Machine Learning for SAT Variable Assignment

an investigation into the efficacy of binary classification and literal-level statistics for identifying valid assignments.

Ruth Bergin

A thesis presented for the degree of
BSc Data Science & Analytics

School of Computer Science | School of Mathematical Science
University College Cork
Ireland
March 2023

Machine Learning for SAT Variable Assignment

an investigation into the efficacy of binary classification and literal-level statistics for identifying valid assignments.

Ruth Bergin

Abstract

The Boolean Satisfiability Problem (SAT) has been seminal for computer science for over half a century. Preprocessing algorithms have proven crucial to developing efficient solvers [17]. Recently, great strides have been made in the application of machine and deep learning techniques to the SAT problem. In particular, binary classification of SAT instances based on their statistical features has shown to have accuracy in excess of 99% [8].

This dissertation presents a preprocessing technique which uses machine learning techniques to guide variable assignment. The pre-solver iteratively assigns variables based on a literal-level heuristic and a SAT/UNSAT random forest classifier. At each step, the pre-solver attains two reduced versions of the instance through probing. The probability of satisfiability is predicted for both using a random forest classifier. The assignment which resulted in a reduced instance with greater probability of satisfiability is chosen. The probing step is then iteratively repeated on the reduced instance.

The probability of satisfiability doubles as a stopping criterion - if it doesn't exceed a user-defined cutoff point, the process terminates and the reduced instance is returned.

This method has been tested on a number of small instances, and can find a complete, satisfying assignment in excess of 98% of instances. When passed to the pre-solver for ten assignments, the reduced instances took on average 52% less time to solve than the original instances. This preprocessing algorithm is not yet competitive in terms of time. However, it shows great promise as a new pre-solving approach, or, when reinforced with backtracking functionality, a full solver.

Declaration

Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly at- tributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: .....
Dated: .....

Acknowledgements

I want to thank my supervisor, Professor Gregory Provan, whose encouragement and insightful suggestions kept me focused throughout this project. The contributions of Dr Andrea Visentin and Marco Dalla made to this project have been invaluable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In spite of decades of study, the Boolean Satisfiability Problem (SAT) remains one of the most popular areas of research in computer science, with prestigious journals and annual conferences dedicated to its exploration. This is due in large part to its use as a tool to enable the solution of fiscally significant real-world problems such as scheduling, circuit finding, route planning, along with more technical applications like deep learning and theorem proving.

As SAT is NP-complete, a plethora of diverse algorithms - collectively known as "solvers" - exist to tackle instances. Solvers can have a range of objectives, including but not limited to:

- satisfiable/unsatisfiable (SAT/UNSAT) binary classification, i.e. whether a complete assignment exists for which the instance evaluates to true.
- identifying complete assignments which satisfy the instance.
- calculating the model count of an instance, i.e. how many complete, satisfying assignments exist.

Even the most performant SAT solvers can be incredibly computationally expensive - both in space and time. As such, a phase of the process called preprocessing (or pre-solving) has become vital to solvers. Completed after encoding but before the main solving algorithm, the purpose of pre-solvers is to simplify instances before they are passed to the solver in order to significantly decrease the overall runtime and/or space complexity.

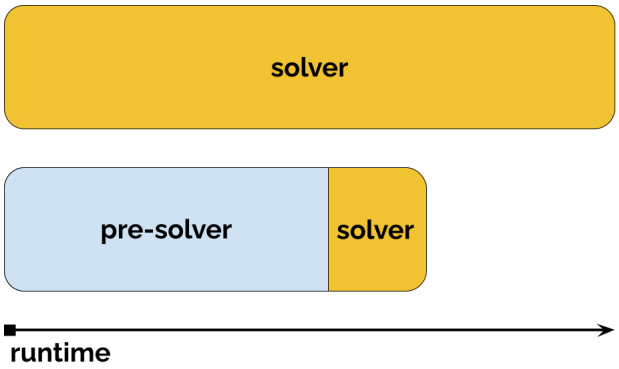

Figure 1.1: Visualisation of effect of preprocessing on runtime

While algorithm efficiency is always key, it is even more crucial in the case of SAT solving. When solvers run alone, the space complexity of the algorithm can

render finding solutions for large instances infeasible. Pre-solvers have shown to allow solvers to complete previously unsolveable instances - one investigation into the effectiveness of preprocessing [17] found that with appropriate application of pre-processing techniques up to 27% more instances could be solved than a solver working alone.

Even though pre-solvers are usually lightweight, they often fully solve a proportion of instances before they terminate [19], which saves on the often heavy encoding cost of full solvers.

Since pre-solvers can reduce the run-time of solvers by up to 33x [16], solve simple instances, and even make gratuitously large instances feasible to solve, most of the top-performing solvers today employ them in some form.

More and more, the applications of machine and deep learning to the field of SAT are being explored [13], [1]. In particular, SAT can be treated as a binary classification task, and learning algorithms trained either on statistical features extracted at the instance level or on graphical representations of the CNF instances have proven able to successfully predict satisfiability with great accuracy [8], [6]. No matter how accurate it becomes, binary classification won't be able to provide a complete assignment or a fail-safe guarantee of satisfiability. However, it may be able to guide the branching decisions of a preprocessing or solving algorithm.

This dissertation presents a proof-of-concept pre-solver, which uses a SAT binary classification technique in a variable elimination algorithm. The pre-solver iteratively assigns variables using a combination of heuristic and machine learning to guide its path. Since it probes down a single path only and may make an assignment that violates satisfiability at any stage, a user-defined level of confidence is used to control when the process terminates. When this confidence level is breached - i.e. the classifier is 85% confident the instance is SAT when the cutoff is 90% - the reduced instance at that point is returned. If this confidence level is not breached, a complete assignment will be found.

The variable to assign is chosen by a literal-level statistical heuristic, which aims to identify the most influential variables - i.e. the variables for which one assignment will maintain the instance's satisfiability, while the other assignment will render the instance UNSAT. The search tree of the instance assignment space is probed at depth one in two directions (True/False assignment of the variable). However, unit clauses are propagated at each "manual" assignment, so the overall reduction of the instance size isn't restricted to one literal per step.

A random forest classifier trained on SATzilla statistical features (including number of clauses and variables, balance statistics, and Horn formula statistics) provides a SAT/UNSAT probability for the resulting instance of both probes. Whichever instance has higher SAT probability is the branch which is chosen as next in the path - see 1.2.

The assignment of the variable is decided via prediction of the probability of satisfiability of the reduced formula. This satisfiability probability is also used to decide whether or not the process should continue or be terminated (returning a reduced instance).

The pre-solver is compared to a benchmark of iterative random assignment to assess its accuracy. Its effect on solver runtime was assessed using X solver, with the benchmark being the time taken for the solver to complete without any preprocessing.

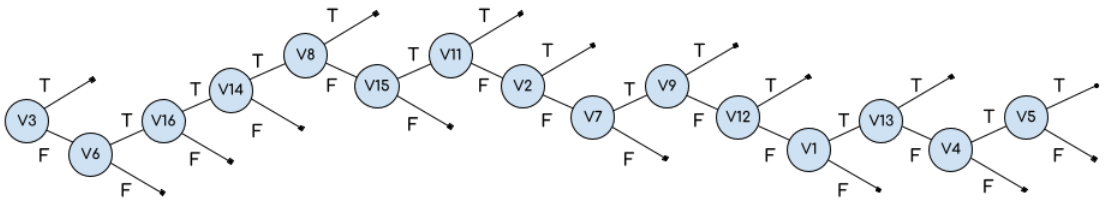

Figure 1.2: Path of variable assignment. The search tree is probed at depth 1 in two directions, but only one branch is pursued.

The pre-solver has been tested on a set of small instances, and has been shown to solve the instances in 90% of cases. In the cases where the instances were reduced but not fully solved, satisfiability was maintained 100% of the time.

In general, this pre-solver performed Yx better than the random benchmark at assigning variables while maintaining satisfiability. Further, due to the carefully chosen branching heuristic, instances with 10 assignments made by the pre-solver have shown to improve runtime by an average of 44%.

As of yet, this pre-solver does not provide an improvement on runtime, and its accuracy needs improvement. However, it shows strong potential as a way to incorporate machine learning into the preprocessing step of SAT solving.

In chapter 2, existing literature surrounding the topic is reviewed, with a focus on pre-solvers and the use of machine learning in the area of SAT. Chapter 3 provides the required context regarding the Boolean satisfiability problem and machine learning. The design and implementation of the preprocessing algorithm is explained in chapter 4. Experimental methods and results are described in chapters 5 and 6 respectively, while chapter 7 discusses conclusions and future work.

Chapter 2

Literature Review

2.1 Pre-Solvers

Preprocessing techniques in SAT, often referred to as pre-solvers, exist to reduce the runtime and memory requirements of SAT solvers [see fig 1.1]. They have been shown in some cases to enable a solution to be found for instances which are otherwise too big to compute [17].

The Handbook of Satisfiability [5] devotes its ninth chapter to the discussion of pre-solvers. Biere et al. posit that preprocessing not only decreases the runtime of state-of-the-art solvers, but also lifts the burden of careful encoding of instances from users who may not be expert. The handbook describes the main methods of pre-solving, summarised below.

The preprocessing methods are categorised into two broad groups - classical and resolution-based. Classical methods deal directly with the SAT instnace, and are usually (relatively) simple, both conceptually and computationally. Resolution-based methods require the resolvents of variables to be computed in order to infer actions to be taken. These algorithms are conceptually and computationally more demanding, but have been shown to be extremely effective [17].

Classical

Classical methods apply logical inferences without rewriting the SAT instance. **Unit propagation:** a unit clause is a clause containing only one literal. It follows that this literal must be satisfied. The assignment of all literals in unit clauses (iteratively, until no such literals remain) is known as unit propagation.
Pure literal elimination: a literal is considered "pure" if its complementary literal doesn't appear at all in the formula. Since a pure literal's assignment will never compromise satisfiability, they can always be satisfied.
Failed literal: If unit propagation following the assignment of l derives a conflict in F, l is said to be a failed literal. The satisfaction of $\neg l$ or the addition of a unit clause $\{\neg l\}$ to F are two ways to address failed literals.
Subsumption: a clause C_1 subsumes another clause C_2 if the set of literals in C_1 is a subset of those in C_2. In this case, C_1 is clearly redundant and does not provide any further benefit to the formula, so can be discarded.
Connected components: the encoding of a CNF instance into a variable hy-

pergraph where hyperedges correspond to clauses often yields several disconnected sub-graphs rather than one fully connected hypergraph. These sub-graphs are totally independent to each other in terms of satisfiability and can be solved separately as smaller instances.

Resolution-Based

Resolution-based methods take action based on new clauses that are inferred by resolving clauses on variables. **Bounded Variable Elimination (BVE):** based on the concept of clause distribution, this method computes all the resolvents on a variable, adds the new clauses to the formula, and removes all the original clauses in which the variable appears. It is bounded in that this is only done if the total number of clauses in the formula doesn't increase.
Techniques based on implication graphs: the representation of binary clauses in a (binary) implication graph yields strongly connected components, i.e. a set of literals whose assignments in every satisfying solution are totally dependent on each other. These sets are known as equivalent literals and can be replaced with a single representative literal.
Hyper binary resolution: the iterative application of resolution to binary clauses and propagation of unit clauses.
Advanced probing techniques: these methods, being similar to the one described in this paper, will be discussed further below.

The handbook also discusses blocked clauses, which are clauses that contain a literal for whom all its resolvents are tautologies. There are several generalisations of blocked clauses, including resolution asymmetric tautologies and covered clauses. The chapter then moves past preprocessing techniques for CNF into structure-based preprocessing, this is however deemed irrelevant to the issue at hand and will not be discussed further.

The aforementioned analysis of the effectiveness of preprocessing [17] found that the most performant pre-solver was a combination of bounded variable elimination, blocked clause elimination, and unhiding (binary implication graph resolution).

Bounded variable elimination, being a variable elimination technique, is quite similar to the methods pursued in this paper. Its origins come from a method called Variable Elimination Resolution (VER) [7]. This algorithm created resolutions between clauses, and uses resulting tautologies to eliminate variables. However, its exponential space complexity made the approach infeasible. One approach, Non-increasing Variable Elimination Resolution (NiVER) [16] effectively addressed this problem by only applying VER in the cases where there wasn't a net increase in the number of literals. It was shown to decrease the overall solving time of large instances in most cases, yielding an improvement of up to 33x in best case. Further, some instances that were too large to be solved when passed directly to the then SOA solvers were solveable after being reduced by NiVER.

NiVER has since been incorporated into many state-of-the-art solvers. In 2005 it was extended by SATElite [9]. This pre-solver enhanced NiVER with subsumption, self-subsuming resolution, and variable elimination by substitution. SATElite went on to be the dominant pre-solver for years.

Also similar to the approach outlined in this paper are probing based techniques. In ProbIt [14], both variable and clause probing is employed to discover implied

assignments, i.e. assignments which are required for satisfiability in all branches of the search tree. This method was shown to increase robustness of the solver, but didn't significantly improve run-time.

Another method called distillation employs probing methods [11] which interleave assignment with propagation, as will be seen later in this paper.

Several sources [5], [11] stress that naive simplification of instances could harm the performance of solvers rather than help it. Variables or clauses which are technically redundant may in fact help the solver to reason out a solution more quickly. Since the method described in this paper does not focus on the elimination of redundant clauses and variables and instead on the assignment of the most influential ones, we hope that this will not be an issue.

2.2 Machine Learning in SAT

The application of machine learning to the SAT problem has in recent years had increasing success. Before diving into the methods used, however, it would be prudent to discuss the statistical features used to train the models.

Subbarayan and Pradhan [16] introduced 84 statistical features to describe CNF instances - including problem size features, graph features, balance features, and more. These became the basis of SATzilla [19], a seminal portfolio algorithm for SAT solving. Classifiers trained on SATzilla statistical features have shown to be successful not only in binary SAT classification [19], [2] but also instance-type classification [3]. The current dominant solver, KISSAT-MAB-HyWALK [10] uses a hybrid walk approach. In order to decide which of its six walking strategies to use, a decision tree trained on instance-level statistics is employed.

Devlin and O'Sullivan [8] conducted an investigation into the accuracy of standard classifiers on SAT/UNSAT classification. They used SATzilla statistical features to train a random forest, a decision tree, a multi-layer perceptron, 1-nearest-neighbour, and a naive Bayes model. Results showed that these classifiers could achieve over 99% accuracy on hard 3-sat instances, and over 90% accuracy on large industry standard benchmarks.

Building on this, Dalla, Visentin, and O'Sullivan [6] used convolutional autoencoders to replace typical feature extraction. Even on as few as 8 latent dimensions, the classifiers trained on the autoencoders far outperformed the classifier trained on SATzilla features.

Of particular relevance to this paper, Wu [18] used SAT/UNSAT binary classification to make partial variable assignments. They used logistic regression to predict the satisfiability of instances, and followed a Monte-Carlo approach to determine the preferred initial value for a fraction of the variables. This method was successful in reducing the runtime of MiniSAT (with a 23% decrease), but did not outperform the default settings overall since the preprocessing algorithm took some time.

Chapter 3

Preliminaries

3.1 Boolean Satisfiability Problem

3.1.1 Propositional Logic

A Boolean variable is one which has only two possible values: {True,False} - or alternatively, {1,0}. These atomic variables can be combined using the operators $[\neg, \wedge, \vee]$ along with punctuation like parentheses to compose larger statements, which themselves can evaluate to either True or False.

3.1.2 Conjunctive Normal Form

Each variable v in a boolean statement has a corresponding pair of **literals**, l and $\bar{l}$.

$$l \iff v \ ; \ \bar{l} \iff \neg v \tag{3.1}$$

Only one literal for each variable can ever be satisfied. As such, $\bar{l}$ is known as the **negation** of l, and vice versa. Given a set of variables X, let a **clause** be the conjunction of a subset of the literals for which X is the basis. Conjunctive Normal Form (CNF) is a boolean statement comprised of a disjunction of clauses. A simple example of a statement in CNF is

$$(x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_4) \wedge (\neg x_3) \tag{3.2}$$

It follows that the statement evaluates to True if and only if **at least one literal from each clause evaluates to True**. An instance is referred to as **SAT** if a complete assignment of variables exists for which the whole statement evaluates to True.

3.1.3 SAT Solvers

The term "SAT solvers" refers to the class of algorithms which deal with SAT instances - including classifying instances as SAT/UNSAT, counting (or estimating) the number of satisfying solutions which exist for an instance, and identifying such solutions.

Variable Assignment Methods

Naturally, there is a strong element of guesswork involved when attempting to assign variables. A **branching heuristic** is the metric or statistic that is used to choose the next variable in the instance to assign.

A unit clause is a clause with only one literal. It is (of course) imperative for that literal to be satisfied, so **unit clause propagation** is a step in many variable assignment solvers. Another common step is the assignment of **pure literals**, i.e. all literals in a statement for which their negations do not appear.

3.2 Machine Learning

3.2.1 Supervised Learning

Supervised learning is a class of algorithms which take as input a set of p input variables (predictors), and output a single value (label), which is the variable of interest. If the label is categorical, this endeavour is known as classification. If numerical, it's called regression. Examples of labels of interest include house prices, tumor diagnosis, and CNF SAT/UNSAT. Classification is done in three broad ways:

Method	Example Algorithms
Probability	Naive Bayes, Discriminant Analysis
Distance	K-Nearest Neighbours, Support Vector Machines
Rule-Based	Decision Trees

Table 3.1: High-level overview of supervised learning methods.

3.2.2 Random Forest Classification

A decision tree is a form of rule-based supervised learning. In the tree, each internal node is a conditional statements and the leaves are the labels. The objective of the tree is to minimise impurity. The Gini index and cross-entropy are two metrics which can be used to form the internal leaves.

Decision trees are easy to train but have low accuracy alone. One popular solution to this is the use of ensemble learning, which follows the "majority rules" concept. In practice, this means that many trees (trained on different data, or using different features, or both) are given the same sample and the predicted class is the one with the highest vote.

Random forest is a method of ensemble learning by which a number of trees are trained, each on a random subset of the data and using a random subset of all of the available features.

3.2.3 Model Performance

In general, classification model performance can be captured simply and effectively in a single metric: how often did the model guess correctly on the training data? In binary classification problems, i.e. when there are two label classes, it is feasible to be more specific. **Sensitivity** is the true positive rate, the proportion of

samples which are correctly assigned as True/Yes/Pass/Present etc. **Specificity** is the true negative rate, the proportion of samples which are correctly assigned as False/No/Fail/Not present etc. This is summarised in 3.2.

	Positive [actual]	**Negative [actual]**
Positive [predicted]	Sensitivity	Type I
Negative [predicted]	Type II	Specificity

Table 3.2: Binary classification performance terminology explained.

Chapter 4

Preprocessing Algorithm

4.1 Design

Iterative variable assignment requires three decisions at each step:

1. which variable should we assign next?
2. which truth value should we assign to it?
3. should we stop here?

4.1.1 Variable Choice

The next variable to assign is decided using a variable-level statistical heuristic which attempts to identify the most influential variable (see section 4.3). In this case, influential is taken to mean a variable which when assigned one way maintains the instance's satisfiability, and when assigned the other way renders the reduced instance UNSAT. Influence is estimated using metrics which measure variable importance (e.g. how many times the variable appears) and variable purity (e.g. the ratio of appearances of the variable's complementary literals).

4.1.2 Truth Value Decision

The truth value is decided using predictions from the random forest classifier. The instance is probed by assigning the chosen variable both ways, and the two resulting instances (after unit propagation) are stored as *shadow* instances. The SATzilla statistical features of both instances are extracted, and used by the random forest to predict the probability of satisfiability. Whichever truth assignment's probing resulted in the instance with the higher probability of satisfiability will be the truth value used in the assignment.

4.1.3 Stopping Criterion

To decide when to stop, at each step the probability of satisfiability found in the previous step is compared to a user-defined cut-off point. If the probability is below the cut-off, the process is terminated and the reduced instance is returned. For example, if the user-defined cut-off point is 0.9, if the SAT probability found by the random forest drops to 85%, the algorithm will immediately terminate.

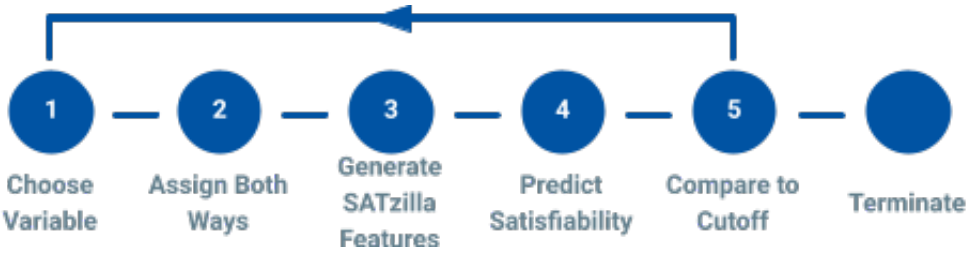

Figure 4.1: Algorithm pipeline

A summary of the solving pipeline is shown below:

algorithm TermReduce
Input: CNF instance ϕ; probability cutoff C
Output: ϕ', reduced or solved instance of ϕ;
function TermReduce(ϕ, C)

while $satProb \geq C$ **do**
  if not first pass **then**
    $\phi \leftarrow assign(\phi, V, assignment)$
  end if
  if $\phi\, solved$ **then**
    return $\emptyset$;
  end if
  $V \leftarrow getMostPureLiteral(\phi)$
  $branchTrue \leftarrow assign(\phi, V, True)$
  $branchFalse \leftarrow assign(\phi, V, False)$
  $assignment, satProb \leftarrow satProbMax(branchTrue, branchFalse)$
end while
return ϕ;

4.2 Binary Classifier

4.2.1 Dataset

The random forest is trained on hand-crafted instances from the benchmark CBS dataset from SATlib [12]. The dataset comprises 1000 random-3-SAT CNFs with controlled backbone size, each with 100 variables and 403 clauses. This dataset was chosen as it is a well known benchmark, all of its instances are SAT, the instance size was ideal for this purpose, and the instances are of reasonable difficulty to solve.

4.2.2 Feature Selection

The features used in the random forest classifier are the SATzilla statistical features [19].

- Problem size features:
  - Number of clauses.
  - Number of variables.

  - Ratio of clauses to variables.
  - Variable Clause Graph features:
  - Variable nodes degrees statistics.
  - Clause nodes degrees statistics.
- Variable Graph features: Node degree statistics.
- Balance features:
  - Bias of positive or negative literals in each clause statistics.
  - Bias of positive or negative occurrences of each variable statistics.
  - Fraction of binary and ternary clauses.
- Proximity to Horn formula:
  - Fraction of Horn clauses.
  - Number of occurrences in a horn clause for each variable statistics.
- DPLL Probing features:
  - Number of unit propagations computed at depths 1, 4, 16, 64 and 256.
  - Search space size estimate: mean depth to contradiction.
  - Estimate of the log of number of nodes.

The final category of features, DPLL probing, proved to be divisive. On the one hand, the calculation of the probing features takes a fixed time of 2 seconds (after which it is cut off). As features are being computed twice (once for each shadow CNF) per step, this slows the algorithm considerably. However, as shown in [15], DPLL probing features have a significant impact on the random forest's accuracy - and in turn, the whole algorithm's performance.

While DPLL probing is an effective tool in binary SAT classification, there are issues aside from runtime. With its absolute cut-off time, the same algorithm run on different machines result in different features for the same instance. If the features for the training set for the random forest are computed on one machine and the pre-solver is run on a different machine, the DPLL features will have a severe negative impact on the classifier's accuracy. This element must be considered when deciding whether to include DPLL features or not.

4.2.3 Pre-Processing

80 instances of the CBS dataset were probed in order to generate a labelled dataset with a variety of sizes and balanced distribution between SAT and UNSAT. The probing is similar to the previously described pipeline:

- A variable is selected at random,
- Two reduced *shadow* CNF instances are generated,
- SATzilla features are extracted from both and stored as training inputs, and
- The output [SAT/UNSAT] of the glucose4 SAT solver is stored as the target class.

The branching can be directed by the solutions computed by the SAT solver to obtain a labelled dataset with the desired SAT/UNSAT balance.

SAT/UNSAT Balance

The tradeoff of sensitivity to specificity has a significant influence on the performance of the pre-solver. A high specificity is crucial to ensure the process terminates as early as possible after a mistake. However, if sensitivity is low, the process will terminate unnecessarily early.

4.3 Variable selection

The approach to variable selection focuses on purity, as inspired by the Davis-Putnam Pure Literal Rule. While the strict purity of a literal is a binary characteristic, it is possible to implement a new measure that evaluates how close a literal is to purity and its importance to the formula resolution. The goal is to select a variable that, if assigned incorrectly, makes the resulting formula unsatisfiable. Among the unfixed variables, we select the one with the higher score computed with the following formula:

$$score(v) = |f(l) - f(\neg l)| \tag{4.1}$$

where l is the literal in which the variable v appears positively, and $\neg l$ is the negated literal. The absolute difference is chosen to maintain the absolute importance of the variable in the heuristic - when the ratio of $f(l)$ to $f(\neg l)$ is used, the components of the heuristic related to variable importance cancel out, leaving only an estimation of literal purity.

Function f is a measure of the literal relevance and purity:

$$f(l) = \frac{number\_appearances(l) * cov(l)}{(avg\_clause\_size(l) + min\_clause\_size(l)) * \sqrt{cov(\neg l)}} \tag{4.2}$$

The inclusion of each element of this equation is explained below.

Number of appearances of l

A direct comparison of this statistic to truly pure literals exists: the ratio of appearances of a pure literal to its negation is ∞. As a result, the number of appearances is included as the basis for the heuristic.

Average clause size of l

This statistic is a measure of literal importance - if a literal appears relatively few times but always in small clauses, it may be more important than a literal which appears many times but in large clauses. Mean was chosen as the measure of central tendency due to its inclusion of outliers.

Minimum clause size of l

This statistic may seem redundant considering the inclusion of the average clause size. However, mean clause size doesn't capture the spread of data. The inclusion of this statistic attempts to penalise literals whose minimum clause size is not much

less than mean clause size, improving the score for literals which may appear in large clauses but also appear in important ones as well.

Covariance of l and $\neg l$

Another measure of how important a literal is to instance satisfaction is its covariance with the negations of crucial literals. For example, if a literal l shares a binary clause with literal k, but $\neg k$ is in a unit clause, then it is imperative to assign $l = True$ to prevent the contradiction of a unit clause. This concept can be generalised to clauses of any size by distributing the weight of a literal to the covariant literals of its negation. To represent this, let the K be the set of literals which share a clause with l.

$$cov(l) = \sum_{k \in K} \frac{f(\neg k)}{number\_appearances(k)} \tag{4.3}$$

The heuristic of each literal is directly proportional to its covariance. However, the heuristic is also inversely proportional to the covariance of its negation. This is so because as explained at the beginning of this section, the difference of the literal heuristics is taken as the variable score, rather than the ratio between them. With the covariance of the literal's negation included as a factor in the literal heuristic, even if the literal is totally dominant over its negation, the covariance of the negation is still taken into account.

4.4 Implementation

Python was chosen for speed and ease of implementation and maintenance of the project. The python library SATfeatPy [15] was used to compute the instance statistical features.

4.4.1 Package Structure

A custom CNF SAT Instance object was created in order to facilitate calculation and storage of statistics on a variable level - something that's generally avoided in the interest of memory in typical SAT instance objects. While it automatically propagates units, it otherwise has no autonomy regarding variable assignment.

The full pipeline leverages this custom CNF object along with the random forest classifier to execute the algorithm described in 4.1.

4.4.2 Technical Challenges

With so many custom objects to keep track of, it was easy to introduce bugs into the code.

While the benchmark CBS dataset was used to train the preprocessor, other datasets were used for verification purposes. These datasets had CNF files that didn't quite match the syntax of the CBS dataset - e.g. clauses were separated by "\n 0 \n" instead of "0\n". Modifying the preprocessor to be flexible in the separators used for clauses was, annoyingly, non-trivial.

Another challenge was the depth of recursion possible with unit propagation. Originally, I had programmed **propagate_units()** to run every time a literal was assigned. However, this caused the maximum recursion depth to be exceeded often. I had to change the way I approached unit propagation, and only called it after manual assignments of literals.

Since negated variables are denoted with a coefficient of -1, it sometimes made sense to represent a variable's assignment to False as -1. However, other times it was only appropriate to use the boolean variable directly. The entire pipeline was custom built, so when I introduced a type error of checking for (variable, -1) in a list of tuples of the form (variable, bool), it took some time to catch this error.

Perhaps one of the silliest mistakes I made was forgetting to abort the assignment if there was a conflict between unit clauses. Leaving out one simple check (if there is a unit clause containing the complementary literal to the one being assigned) rendered the entire algorithm completely incorrect.

Chapter 5

Experimental Methods

5.1 Datasets

5.1.1 CBS

This is a random sample of 50 instances from the CBS benchmark dataset. These instances were excluded while generating the training set for the random forest classifier.

5.1.2 Verification

A new dataset, VERIFICATION, is introduced, with both SAT and UNSAT instances. Classes of problems like clique, graph colouring, dominant set, and others are represented. The number of literals in the original instances vary from 24 to 222. The random forest is re-trained as before on a subset of these instances.

A random sample of 50 SAT and 50 UNSAT instances is used for experimentation. As above, these instances were excluded while generating the training set for the random forest classifier.

5.2 Control Configuration

The configuration of the pre-solver, unless otherwise specified, is detailed below.

	Reference	Description
Feature Set	full	Include SATzilla base & DPLL probing features.
Training Set Class Balance	50/50	Equal distribution of SAT/UNSAT instances.
SAT Probability Cut-Off	0.5	Process terminates when SAT probability < 0.5.
Heuristic	complete	The heuristic is as described in 4.3.

Table 5.1: Default Configurations

5.3 Metrics

5.3.1 Assignment Validity & Completeness

This measure tracks the overall quality and accuracy of the pre-solver. The reduced instances returned from the pre-solver will be categorised as follows:

SUCCESS	complete assignment found
PARTIAL SUCCESS	partial assignment found & satisfiability maintained
FAILURE	satisfiability not maintained

Table 5.2: Levels of success of pre-solver

The proportion of reduced instances in each category is compared.

5.3.2 Effect on Pre-Solver Runtime

This metric measures the empricial effectiveness of the algorithm as a pre-solver.

$$1 - \frac{\text{reduced instance runtime}}{\text{original instance runtime}} \tag{5.1}$$

Reduced instances of the pre-solver after a fixed number (10) of assignments are saved. The glucose solver [4] is run on both the original and reduced instances. The percentage reduction in solver runtime between the reduced instances and the original instances is calculated.

5.3.3 Number of Manual Assignments Required for Complete Solution

This metrics measures, in the cases where instances were solved completely, how well-chosen the variables are. Manual assignment refers to variables that were assigned outside of unit propagation. The more crucial variables are assigned at the beginning, the fewer assignments (excluding propagation) will be necessary to find a complete assignment.

5.4 General Performance

The pre-solver will be run on default configurations and the three metrics from above will be measured. This will be repeated for the CBS dataset and for the verification dataset. A further experiment will be carried out on the verification dataset alone:

5.4.1 Recognition of UNSAT instances

This experiment will be carried out on 50 UNSAT instances from the verification dataset. As there are no UNSAT instances in the CBS dataset, it will be excluded from this experiment.

Hypothesis

> **The number of assignments made before termination when the instance is originally UNSAT will be less than 3.**

Assumption: If the classifier has specificity of even 90%, the probability of the classifier making an error n consecutive times will be $(0.1)^n$. In almost all instances, the classifier should recognise an UNSAT instance and terminate early within three assignments.

This assumption is by no means correct, as there may be an UNSAT instance whose feature characteristics aren't reflected in the training set, and whose feature characteristics are unaffected by assignment.

Experiment

Run the preprocessor on 50 UNSAT instances from VERIFICATION.

Metric

The number of assignments before termination.

5.5 Machine Learning Experiments

The CBS dataset alone is used for the experiments in this section.

5.5.1 Class Balance

Hypothesis

> **A training dataset skewed in favour of UNSAT instances will have better performance than a balanced training set.**

Considering that the probability of satisfiability acts as stopping criterion, a classifier skewed in favour of UNSAT classification will be more likely to exit early. An imbalanced dataset will cause fewer instances to be fully solved, but will also lower the proportion of failures. It is far more important to maintain satisfiability than to solve a higher proportion of instances, so a skewed training set would have better performance.

Experiment

Run the preprocessor three times, each with a training set of size 3000. The training sets will differ by class balance:

- **Balanced:** there is a 50/50 balance of SAT/UNSAT instances in the training set.
- **Skewed:** the training set is skewed 75/25 in favour of UNSAT instances.
- **Severely skewed:** the training set is skewed 90/10 in favour of UNSAT instances.

Metric

The performance of each experiment will be measured using metric 5.3.1.

5.5.2 Feature Selection

Hypothesis

> **The preprocessor run on a training dataset which includes DPLL probing will perform better than one with only SATzilla basic statistical features as training set.**

The probability of satisfiability predicted by the random forest classifier is used to guide assignments. It has been shown [15] that the inclusion of DPLL probing statistics as predictor variables greatly increases classifier accuracy. Therefore, the preprocessor which includes DPLL will have much better performance than one configured to exclude it.

Experiment

Run the preprocessor twice.

- **Base:** Only SATzilla base statistical features are used in the classifier.
- **DPLL:** Both SATzilla base and DPLL probing features are used to predict satisfiability.

Metric

The performance of each experiment will be measured using metric 5.3.1.

5.6 Algorithm Configuration Experiments

5.6.1 Heuristic

Hypothesis

> **The preprocessor using a literal-level heuristic including the number of appearances of literals, clause size statistics of literals, and the covariance of literals will perform better than the preprocessor using a heuristic that includes only a strict subset of those metrics.**

The inclusion of each of the above metrics was explained in section 4.3. More information in the heuristic will allow it to choose more important variables, so the most complex heuristic will perform best

Experiment

Run the preprocessor four times, with a different heuristic each time. Each heuristic (aside from random) is the absolute difference of a function f of the complementary literals:

$$h = |f(l) - f(\neg l)| \tag{5.2}$$

The function f differs based on each heuristic.

- **Random:** the variable to be assigned is chosen randomly.
- **Appearances:** the absolute difference in number of appearances between complementary literals is taken.

$$f(l) = number\_appearances(l) \tag{5.3}$$

- **Importance:** Number of appearances and clause size statistics are incorporated into the heuristic.

$$f(l) = \frac{number\_appearances(l)}{min\_clause\_size(l)} \tag{5.4}$$

- **Complete:** Number of appearances, clause size statistics, and literal covariance are all included.

$$f(l) = \frac{number\_appearances(l) * cov(l)}{(avg\_clause\_size(l) + min\_clause\_size(l)) * \sqrt{cov(\neg l)}} \tag{5.5}$$

Metric

The performance of each experiment will be measured using metric 5.3.3.

5.6.2 Cutoff

Hypothesis

The optimal cut-off point for the pre-solver is 0.7.

The higher the cut-off point, the fewer instances the pre-solver will fully solve or violate satisfiability. Since satisfiability maintenance is more important than finding a complete assignment, a higher cut-off point is better. However, if the cut-off is too high, the pre-solver will never have much impact on the instances.

Experiment

Run the preprocessor ten times, with a cut-off probability ranging from 0% to 90% in increments of 10%.

Metric

The performance of each experiment will be measured using metric 5.3.1.

Chapter 6

Results

6.1 General Performance

These results measure the general performance of the pre-solver.

6.1.1 Assignment Validity & Completeness

	Success	Partial Success	Failure
CBS	0.9	0.1	0
verification	0.9	0.0816	0.0204

Table 6.1: Proportion of successes, partial successes, and failures of the pre-solver run on 50 instances with default configurations.

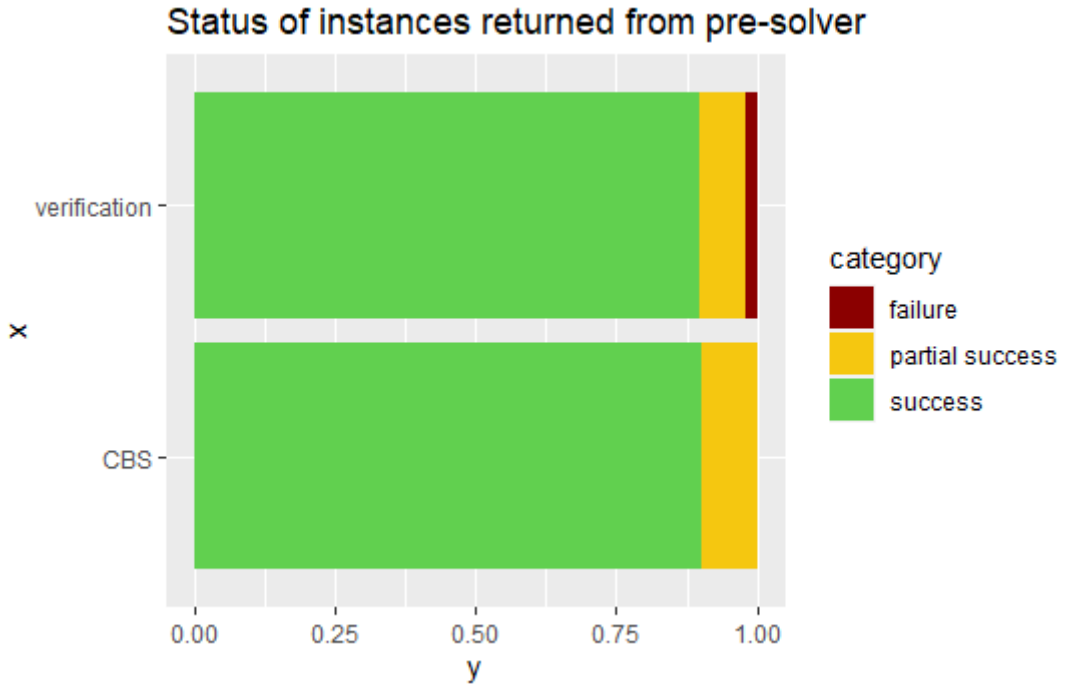

Figure 6.1: Barplot of general performance of pre-solver, by dataset.

6.1.2 Effect on Runtime

There was a mean reduction in runtime of 43.67% for the CBS dataset. The verification dataset had a mean reduction in runtime of 52%.

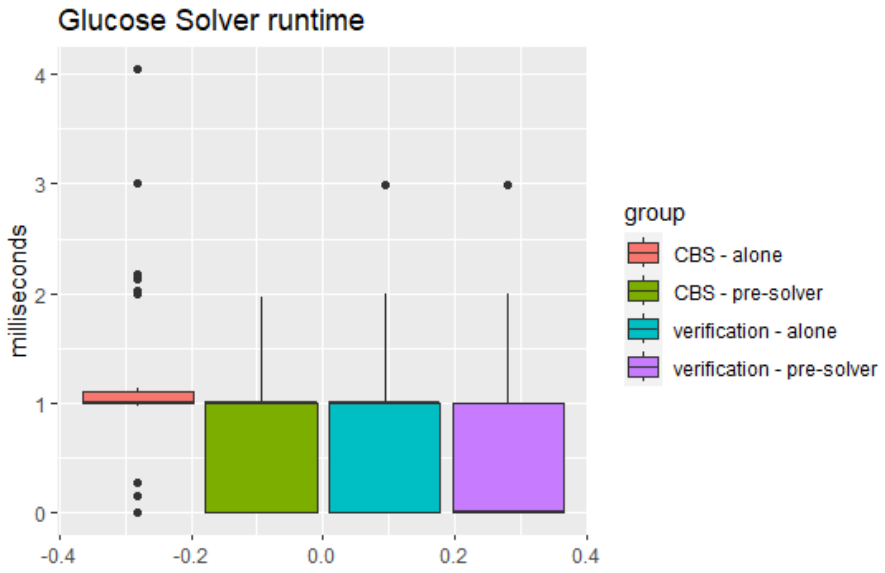

Figure 6.2: Comparison of solver runtimes on original and reduced instances, by dataset.

6.1.3 Number of Manual Assignments Required for Complete Solution

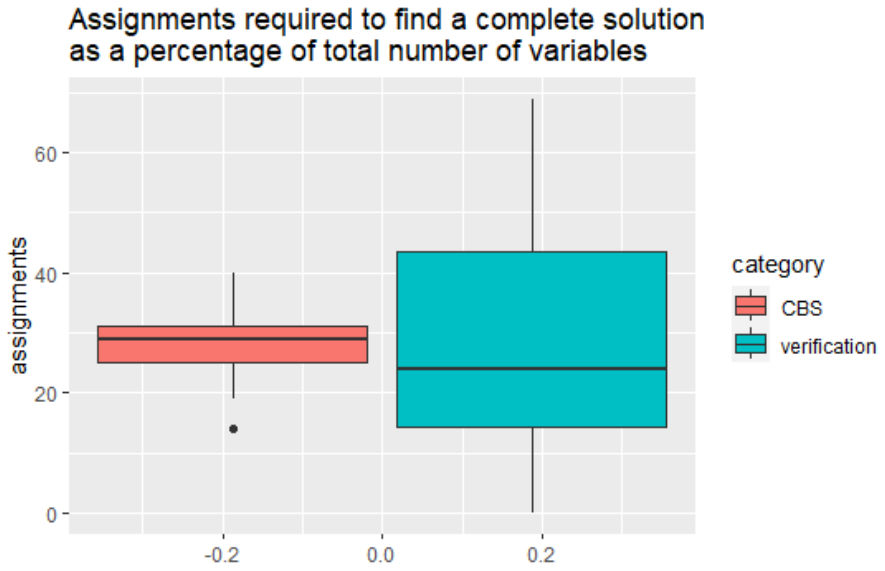

Figure 6.3: Number of assignments made before a complete solution was found as a percentage of total variables, by dataset.

6.1.4 Recognition of UNSAT Instances

This experiment measures the ability of the pre-solver to recognise that the original instance is UNSAT and to terminate accordingly.

	min	median	mean	max
proportion	0	0	0.73	13

6.2 Machine Learning

These experiments measure the effect different configurations of the random forest classifier have on the performance of the pre-solver.

6.2.1 Class Balance

	Success	Partial Success	Failure
Balanced	90	10	0
Skewed	90	10	0
Severely Skewed	78	22	0

Table 6.2: Success rate by SAT/UNSAT class balance in the training set.

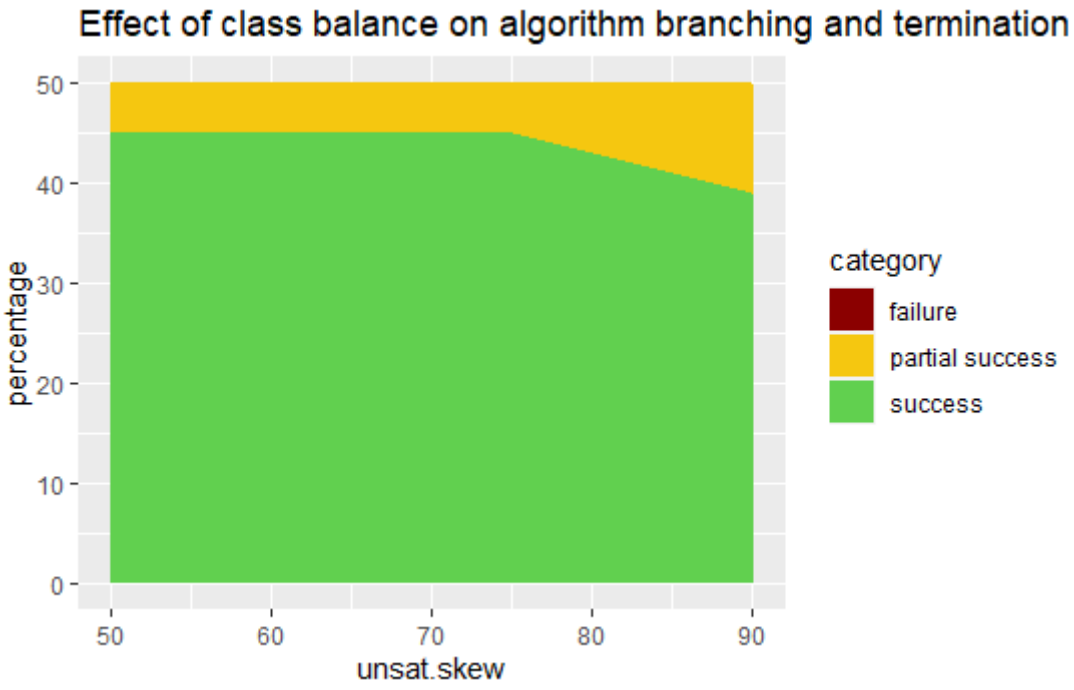

Figure 6.4: Experiment 5.5.1 - proportion of skewness towards UNSAT instances in training set.

6.2.2 Feature Selection

	DPLL	Base
success	90	4
partial success	10	22
failure	0	74

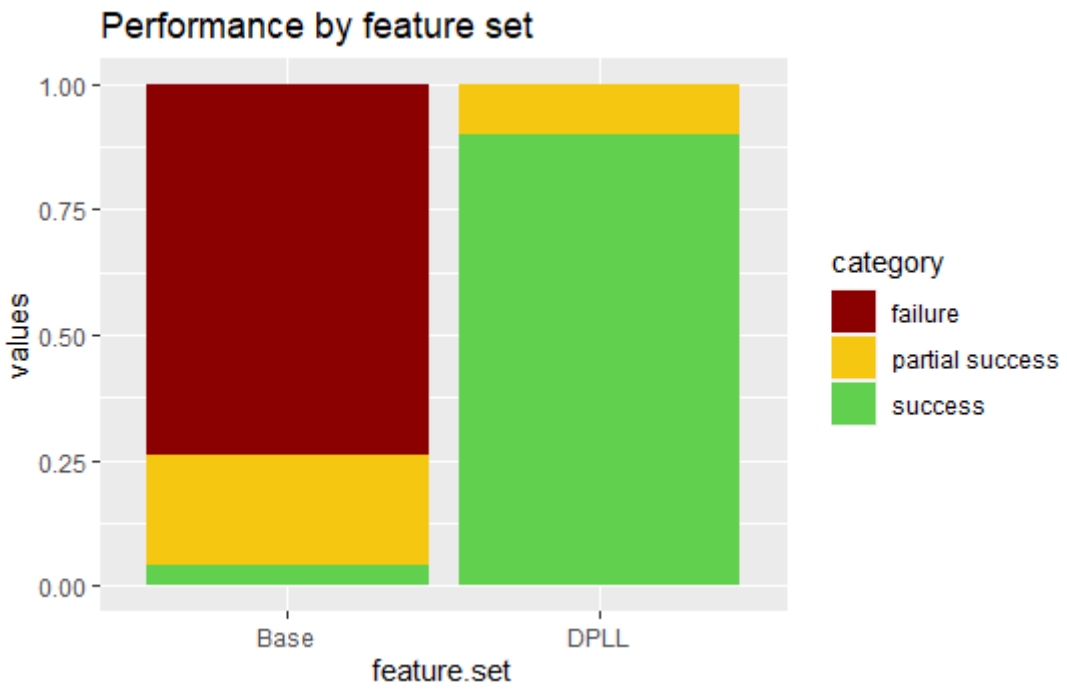

Figure 6.5: Experiment 5.5.2 - comparison of performance between feature sets including and excluding statistics from DPLL probing.

6.3 Algorithm Configuration

These experiments measure the effect different configurations of the algorithm have on the performance of the pre-solver.

6.3.1 Heuristic

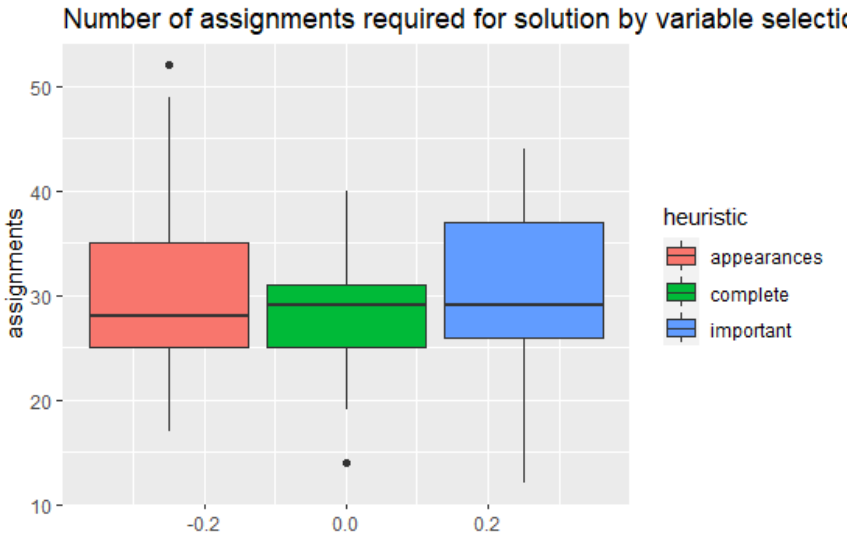

Figure 6.6: Experiment 5.6.1 - CBS - comparison of the effect heuristics of different complexity have on the number of assignments required before a complete solution is found.

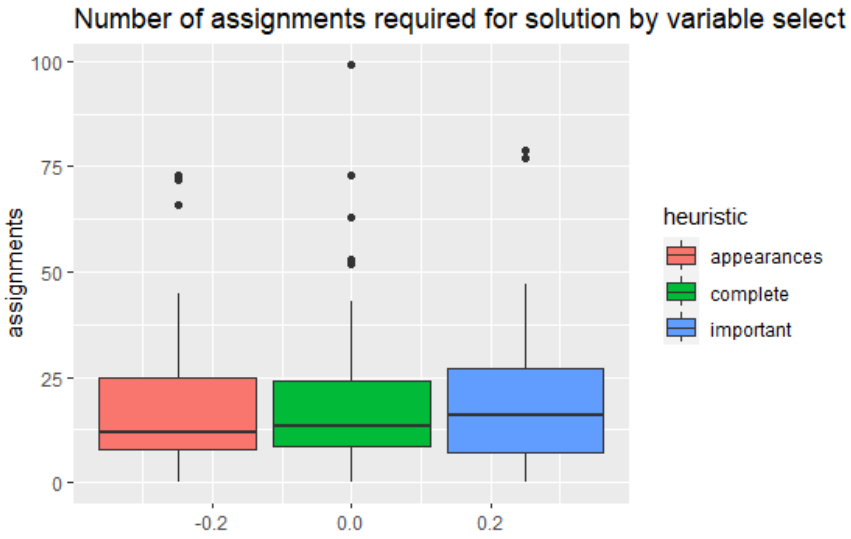

Figure 6.7: Experiment 5.6.1 - verification - comparison of the effect heuristics of different complexity have on the number of assignments required before a complete solution is found.

6.3.2 Cutoff

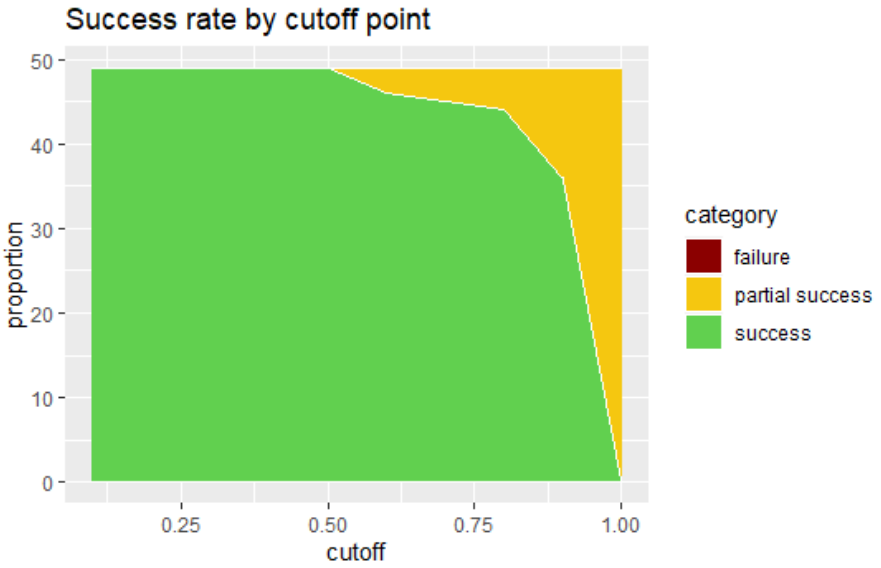

Figure 6.8: Experiment 5.6.2 - CBS - comparison of the effect the cutoff point has on assignment completion.

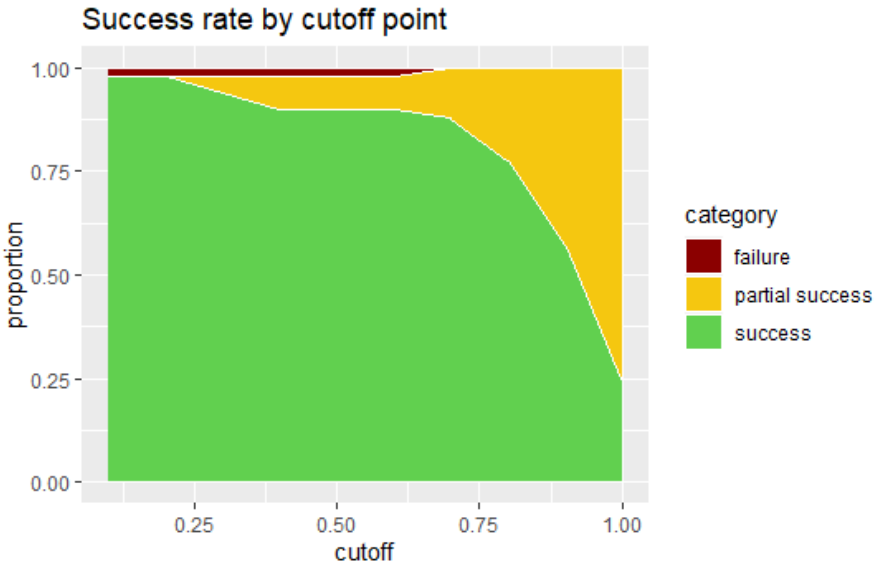

Figure 6.9: Experiment 5.6.2 - verification - comparison of the effect the cutoff point has on assignment completion.

Chapter 7

Conclusions & Future Work

7.1 Discussion of Results

7.1.1 General Performance

Overall, in light of the fact that this algorithm makes assignments based on a heuristic, its performance is excellent. Random assignment solves 0% of instances, whereas at its best this algorithm solved 100% of instances. The pre-solver did violate satisfiability for one instance of the verification dataset, so there is still clearly room for improvement.

The instances were small, so the glucose4 solver had a runtime of several milliseconds at most. This made it difficult to assess the effect the pre-solver had on runtime. Some instances were solved so quickly that the runtime measured as 0.000 seconds - basically instantaneous! However, in aggregate, there was still a perceptible difference in solver runtime between the original and reduced instances. In future, this effect could be assessed further on larger instances, when the algorithm has been optimised.

The number of assignments required to find a complete solution (as a proportion of the total number of variables) averaged between 20-30%. This metric of course depends greatly on the class of problem as well as the performance of the pre-solver, as can be seen with the difference in variablity between the results for the CBS and verification datasets. However, the fact that less than a quarter of variables needed to be manually assigned on average is promising. The most time-intensive element of the algorithm is the calculation of instance features (particularly DPLL probing), so the lower the number of manual assignments required to find a complete solution, the better. Further iteration on the literal-level heuristic used could yield improvement in this area.

The worst-case number of assignments taken to recognise an instance which was originally UNSAT, 13, is disappointing. It is over 4x worse than the predicted worst case. However, the average case number of assignments offers some solace, with a median of 0. In most cases, the algorithm terminated immediately. Variegation of the training set in terms of UNSAT instances is recommended to improve the worst-case scenarios.

7.1.2 Machine Learning

As satisfiability wasn't violated by the pre-solver in any of the instances in experiment 5.5.1, it's difficult to glean much insight from the results. It seems that the classifier will perform best when trained on a balanced or slightly-skewed dataset, but more experimentation is strongly recommended to deepen understanding in this area.

While DPLL probing features are expensive to compute, they are clearly worth the wait. The performance of the pre-solver when the classifier was trained on SATzilla base features alone yields a shocking 74% failure rate. Despite its challenges, the inclusion of DPLL probing features in the training set is a foregone conclusion.

7.1.3 Algorithm Configuration

Surprisingly, the heuristics of varying complexity performed equivalently, at least in terms of average number of assignments required to find a solution. The complete heuristic does seem to have lower variability in the number of assignments required, but that's not necessarily a feature to strive for. More experimentation is required, but as per Occam's Razor, the simplest heuristic could be the best.

There is a steady curve in the boundary between successes and partial successes as they vary by cutoff point. This curve levels out around the 70-75% mark, which suggests that this may be the optimal cutoff point for when the pre-solver is trained on a SAT/UNSAT balanced dataset.

7.2 Future Work

7.2.1 Optimisation

This algorithm is highly computationally expensive, in terms of both space and time. It wasn't feasible to experiment on instances with more than a thousand clauses. This dissertation presents a persuasive proof-of-concept, however the pre-solver would need to be significantly optimised in both space and time complexity before ever being considered competitive.

7.2.2 Backtracking Functionality

Despite a success rate of finding complete, satisfying assignments in excess of 98%, this approach can only be considered a pre-solver. This is because only one branch of the search tree is explored. As a full solver, the algorithm would need a backtracking component to search other branches of the search tree in the rare cases where the classifier makes an error. Without this, the algorithm remains purely heuristic and can act only as a pre-solver.

7.2.3 Reinforcement Learning Integration

This algorithm could be represented as a Markov process, where the state-space is the instance and all its reduced versions, and the action space consists of {variable assignment, termination}. Integrating this algorithm into a deep Reinforcement

Learning model with more actions - like applying resolutions, checking for tautologies, and checking for subsumptions - could yield an extremely powerful solver.

7.3 Conclusion

This dissertation investigates the use of SAT binary classification to guide branching decisions in a probing-based pre-solver. The preprocessing algorithm iteratively assigns variables using a random forest classifier and a literal-level heuristic. The predictions of the classifier are also used as stopping criterion, allowing for a complete, valid assignment to be found in some cases.

Experiments on small instances show that this approach has great promise as a preprocessing algorithm, yielding full solutions most of the time. However, as of yet, the algorithm is far too inefficient in both space and time complexity to be competitive. Further, more experimentation is required to find the optimal literal-level heuristic, training dataset class balance, and stopping criterion.

These findings open up several avenues for research, including implementation of backtracking functionality and integration into a larger deep Reinforcement Learning model. While in its current capacity the algorithm could only be used as a preprocessing step in a full solver, with extension this algorithm could itself be the basis of a full solver.

Bibliography

[1] Ibrahim Abdelaziz et al. "Learning to guide a saturation-based theorem prover". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.1 (2022), pp. 738–751.

[2] Enrique Matos Alfonso and Norbert Manthey. "New CNF Features and Formula Classification." In: *POS@ SAT*. 2014, pp. 57–71.

[3] Carlos Ansótegui et al. "Structure features for SAT instances classification". In: *Journal of Applied Logic* 23 (2017), pp. 27–39.

[4] Gilles Audemard and Laurent Simon. "On the Glucose SAT solver". In: *International Journal on Artificial Intelligence Tools* 27 (Feb. 2018), p. 1840001. DOI: 10.1142/S0218213018400018.

[5] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. "Preprocessing in SAT Solving." In: *Handbook of Satisfiability* 336 (2021), pp. 391–435.

[6] Marco Dalla, Andrea Visentin, and Barry O'Sullivan. "Automated SAT Problem Feature Extraction using Convolutional Autoencoders". In: Institute of Electrical and Electronics Engineers (IEEE), Dec. 2021, pp. 232–239. ISBN: 9781665408981.

[7] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[8] David Devlin and Barry O'Sullivan. "Satisfiability as a Classification Problem". In: *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science* (05 2008).

[9] Niklas Een and Armin Biere. "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: vol. 3569. June 2005, pp. 61–75. ISBN: 978-3-540-26276-3. DOI: 10.1007/11499107_5.

[10] Armin Biere Mathias Fleury. "GIMSATUL, ISASAT, KISSAT". In: *SAT COMPETITION 2022* (), p. 10.

[11] Hyojung Han and Fabio Somenzi. "Alembic: An Efficient Algorithm for CNF Preprocessing". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: Association for Computing Machinery, 2007, pp. 582–587. ISBN: 9781595936271. DOI: 10.1145/1278480.1278628. URL: https://doi.org/10.1145/1278480.1278628.

[12] Holger Hoos and Thomas Stützle. "SATLIB: An online resource for research on SAT". In: Apr. 2000, pp. 283–292.

[13] Vitaly Kurin et al. “Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver?” In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9608–9621.

[14] I. Lynce and J. Marques-Silva. “Probing-based preprocessing techniques for propositional satisfiability”. In: *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence*. 2003, pp. 105–110. DOI: 10.1109/TAI.2003.1250177.

[15] Benjamin Provan-Bessell et al. “SATfeatPy–A Python-based Feature Extraction System for Satisfiability”. In: *arXiv preprint arXiv:2204.14116* (2022).

[16] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. “NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. Hoos and David G. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 276–291. ISBN: 978-3-540-31580-3.

[17] Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer. “Effectiveness of pre- and inprocessing for CDCL-based SAT solving”. In: *CoRR* abs/1310.4756 (2013). arXiv: 1310.4756. URL: http://arxiv.org/abs/1310.4756.

[18] Haoze Wu. “Improving SAT-Solving with Machine Learning”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 787–788. ISBN: 9781450346986. DOI: 10.1145/3017680.3022464. URL: https://doi.org/10.1145/3017680.3022464.

[19] Lin Xu et al. “SATzilla: portfolio-based algorithm selection for SAT”. In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606.

Appendix A

Algorithm Implementation

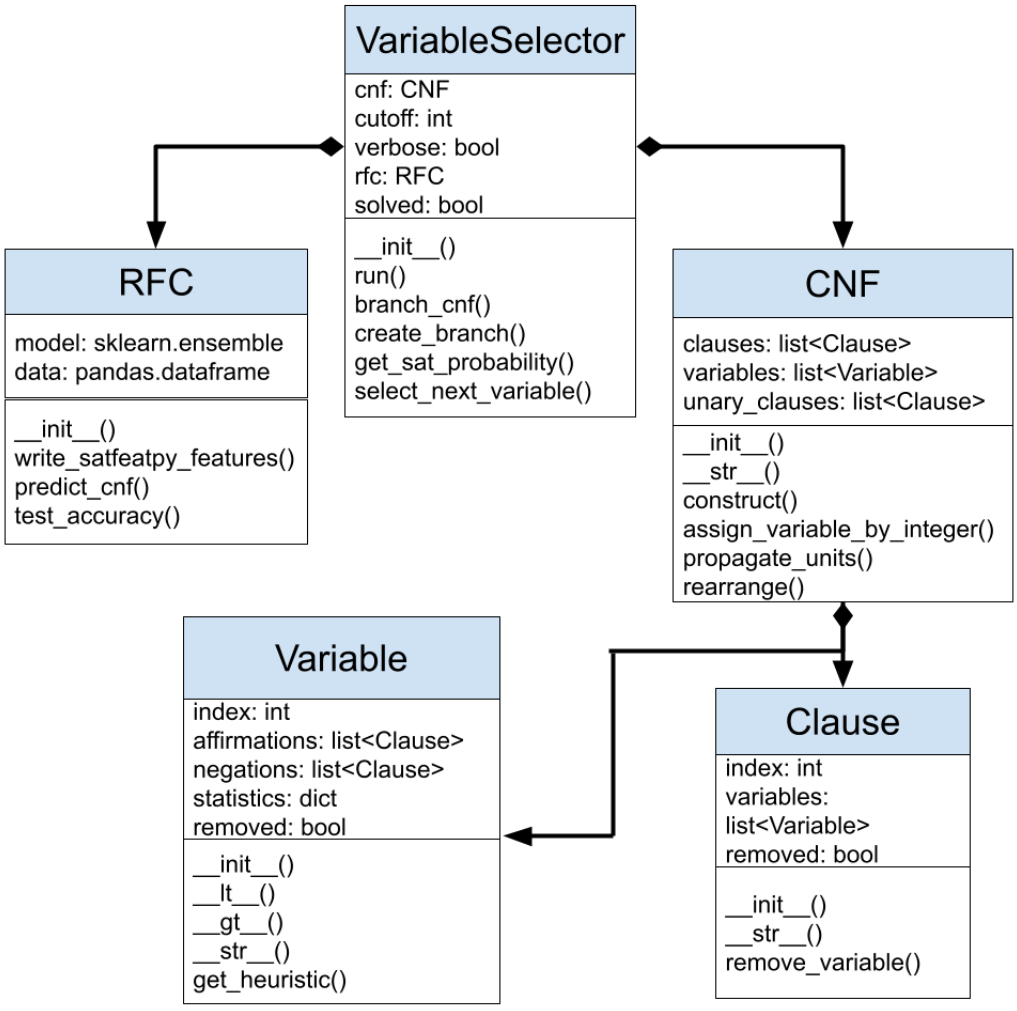

Figure A.1: Simplified UML diagram of package