both SAT and CP have widely adopted variations of the backtrack search algorithm as the most prominent approach.

2.1.3. Integration of SAT and CSP

While originally SAT and CP were commonly treated as separate fields of study, they have grown towards each other over the past couple decades. This proximity became further apparent when researchers discovered that SAT and CSP instances can easily be translated from one to the other [44].

With researchers realising the similarity between SAT and CP more and more researchers have attempted to transfer successful methods from one domain to the other [4]. With SAT solvers being incredibly efficient, researchers have even started to propose encoding difficult CSPs in SAT. This efficiency can be accredited to SAT's conflict analysis in conjunction with the Variable State Independent Decaying Sum (VSIDS) heuristic [26]. Unsurprisingly, researchers have made an effort to try and integrate conflict learning with CP [23]. Clause learning is usually referred to as no-good learning in a broader context. Multiple efforts have been made over the years to integrate no-good learning learning into CP solving [14, 18, 35]. However, while related techniques such as intelligent backtracking approaches have been successfully adopted for CP, conflict learning has remained far more effective within the SAT domain. The explanation for this difference is the fact that the specialised clausal structure of SAT problems is much more effective at representing conflicting assignments than the general and complex structures found in CP.

A solution to this problem was proposed by Stuckey [42], his work describes a SAT/CP hybrid approach where parts of the CP solving paradigm are mapped to clauses in a SAT solver. This hybrid approach is based on a technique named Lazy Clause Generation (LCG). This technique uses finite domain propagation solving from CP and combines it with SAT's conflict learning ability. It works by generating SAT clauses representing finite domain propagators, based on these clauses an inference graph is created and used to prevent searching similar parts of the problem.

2.2. Emergence of machine learning

In the last decade increasingly much research has been conducted to explore the potential of integrating machine learning into traditional SAT and CP solvers. Recent advances in the field of machine learning have stimulated researchers to try and explore different ways of applying machine learning to improve traditional research. The promising results and versatility of machine learning has further contributed to the popularity of this field. The major benefit of adopting machine learning is that, even though training may take considerable time and space, once a model has been trained the prediction or classification can be performed in constant time. This may have practical benefit as models can be trained during idle times allowing for a more effective utilisation of computing resources. Although many use cases for machine learning are described in SAT or CP related research, roughly five ways of integrating machine learning with traditional methods can be distinguished.

Satisfiability prediction Machine learning can be used to predict whether an instance is satisfiable or not. This application has been identified and realised for both SAT [9] and CSP [46]. Determining satisfiability is considered to be NP-Complete for both SAT and CSP, meaning that it can be very costly to use traditional methods for this. Machine learning, on the other hand, is able to provide a prediction virtually instantaneously with very high accuracy. For example, the work by Devlin and O'Sullivan [9] claims that they are able to achieve over 90% accuracy for large industrial benchmarks using standard learning techniques like random forests. This means that for any application where absolute certainty is not required machine learning can be used to significantly speed up the process. Research by Selsam et al. [40] has shown that it is even possible to extend satisfiability prediction to end-to-end solving.

End-to-end solving Another popular application of machine learning is using a model to predict the output of a solver, effectively proposing a direct solution to the problem. Research has shown that it is possible for machine learning to solve both SAT [40] and CSP instances [2]. Interestingly, even if the machine learning model fails to find a proper solution to a problem, the predicted assignment can still be useful as a starting point for a traditional complete solver. For example the work by Wu [45] describes an approach where the run-time of SAT solver is reduced by using an initial SAT formula which is generated using machine learning.

Heuristics learning Perhaps the most widely studied application of machine learning is to improve existing solvers. Most solvers use heuristics to guide their search, traditionally, heuristics were hand-crafted based on the expertise gathered over years of domain knowledge. Machine learning; however, has the ability to quickly analyse and learn from innumerable amounts of data to automatically learn or improve heuristics to that work well for a specific instance. An example of such approach is proposed by Selsam and Bjørner [39] where a neural network architecture is used to initialise the values of the VSIDS heuristic for SAT. Another example would be the work of Song et al. [41] where machine learning is deployed to automatically learn variable ordering heuristics for solving CSP instances.

Portfolio selection Portfolio selection refers to choosing between algorithms or search strategies based on the instance that needs to solved. This technique is especially useful for CP as a broad range of strategies have been developed which all perform differently depending on the characteristics and structure of the problem. Using machine learning to automatically choose the best strategy can result in incredibly good performing and robust solution. Research by Guerri and Milano [17] shows that machine learning is very capable at deciding when to use an constraint programming approach or when to use an integer programming approach to solve an instance. Even without the availability of multiple algorithms this technique can also be used to learn when to use a certain search strategy [13].

2.3. Related work

This research described in this document is aimed at using machine learning to improve the hybrid SAT/CP solver Chuffed. To the best of my knowledge, no other machine learning approaches have been used to improve hybrid SAT/CP solvers. Instead, the final method deployed and analysed in this research primarily draws inspiration from the work by Selsam and Bjørner on using machine learning for SAT solving [39]. Their work describes how the existing neural network architecture called NeuroSAT, which was developed for predicting satisfiability on SAT instances [40], is modified to predict unsatisfiable cores. An unsatisfiable core is an the smallest unsolvable subset of variables, also referred to as minimum unsatisfiable core (MUC). The work of Selsam and Bjørner [39] suggests that when a variable is being classified as being part of a MUC it means that it should be prioritised during solving. Following this intuition they integrated their modified machine learning model with the existing SAT solvers MiniSat, Z3 and Glucose. They do so by using the predictions to initialise the values for the VSIDS heuristic. Using this approach they manage to solve between 6 and 20% more problems from SAT-COMP 2018 compared to the unmodified versions within the same time. Other work has shown that it is possible create a generic problem class model to which allows using conflict learning across different instances of the same problem type for Chuffed [5], which indicates that there may, in fact, be learn-able concepts across instances.

The approach described in Chapter 4 continues on the method proposed by Selsam and Bjørner [39] by developing a procedure to examine if the improvements obtained for SAT solving can be extended to improve the CP solver Chuffed.

3
Initial experiments

Before committing to the research approach described in Chapter 4 some exploratory research was done to explore different possibilities for machine learning to improve hybrid SAT/CP solving. This section briefly describes the experiments conducted as part of this preliminary research.

3.1. Satisfiability prediction

As a lazy-clause-generation solver, Chuffed relies on SAT to perform conflict analysis. Initially some research was done to investigate the possibility of using satisfiability prediction on SAT instances to leverage the conflict analysis component of Chuffed. For this approach the following two questions need to be answered:

3.1.1. Can satisfiability be accurately predicted?

While research already exists on predicting satisfiability on SAT instances [47], it is not yet a given that it is possible to accurately predict satisfiability on SAT instances that are generated during the solve procedure of Chuffed. For this classification to be any useful to Chuffed requires the machine learning model to to acquire the features in constant time and accurately make predictions. Given the complex nature of CP problems, it is not unlikely that Chuffed produces large and difficult SAT instances.

To validate the capability of machine learning to predict satisfiability in Chuffed's setting, multiple different classifiers (Random forest, SVM, MLP, DNN) are trained and cross-validated on a dataset containing 600 3SAT instances at phase transition, which are obtained through the public library SATLib [19]. The models were initially trained on 32 features, including features such as: minimum/maximum occurrences, horn rate, amount of positive/negative literals and co-occurrences of variables. These selected features were based on the results presented in earlier work on SAT classification [47, 48]. Additionally only shallow features were selected, meaning that they can be extracted with minimal overhead, typically within a single pass over the instance.

Initially, no single classifier managed to achieve an accuracy higher than 50%. These results indicated that the features did not sufficiently correlate to the satisfiability of the SAT instance. Particularly, 11 features did not have any influence on the prediction output. Therefore, these 11 features were replaced with a new feature, which represented the average ratio of positive versus negative occurrences of literals. This specific feature was chosen because it appeared to be very effective according to the results of Xu et al. [48].

Using the new 22 features, the same experiment was repeated. This resulted that the random forest classifier performs the best on the data by correctly classifying 60% of the instances. While 60% does not seem convincing, it is actually quite promising given the difficulty of the instances. For actual instances, as encountered with CP solving, a higher accuracy may be expected as they likely have more similarities in structures. Additionally more features could be added based on the linear relaxation of the SAT instance.

3.1.2. Does satisfiability prediction benefit Chuffed?

The next step is to investigate the possibility of integrating satisfiability in the solving process of Chuffed. Like any lazy-clause-generation (LCG) solver, Chuffed generates clauses during each propagation step. This clause is used to explain the propagation, it does so by identifying which variables would otherwise be in conflict and creates a clause containing the conflicting variables. The clause is then used to build and store a redundant constraint, also called a no-good, which prevents exploring similar parts of the search tree where those same variables would end up in a conflict.

Unfortunately, there seemed to be no obvious way to use satisfiability prediction on the clauses generated by the LCG solver. The main reason for this is the fact that these clauses are temporarily created and used to build a no-good, but are never stored themselves. Another reason is the fact that by definition all clauses generated during search are unsatisfiable which makes it very difficult to provide sensible guidance for the solver based on the machine learning predictions.

3.1.3. Conclusion

The results obtained during the experiments do not contradict the possibility of using shallow features to classify SAT instances generated by Chuffed. However, no obvious way could be discovered to use SAT classification for improving the performance of Chuffed.

4 Methodology

As described in Chapter 1, the objective of this research is to try and use unsatisfiable core learning to create an improved version of the Lazy Clause Generation (LCG) solver named Chuffed [6]. This chapter explains the steps that have been taken towards accomplishing this goal. The first section briefly explains the general approach used for this research. The next section describes which data was required, how it was acquired and what the characteristics are for the obtained data. The third section covers the machine learning model which is deployed for this research, why it is chosen and how it is implemented. Lastly, the final section explains how the approach was implemented in such a way that the machine learning output could be used for improving the solver Chuffed.

4.1. Approach

This section covers the intuition behind the approach used for this research and briefly summarises the complete procedure. A more detailed description on the execution of the steps mentioned in this section is provided in the corresponding subsequent sections.

4.1.1. Intuition

The intuition behind this research is based on the observation that the activity score for the Variable State Independent Decaying Sum (VSIDS) represent the frequency of a variable being part of a conflict. Normally the score for each variable is initialised at zero and incremented when the corresponding variable occurs as part of a conflict. Consequently, the scores do not initially provide any information to the solver but they gradually become more useful. However, with machine learning the activity scores can be directly initialised based on predictions. To achieve this a machine learning model is trained to predict the likelihood of variables to be part of a conflict. Having the VSIDS heuristic initialised immediately may benefit the solver in the early stages of solving an instance thus reducing the total run-time.

This approach was originally proposed by Selsam and Bjørner [39], they use a machine learning technique called unsatisfiable core learning to predict which variables belong to a so called minimal unsatisfiable core (MUC). Logically, variables which are part of such unsatisfiable core are more likely to occur in a conflict, which matches with the conflict driven nature of the VSIDS heuristic. The results they obtain by using unsatisfiable core learning to set the values of the VSIDS heuristic show a considerable improvement for the performance of SAT solvers.

Being a hybrid solver combining SAT and CP allows Chuffed to use conflict driven heuristics similar to the ones typically used for SAT solving. In particular, Chuffed also uses the same VSIDS heuristic which is used for SAT. This aim of this research is to answer the question whether unsatisfiable core learning can also be used to initialise the VSIDS scores for Chuffed to obtain an improvement similar to the results presented by Selsam and Bjørner [39].

4.1.2. High-level Procedure

This section provides a high-level overview of the steps constituting the research process. Generally speaking, this research can be roughly divided into five different steps, these steps are introduced and described below.

1. **Acquisition of the data** This research required a dataset for both unsatisfiable and satisfiable instances. Contrary to datasets for SAT, no large datasets were available which contained more than a dozen distinct unsatisfiable industrial CP instances. Therefore, the first step of this research was to develop a procedure to obtain sufficient data. Ultimately this procedure involved modifying instances from the MiniZinc benchmark suite [29] to become unsatisfiable.

2. **Identifying and developing a machine learning model**

   After gathering the data, the next step was to implement a machine learning model that would be capable of using this data to predict the likelihood of a variable belonging to an unsatisfiable core. While the model used by [39] has shown to work well for unsatisfiable core learning on SAT instances, a more complex model was chosen to account for the more general nature of CSP instances.

3. **Extract features from the data**

   The unsatisfiable CSP instances are not directly usable to the machine learning model, they first need to be converted to features that represent the instances. For this the data instances need to be parsed and characteristic metrics need to be extracted.

4. **Predict unsatisfiable variables of a satisfiable instance**

   When the machine learning model is trained on the unsatisfiable dataset it can be used to output prediction scores for each variable of an instance, even if the instance is actually satisfiable. The output of the model is expected to have a correlation with how much this variable contributes to the solving complexity, this means that it makes sense to prioritise branching to variables which were classified to be unsatisfiable with high confidence. The VSIDS heuristic allows for an easy way of implementing this priority by initialising the scores for each variable with their respective classification confidences.

5. **Evaluation**

   To determine if the selected approach actually improves the performance a modified version of Chuffed, which uses the machine learning predictions, needs to be developed. This modified version should than be compared against the original unmodified version of Chuffed on a statistically significant amount of test data.

While this section gives some insight into how this research took shape it omits many important details, for a more elaborate description refer to the corresponding sections, which constitute the remainder of this chapter.

4.2. Data

As is the case for any computer science experiment it is essential to use the right type of data. This research required a dataset containing a substantial amount of unsatisfiable and satisfiable constraint programming (CP) instances. To avoid testing on trivially simple instances the decision was made to use a dataset containing industrial instances which is the best way to simulate the performance of the proposed approach in a real application. Ideally, the dataset should also contain data for multiple different problem types in order to compare performances. To the best of my knowledge, the only two public CP benchmark datasets that contain a vast amount of industrial instances from multiple problem types are the MiniZinc benchmark suite [29] and CSPlib [12]. However, neither of these datasets contained sufficiently many unsatisfiable instances to train any machine learning model on. Nevertheless, multiple ways are investigated to still be able to create a dataset containing unsatisfiable instances. The first subsection discusses two possibilities for obtaining unsatisfiable data that have been considered, but were deemed insufficient for the purpose of this research. It is followed by a detailed explanation of the data acquisition procedure which was actually used. The last subsection shows some notable characteristics of the resulting dataset and briefly discusses their impact on this research.

4.2.1. Alternatives

To resolve the lack of unsatisfiable instances the following two options have been considered, but were ultimately not included in this research.

Generate unsatisfiable instances One common solution would be to just generate instances. There are two ways of doing so, random generation or procedural generation.

Achieving good results on random instances does not imply that the same approach will work for real-world applications. To get any scientifically relevant results on random instances they need to be provably difficult to solve and classify. Although there are some ways to realise these properties, such as generating instances at phase transition [48], this comes at the cost of having unstructured data. Similarly to the findings discussed in 3 it may be very difficult to machine learning techniques to achieve adequate performance.

Procedural generation, on the other hand, allows for more control over the structure of the generated instances. Unfortunately, designing a procedure which is able to output realistic instances in terms of structure and complexity would be beyond the scope of this research.

Add constraints to make instances unsatisfiable Another way to obtain unsatisfiable instances would be to use a satisfiable instance and continue adding constraints until it becomes unsatisfiable. This approach does introduce some problems, however. By design, the instances will always fail on at least some of these added constraints, since they were satisfiable before adding them. This introduces the problem that any machine learning model would learn a bias towards the artificial constraints, even if the artificial constraints were explicitly excluded from the unsatisfiable cores on which the machine learning model is trained. Because of this possible bias it is hard to tell if the classifier is actually learning something useful for the solver to use or something arbitrarily related to the added constraints.

4.2.2. Acquisition

The actual method for creating unsatisfiable data, which was performed for this research, was to use constraint optimisation problems (COP) from the MiniZinc benchmark suite [29] and transform them into unsatisfiable instances. This MiniZinc dataset originally contains 133 different problem types written in the standard MiniZinc language with separate model and data files. These files were processed according to the following steps in order to obtain the data used for this research.

1. First the full MiniZinc benchmark dataset was downloaded [29].
2. All model and data files were then flattened to obtain flatzinc and path files, this was done using the following command: *minizinc –solver chuffed –output-paths-to-file path_to/output.paths -I path_to/mznlib -o path_to/output.fzn -c path_to/minizinc.mzn path_to/data.dzn*.
3. Thereafter any instance containing the keyword “Satisfy” was removed from the dataset, meaning that only minimization and maximization instances remain.
4. The remaining instances were solved using a three hour timeout: *fzn-chuffed -s -f -t 10800000 path_to/instance.fzn* and the optimal objective value was saved to a file.
5. Any instances which could not be solved, such as the ’trucking_hl’ problem type, were removed from the dataset.
6. A copy of the resulting dataset, which contained the flatzinc files, was made. The instances in this copy were modified by setting an impossible domain for the objective value. Specifically, the upper bound of the objective value was set to be less than optimal value for minimization problems and the lower bound of the objective value was set to be higher than the optimal value for maximization problems. This caused all instances within this copy of the dataset to become unsatisfiable.
7. Since the machine learning model is trained on the unsatisfiable cores the variables belong to such core had to be extracted from the unsatisfiable instances. This was done by using a MiniZinc command which uses the previously stored path files and the unsatisfiable flatzinc files: *findMUS*

-a –ignore-sat-model path_to/instance.fzn path_to/paths.paths. The output of this command provided a set of variable identifiers for each unsatisfiable core. Unfortunately, this command was quite computationally demanding and did not work for all instances, this command successfully completed on only approximately 52% of the data.

Note that the listed steps are only the final steps which ended up contributing to the final result. A lot of trial and error went prior to discovering exactly which steps to take and how to perform them. In total it took several months to perform all steps from start to finish. However, using the final configuration, repeating all of the aforementioned steps would only take about a week combined. Especially extracting the unsatisfiable cores and solving all instances to optimality took a considerable amount of time. Initially all of these steps were executed on a personal computer. In the interest of time the solving of instances for the optimal value and unsatisfiable cores was later continued on a more powerful virtual machine provided by the Delft university of technology. This virtual machine allowed 16 threads to run in parallel, speeding up the process significantly. Even still these processes were slowed down by unforeseen errors during solving, these errors would occasionally cause one or more threads to freeze. Eventually, the scripts used for the execution of these steps had to be changed to handle these errors properly. Additionally, some intermediate steps had to be adapted along the way to better connect to subsequent steps.

4.2.3. Data characteristics

One of the most important things towards understanding research is to understand the data. This section is therefore dedicated to explaining the most important characteristics which may have impact on future applications of the data. The process of acquiring this data is described in in the previous section 4.2.2, this chapter discusses the data resulting from this process.

The definitive data which is used for this research comprises two datasets; one with satisfiable constraint optimisation problems (COP) and one containing unsatisfiable versions of the same instances. Additionally, for the unsatisfiable dataset information is stored on the unsatisfiable cores of over half the instances. The satisfiable dataset contains 13667 flatzinc files, which are distributed across 84 different problem types. The unsatisfiable dataset, which is the result of converting the satisfiable instances to unsatisfiable ones, contains slightly less instances at 12133 in total. This total ended up slightly lower because the procedure for generating unsatisfiable instances required solving the satisfiable version, which was not possible for all instances within a three hour time limit. For only 8057 out of these 12133 instances it was possible to extract unsatisfiable core data within the three hour time limit. For a complete overview of the amounts of data for each problem type refer to Appendix A.

While having 8057 instances to train on would normally be more than enough, just the sheer volume itself does not give any insight in the quality of the data. Another important factor to consider is the distribution of the data across the different problem types. Figure 4.1 shows a pie-chart of this distribution.

The pie-chart clearly shows that the Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP) dominates almost the entire dataset. The presence of such a dominant problem type is might introduce the issue that the machine learning model fails to learn any patterns from other problem types by focusing too much on the MRCPSP data.

Another difficulty is introduced when examining the distribution of the run-times of instances in the dataset. The data generation procedure was limited by the timeout of three hours for solving and extracting unsatisfiable core data, thus any instance that would require longer would be excluded from the final dataset. However, even accounting for the lack of such large instances, the vast majority of the data instances are solvable in less than a tenth of a second. The distribution of the run-times of the satisfiable instances can be seen in Figure 4.2.

For technical reasons this figure does not include instances which did not solve due to an error. Approximately five to six percent of the original instances did not finish solving due to an error.

The machine learning model uses this data to learn and predict whether a variable belongs to an unsatisfiable core. Therefore, it is also interesting to state that the unsatisfiable dataset contains a total 1532444 variables across all instances and 623293 of them belonged to an unsatisfiable core which amounts to roughly 40.67%. This means there is a slight class imbalance, but no class significantly dominates the other in such way that it will have a negative impact on the final predictions.

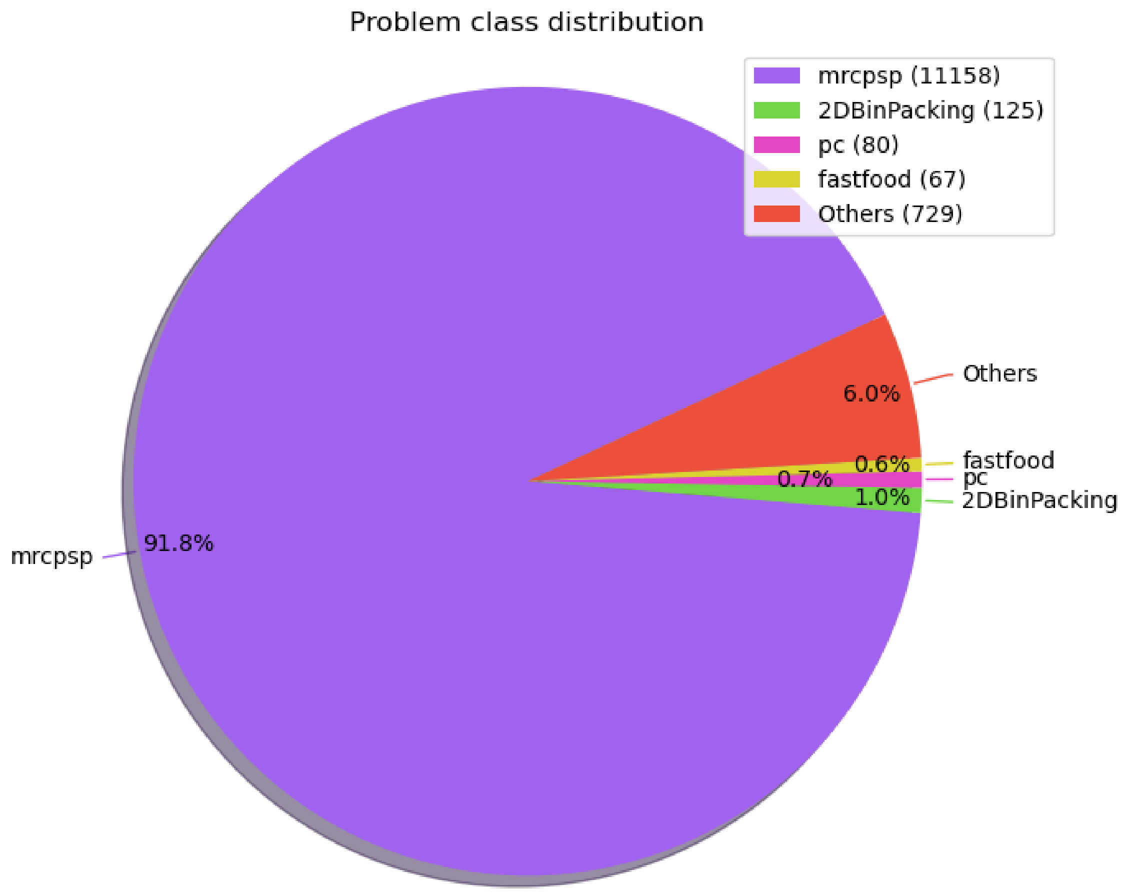


Figure 4.1: A pie-chart showing the distribution of the most common problem types.

4.3. Machine learning

It is important to realise that, similar to the research by Selsam and Bjørner [39], it is not the intention to achieve the best possible prediction scores. Perfect predictions are not necessarily more informative for the solver. For instance, it may be the case that the smallest unsatisfiable core includes all variables, which gives no meaningful information to the solver to branch on. Instead, the presumption is that the machine learning model will have imperfect predictions. The assumption is that the prediction confidence of a variable belonging in an unsatisfiable core correlates well with the asset of branching on that variable. In fact, having perfect predictions would even undermine the point of this research, since the goal is to use predictions on satisfiable instances to improve a branching heuristic.

For the aforementioned reason there is no emphasis on obtaining the best possible classification accuracy. Instead a machine learning model which works well for the available data is chosen and configured to obtain at least a decent accuracy, but is not extensively fine tuned.

The remainder of this section proves an in depth description of the selected machine learning procedure. It starts with the features which were selected from the data and is followed by a description and justification of the chosen machine learning model.

4.3.1. Model

As for the actual machine learning model, it made sense to use a model that has already proven to work for classifying CP instances in earlier research. Unfortunately, the publicly available models were often only trained for binary CSP, a variant of CSP where all variables are restricted to be Boolean. In reality CP instances, including the MiniZinc benchmark suite, are rarely binary. Theoretically, it is possible to convert between binary and non-binary CSP, which would retain the possibility of using of a

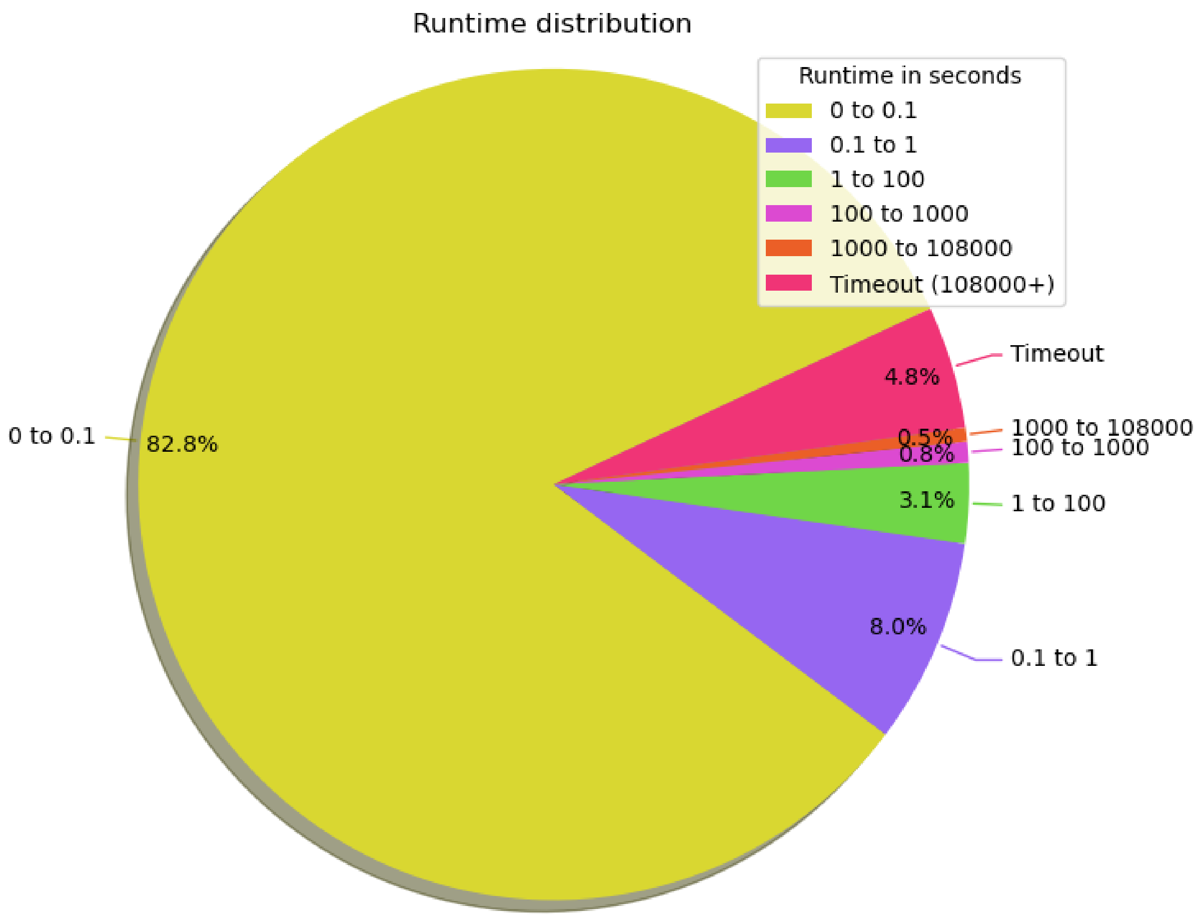


Figure 4.2: Distribution of six different run-time intervals for satisfiable instances.

binary CSP model. However, doing so would not a viable approach because of the massive translation overhead. Another approach would be to use the architecture of a good performing model on binary CSP and modifying it to work with non-binary instances. Based on the work presented by Samaras and Konstantinos [37] there is, unfortunately, no guarantee that such model would also work well for CP in general.

Because of the aforementioned reasons, a model needed to be implemented which had not yet been used for CP instances. Designing such model from scratch to would require considereable time, instead an existing architecture was selected for which an existing implementation was available to modify.

Even though the original research to unsatifiable core prediction for SAT [39] uses a simple version of the MPNN model used in earlier work on predicting satisfiability [40], the decision was made to use the architecture of a Graph Convolutional Neural Network (GCN). Both MPNNs and GCNs are known for working particularly well on undirected graph structures. Both SAT and CP instances can be represented with an undirected graph where the variables are represented by nodes and the constraints are represented with vertices between the nodes corresponding to the variables that are restricted by the constraint. For SAT it would suffice to represent an instance as a graph containing a node for each literal. However, using the same approach for CP would impose an exponential increase in the size of each graph as it would have to contain a node for each possible value in the domain of each variable. Contrarily, using a GCN allows individual features representing the domain and neighbours of each node, which means the graph representation can be reduced to just having a single node for each variable.

The GCN model which has been used for this research was originally implemented by Kipf [25] and was made available at `https://github.com/tkipf/gcn`.

A GCN works by learning a function of the features on a graph. In this case no actual graph was constructed but it is sufficient that the data is structured in such way that it could be represented with a graph. A simplified overview of the architecture is shown in Figure 4.3.

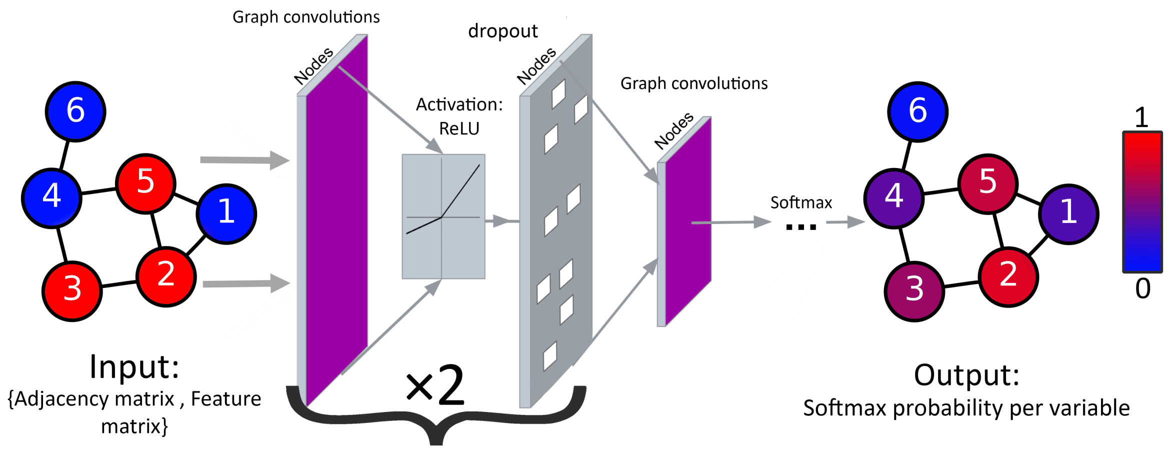


Figure 4.3: Visualisation of the Graph Convolutional Network architecture

The input of this GCN model is:

1. **A feature matrix of size N×D** Here N represents the number of variables and D the number of selected features.
2. **An adjacency matrix of size N×N** In this matrix variables are considered adjacent if they co-occur in a constraint.
3. **The labels in an N×C matrix** Here C represents the number of output classes, in this case 2; one for variables which are part of an unsatisfiable core and the other for variables which are not.

The output of the model is a **N×C** matrix which contains the output of the soft-max function for each variable. The resulting values from soft-max can be interpreted as the probability for each variable to belonging to each class. This research only considers two distinct classes; a variable either belongs to an unsatisfiable core or it does not. This means that the probability of belonging to the first class equals one minus the probability of belonging to the second class. Therefore it is possible to express the output of the machine learning predictions with a single value. For the remainder of this document this value will be referred to as the prediction confidence, representing the probability of a variable belonging to an unsatisfiable core.

The original implementation of the GCN model by Kipf [25] did not work for the data described in the previous section 4.2.3, therefore the data processing part of the model was rewritten accordingly. Besides this modification, no further changes were made to the architecture. The chosen parameters of the model will be covered in Section 4.3.3.

4.3.2. Features

For any data to be usable for any machine learning model, the first step is to select features from the data. Determining representative features is paramount to the effectiveness of the model.

The features were selected to provide as much information on a variable as possible while minimising the time required to extract them. Ultimately, the features used for this research are:

1. Categorical features indicating if a variable is declared as a Boolean, integer, float or set.
2. Minimum value of the variable domain.
3. Maximum value of the variable domain.
4. The range of the variable domain.

5. A set of identifiers of variables which co-occur in some constraint.

The categorical features were implemented using one-hot-encoding which means a binary feature is used for each category to indicate whether it applies to the variable. With sufficient knowledge on the implementation of Chuffed these features can be integrated in the flatzinc parsing process of Chuffed with very minimal overhead. However for this research the features were extracted through an external Python script specifically written for this purpose. This script was used to parse any set of flatzinc files and store the extracted features in a format which is directly usable by the modified GCN model described in 4.3.1.

4.3.3. Configuration

The performance of most machine learning architectures varies depend largely on the selected parameters. For this research the parameters of the model were set based on a couple empirical trials.

1. Learning rate: 0.3
2. Number of epochs: 200
3. Number of units in the first hidden layer: 16
4. Dropout rate: 0.1
5. Weight decay: 5e^{-4}
6. Tolerance for early stopping: 10

With this configuration and the selected features the training would normally terminate before reaching 100 epochs due to early stopping. This means that, with the given parameters, it took around 90 epochs for the accuracy of the model to not significantly improve anymore. At this stage the model must have reached either a local optimum or the global optimum. The accuracy achieved after this training phase varied roughly between 0.70 and 0.80, depending on the training data. While these results are definitely not disappointing given the complexity of the data, it is worth noting, however, that a high accuracy does not necessarily translate to being useful as a branching heuristic. Therefore, while it may have been possible to improve these results using different models or parameters, no further efforts have been made towards achieving the best possible predictions. For these reasons the resulting model was used for all following experiments.

4.4. Implementation

With both the data and the machine learning model covered, the remaining part was to combine them in such way that the output can be used by Chuffed as a heuristic. This section covers the integration of the prediction confidences with the VSIDS heuristic of Chuffed. This section is divided in a part about the integration of all the components and a part describing how the parameters of Chuffed had to be configured.

4.4.1. Integration

A modified version of Chuffed had to be created to allow for the integration of the machine learning output with Chuffed. This was done by creating a local repository with Chuffed's code as available at `https://github.com/chuffed/chuffed` on the 3rd of November 2019. In total three copies of Chuffed were used, two of them were modified to use the machine learning output the other was configured exactly the same but without the machine learning integration. Chapter 5 explains the difference between the machine learning enhanced version and compares them against the unmodified version of Chuffed.

The machine learning model described in Section 4.3.1 outputs the prediction confidences of a variable belonging to an unsatisfiable core. Because this model was implemented using external feature extraction the output would not be directly accessible within Chuffed's code. Therefore, the output needed to be integrated with the Chuffed solver using an intermediate data file. For each instance in the test-set a comma-separated file was created containing the prediction confidences alongside the flatzinc identifier of tat variable. Internally, Chuffed refers to the same flatzinc identifiers when creating the variables which represent the CP problem. Whenever a variable is created Chuffed also assigns a VIDS activity score to that variable. By default this activity score gets assigned a zero. By making use of the previously created comma-separated file containing it was possible to match the variables created internally in Chuffed with the prediction confidences from the machine learning model. This allowed Chuffed to be modified in such way that, instead of assigning zero, it was now possible to initialise the VSIDS with different activity scores depending on the prediction confidences.

Besides the previously mentioned modification and the parameters discussed in the following Section 4.4.2 no further changes were made to version of Chuffed that were used during this research.

4.4.2. Chuffed's Parameters

For the approach to work it was necessary that some of Chuffed's parameters were changed in order to actually use the VSIDS heuristic. By default Chuffed is configured to only use search annotations. The free search procedure of Chuffed will start the solving procedure with the VSIDS heuristic disabled, the VSIDS will automatically be enabled once a predetermined number of conflicts is reached. By just enabling free search, which can be done through the *-f* command-line option, Chuffed would still use annotations until at least a billion conflicts have been encountered. Realistically, Chuffed can manage around 20.000 conflicts per second, meaning that it would use annotations for the first one and a half hour before switching to VSIDS. With the VSIDS being initialised with machine learning predictions it made sense to set the number of required conflicts to a much smaller number, in this case a value of 100 was chosen. This has the effect that the solver uses VSIDS almost immediately and therefore takes advantage of the learned initialisation to make branching decisions.

5 Experiments

The most straightforward way to verify if the machine learning output is actually informative to the solver is to conduct experiments. This chapter covers all relevant experiments that contribute towards evaluating the machine learning integration into Chuffed. More experiments have been conducted throughout the duration of this research in order to figure out how to correctly set up the procedure and choose the right parameters for Chuffed. However, these preliminary experiments are excluded from this document because they do not provide any additional insight in the performance of the final version. First an overview is given of the general procedure which is used throughout all experiments. The second section of this chapter describes the conducted experiments and discusses their results.

5.1. Experimental setup

The aim of the conducted experiments was to find out if machine learning predictions are any informative for the solver and to determine the effect of training on different problem types. In order to determine the effect of training on different problems, all the experiments conducted in this document are all directed at solving instances from a single problem type, but different training sets are used. As a result the following experimental setup was created.

For each experiment three different versions of Chuffed were compiled, these will be referred to as *Chuffed0_OG*, *Chuffed1_Ex* and *Chuffed1_Inc*. While all three versions have the same configuration, they are different in the way the machine learning was integrated. Besides the configuration Chuffed0_OG was left completely unmodified, and serves purpose as a benchmark. Chuffed1_Ex was modified to have the VSIDS scores initialised with the predictions obtained from training on a training set which contained only instances from different problem types. Similarly, Chuffed1_Inc was modified to initialise the VSIDS scores with predictions from training on all training instances, including from the same problem type. These different version were used to solve different instances from the selected problem type and their performance was compared against each other.

All experiments conducted for this research follow the same general procedure, using the data described in section 4.2 and the Graph Convolutional Network (GCN) covered in section 4.3.1. Additionally, all experiments discussed in this chapter are performed on the virtual machine provided by the Delft university of technology. This virtual machine uses an Intel®Xeon Gold 6248 CPU @ 2.50 GHz with 16 cores and has access to 32GB RAM.

Any machine learning approach relies on training on a sufficiently large data set. For this reason the experiments discussed in this section are conducted on the four largest problem classes. As described in the previous chapter 4.2.3 the majority of the data belongs to the Multi-mode Resource-Constrained Project Scheduling (MRCPSP) problem type. Therefore it made sense to start experimenting with MRCPSP instances, thereafter some more experiments were conducted to compare the performance on different problem types. The general procedure for each experiment is visualised in the flowchart shown in Figure 5.1.

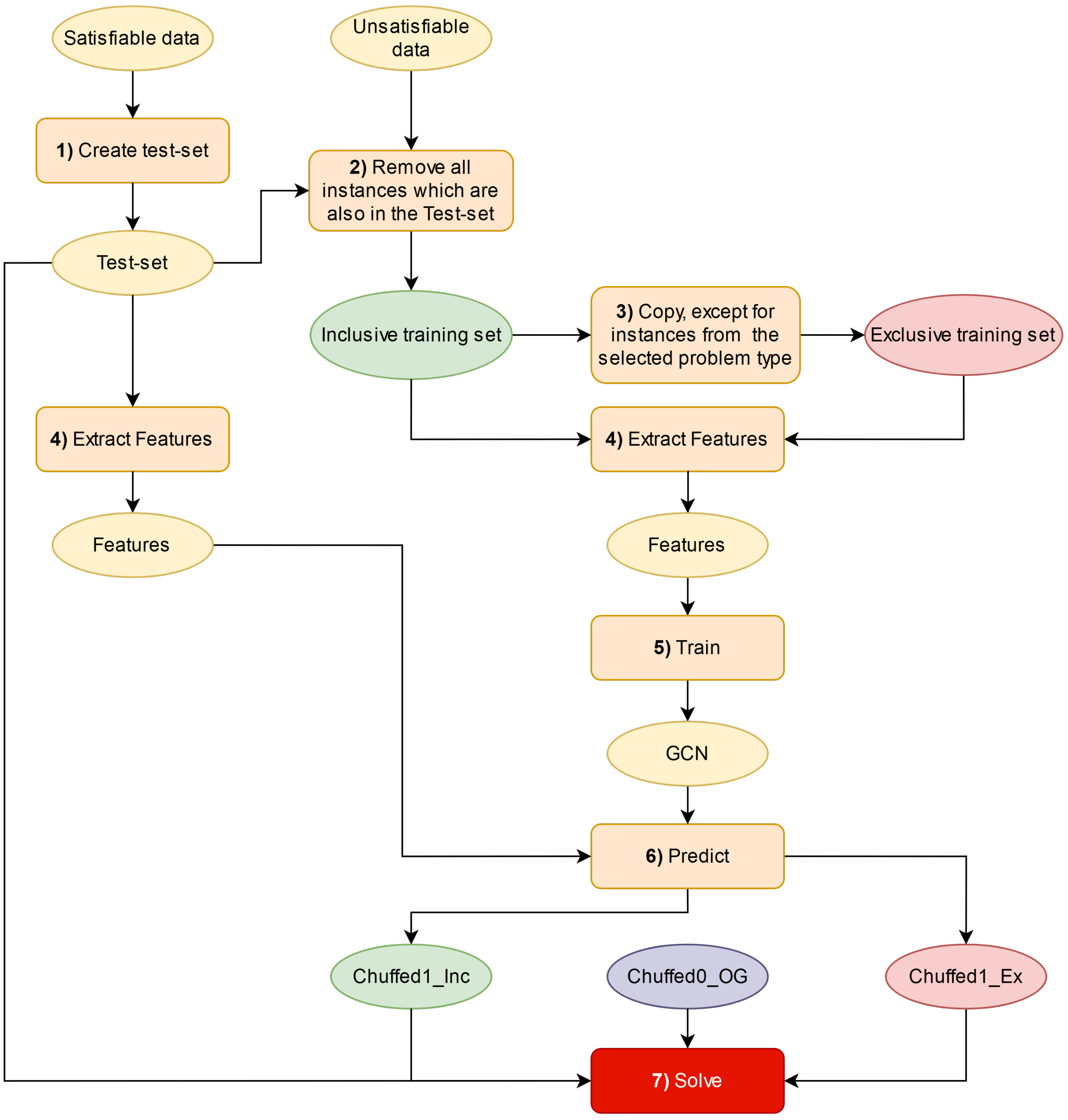


Figure 5.1: Box-plot showing the total run-time of all test instances averaged over 10 runs.

The following list provides a more elaborate explanation for the steps shown in Figure 5.1:

1. **Create test-set** The first part of the procedure was to construct a test-set which by setting aside a selection of instances from the selected problem type. The instances are drawn from the satisfiable dataset described in Section 4.2, these instances will be solved using different version of Chuffed to compare the difference in performance.
2. **Remove instances from unsatisfiable data to create the training data** Since the unsatisfiable dataset, which is used for training, is obtained by direct modification of instances from the satisfiable dataset it may occur that the unsatisfiable instance in the training set corresponds to a satisfiable instance from the test-set, for good practise any such instance is not included in the training data.
3. **Create two copies of the training data** Two different training sets were used for the experiments. The first training set contained all unsatisfiable instances, including those of the same problem type. This training set will be referred to as the inclusive training set. The second, exclusive, training set contained only instances from the unsatisfiable dataset which belonged to problem classes other than the selected problem type. Consequently, the exclusive training set is a direct

subset of the inclusive training set and could be created by making a selective copy of the inclusive training set.

4. **Parse all instances to extract features** The GCN model requires features to be extracted from both training- and test datset as well as labels for the training dataset. Therefore the next step in the procedure is to parse the instances using an external Python script and store the corresponding features and labels.
5. **Train GCN model on the extracted features** The GCN is trained separately on the features extracted from the inclusive and exclusive training sets.
6. **Create predictions for the test-set** After training the GCN on either the inclusive or exclusive training sets the model is used to make predictions using the features from the test-set. These predictions are then used to initialise the VISDS scores for the machine learning enhanced versions; Chuffed1_Ex and Chuffed1_Inc.
7. **Solve the instances from the test-set** Finally, all instances from the test-set are supplied to each of the previously described versions of Chuffed, *Chuffed0_OG*, *Chuffed1_Ex* and *Chuffed1_Inc*, and the resulting performances are monitored and analysed.

Besides run-time, the following statistics have been recorded for each run: a flag indicating timeout, the number of visited nodes and the best found objective value. An example of the output for a single run is shown in Appendix B.1. However, these statistics did not contribute any information and are excluded for the remainder of this chapter.

5.2. Exploratory trial

Initially, an exploratory trial was performed to verify that the developed methods work as intended and to see if any performance is gained using the machine learning enhanced versions. For this experiment all instances from the test set were solved with all three Chuffed version for a total of 10 times per instance. The box-plot shown in figure 5.2 gives an overview of the results of this trial.

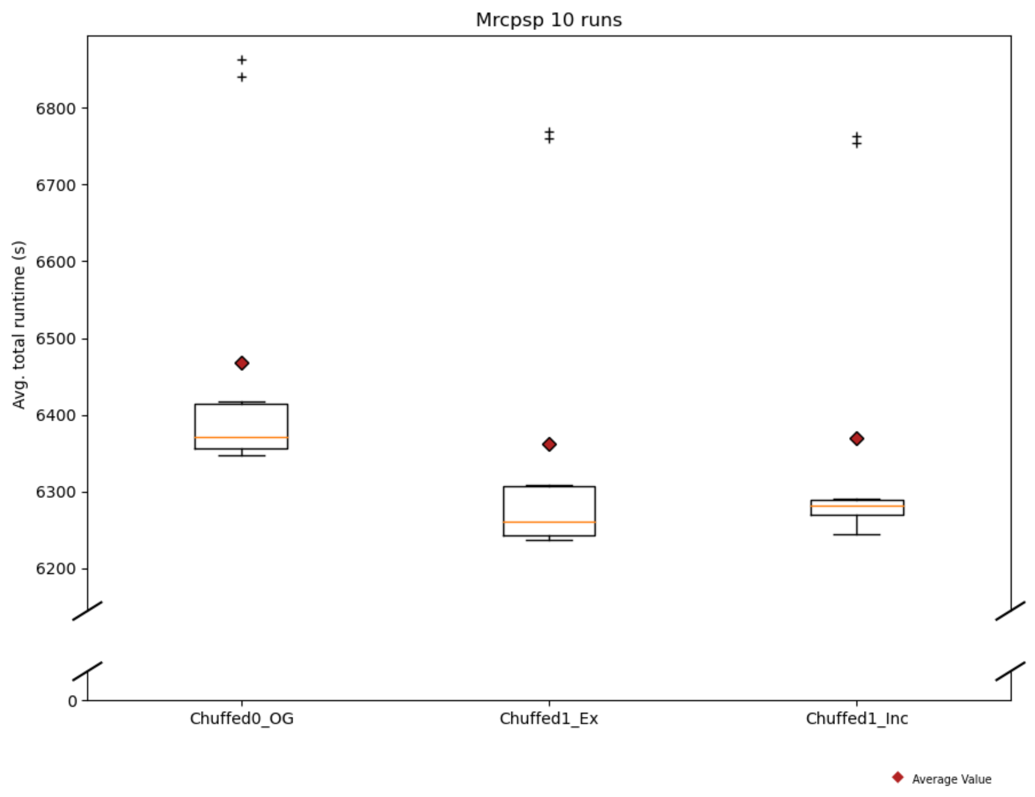


Figure 5.2: Box-plot showing the total run-time of all test instances averaged over 10 runs.

Figure 5.2 shows that both machine learning enhanced version of Chuffed outperform the version which does not use machine learning. However, while these results may seem promising, there is a lot of variance going on due to the outliers, this could indicate that these results may be different when repeating this experiment. Therefore, these results do not provide sufficient statistical evidence to make any strong scientific claims. Since the experiment was conducted on a virtual machine with no other active processes it was safe to assume that there were no significant external influences which may have caused this variance in run-time. This means that the variance is likely caused by Chuffed's solving procedure thus the only way to reduce this variance is to repeat the same experiment with increased number of measurements.

In an attempt to minimise the variance the amount of runs have, therefore, been increased to 100 per instance for any further experiment.

5.3. Results

Further experiments were conducted to verify that the machine learning enhanced version outperforms the unmodified version. The same procedure used for the exploratory trial 5.2 was repeated on the four largest problem types: MRCPSP, Bin-packing, Price-collecting and Fastfood. The goal was to find out if machine learning has a different effect on the performance for different problem types. Once again, the same three versions *Chuffed0_OG*, *Chuffed1_Ex* and *Chuffed1_Inc* were used to investigate the effect of training on instances from different problem types. To address the large variance encountered during the exploratory trial 5.2 all following trials were run a total of 100 times instead of 10. The results of these 100 runs for the four largest problem types are presented in the following subsections.

Thereafter, Chapter 6 provides an analysis and interpretation of the results presented in these subsections.

5.3.1. MRCPSP

The box-plot in figure 5.3 below shows the distribution of the the total run-time of all instances from the test-set averaged over the 100 runs.

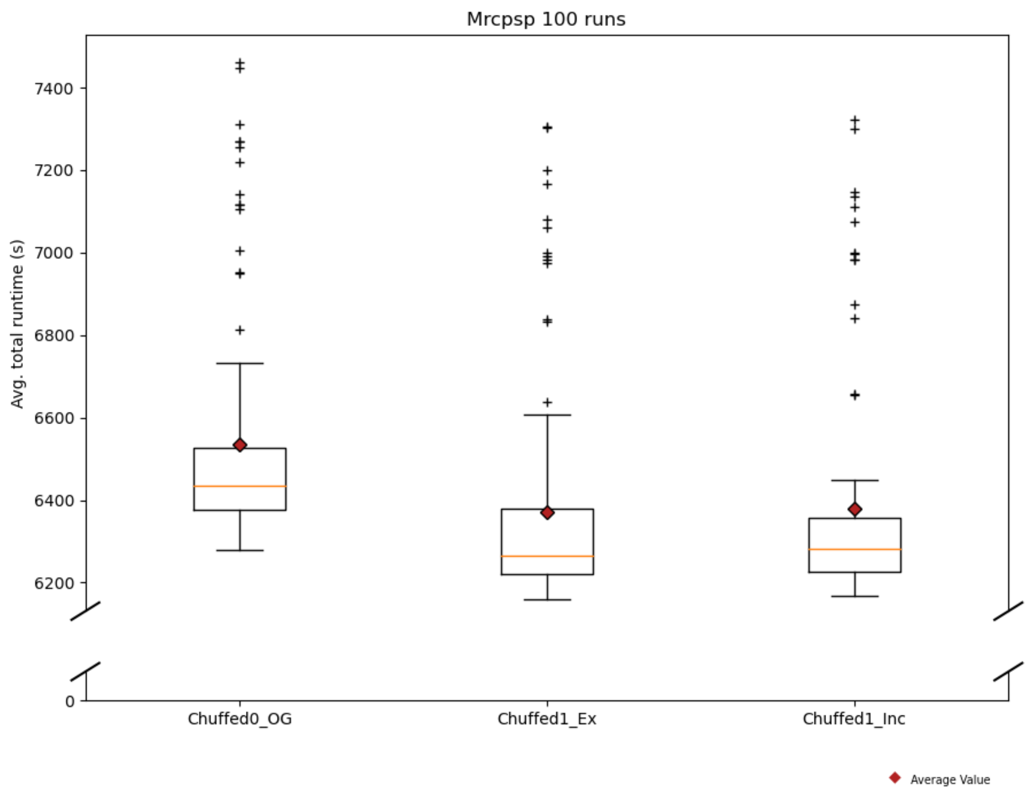


Figure 5.3: Box-plot showing the total run-time of all test instances averaged over 100 runs.

A more detailed summary of the results for this experiment is presented in table 5.1, which shows the average run-time over 100 runs for each of the instances from the test-set as well as some statistics on the total run-time.

	Chuffed0_OG	**Chuffed1_Ex**	**Chuffed1_Inc**
Instances	Avg. runtime(s)	Avg. runtime(s)	Avg. runtime(s)
mrcpsp10900	4.507	4.356	4.461
mrcpsp36	2.399	2.428	2.410
mrcpsp4425	311.565	296.139	302.595
mrcpsp4777	5274.736	5153.284	5155.367
mrcpsp4871	892.922	865.954	865.404
mrcpsp4960	32.713	32.241	32.099
mrcpsp7051	16.091	15.884	16.028
mrcpsp896	0.152	0.155	0.189
mrcpsp9880	0.236	0.241	0.240
mrcpsp9994	0.033	0.034	0.035
Total(s)	6535.354	6370.715	6378.829
Standard Deviation	282.493	273.983	271.103
Relative(%)	100.0%	97.5%	97.6%

Table 5.1: Average run-time per mrcpsp instance across 100 runs.

5.3.2. Bin-Packing

A box-plot showing the results of 100 runs on the bin-packing instances is presented in figure 5.4.

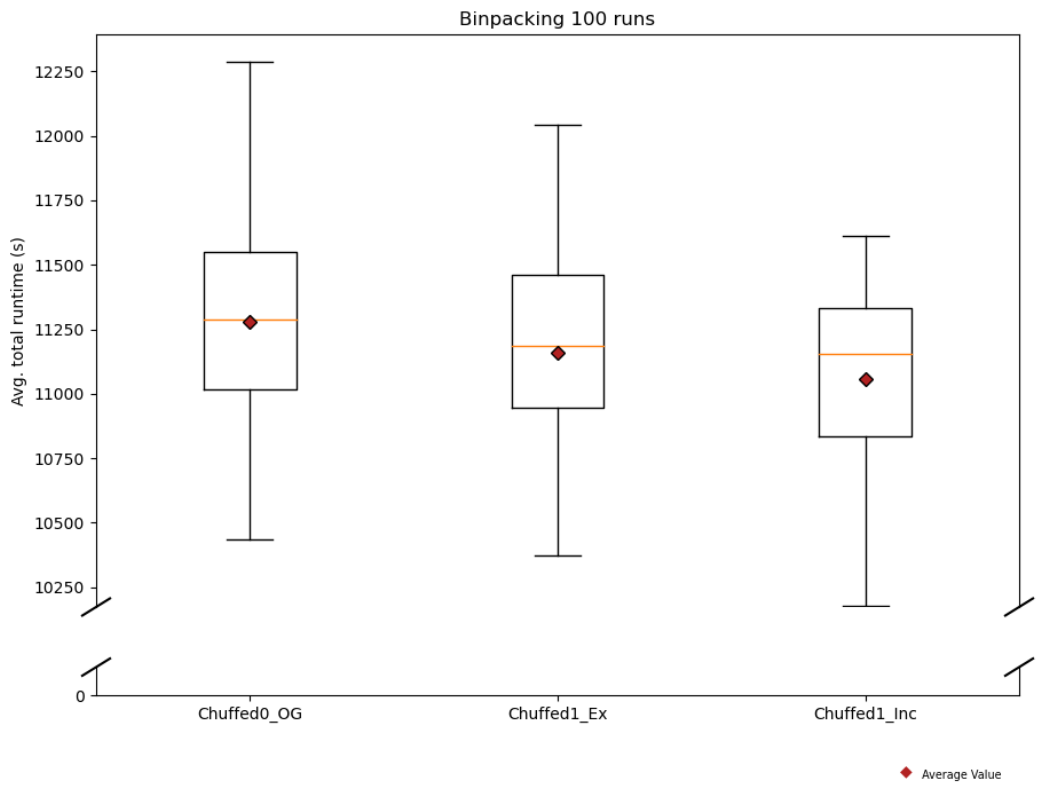


Figure 5.4: Box-plot showing the total run-time of all test instances averaged over 100 runs.

The following table 5.2 shows the average run-time per instance as well some statistics on the total run-time.

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
2DLevelPacking238	171.700	151.000	152.580
2DLevelPacking23	1563.956	1499.611	1512.328
2DLevelPacking492	1221.866	1275.854	1237.965
2DPacking13	5065.462	5037.534	5025.021
2DPacking165	683.933	708.044	641.285
2DPacking168	2511.413	2430.075	2431.017
2DPacking62	58.744	57.180	57.587
Total(s)	11277.074	11159.298	11057.783
Standard Deviation	381.016	359.230	347.639
Relative(%)	100.0%	99.0%	98.1%

Table 5.2: Average run-time per bin-packing instance across 100 runs.

5.3.3. Price-collecting

A box-plot showing the results of 100 runs on the price-collecting instances is presented in figure 5.5.

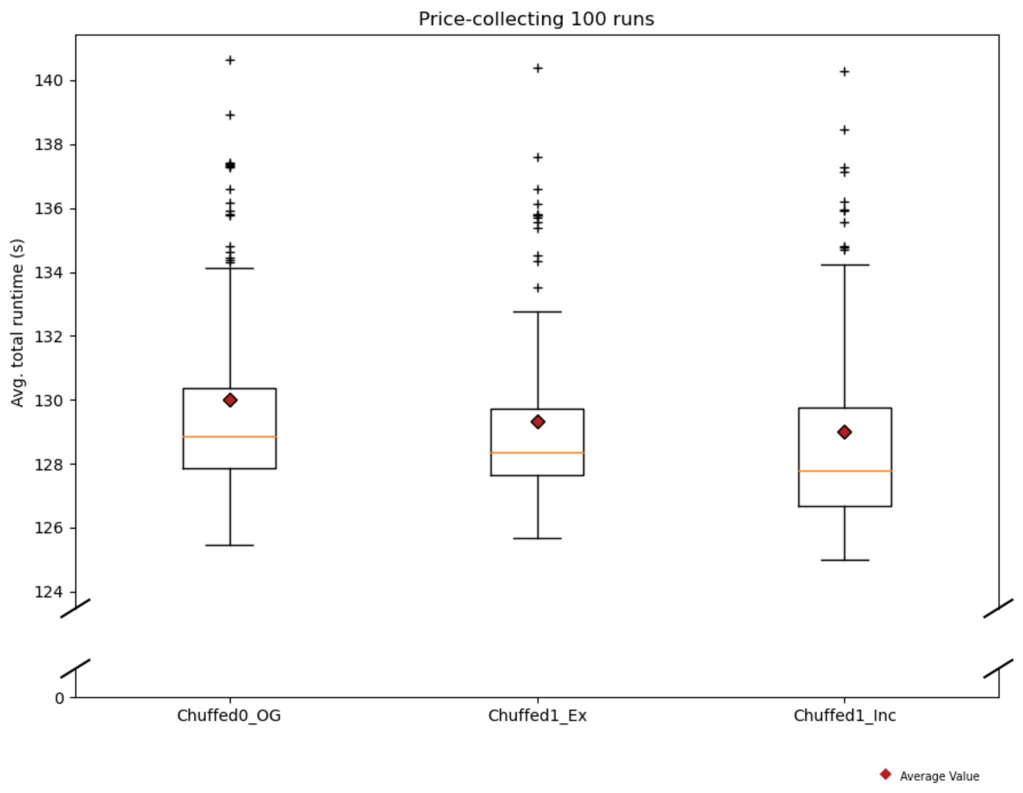


Figure 5.5: Box-plot showing the total run-time of all test instances averaged over 100 runs.

The following table 5.3 shows the average run-time per instance as well as some statistics on the total run-time.

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
pc52	12.211	12.326	12.152
pc56	8.777	8.750	8.750
pc58	15.283	15.409	15.431
pc61	10.501	10.648	10.644
pc65	12.111	11.908	12.049
pc73	42.743	42.407	42.948
pc77	7.886	8.013	8.040
pc79	20.479	20.631	20.709
Total(s)	129.991	130.092	130.722
Standard Deviation	3.373	2.895	3.452
Relative(%)	100.0%	100.1%	100.6%

Table 5.3: Average run-time per price-collecting instance across 100 runs.

5.3.4. Fastfood

A box-plot showing the results of 100 runs on the fastfood instances is presented in Figure 5.6.

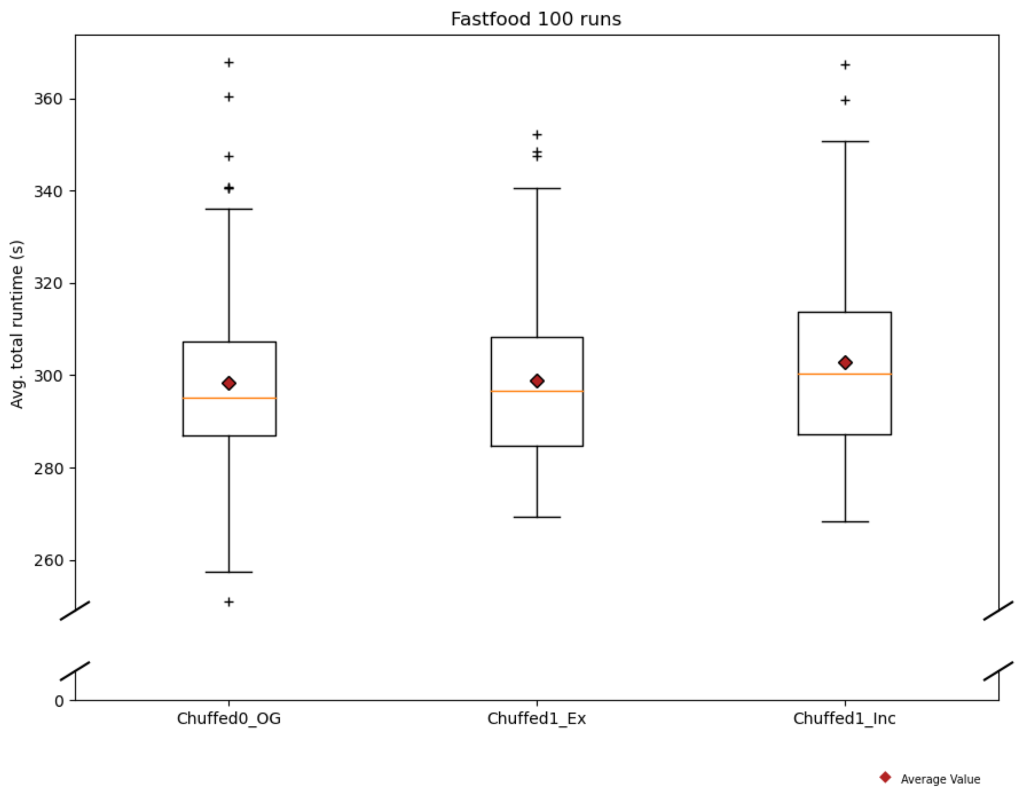


Figure 5.6: Box-plot showing the total run-time of all test instances averaged over 100 runs.

The following table 5.4 shows the average run-time per instance as well as a some statistics on the total run-time.

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
fastfood15	30.568	31.614	31.580
fastfood17	23.212	25.660	23.382
fastfood20	8.492	6.361	6.867
fastfood36	6.414	6.464	6.396
fastfood53	80.526	86.375	86.946
fastfood58	49.063	40.309	45.349
fastfood61	18.369	20.553	20.467
fastfood74	81.844	84.775	82.858
Total(s)	298.487	302.111	303.844
Standard Deviation	20.318	18.402	19.197
Relative(%)	100.0%	101.2%	101.8%

Table 5.4: Average run-time per fastfood instance across 100 runs.

6 Discussion

This section analyses the results presented in chapter 5 and discusses their significance. First an analysis is given on the statistical significance of the observed results, thereafter the results are interpreted within the context of this research.

6.0.1. Statistical significance

With 100 runs per instance it becomes visible that the outliers, which have been discussed in section 5.2, are still present. However, they they do appear consistently for version of Chuffed. Given the large amount of time required for performing these trials it would not be feasible to further increase the amount of runs to the point where the outliers are completely flattened.

Instead, to confirm the statistical significance of these observations a two-tailed t-test [24] is conducted to prove that the results are significantly different. The results of this t-test are shown in Table 6.1.

MRCPSP		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	4.163	$4.693e^{-5}$
Chuffed0_OG - Chuffed1_Inc	3.978	$9.761e^{-5}$
Chuffed1_Ex - Chuffed1_Inc	-0.209	0.834

Binpacking		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	2.238	0.026
Chuffed0_OG - Chuffed1_Inc	4.230	$3.577e^{-5}$
Chuffed1_Ex - Chuffed1_Inc	-2.020	0.045

Price-collecting		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	-0.226	0.821
Chuffed0_OG - Chuffed1_Inc	-1.506	0.134
Chuffed1_Ex - Chuffed1_Inc	-1.390	0.166

Fastfood		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	-1.316	0.190
Chuffed0_OG - Chuffed1_Inc	-1.907	0.058
Chuffed1_Ex - Chuffed1_Inc	-0.648	0.518

Table 6.1: T-Test analysis

The p-values shown in Table 6.1 mean that for the results of the two selected version pairs, under the hypothesis that they follow the same distribution, the probability of obtaining less similar results is less than the P-Value for this version pair. Obtaining very small P-Values indicate that there is evidence for the hypothesis to be rejected, meaning that the distributions are in fact significantly different.

For MRCPSP the probability for obtaining less similar results compared to Chuffed0_OG is less than 0.01% for Chuffed1_Inc and less than 0.005% for Chuffed1_Ex, which provides sufficient statistical evidence to conclude that they outperform Chuffed0_OG. There is, however, no sufficient statistical evidence to conclude any significant difference between the results obtained with Chuffed1_Inc and Chuffed1_Ex. The probability of obtaining less similar results with identical distribution amounts to more than 83%. With bin-packing, in addition to the difference between the machine learning enhanced version compared to Chuffed0_OG, there is also a statistically significant difference between Chuffed1_Ex and Chuffed1_Inc.

For price-collecting and fastfood there is insufficient evidence to conclude any significant differences between results of the different Chuffed versions.

6.0.2. Interpretation

The statistical analyses in the previous section 6.0.1 confirms the observation that the machine learning enhanced versions of chuffed Chuffed1_Inc and Chuffed1_Ex both outperform the unmodified version Chuffed0_OG on the MRCPSP and bin-packing instances.

The fact that Chuffed1_Inc and Chuffed1_Ex achieve such similar performance is particularly interesting. This indicates that the machine learning generalises well enough to be able to learn informative concepts from other problem types. This is especially surprising because MRCPSP represents over 91% of the training data, meaning that similar performance can be achieved not only without learning from any similar problem but also with less than 9% of the data available.

While it is to be expected, given the available data, that MRCPSP achieves the best performance, it is still interesting that an improvement is obtained for bin-packing. It is worth noting that, despite being the 2nd largest class, bin-packing is still significantly smaller than MRCPSP with unsatisfiable core data only being available for 17 '2DPacking' instances and 59 '2DLevelPacking' instances. Although very limited training and test data was available, both machine learning enhanced version of Chuffed were still able to achieve better overall performance on bin-packing than the unmodified version, Chuffed0_OG. The difference in performance has however has somewhat decreased in comparison to MRCPSP.

Interestingly, for bin-packing, there is a larger difference between the performance of Chuffed1_Inc and Chuffed1_Ex. This may indicate that bin-packing shares less learn-able concepts with other problem types than MRCPSP.

While the experiments on MRCPSP and bin-packing show promising results, the results for price-collecting and fastfood are a lot less convincing, however. There are multiple possible explanations for this observation.

One explanation could be that there was just very limited data available for these problem types. As discussed in section 4.2.3 the data-set is dominated by MRCPSP instances. Despite being the 3rd and 4th largest problem types there were only 65 training instances available for price-collecting and 33 for fastfood. However Chuffed1_Ex, which only used training data excluding MRCPSP instances, has shown promising performance despite only using around 9% of the data.

A more probable explanation is that, none of those instances required considerable solving time. The average run-time per MRCPSP instance stated in table 5.1 indicate that the machine learning integration works better for sizeable instances. Therefore it is most likely that lack of improvement on price-collecting and fastfood is not because they are less similar to other problem types but because the tested instances were not sufficiently large.

To further investigate the relation between run-time and the effect of machine learning the relative run-time of the machine learning enhanced versions have been plotted against the run-time of the unmodifed version of Chuffed, which can be seen in figure 6.1.

While there is an obvious declining trend visible in the plot, there is no clear linear or exponential relation between the machine learning improvement and the corresponding run-times. It does indeed show that the machine learning version generally perform worse on instances which take less than around four seconds to solve.

The spike at the start is caused by the mrcpsp896 instance which was solved relatively slow with Chuffed1_Inc, the reason for this could be that the concepts learned from similar MRCPSP instances do not work well for this particular instance.

Another interesting observation is that for very large instances the relative run-time starts to slightly increase again. While there are not enough data points to confirm this observation, it would make sense given that the effect of initialising the VSIDS heuristic would diminish as more conflicts are encountered.

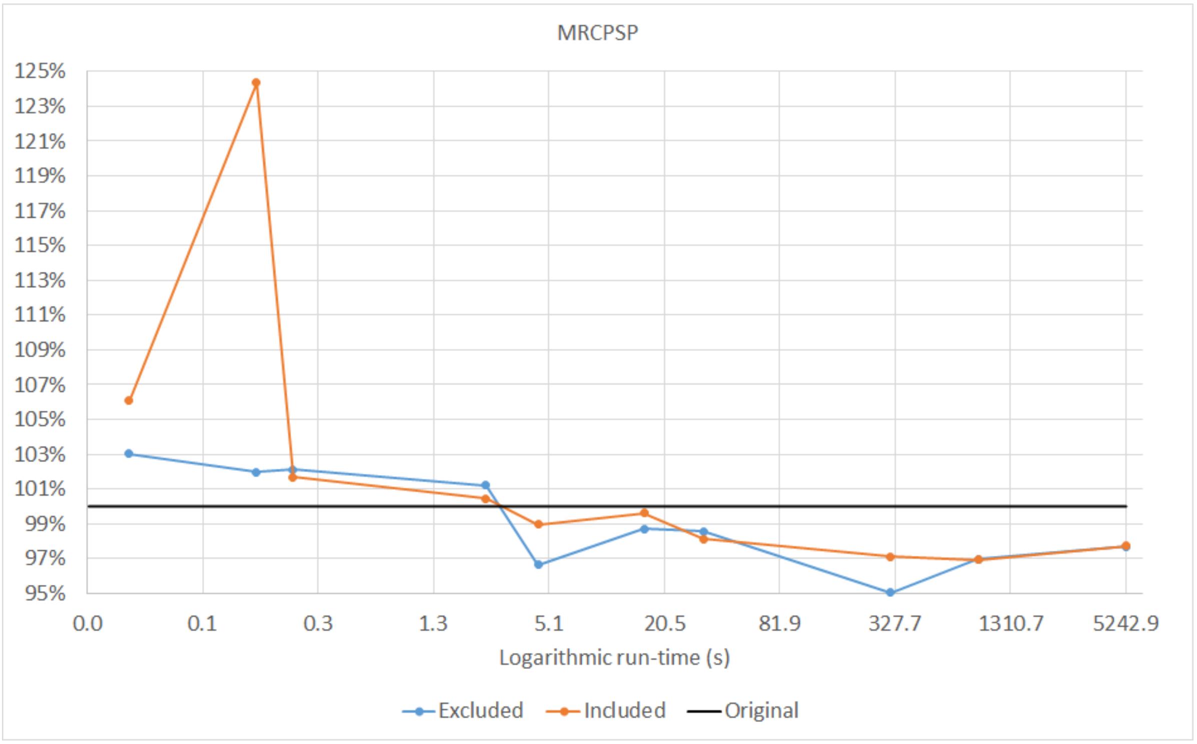


Figure 6.1: Relative run-time of Chuffed1_Ex and Chuffed1_Inc on MRCPSP instances plotted against the run-time of Chuffed0_OG.

7 Conclusion

In this chapter the main findings of the research are covered as well as their implications to the scientific field of research. This is done by first answering the research questions stated in 1.4. The next section explains the scientific impact of the findings. Finally, some suggestions are presented for future research on this topic.

7.1. Answers to the research questions

In this section an attempt is made to answer the research questions stated in section 1.4 according to the observations discussed in chapter 6.

- **Is it possible to reduce the run-time of solving a CP instance by integrating machine learning in the LCG solver Chuffed?** As seen most evidently on the larger instances of the MRCPSP and bin-packing problem types the machine learning integration is indeed able to reduce the run-time of solving an CP instance. The total run-time for all the MRCPSP instances in the selected test-set, for which most data was available, was proven with over 99.99% certainty to be significantly less compared to the run-time of the unmodified version. On average a 2.5% increase in performance was achieved with a standard deviation of approximately 4%.
- **Does the impact of using machine learning on the run-time of Chuffed differ across different problem types?** The impact of machine learning seems to depend mostly on the difficulty of the instance. For instances that take less than a four seconds the overhead of initialising VSIDS may outweigh the benefit gained from it. Because of limited available data it was not possible to determine a significantly different impact on performance across different problem types.
- **Does the impact of using machine learning depend on the type of problems in the training set?** For the performance gained on the MRCPSP instances it made no significant difference whether the Graph Convolutional Network model was trained exclusively on different problem types or also on MRCPSP instances. This suggests that the chosen model was able to generalise between the different problem types and still learn concepts which are useful to the solver. However, for the bin-packing problem type the performance was adversely affected by excluding any bin-packing instances from the training set. Therefore it may be concluded that it is possible to generalise between different problem types, but the performance may depend on the selected problem types.

7.2. Scientific impact

To this date, no other research has shown the possibility of using machine learning for Lazy Clause Generation (LCG) solvers. This study shows that it is possible to use machine learning approaches which are designed for solving SAT instances to improve LCG solving techniques. Specifically, this research has shown that it is possible to use unsatisfiable core learning, which originates from the work of Selsam and Bjørner [39], for improving the performance of the LCG solver Chuffed. With LCG approach dominating recent benchmarks it is interesting that the proposed approach is able to consistently achieve an improved performance on sizeable instances, even if only by a small margin. Most importantly, the findings of this study may encourage more researchers to try and adapt existing machine learning approach to further improve the performance of hybrid solving techniques.

7.3. Future research

This section gives some suggestions and points of attention for researchers who have interest in continuing this line of research.

- For this approach to be used in practical context the possibilities for integrating the classification part directly into the solver should be investigated, this would require embedding the feature extraction part directly into the solver.
- In order to examine the effect across different problem types this experiment it may be valuable to repeat this study with more evenly distributed data.
- While the Graph Convolutional Network model proved sufficient for this study, it may be worth it for any future research which involves adapting a similar approach to evaluate the effect of using different machine learning architectures.
- It may be interesting to determine the effect of using the periodic refocussing technique described in the original paper [39] and examine if it would provide better performance for instances which take longer than a couple of hours to solve.
- The nature of this approach may reduce the time required to determine unsatisfiability on unsatisfiable instances, this may be interesting to further investigate.
- Another and possibly more successful way to improve LCG solvers would be to learn the best configuration of the solver's parameters given a certain instance.
- Machine learning may also be useful for predicting no-goods or their activity scores.

A Available data instances

Problem types	FlatZinc # Files	UnSAT # Files	MUC # Files
mrcpsp	11182	11158	6425
2DLevelPacking	500	99	59
pc	80	80	65
fastfood	89	67	33
rect_packing_opt	61	48	40
amaze	47	47	30
open_stacks_	50	27	21
2DPacking	500	26	17
TableLayout	26	26	14
depot_placement	24	24	23
ship-schedule	24	24	18
ghoulomb	25	23	18
pattern_set_mining	32	20	8
tpp	20	20	18
gfd-schedule	20	19	9
roster_model	20	19	0
mspsp	20	19	15
filter	29	19	14
jobshop	82	18	8
rcpsp_max	20	18	11
mario	16	16	9
evilshop	16	16	8
cc_base	20	13	3
rcpsp	13	12	9
radiation	23	11	6
handball	10	10	0
rcmsp	20	10	8
golomb	10	9	9
parity-learning	9	9	7
shortest_path	10	9	9
still_life	14	9	6
still_life_free	10	9	5
still_life_no_border	10	9	6
rcpsp-wet	12	8	6
p1f	15	7	6
tdtsp	11	7	3
league	20	7	6
tcgc	7	7	4
cargo_coarsePiles	11	6	4
hrc	6	6	6
mznc2017_cargo	11	6	3
mapf	6	6	0
still_life_full_border	6	6	6
maximum-dag	6	6	5
dcmst	6	5	4
mknapsack_global	7	5	2
bus_scheduling	12	5	0
openshop	5	5	0
model	5	5	0
steelmillslab	6	5	0
train	15	5	2
group	6	5	0
sugiyama	5	5	5
sugiyama2	5	5	5
still-life	5	5	3
linear-to-program	5	5	4
trucking	15	5	5
smelt	5	4	3
crossword_opt	6	4	0
community-detection	7	4	1
rel2onto	6	4	4
mapping	6	4	2
oc-roster	5	3	1
fjsp	5	3	2
mqueens	5	3	2
pattern_set_mining	5	2	1
mznc2013_league	4	2	2
nfc	6	2	2
photo	2	2	2
mznc2017_aes_opt	6	2	0
spot	10	2	0
talent_scheduling	33	2	0
template_design	7	2	0
triangular	5	1	0
city-position	6	1	1
cutstock	121	1	1
ttppv	10	1	0
vrp	74	1	1
freepizza	5	1	1
road_naive	11	1	0
GridColoring	10	1	1
celar	10	0	0
cvrp	5	0	0
gbac	10	0	0
jp-encoding	10	0	0
trucking_hl	5	0	0
largecumulative	5	0	0
opd	11	0	0
wcsp	5	0	0
zephyrus	6	0	0
Total	13667	12133	7037

Table A.1: Total amount of data files processed

B Single run example

Version	Instance	Timeout	Run-time	Nodes	Objective
chuffed0_OG	mrcpsp10900	FALSE	5.623	39201	41
chuffed1_Ex	mrcpsp10900	FALSE	5.115	39201	41
chuffed1_Inc	mrcpsp10900	FALSE	3.779	39201	41
chuffed0_OG	mrcpsp36	FALSE	2.877	35302	48
chuffed1_Ex	mrcpsp36	FALSE	2.493	35302	48
chuffed1_Inc	mrcpsp36	FALSE	2.717	35302	48
chuffed0_OG	mrcpsp4425	FALSE	369.861	1424755	42
chuffed1_Ex	mrcpsp4425	FALSE	313.295	1424755	42
chuffed1_Inc	mrcpsp4425	FALSE	349.088	1424755	42
chuffed0_OG	mrcpsp4777	FALSE	5125.027	23222528	42
chuffed1_Ex	mrcpsp4777	FALSE	5029.44	23222528	42
chuffed1_Inc	mrcpsp4777	FALSE	5041.073	23222528	42
chuffed0_OG	mrcpsp4871	FALSE	1001.316	4952765	37
chuffed1_Ex	mrcpsp4871	FALSE	989.826	4952765	37
chuffed1_Inc	mrcpsp4871	FALSE	983.291	4952765	37
chuffed0_OG	mrcpsp4960	FALSE	38.007	228056	35
chuffed1_Ex	mrcpsp4960	FALSE	39.07	228056	35
chuffed1_Inc	mrcpsp4960	FALSE	39.698	228056	35
chuffed0_OG	mrcpsp7051	FALSE	14.516	92240	36
chuffed1_Ex	mrcpsp7051	FALSE	16.294	92240	36
chuffed1_Inc	mrcpsp7051	FALSE	13.592	92240	36
chuffed0_OG	mrcpsp896	FALSE	0.176	2630	27
chuffed1_Ex	mrcpsp896	FALSE	0.256	2630	27
chuffed1_Inc	mrcpsp896	FALSE	0.256	2630	27
chuffed0_OG	mrcpsp9880	FALSE	0.239	4302	45
chuffed1_Ex	mrcpsp9880	FALSE	0.246	4302	45
chuffed1_Inc	mrcpsp9880	FALSE	0.262	4302	45
chuffed0_OG	mrcpsp9994	FALSE	0.075	950	31
chuffed1_Ex	mrcpsp9994	FALSE	0.077	950	31
chuffed1_Inc	mrcpsp9994	FALSE	0.061	950	31

Table B.1: Output for a single run on MRCPSP instances.

C
Scientific paper

This appendix contains a draft of the scientific paper to be used for conference submission, which was written together with my supervisor Neil Yorke-Smith.

Unsatisfiable Core Learning for Chuffed

Ronald van Driel and **Neil Yorke-Smith**
Algorithmic group, Delft University of Technology, Netherlands
ronald_van_driel@outlook.com and n.yorke-smith@tudelft.nl[*]

Abstract

Contemporary research explores the possibilities of integrating machine learning (ML) approaches with traditional combinatorial optimisation solvers. Since optimisation hybrid solvers, which combine propositional satisfiability (SAT) and constraint programming (CP), dominate recent benchmarks, it is surprising that the literature has yet to develop machine learning approaches for hybrid CP–SAT solvers. We identify a recent technique called unsatisfiable core in the SAT literature as promising to improve the performance of the hybrid CP–SAT lazy clause generation solver Chuffed. We leverage a graph convolutional network (GCN) model, trained on an adapted version of the MiniZinc benchmark suite. The GCN predicts which variables belong to an unsatisfiable cores on CP instances; these predictions are used to initialise the activity score of Chuffed's Variable-State Independent Decaying Sum (VSIDS) heuristic. We benchmark the ML-aided Chuffed on MiniZinc benchmark suite and find a robust 2.5% gain over baseline Chuffed. This paper thus presents the first, to our knowledge, successful application of machine learning to improve hybrid CP–SAT solvers.

1 Introduction

Both propositional satisfiability (SAT) and constraint programming (CP) are immensely popular automated reasoning technologies, they form the foundation of many real-life problems such as: scheduling, logistics and resource allocation (Rossi, Van Beek, and Walsh 2006). Solving SAT or CP instances as quickly as possible is therefor of great economical interest to many companies. The popularity of SAT and CP is reflected in the vast amount of research that has been conducted over the years to improve SAT and CP solvers. Recently, Stuckey proposed a new CP solving technique called Lazy-Clause-Generation (LCG) (Stuckey 2010), this technique combines the conflict learning ability from SAT solvers with finite domain propagation from CP solvers, essentially creating a hybrid SAT/CP solver. These hybrid solvers have shown to be capable of beating modern state-of-the-art solvers.

2 Approach

This section provides a high-level design of the proposed approach for improving Chuffed with machine learning.

[*]Contact author

Similar to SAT solvers Chuffed is able to use the Variable-State Independent Decaying Sum (VSIDS) heuristic. VSIDS is usually implemented by keeping track of an activity score for each variable which indicates the value of branching on that variable. Normally the score for each variable is initialised at zero and incremented when the corresponding variable occurs as part of a conflict. To emphasise variables visited recently the activity scores are periodically decreased.

Consequently, the scores do not initially provide any information to the solver but they gradually become more useful. Chuffed typically uses search annotations before switching to VSIDS for making branching decisions. However, with machine learning the activity scores can be directly initialised, which may benefit the solver also in early stages of the solving procedure.

To achieve this a Graph Convolutional Network model is trained on unsatisfiable instances to make a prediction on which variables belong to an unsatisfiable core. An unsatisfiable core is a minimal subset of variables which can not not be simultaneously satisfied. The trained model is then used to classify the variables of an instances which needs to be solved and the softmax probabilities of this classification are used to initialise Chuffed's VSIDS scores.

3 Data

The approach described in section 2 requires two different datasets containing CP instances. One of these datasets should only contain unsatisfiable instances to train on, the other one should contain satsifiable instances to solve using this approach for evaluation.

The MiniZinc benchmark suite (MiniZinc 2016) was used to supply over 13.000 satsifiable instances for evaluation. However, to the best of my knowledge, no public CP dataset contained sufficiently many unsatisfiable instances for training any machine learning model on. Therefore the constrain optimisation problem (COP) instances from the MiniZinc benchmark suite were also modified to become unsatisfiable. This was done by first solving them for their optimal value. Then the original instance was modified by setting the domain of the objective variable to only include values better than the optimal value, which makes the instances unsatisfiable. While less computationally intesive alternatives exist, this procedure was selected with the intention to not introduce any unwanted bias for learning the unsatisfiable cores.

Using this procedure allowed to create both the satisfiable datset as well as the unsatisfiable dataset. For the unsatisfiable dataset the labels were generated using MiniZinc's 'findMUS' command. Ultimately the datasets contained 13667 instances for which features were available and 8057 instances for which labels could be extracted. Unfortunately over 90% of the data belonged to a single problem type being the Multi-mode Resource-Constrainted Project Scheduling Problem (MRCPSP), and over 80% of the data were instances which could be solved in less than 0.1 second. Because of that it was challenging to find enough sizeable instances to train and test on.

4 Implementation

This section covers the technique used to integrate machine learning with the LCG solver Chuffed. It is important to note that, similar to the approach proposed by Selsam and Bjørner (Selsam and Bjørner 2019), it is not the intention to achieve the best possible predictions. The reason for this is that more accurate predictions do not necessarily imply that they are more useful for the solver. In fact, if all variables of a satisfiable instance would be correctly classified as not being part of an unsatisfiable core with 100% certainty, it would not provide any information to the solver at all. Instead, the assumption is that the confidence of classifying a variable to be part of an unsatisfiable core correlates with the effectiveness of branching on that variable.

4.1 Features

Using an external Python script the available data was translated to a feature representation which the GCN model is able to use. For this research the following features were used:

1. Categorical features indicating if a variable is declared as a Boolean, integer, float or set.
2. Minimum value within the variable domain.
3. Maximum value within the variable domain.
4. The range of the variable domain.
5. A set of identifiers of variables which co-occur in some constraint.

4.2 Model

The implemented GCN model is taken from (Kipf and Welling 2016) which was made available at `https://github.com/tkipf/gcn`. A GCN works by learning a function of the features on a graph. In this case no actual graph was constructed but it is sufficient that the data is structured in such way that it could be represented with a graph.

The input of this GCN model is:

1. **A feature matrix of size N$\times$ D** Here N represents the number of variables and D the number of selected features.
2. **An adjacency matrix of size N$\times$ N** In this matrix variables are considered adjacent if they co-occur in a constraint.
3. **The labels in an $N\times$ C matrix** Here C represents the number of output classes, in this case 2; one for variables which are part of an unsatisfiable core and the other for variables which are not.

The output of the model is a $N\times$ **C** matrix which contains the soft-max output which can be interpreted as the probability for each variable to belonging to each class. Because this research only considers two classes, it is possible to express the output of the machine learning predictions with a single value, which is the prediction confidence of a variable belonging to an unsatisfiable core.

A simplified overview of the architecture is shown in Figure 1.

For this research the following parameters of the model were set based on a couple of empirical trials:

- Learning rate: 0.3
- Number of epochs: 200
- Number of units in the first hidden layer: 16
- Dropout rate: 0.1
- Weight decay: 5e^{-4}
- Tolerance for early stopping: 10

Using the aforementioned configuration of the GCN model, usually the early stopping criteria would be reached at around 100 epochs at which point an accuracy varying between 0.70 and 0.8 was achieved. It may be possible to achieve even better predictions but doing so would serve little purpose for the goals of this research.

4.3 Integration

A copy of Chuffed's code, which is available at `https://github.com/chuffed/chuffed`, was made at on the 3rd of November 2019. This code was modified to use the predictions from the GCN described in Section 4.2. The GCN was first trained to be able to output the prediction confidences of a variable belonging to an unsatisfiable core. For each instance in the test-set a comma-separated file was created containing the prediction confidences alongside the corresponding flatzinc identifier of that variable. Internally, Chuffed refers to the same flatzinc identifiers when creating the variables which represent the CP problem. By using the previously created comma-separated file it was possible to import the prediction confidences and match them with the variables created internally in Chuffed. This means it was now possible to initialise the VSIDS activity scores with respect to the prediction confidences instead of always assigning zero.

5 Evaluation

The research covered in this paper tries to answer the following research questions:

1. **Is it possible to reduce the run-time of solving a CP instance by integrating machine learning in the LCG solver Chuffed?**

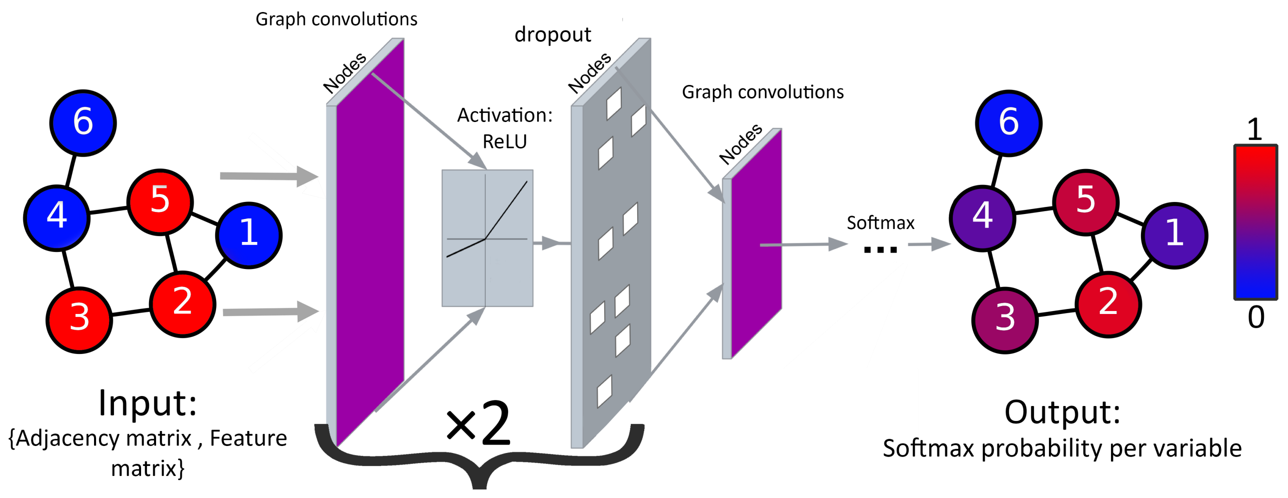


Figure 1: Visualisation of the Graph Convolutional Network architecture

2. **Does the impact of using machine learning on the run-time of Chuffed differ across different problem types?**
3. **Does the impact of using machine learning depend on the type of problems in the training set?**

This section cover the experiment which has been conducted to evaluate the effectiveness of the proposed approach. For this experiment three different versions of Chuffed were compiled, these will be referred to as *Chuffed0_OG*, *Chuffed1_Ex* and *Chuffed1_Inc*. All three of them were configured to switch to VSIDS as soon as 100 conflicts have been encountered. While all three versions have an identical configuration, they are different in the way the machine learning was integrated. Chuffed0_OG was otherwise left completely unmodified, and serves purpose as a benchmark. Chuffed1_Ex was modified to have the VSIDS scores initialised with the predictions obtained after being trained on a training set which contained only instances from other problem types. Similarly, Chuffed1_Inc was modified to initialise the VSIDS scores with predictions after being trained on all training instances, including from the same problem type.

These three different version were used to solve different selected test-sets containing instances from the four largest problem types: MRCPSP, Bin-packing, price-collecting and fastfood. Finally, their run-times were stored and compared against each other. This experiment was run entirely on a virtual machine provided by the Delft university of technology. This virtual machine uses an Intel®Xeon Gold 6248 CPU @ 2.50 GHz with 16 cores and has access to 32GB RAM.

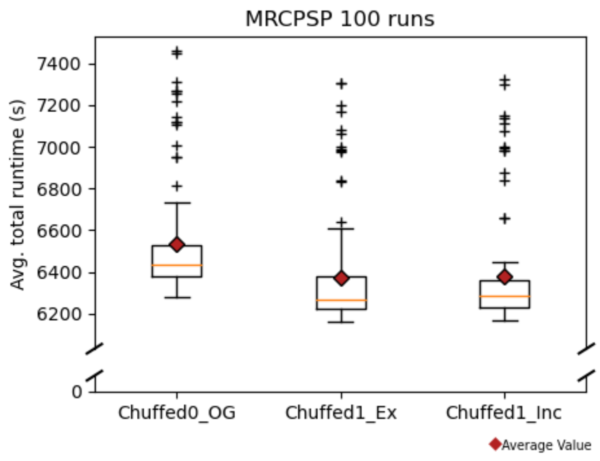
The box-plot in Figure 2 shows the resulting distribution of the the total run-times of all instances from the each of the four largest problem types, averaged over a total of 100 runs.

A more detailed summary of the results for this experiment is presented in Table 5, which shows the average run-time over 100 runs for each of the instances from the test-set as well as some statistics on the total run-time.

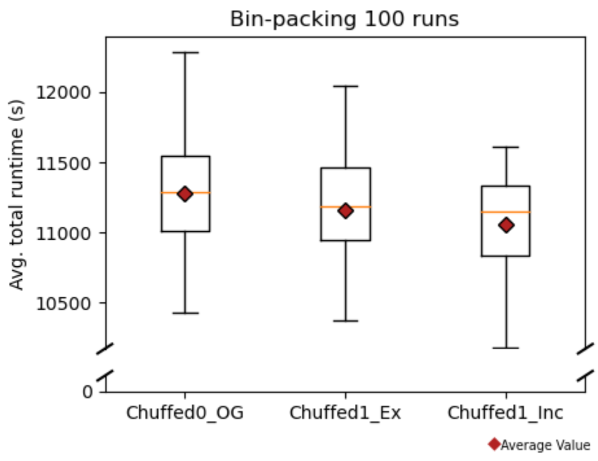
To confirm the statistical significance of the presented results a two-tailed t-test (Kim 2015) is performed. The results of this t-test are shown in Table 6.

The t-test analysis shows that the machine learning enhanced version significantly outperform the unmodifed version for both MRCPSP and Bin-packing instances. The probability for obtaining less similar results on the MRCPSP test-set compared to Chuffed0_OG is less than 0.01% for Chuffed1_Inc and less than 0.005% for Chuffed1_Ex. For Bin-packing these probabilities are 2.6% and 0.0036% respectively. This means the hypothesis that they follow the same distribution as Chuffed0_OG can be rejected with over 99.99% certainty for MRCPSP and 97% certainty for Bin-packing. Therefore, it makes sense to conclude that the machine learning enhanced versions both outperform the unmodifed version on MRCPSP and Bin-packing. There is, however, no sufficient statistical evidence to conclude any significant difference between the results obtained with Chuffed1_Inc and Chuffed1_Ex for MRCPSP. The probability of obtaining less similar results with identical distributions is over 83%. However, for bin-packing, in addition to the difference between the machine learning enhanced version compared to Chuffed0_OG, it is also 95% certain that there is a statistically significant difference between Chuffed1_Ex and Chuffed1_Inc. This may indicate that bin-packing shares less learn-able concepts with other problem types than MRCPSP.

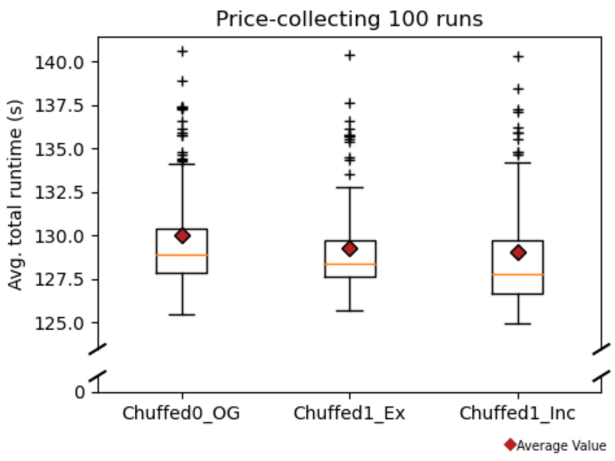
For price-collecting and fastfood there is insufficient evidence to conclude any significant differences between results of the different Chuffed versions. The most likely explanation is that, because of the limited data available for these problem types, none of the price-collecting or fastfood instances required considerable solving time. The average run-time per instance stated in Table 5 indicate that the machine learning integration works better for sizeable instances. Therefore it is most likely that lack of improve-



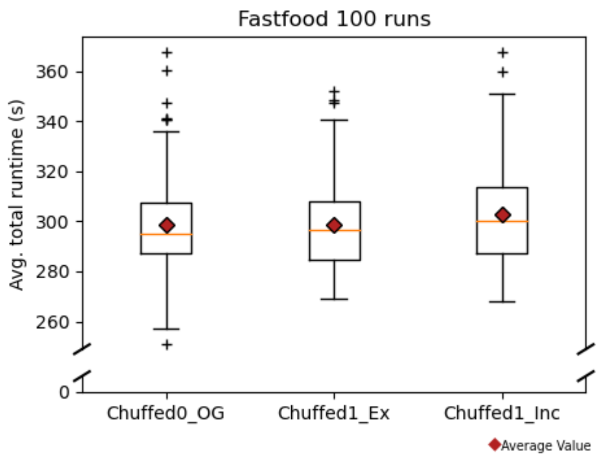
(a) MRCPSP



(b) Bin-packing



(c) Price-collecting



(d) Fastfood

Figure 2: Box-plots showing the total run-time of all test instances averaged over 100 runs for the four largest problem types.

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
mrcpsp10900	4.507	4.356	4.461
mrcpsp36	2.399	2.428	2.410
mrcpsp4425	311.565	296.139	302.595
mrcpsp4777	5274.736	5153.284	5155.367
mrcpsp4871	892.922	865.954	865.404
mrcpsp4960	32.713	32.241	32.099
mrcpsp7051	16.091	15.884	16.028
mrcpsp896	0.152	0.155	0.189
mrcpsp9880	0.236	0.241	0.240
mrcpsp9994	0.033	0.034	0.035
Total(s)	6535.354	6370.715	6378.829
Standard Deviation	282.493	273.983	271.103
Relative(%)	100.0%	97.5%	97.6%

Table 1: MRCPSP

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
2DLevelPacking238	171.700	151.000	152.580
2DLevelPacking23	1563.956	1499.611	1512.328
2DLevelPacking492	1221.866	1275.854	1237.965
2DPacking13	5065.462	5037.534	5025.021
2DPacking165	683.933	708.044	641.285
2DPacking168	2511.413	2430.075	2431.017
2DPacking62	58.744	57.180	57.587
Total(s)	11277.074	11159.298	11057.783
Standard Deviation	381.016	359.230	347.639
Relative(%)	100.0%	99.0%	98.1%

Table 2: Bin-packing

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
pc52	12.211	12.326	12.152
pc56	8.777	8.750	8.750
pc58	15.283	15.409	15.431
pc61	10.501	10.648	10.644
pc65	12.111	11.908	12.049
pc73	42.743	42.407	42.948
pc77	7.886	8.013	8.040
pc79	20.479	20.631	20.709
Total(s)	129.991	130.092	130.722
Standard Deviation	3.373	2.895	3.452
Relative(%)	100.0%	100.1%	100.6%

Table 3: Price-collecting

Instances	**Chuffed0_OG** Avg. runtime(s)	**Chuffed1_Ex** Avg. runtime(s)	**Chuffed1_Inc** Avg. runtime(s)
fastfood15	30.568	31.614	31.580
fastfood17	23.212	25.660	23.382
fastfood20	8.492	6.361	6.867
fastfood36	6.414	6.464	6.396
fastfood53	80.526	86.375	86.946
fastfood58	49.063	40.309	45.349
fastfood61	18.369	20.553	20.467
fastfood74	81.844	84.775	82.858
Total(s)	298.487	302.111	303.844
Standard Deviation	20.318	18.402	19.197
Relative(%)	100.0%	101.2%	101.8%

Table 4: Fastfood

Table 5: The average run-time per instance of the four largest problem types, averaged over 100 runs.

ment on price-collecting and fastfood is not because they are less similar to other problem types but because the tested instances were not sufficiently large.

6 Related work

This study continues on the work of Stuckey (Stuckey 2010) who proposed a hybrid SAT/CP solver based on Lazy-Clause-Generation (LCG). LCG combines finite domain propagation with the conflict learning ability of SAT. Because of this LCG solvers are able to use conflict driven heuristics such as VSIDS, which were originally developed for SAT solver Chaff (Moskewicz et al. 2001).

Multiple approaches have been proposed to combine machine learning with traditional SAT or CP solvers. For example, (Song et al. 2019) shows that machine learning can be used to automatically learn variable ordering heuristics for CSP solving. Moreover, Guerri et al. (Guerri and Milano 2004) has shown that machine learning is very capable of selecting solving strategies.

However, to the best of my knowledge, there has not been any research to combining machine learning with hybrid SAT/CP solvers, so far. This study mostly draws inspiration from the work by Selsam and Bjørner (Selsam and Bjørner 2019). This work describes how they use a technique called unsatisfiable core learning where to initialise the values of the Variable-Sate Independent Decaying Sum (VSIDS) heuristic for a selection of well-known SAT solvers. With this approach they manage to solve between 6 and 20% more instances within the same amount of time compared to the original solver.

7 Conclusion

As seen most evidently on the larger instances of the MRCPSP and bin-packing problem types the machine learning integration is possible to use this procedure to reduce the run-time of solving an CP instance. The total run-time for all the MRCPSP instances in the selected test-set, for which most data was available, was proven with over 99.99% certainty to be significantly less compared to the run-time of the unmodified version. On average a 2.5% increase in performance was achieved with a standard deviation of approximately 4%.

The impact of machine learning seemed to depend mostly on the difficulty of the instance. For instances that take less than a four seconds the overhead of initialising VSIDS may outweigh the benefit gained from it. Because of limited available data it was unfortunately not possible to determine a significantly different impact on performance across different problem types.

For the performance gained on the MRCPSP instances it

MRCPSP		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	4.163	$4.693e^{-5}$
Chuffed0_OG - Chuffed1_Inc	3.978	$9.761e^{-5}$
Chuffed1_Ex - Chuffed1_Inc	-0.209	0.834

Binpacking		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	2.238	0.026
Chuffed0_OG - Chuffed1_Inc	4.230	$3.577e^{-5}$
Chuffed1_Ex - Chuffed1_Inc	-2.020	0.045

Price-collecting		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	-0.226	0.821
Chuffed0_OG - Chuffed1_Inc	-1.506	0.134
Chuffed1_Ex - Chuffed1_Inc	-1.390	0.166

Fastfood		
Version Pair	T-Stat	P-Value
Chuffed0_OG - Chuffed1_Ex	-1.316	0.190
Chuffed0_OG - Chuffed1_Inc	-1.907	0.058
Chuffed1_Ex - Chuffed1_Inc	-0.648	0.518

Table 6: T-Test analysis

made no significant difference whether the Graph Convolutional Network model was trained exclusively on different problem types or also on MRCPSP instances. This suggests that the chosen model was able to generalise between the different problem types and still learn concepts which are useful to the solver. However, for the bin-packing problem type the performance was adversely affected by excluding any bin-packing instances from the training set. Therefore it may be concluded that it is possible to generalise between different problem types, but the performance may depend on the selected problem types.

All things considered, this study shows that it is possible to use machine learning approaches which are designed for solving SAT instances to improve LCG solving techniques. Specifically, this research has shown that it is possible to use unsatisfiable core learning, which originates from the work of Selsam and Bjørner (Selsam and Bjørner 2019), for improving the performance of the LCG solver Chuffed. With LCG approach dominating recent benchmarks it is interesting that the proposed approach is able to consistently achieve an improved performance on sizeable instances, even if only by a small margin.

Our work demonstrates the first, to our knowledge, successful application of machine learning to aid a CP–SAT optimisation solver. This paper opens the door to further research:

- For this approach to be used in practical context the possibilities for integrating the classification part directly into the solver should be investigated, this would require embedding the feature extraction part directly into the solver.
- In order to examine the effect across different problem types this experiment it may be valuable to repeat this study with more evenly distributed data.
- While the Graph Convolutional Network model proved sufficient for this study, it may be worth it for any future research which involves adapting a similar approach to evaluate the effect of using different machine learning architectures.
- It may be interesting to determine the effect of using the periodic refocussing technique described in the original paper (Selsam and Bjørner 2019) and examine if it would provide better performance for instances which take longer than a couple of hours to solve.
- The nature of this approach may reduce the time required to determine unsatisfiability on unsatisfiable instances, this may be interesting to further investigate.
- Another and possibly more successful way to improve LCG solvers would be to learn the best configuration of the solver's parameters given a certain instance.
- Machine learning may also be useful for predicting nogoods or their activity scores.

References

[Guerri and Milano 2004] Guerri, A., and Milano, M. 2004. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, volume 16, 475.

[Kim 2015] Kim, T. K. 2015. T test as a parametric statistic. *Korean journal of anesthesiology* 68(6):540.

[Kipf and Welling 2016] Kipf, T. N., and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *CoRR* abs/1609.02907.

[MiniZinc 2016] MiniZinc. 2016. The minizinc benchmark suite.

[Moskewicz et al. 2001] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, 530–535.

[Rossi, Van Beek, and Walsh 2006] Rossi, F.; Van Beek, P.; and Walsh, T. 2006. *Handbook of constraint programming*. Elsevier.

[Selsam and Bjørner 2019] Selsam, D., and Bjørner, N. 2019. Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR* abs/1903.04671.

[Song et al. 2019] Song, W.; Cao, Z.; Zhang, J.; and Lim, A. 2019. Learning variable ordering heuristics for solving constraint satisfaction problems.

[Stuckey 2010] Stuckey, P. J. 2010. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6140 LNCS(June):5–9.

Bibliography

[1] Or-tools | google developers, 2016. URL `https://developers.google.com/optimization`.

[2] H-M. Adorf and Mark D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 917–924. IEEE, 1990.

[3] Alexander Bockmayr and John N. Hooker. Constraint programming. *Handbooks in Operations Research and Management Science*, 12:559–600, 2005.

[4] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys (CSUR)*, 38(4):12–es, 2006.

[5] Geoffrey Chu and Peter J Stuckey. Inter-instance nogood learning in constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 238–247. Springer, 2012.

[6] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2018.

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

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

[9] David Devlin and Barry O'Sullivan. Satisfiability as a Classification Problem. *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*, 2008.

[10] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[11] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Principles and Practice of Constraint Programming - CP 2009 - 15th International Conference, CP 2009, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 352–366. Springer, November 2009. ISBN 3642042430. doi: 10.1007/978-3-642-04244-7_29. 15th International Conference on Principles and Practice of Constraint Programming, CP 2009 ; Conference date: 20-09-2009 Through 24-09-2009.

[12] Ian P. Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.

[13] Ian P. Gent, Chris Jefferson, Lars Kotthoff, Ian Miguel, Neil C.A. Moore, Peter Nightingale, and Karen Petrie. Learning when to use lazy learning in constraint solving. *Frontiers in Artificial Intelligence and Applications*, 215:873–878, 2010. ISSN 09226389. doi: 10.3233/978-1-60750-606-5-873.

[14] Matthew L. Ginsberg. Dynamic backtracking. *journal of artificial intelligence research*, 1:25–46, 1993.

[15] Carla P. Gomes and David B. Shmoys. The promise of lp to boost csp techniques for combinatorial problems. In *Proc., Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02), Le Croisic, France*, pages 25–27, 2002.

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

[17] Alessio Guerri and Michela Milano. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, volume 16, page 475, 2004.

[18] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 169–177, 2000.

[19] Holger H. Hoos and Thomas Stützle. Satlib–the satisfiability library. *Web site at: http://www. satlib. org*, 1998.

[20] Holger H. Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[21] Holger H. Hoos and Edward Tsang. Local search methods. In *Foundations of Artificial Intelligence*, volume 2, pages 135–167. Elsevier, 2006.

[22] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Aaai*, volume 7, pages 1152–1157, 2007.

[23] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In *AAAI*, volume 5, pages 390–396, 2005.

[24] Tae Kyun Kim. T test as a parametric statistic. *Korean journal of anesthesiology*, 68(6):540, 2015.

[25] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL `http://arxiv.org/abs/1609.02907`.

[26] Jia Hui (Jimmy) Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. *CoRR*, abs/1506.08905, 2015. URL `http://arxiv.org/abs/1506.08905`.

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

[28] Zbigniew Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. *Evolutionary programming*, 4:135–155, 1995.

[29] MiniZinc. The minizinc benchmark suite, 2016. URL `https://github.com/MiniZinc/minizinc-benchmarks`.

[30] MiniZinc. Challenge2017, 2017. URL `https://www.minizinc.org/challenge2017/results2017.html`.

[31] MiniZinc. Challenge2019, 2019. URL `https://www.minizinc.org/challenge2019/results2019.html`.

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

[33] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[34] Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Springer, 2015.

[35] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[36] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[37] Nikos Samaras and Konstantinos Stergiou. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *CoRR*, abs/1109.5714, 2011. URL `http://arxiv.org/abs/1109.5714`.

[38] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www. gecode. org*, pages 11–13, 2006.

[39] Daniel Selsam and Nikolaj Bjørner. Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 2019. URL `http://arxiv.org/abs/1903.04671`.

[40] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[41] Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems, 2019.

[42] Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6140 LNCS(June):5–9, 2010. ISSN 03029743. doi: 10.1007/978-3-642-13520-0_3.

[43] Peter J.M. Van Laarhoven and Emile H.L. Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[44] Toby Walsh. Sat v csp. In *International Conference on Principles and Practice of Constraint Programming*, pages 441–456. Springer, 2000.

[45] Haoze Wu. Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 787–788, 2017.

[46] Hong Xu, Sven Koenig, and T. K. Satish Kumar. Towards effective deep learning for constraint satisfaction problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11008 LNCS:588–597, 2018. ISSN 16113349. doi: 10.1007/978-3-319-98334-9_38.

[47] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[48] Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Predicting satisfiability at the phase transition. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.