An Analysis of the Effect of Community Structure on SAT Solver Performance

by

Zack Newsham

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electronic and Computer Engineering

Waterloo, Ontario, Canada, 2015

Abstract

Despite enormous improvements in Boolean SATisfiability solver performance over the last decade, it is still unclear why specific input formula are slow to solve, when other similarly specified formula execute more quickly. This work explores the relationship between the community structure of a SAT formula and its execution time on several state-of-the-art solvers. We explore the analysis of this data from a number of directions; first, we explore the relationship between the well known clause-variable ratio result, and community structure in randomly generated instances. Second, we perform a standard linear regression on data obtained from the 2013 SAT competition. Third, we present a visualisation tool and data repository for viewing the structure of a SAT formula. Fourth, we explore the effect of hardware constraints on the solution time of instances across various machines. Finally, we explore survival analysis, a technique that is new to the field of Boolean SATisfiability. By collating the results from each of these experiments, we have determined that the community structure is critical in determining the solution time of a SAT formula, more important than the clause-variable ratio of the formula. While this work is not a complete explanation of the varying solution time of SAT formulae, it has provided us with significant insight for further research to answer the question: why different similarly specified formula have such different solution times?

Acknowledgements

I would like to thank my supervisors Sebastian Fischmeister and Vijay Ganesh, as well as the kernel development team at QNX for their help with implementation and Jean-Chrisophe Petkovich for his help with the DataMill experiment. Additionally, I'd like to thank William Lindsay for his help with the development of SATGraf and Professor Thomas Duever for his help with the details of the linear regression experiments. Finally I'd like to thank Gina Hickman, for her support and proof reading abilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Boolean satisfiability (SAT) problem is widely regarded as one of the most important problems in computer science. The problem is known to be NP-complete [43], and an efficient algorithm to solve it can have profound impact on computer science and society as a whole. While the problem is believed to be intractable [21] in general, in the past decade there has been dramatic progress in designing and implementing efficient algorithms that perform very well on large instances obtained from practical applications.

In this thesis I will explore the relationship between the community structure of a Boolean satisfiability formula and its solution time. The community structure of an input formula (or instance) can be loosely defined as how well clustered the variables in the formula are, with respect to the clauses that connect them.

Problem Statement: The intention of this work is to provide a detailed analysis of the effects of structure — specifically community structure — on the solution times of Boolean satisfiability formula. We will also compare this model with previously considered models, such as the phase transition in the clause-variable ratio.

It has long been known that the structure of a SAT instance has some effect on the performance of SAT solvers. The work by Gent and Walsh [18] as well as the work by Coarfa, et al [14] has shown that there is some connection between the clause-variable ratio (CVR) of random SAT instances and their performance. However this relationship changes with the use of different solver algorithms, and implementations. In addition to this, further work by Levi, et al [7, 6] has shown that industrial SAT instances exhibit a better community structure than other types of SAT formula (e.g., random or hand-crafted).

While attempts have been made to explain the link between structure of a SAT instance and its solution time, these have been limited to solver specific analysis and typically basic factors, such as the clause-variable ratio. This work shows that SAT instances have an inherent community structure, and that in many cases this community structure is correlated with the solution time of that instance. We go on to show that this is the case for a variety of state-of-the-art solvers under numerous testing conditions. Further, we present an analysis technique that is new to the field of SAT solver performance, and has enabled us to identify specific characteristics of an input formula that affect the performance of certain SAT solvers.

Motivation: Understanding the structure present in SAT instances, and the relationship between this structure and SAT solver performance should enable practitioners to develop more predictable and potentially faster SAT solvers. As SAT solvers are used in all fields of software engineering, for example program synthesis[20] and vulnerability testing[26], an improvement in SAT solver technology can be translated into an improvement in the implementation of software engineering techniques.

Contributions:

- First, We performed a regression analysis on approximately 800 instances from the 2013 SAT Competition executed on the Minipure solver, with additional results considered from other competitions and solvers. These were analysed while paying particular attention to the community structure of the instances. We found a strong correlation between measures of community structure and Minipure performance. These results are described in Chapter 3

- Secondly, we performed an analysis of the community structure present in random instances, and the relationship of such a structure with the clause-variable ratio(CVR). This experiment was performed twice with different datasets, providing the same result - that random instances with $0.05 \leq Q \leq 0.13$ are harder to solve than similar formula with a community structure outside of this range. This was found to be the case, regardless of the instances CVR. This result is described in Chapter 4

- Thirdly, we developed a tool for viewing the evolution of the community structure of a SAT formula, during solution time. This tool named SATGraf is described in Chapter 5

- Fourthly, we introduce survival analysis as a tool for evaluating the execution time of different classes of trials. For example, trials performed on different solvers or machines, or instances with different ranges CVR. We were able to confirm previous hypothesis using this technique, and gain extra detail on others. These results are described in Chapter 6

- Our final contribution is An initial explanation of the varying performance seen when running SAT instances on different machines. In this study we consider approximately 14,000 pre-simplified instances and their running times on 28 different machines and five solvers. We found that solution time of a particular instance on a particular solver, is not purely a product of the solving machines CPU speed, or other basic factors such as RAM and cache sizes. These results are described in Chapter 7

- In addition to these main results, we present SATBench. A user editable repository for storing and analysing SAT instances from various sources, utilising numerous (currently approximately 40) different parameters.

Summary of Thesis In order to understand the link between community structure and the performance of SAT instances, it is necessary to define certain terms. The study of community structure in complex networks/graphs is not new and many algorithms have been proposed to represent this structure as a single metric, commonly referred to as the Q metric. In order to utilise such algorithms it was necessary to convert the instances under analysis into graphs. For this task we chose to represent a SAT instance as its variable-incidence graph, where variables in the formula become vertices in the graph. An edges between two vertices if the variables they represent are present within a clause of the input formula.

In this work we use two such proposed algorithms, the Clauset-Newman-Moore (CNM) method and the Online (OL) community detection algorithm.

These are described in detail in Chapter 2, however it suffices to say for now that both these algorithms define the community structure of a SAT instance as a ratio of the number of between community edges to the number of within community edges. A community is a sub graph of the variable-incidence graph that contains more internal edges than outgoing ones.

We explain the link between community structure and SAT solver performance by utilising a number of analytical techniques. For example, we use simple correlation metrics, measures of fit, standard linear regression and survival analysis. These individual methods are described in detail in Chapters 3 and 6, however the purpose of each analysis is the same: to reduce a complex set of input parameters and provide a single measure for the quality of the relationship between two factors. In our case the response factor for each of these analysis techniques is almost always the wallclock execution time of a SAT solver, however in some analysis, we use the measure of the relationship between two factors as a response variable, such as when considering how effective certain prediction models are for different types of input formula as in Chapter 7.

Structure of Thesis: This thesis contains five distinct undertakings, all related to the study of the community structure of a SAT instance, and the connection between that structure and an instances solution time.

In order to understand the work presented in this document, a certain amount of background knowledge is required of the reader. Much of the relevant information can be found in Chapter 2, where we describe the different regression techniques used in the experiments, as well as defining community structure. In addition to this, each chapter presents any specific relevant information required.

In Chapter 3 we explore the utility of certain input characteristics — such as the Q value, and the clause-variable ratio — on the solution time of the Minipure[42], MiniSAT[15] and Glucose[8] solvers, when considering input formula and timing data taken from the 2013 and 2014 SAT competitions [1].

In Chapter 4 we endeavour to explain the relationship between the well understood CVR result [18, 14] and the recently explained Q result [36] when considering randomly generated SAT formula

In Chapter 5 we describe our tool SATGraf, a tool designed to allow the visualisation of the community structure of a SAT instance, as well as the evolution of that SAT instance while it is being solved.

In Chapter 6 we discuss an analysis technique that to our knowledge has not been considered in the field of SAT solver performance, survival analysis. In survival analysis it is possible to describe the probability of a solver still running after a certain amount of time, when considering different input factors, such as the clause-variable ratio Q or other less obvious factors.

Finally, in Chapter 7 we discuss performance differences when solving the same instance on different hardware and solvers, and provide an initial insight into the effects of CPU speed, RAM size and cache sizes on the performance of different solvers, and different types of instances.

Chapter 2

Background

In this chapter, we provide some background on regression analysis, the concept of the community structure of graphs and how it relates to SAT formula.

1 Boolean satisfiability

The Boolean satisfiability problem (referred to as SAT) is the quintessential NP-complete problem. In short, SAT takes an input formula of clauses and variables, and tests if there is an assignment to each of the variables that allows the formula to evaluate as true. Each variable has only a true or false representation, and may appear as both in a formula, in separate clauses. When a variable is represented with a value it is referred to as a literal. There are multiple representations for SAT problems, the most commonly used is conjunctive normal form (CNF). In CNF clauses are sets of variables joined by disjunction, the entire formula is made up of the conjunction of all the clauses, as opposed to disjunctive normal form (DNF) which is the opposite. A formula is said to be satisfiable if every clause in the problem is satisfied, a clause is satisfied if at least one literal in the clause has a satisfying assignment. For example, consider Equation 2.1. This example is displayed in dimacs format, where each variable is numbered. A positive number specifies that literal requires a true assignment to the variable, a negative literal specifies that a false assignment is required, clauses are terminated with zeros. A satisfying assignment to this formula could be $-1, 2, -3, -4$. Compare this to the highly trivial Equation 2.2, which has no satisfying assignment.

$$\begin{array}{r} 1\ 2\ 3\ 0 \\ -1\ -2\ 0 \\ 3\ -1\ 4\ 0 \\ -3\ 2\ -1\ 0 \\ -4\ 0 \end{array} \tag{2.1}$$

$$\begin{array}{r} 1\ 2\ 0 \\ -1\ 0 \\ -2\ 0 \end{array} \tag{2.2}$$

While in these highly simplified examples the problem may appear simple, these formulas can contain millions of clauses and variables, the problem quickly becomes complicated. As SAT formula became larger, new techniques have been developed to solve them. The current state-of-the-art solvers are conflict driven clause learning (CDCL) solvers [25].

In a CDCL solver there are two phases, initially the solver chooses a variable (based on some heuristic) and assigns it true/false. After this any other variable that can be assigned through propagation are set. This can be caused either because a variable now only appears as true/false (a pure literal) or due to constraints. For example, considering Equation 2.1, if the variable 1 were assigned to true, the solver would know to assign 2 to false to satisfy clause two. The second phase in CDCL solvers is when a conflict occurs. A conflict is defined as the solver finding a set of assignments that does not satisfy the formula, discovered when the next variable chosen propagates a variable assignment that conflicts with a previous assignment. Considering Equation 2.1 if the variable 1 is assigned to true, forcing 2 to be false, which in turn forces 3 to be false, however clause three specifies that either 3 must be true, or 1 must be false. This creates a conflict. The second phase of CDCL solvers is to learn from this conflict, this is achieved by finding the minimal set of assignments that led to this conflict, in our case $1, -3$. This clause is inverted and added to the formula. The solver then resets the state of the solver to the assignment before the first assignment in the conflict clause.

CDCL solvers can generate millions of conflict clauses, this can quickly consume the available memory. As such clauses are periodically deleted, unfortunately this removes the completeness property from CDCL solvers, as it is possible that a deleted conflict clause will be re-discovered, causing the solver to enter an un-escapable loop.

For more information on the Boolean satisfiability problem, consider reading the handbook of satisfiability [9]

2 Community structure of SAT formulae

The idea of decomposing graphs into *natural communities* [13, 45] arose in the study of complex networks such as the *graph of biological systems* or the Internet. Informally, a network or graph is said to have community structure if the graph can be decomposed into sub-graphs where the sub-graphs have more internal edges than outgoing edges. Each such sub-graph (aka module) is called a community. Modularity is a measure of the quality of the community structure of a graph. Figure 5.1 is an example of a graph with good community structure. the coloured edges represent edges within a single community, the white edged represent inter-community edges. The idea behind this measure is that graphs with high modularity have dense connections between nodes within a sub-graph but have few inter-module connections. It is easy to see that maximising modularity is one way to detect the optimal community structure inherent in a graph. Many algorithms [13, 45] have been proposed to solve the problem of finding an optimal community structure of a graph, the most well-known among them being the one from Girvan and Newman [13]. The quality measure for optimal community structure is often called the Q value, and we will continue to refer to it as such. There are many different ways of computing the Q value and we refer the reader to these paper [13, 45, 28] for a complete explanation of the individual algorithms, a brief explanation of those used in this work is provided in Section 3.

We experimented with two different algorithms; the Clauset-Neuman-Moore (CNM) algorithm [13] and the online community detection algorithm (OL) [45]. While we did find that the CNM algorithm resulted in a better community structure — evidenced by a small number of communities with few links between them — we chose the OL algorithm because of its vastly superior run time. This was of particular importance due to the sheer size and number of SAT instances we processed. Our initial experiments were conducted with an implementation of the CNM algorithm, then repeated with the OL algorithm. The results we present in Section 2 were observed, regardless of the choice of algorithm.

3 Linear regression

In this thesis we make use of linear regression techniques for the result that correlates the Q value and number of communities with the running time of the MiniSAT CDCL SAT solver.

Linear regression can be used to determined the relationship between the factors and variables based on a provided model. Given multiple independent factors and a single dependent variable. For the scope of this paper the dependent variable will always be $log(time)$, while the independent factors (such as Q value, number of communities, variables, and clauses) will be appropriately specified for each experiment. This model can either look only at the main effects of the factors specified, or at both the effects of factors and the interactions between them.
We provide a few important definitions below:

ANOVA stands for analysis of variance. In the scope of this paper, it is generated by the linear regression, and used to understand the influence that specific factors and interactions between factors have on the dependent variable.

R^2 represents the amount of variability in the data that has been accounted for by the model and is used to measure the *goodness of fit* of the model. It ranges from zero to one with one representing a perfect model. Due to the nature of the calculation, the R^2 value will increase when additional factors are added to the model. In this paper, we refer to the *Adjusted* R^2 which is modified to only increase if an added factor contributes positively to the model.

Confidence Levels are used to specify a certain level of confidence that a given statement is true. They can be used to calculate the likelihood of a given set of input values resulting in a given output (e.g., time), or they can be used to estimate the likelihood that a factor in a model is significant. They are measured in percent, typical values are 99.9, 99 and 95. Any result with a confidence level below 95% is considered unimportant in the context of this paper.

Confidence Intervals are used to provide a range for a value at a given confidence level, which is usually set to 95% or 99%. They show that with a given percentage probability, an estimated value will lie within a certain range.

Kolmogorov–Smirnov test is used to provide quantitative assurances that a provided sample belongs to a specified distribution. It results in a value between zero and one, with values approaching zero indicating that the provided sample does belong to the specified distribution.

Residuals is the difference between a fitted dependent variable and the corresponding provided dependent variable. It represents the amount of error for a given set of input factors when calculating the output.

There are numerous resources available for further information on statistical analysis including linear regression such as [27].

Chapter 3

Regression analysis of impact of community structure on SAT solver performance

We pose the question: is there a connection between graph theory and performance of SAT solvers? There have been numerous attempts at characterising structure in SAT instances, the most famous of these is the result showing that hard random instances have a CVR of approximately 4.2[18]. However, this result only characterises the structure of random instances, and does not lend any insight to the structure of industrial or crafted instances. The results presented in this chapter utilise graph theoretical concepts, such as community structure, to characterise SAT instance structure. We found that there is a correlation between the quality of the community structure and SAT solver performance. Our initial work in this area utilised linear regression to determine which input characteristics of a SAT formula were most predictive of the solution time for a specific solver. This resulted in some interesting results as published in [36]. This chapter builds on those results by considering different datasets from different SAT competitions and solvers.

1 Experimental setup

The data for this analysis was taken from the 2013 SAT competition and looked at instances from the application, hard combinatorial and random categories. Our initial experiments were published in the SAT 2014 conference[36] and utilised the OL algorithm to detect the community structure and used the timing data published in the competition[1]. This data was obtained by running the solvers under analysis on a 2x Quad core Intel Xeon 2.83GHz with 16GB RAM running scientific Linux 2.6.18. When discussing time in this chapter, we refer to the wallclock time taken for the solver to either return a correct result, or to time-out. A time-out of 5200 seconds was specified. Due to memory constraints it was not possible to detect the community structure for each instance. As such in these initial experiments we were left with 835 instances to consider.

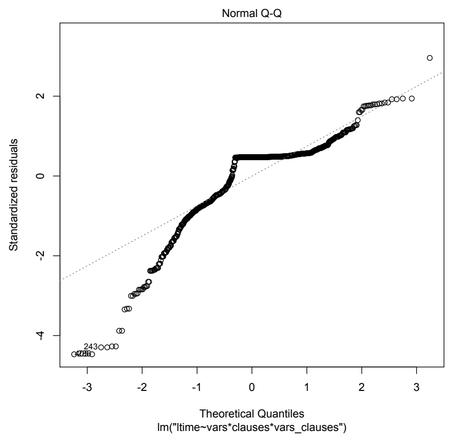
The analysis was performed using log(time) rather than raw recorded time due in part to the presence of a large number of time-outs. A time-out occurs when a user imposed deadline for a solution is passed. It is possible that a solution would have been found within the next second, or possibly not for many hours (or longer). The wide distribution between the slowest solved instances and the time-outs was an additional contributing factor to our decision to use log(time). This would have severely skewed the distribution if raw recorded time had been used. In addition to this we standardised our data to have a mean of zero and a standard deviation of one, which is standard practice when using parameters that have large differences in scale to ensure that significance is not awarded to a variable based on scale alone.

2 Results

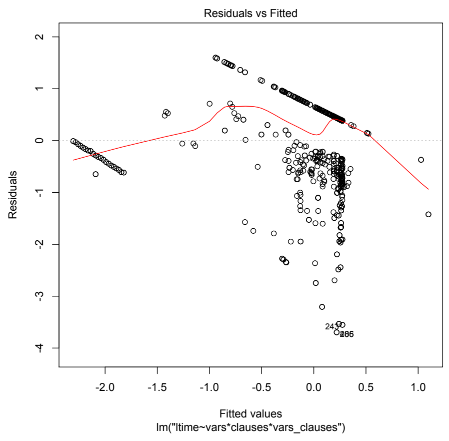
After formatting the data as described, we fitted a linear regression model to it using a stepwise regression technique to choose the best model, which was identified as:

$$log(time) = |V| \oplus |CL| \oplus \mathrm{Q} \oplus |CO| \oplus \mathrm{QCOR} \oplus \mathrm{CVLR} \tag{3.1}$$

Where V is the set of all variables in a formula, CL is the set of all clauses, CO is the set of all communities, QCOR is the ratio of $\frac{Q}{|\mathrm{CO}|}$, and VCLR is the ratio of $\frac{|V|}{|\mathrm{CL}|}$. In this model the $\oplus$ operator denotes that the factor and all interactions with all other factors were to be considered. Performing the regression resulted in a residual vs fitted plot, shown in Figure 3.2a where the x-axis shows the fitted values and the y-axis shows the residuals. As well as a normal quantile plot shown in Figure 3.2b where the x-axis shows the standardised residuals plotted against a randomly generated normally



(a) Plot of normal and theoretical quantiles



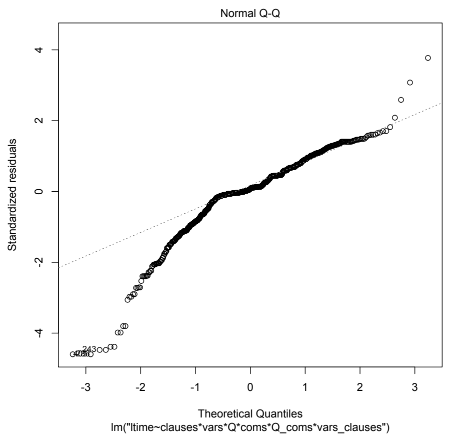
(b) Residuals vs Fitted values

Figure 3.1: Plots for the model without community metrics $R^2 = 0.3148$

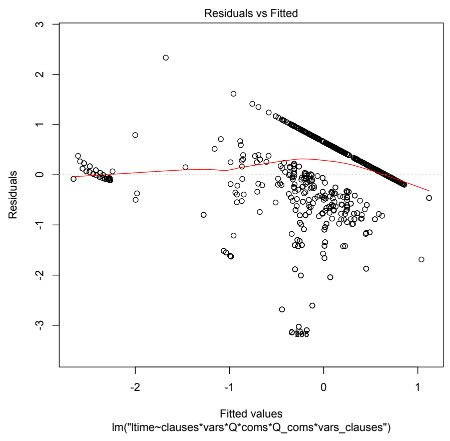
distributed sample with the same mean and standard deviation on the y-axis. In the normal quantile plot, the presence of a slight curve in the line is indicative that the distribution of the data is non normal. However, when we consider Figure 3.1 which shows the same data plotted for the previously available model 3.2 we can see that its data is at least equally non-normal. In this situation calculating confidence intervals is non straightforward and a method to allow accurate estimates is unknown to us, as such we are unable to measure confidence intervals for the accuracy of the model. In addition to this, the residual plot shows that the data may be biased, but at least has relatively even variance. Unfortunately, the presence of the time-out results has played a role in the biased nature of the experiment. However dropping them from the results entirely leads to a bias in the opposite direction.

The adjusted R^2 of our model 3.1 is 0.5159. While this relatively low R^2 value indicates that there is some factor we have not considered, our model is far better than any previous model, which relied only on number of variables and clauses. This model, which takes the form of

$$log(time) = |V| \oplus |CL| \oplus \text{CVLR} \tag{3.2}$$



(a) Plot of normal and theoretical quantiles



(b) Residuals vs Fitted values

Figure 3.2: Plots for the model 3.1 including community metrics $R^2 = 0.5159$

is also given for comparison and results in an adjusted R^2 of 0.3148 — making it significantly less predictive than our model. In addition to this, the more distinct S shape, and presence of sharp curves in Figure 3.1 shows that the distribution of the data is less normal. This result is confirmed when using the Kolomogorov-Sminov (KS) method to test *goodness of fit*. Our model 3.1 results in a KS value of 0.1283 compared with the previously available model 3.2 which gave a KS value of 0.3154. As discussed, this lack of normality makes it difficult to estimate confidence intervals for the results. However, it is possible to rank the factors by importance (because the data was standardised prior to regression). The results in Figures 3.2a and 3.2b show that our model, while not perfect, is a major step towards being a predictor for solve time. This is confirmed when viewing the results of the regression shown in the Table 3.1 (This table can also be viewed from our website [35]). This table shows each significant factor (Appendix C shows the table for all factors) and the probability that its significance is random ($Pr(>|t|)$).

The main result we found from the regression is that the Q factor is involved in every one of the significant interactions at a 99.9% confidence level.

In addition to this we found that $|V|$ (number of variables) alone is not significant, and |CL| (number of clauses) alone is only marginally significant. Furthermore, |CO| (number of communities) proved to be the most predictive effect, as well as being involved with numerous other interactions that are also significant.

3 Further results

After evaluating this initial result, we decided to explore additional datasets from the SAT competition, specificity different years and solvers. Unfortunately, due to unforeseen circumstances we were unable to directly reuse the data obtained in the previous result for the SAT 2013 Minipure solver. Instead we re-ran the community detection algorithm from approximately the same subset of formula from the 2013 SAT competition and used this data for analysis where relevant. Due to the different physical characteristics of the machines used to run the community detection algorithm, the subset of results used here is slightly different than those used in the previous section, additionally due to a randomness that is present in the community detection algorithm, the Q value and number of communities also differs slightly. Due to these small changes the R^2 of the repeated result is 0.5198, a very small difference.

The full details of the changes to the data can be found in the errata attached to [36], however as an overview, we re-ran the community detection algorithm on 935 instances [32] then paired the number of variables and clauses with the original data to give us an approximation at the same data

Factor	$Pr(>\|t\|)$	Sig
$\|CO\|$	0.000121	***
$\|CL\| \odot Q \odot QCOR$	0.000492	***
$\|CL\| \odot Q$	0.000523	***
$\|CL\| \odot Q \odot \|CO\| \odot QCOR \odot CVLR$	0.000702	***
$\|CL\| \odot Q \odot \|CO\|$	0.000719	***
$Q \odot QCOR$	0.000881	***
Q	0.000947	***

Table 3.1: List of the factors with 99.9% significance level. $\odot$ indicates an interaction between two or more factors and *Sig* stands for Significance. The full table is listed in Appendix Table C

set. It was important to attempt to repeat the same dataset as the newer dataset had far more time-outs than the original dataset. We were ultimately able to match 710 instances from our new dataset, to those instances in the original dataset. As such, these 710 instances are used in the repeated regression for the SAT 2013 competition solvers.

In an attempt to confirm the result described in Section 2 we re-ran the same linear regression on the equivalent formula set from the 2014 SAT competition [2]. Unfortunately we ran into some issues with this. Firstly, the timing data from the SAT competition random instances is only available for satisfiable instances, Which severely reduces the size of this portion of data, it also results in an unfair analysis due to the lack of any UNSAT formula. Secondly Minipure did not compete in the 2014 competition, meaning that a secondary solver had to be used for timing data, as stated in [14] different algorithms and implementations of those algorithms display different timing characteristics — in some cases vastly different — as such it is not possible to directly compare the results of our model on the Minipure solver, with data obtained from any other solver. To mitigate this, we chose Glucose due to our familiarity with it and similar lineage, both Minipure and Glucose were derived from variants of the MiniSAT solver. However, even then Glucose did not compete in the random category. In fact no single solver competed in all three categories. These reasons combined with the different physical characteristics of the machines used to run the trials, as well as the differing time-outs (5200 seconds vs 6000 seconds) makes it very difficult to compare these results as apples to apples. To mitigate this we also considered the Glucose timing data from the 2013 SAT competition, and in some cases will limit our hypothesis to the application and crafted formula from the relevant SAT competitions, as we have solver complete datasets for these. The results of these experiments are detailed below in Table 3.2.

The most noticeable result from these experiments is that the Glucose results are far weaker than the Minipure result. There are many possible reasons for this, including different functionality of the solver, however the most likely explanation is that the absence of random instances in the Glucose regressions skewed the result. The less significant difference between Glucose 2013 and Glucose 2014 is likely the increased time-out, increasing the non-normality of the data under analysis.

In addition to the differing R^2 for the solvers in each competition, we noticed the R^2 for each individual category — particularly with the Glucose solver — the application and hard combinatorial/crafted categories, was

Competition	Subset	Solver	#Formula	#Completed	R^2
2013	All	Minipure	709	274	0.5198
2013	Application	Minipure	228	127	0.5058
2013	Crafted	Minipure	232	147	0.7032
2013	Random	Minipure	249	0	N/A
2013	All	N/A	0	0	N/A
2013	App+Crafted	Glucose	538	350	0.3332
2013	Application	Glucose	264	169	0.5403
2013	Crafted	Glucose	273	180	0.6276
2013	Random	N/A	0	0	N/A
2014	All	N/A	0	0	N/A
2014	App+Crafted	Glucose	527	352	0.2990
2014	Application	Glucose	249	183	0.5702
2014	Crafted	Glucose	278	169	0.547
2014	Random	N/A	0	0	N/A

Table 3.2: Details of the regression scores for the eight subsets of data considered

stronger than the R^2 for the combined dataset. This would suggest that the Glucose solver behaves very differently when solving industrial vs crafted instances, in a way that cannot be explained by any of the four factors we have considered. While a similar pattern was observed with the Minipure solver, the R^2 of the random results is much lower than that of the overall competition, and likely some type of "averaging" is occurring when the full dataset is considered. It is possible that if we had experimental data for the Glucose solver on random instances, we would notice a similar pattern. However, this does not explain the pattern of application and crafted being more predictable alone than together.

4 Conclusions and future work

In this chapter we have discussed the relationship between the community structure of SAT instances and their performance on the Minipure solver. We further describe that this community structure is more important in determining solution time than previously considered metrics, such as the clause-variable ratio (CVR).

There are a number of directions we intend to take this work. First, as described in this chapter, the non-normality of the data precludes us from making confidence intervals on our predictions, to counter this we are going to look at different distributions for our regression. From our initial work in this area the most promising of these is a Gamma regression.

In addition to utilising different regression distributions, our results have shown clearly that the structure of random instances is different from both crafted and application instances. We intend to explore this by developing category specific models. To accomplish this we need to find significantly more instances that do not time-out.

Chapter 4

Relationship between CVR and community structure in random SAT instances

Previous attempts to uncover the relationship between the structure of a random SAT instance and its performance have discovered that hard random instances have a clause-variable ratio of approximately 4.2 [18]. However, further results in this area have shown that this CVR range is dependant not only on the algorithm used, but the implementation of the algorithm [14]. In addition to this, our previous work in this area has found a similar phase transition with respect to community structure [36]. However, we found that there was no direct correlation between community structure and CVR, and some instances identified as hard by the CVR model were identified as easy by our community based, and vice versa. This chapter further explores this relationship and we find that a combined model of CVR and community structure is more descriptive.

1 Experimental setup

The first version of this experiment, published in [36] considered approximately 550,000 randomly generated formula and their running time on the MiniSAT 2.2 solver. In doing so we discovered that there is a large increase in average solution time when $0.05 \leq Q \leq 0.13$. Unfortunately due to the random nature of the experiment, and the number of instances utilised it was difficult to perform a more in depth analysis on this data, without repeating the experiment. As such, the results presented here will be for a repeated experiment where we captured much more information about a smaller set of instances. For this later experiment we considered 4887randomly generated instances.

The formulae were generated by varying the number of variables from 500 to 2000 in increments of 100, the number of clauses from 2000 to 10,000 in increments of 1000, the desired number of communities from 20 to 400 in increments of 20, and the desired Q value, from zero to one in increments of 0.01. Each individual trial was repeated three times with the same characteristics. This was necessary due to the non-deterministic nature of the generation technique. We then randomly sampled 4887instances from this set. The resulting instances were ran in a random order on machine 27 from Appendix Table A

To generate a specific 3-CNF instance we perform the following actions: Let the set of variables be denoted as $\mathbb{V} = \{V_i \mid 0 \leq i < n_v\}$ where n_v is the desired number of variables. Similarly, let the set of groups be $\mathbb{G} = \{G_x \mid 0 \leq x < n_g\}$ where n_g is the desired number of groups. A group is a rough estimate of a community, and is used only to guide the generator in producing a problem with a specific structure. First, we assign variables to groups such that each group $G_x = \{V_y \mid y = r_v * |G| + x; 0 \leq r_v < \frac{|V|}{|G|}\}$, where r_v is randomly selected. This creates n_g groups each containing $\frac{n_v}{n_v}$ randomly selected variables. Next, we generate the set of clauses $\mathbb{C} = \{C_z \mid 0 \leq z < n_c\}$ as follows, where n_c is the desired number of clauses such that $C_z = \{V_{z1} \vee V_{z2} \vee V_{z3}\}$. Each clause is constructed as follows: First, a group G_x and a variable $V_{z1} \in G_x$ are randomly selected. This is followed by a selection of another variable V_{z2} from either G_x or V with probability of q of being selected from G_x. Finally, a third variable V_{z3} is selected from either G_x or V with probability of q of being chosen from G_x. The value q (lies between 0 and 1) can be used as a rough estimator of Q, the modularity of the formula and is provided as input to the generator. The result is a randomly-generated 3-CNF formula.

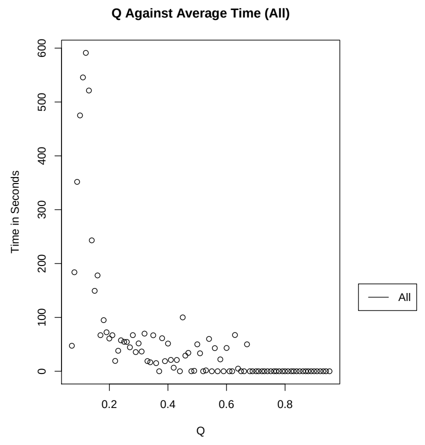
During our analysis of the instances, we discovered that our random generation technique resulted in far more results in the $0.05 \leq Q \leq 0.12$ range than in any other range. To ensure an unbiased result we performed basic analysis on a stratified random sample taken uniformly across the range of Q. A stratified sample ensures that a uniform number of samples are taken from each sub-population of the data, this ensures a fair analysis when the sizes of the sub-populations vary. From the original 545,000 results, 2250 were randomly sampled, with 250 results taken from each range of 0.1; as there were no results with $Q > 0.9$ this range was not included in the sample. This process ensured there was no bias in the results based purely on frequency. The resulting sample is shown in Figure 4.2 which shows that when $0.05 \leq Q \leq 0.12$, the formula take far longer to solve. While this range is slightly different from the results of the full dataset ($0.05 \leq Q \leq 0.13$) this can be explained by the reduced dataset.

In both the original and our extended analysis, we noticed that a huge number of instances — approximately 90% in the first version, and approximately 76%in the second version — finished in less than one second. While to a certain extent this is to be expected, as per the random clause-variable result[18], we wanted to be sure we had not generated a large number of instances that were trivial, i.e., solvable by simplification. Unfortunately it was impossible to do this with the initial experiment, as we did not record this information, however, we were able to utilise a subset of the second experiment's results that could not be solved by simplification. In doing this we found that approximately 40%were solved by simplification and were therefore excluded from the analysis. However even these instances considered alone, which will be discussed further in subsequent chapters, show a community related trend.

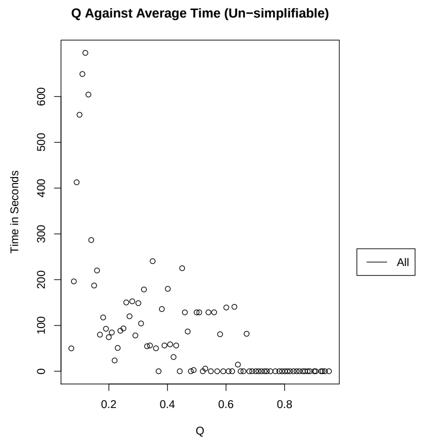
2 Results

As you can see from Figure 4.1 the results are very similar whether we include or exclude the instances that can be solved by simplification. The peak for both datasets is approximately $Q = 0.12$, the main difference is how dispersed the remainder of the dataset is. As would be expected with the instances that can be solved by simplification included the average time for the remainder of the instances decreases, making the trend appear stronger. It was interesting to us that only a little over half of the instances that finish in under one second were solved by simplification, this is likely because those fast non simplifiable instances had relatively few conflict clauses.

In addition to the small difference in distribution when excluding the instances that can be solved by simplification, we found that our original



(a) All random instances

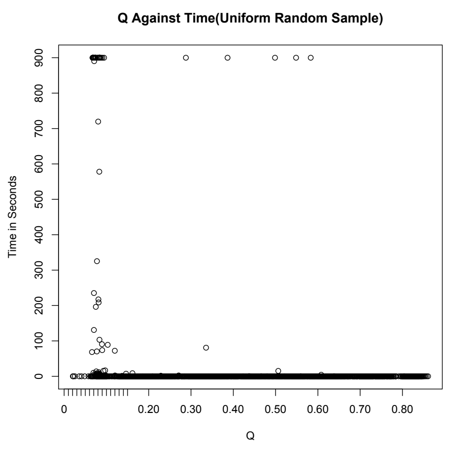


(b) Random instances unsolvable by simplification

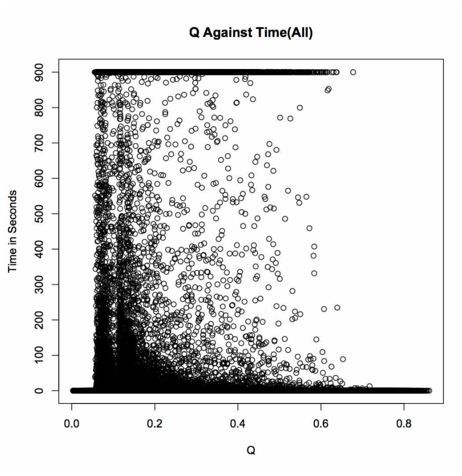
Figure 4.1: Q against average time for randomly generated instances

dataset presented almost identical results even when far more instances were considered. The overall result for that experiment was $0.05 \leq Q \leq 0.13$. The largest difference between these instances is the absence of any $Q < 0.05$ results in our second experiment. We think this is due to a minor error in the OL algorithm, which we fixed in this second experiment as discussed in the errata of [36].

It has long been known that the clause-variable ratio (CVR) in random instances is a factor of longer solve times in random instances. In [18] the authors show that there exists a phase transition between easy UNSAT instances, to "hard" instances, to easy SAT instances around CVR = 4.26. However, in our trials we have not found this to be the case. When considering the graphs in Figure 4.3b we noted that while the absolute peak of the CVR did reside around the CVR = 4.26, the range for "hard" instances was quite wide. Even just considering those instances that timed out, we found this range was $3.25 \leq \text{CVR} \leq 8.70$ with a relatively large number of those time-outs residing outside of the CVR = 4.26 range. Only 83 time-out instances had a CVR of between $4.2 \leq \text{CVR} < 4.4$ compared with 96 between $3.8 \leq \text{CVR} < 4.0$, as well as 193 $4.0 \leq \text{CVR} < 5.0$. While this is only considering the time-out data, we see a similar result when consider-



(a) Stratified Sample



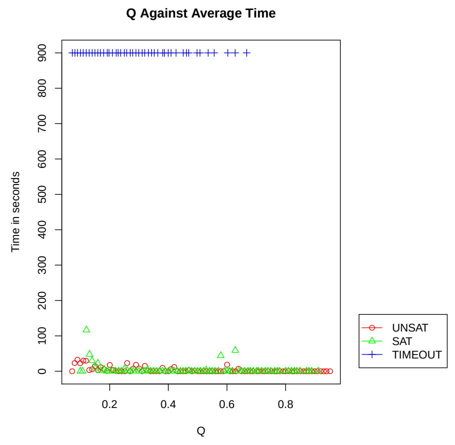
(b) All instances

Figure 4.2: A plot of Q against time, for the original dataset (550,000 instances)

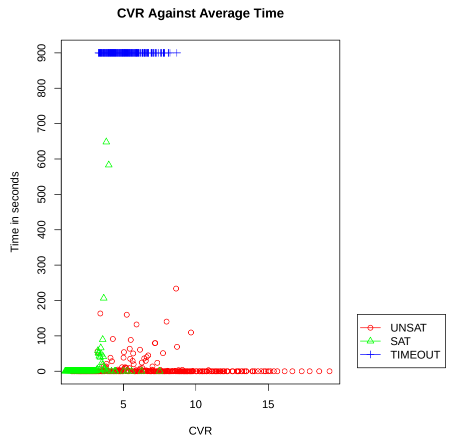
ing the satisfiable results, showing an absolute peak in average execution time at CVR = 3.8 with a peak in execution time for SAT instances existing between $3.5 \leq \text{CVR} < 4.0$. Strangely, we see an entirely different "peak" when looking at the un-satisfiable instances, where the hardest instances reside around CVR = 8.0, but the majority of slightly easier instances reside around CVR = 6.0. This result is more in line with [14] where the authors show that the phase transition may start earlier than anticipated. Their figures show that when CVR = 3.8 the solution time of a formula becomes exponential in the size of the variables; whereas at CVR = 3.7 the solution time is polynomial in the size of the variables.

When we consider the same plot of Q against average time Figure 4.3a we see a similar pattern, a large number of time-outs — 700 out of a total of 839 — reside between $0.05 \leq Q < 0.16$. However, the peak of non time-out instances for both SAT and UNSAT is much lower between 50 and 100 seconds — but both within the target range of Q. Additionally, there is no phase transition when considering Q. SAT and UNSAT instances appear throughout the range of Q all the way from 0.05 to 0.95 (the complete set).

In addition to the phase transition result, Ansotegui et al have shown a correlation between slow instances with a CVR = 4.25 and slow instances with a $Q = 0.15$[6]. When looking at the frequency of Q within the phase



(a) Q against time



(b) CVR against time

Figure 4.3: Q and CVR against time, displayed according to solver result

transition range ($4.23 \leq \text{CVR} \leq 4.27$), we can confirm that far more instances have a bad Q than all the other ranges together, we found that 36 instances had $Q \leq 0.14$ and 26 had $Q > 0.14$. However, when looking at the same frequency distribution of Q, but outside the phase transition range, we found there were still a large number of instances that show bad Q and slow solution time, despite them not being within the phase transition range.

One possible explanation for this is that in our experiment we are working on a relatively low time-out of 900 seconds (used due to its use in the 2012 SAT Competition), however it is possible that a different peak would have been seen if we had allowed instances to continue past this point. In this situation it is possible that the instances which timed out in this experiment, and were outside the $4.23 \leq \text{CVR} \leq 4.27$ range would have been solved, and those instances within this CVR range would still have timed out. Thus showing a more distinct peak in this range.

Another potential explanation for hard instances outside of the phase transition range could be the scale of the results; in [14] the results state that a CPU time limit of 10,800 seconds was imposed. This implies that they were measuring CPU time rather than wallclock time. As mentioned in our experimental setup, we use wallclock time in all our experiments, as it more accurately represents the time spent solving a solution, and as our results in Chapter 7 show, the speed at which an instance is solved does not rely entirely on the CPU speed of the machine; meaning that in some cases RAM and cache sizes/speeds are also a factor. These factors are not considered when measuring only CPU time. In addition to the difference in time measures, we also noted that the authors opted for an analysis of the median time. As the majority of our results are shown with time-outs in a separate trend, utilising the median was not required to handle the bias a single high result may have on a dataset. We did also note, despite the author's assertion that the median and the mean are typically quite close to one another, that our median (4 milliseconds) differs greatly from our mean (160.71 seconds). Additionally, the CPU speed (and other factors) of the solving machine in our experiment are vastly different from theirs; it is very likely that instances that would have timed out in their experiments would have completed successfully in ours, this combined with the huge advancements in SAT solver technology over the past decade, makes it difficult to directly compare our results with theirs. To support this, the result in [14] states that the time complexity of the solution varies with the choice of solver.

A further possible explanation for this difference is that the authors of[18] noted that the phase transition range is very tight around CVR $= 4.26$, where we found that it was much more dispersed, though still centring around this range. This would explain the discrepancy where a large number of instances outside of the phase transition range described therein still had a slow solution time. To test this hypothesis we looked at the same frequency of Q for instances outside our range of "hard" CVR ($3.8 \leq$ CVR ≤ 8.0) and found there were approximately 30% fewer instances with bad Q outside of this range. However, we still saw a significant peak in execution time of those instances with non-bad CVR and bad Q. This leads us to the conclusion that in our dataset, CVR and Q are not tightly correlated, and in fact are separate predictors of solution time. If CVR and Q had been tightly correlated, it would be unnecessary to compute Q as CVR would be an estimator of it. Additionally, it would not be possible to combine the factors for a more accurate model.

In order to confirm this we plotted a graph showing the combined Q and CVR "scores" where individual scores represent the instance's distance from $Q = 0.13$ and CVR $= 5.0$ respectively, these were the values we found optimal for the full dataset. We found slightly different values optimal for instances that could not be solved by simplification ($Q = 0.09$ and CVR $= 5.15$), however the result is very similar. We then plotted the product of these two scores against average execution time for instances, showing different trends for the

SAT vs UNSAT instances. The results of which are shown in Figure 4.4, where the x-axis shows the combined “scores” and the y-axis shows the average execution time in seconds for instances that have the same “score”. This figure shows that instances with a low “score” are typically harder to solve. We can also see that every instance, which did not finish trivially fast, resides within this range:

$$\text{score}(\text{Q}) \cdot \text{score}(\text{CVR}) < 0.6$$

. We also show a good distribution of SAT vs UNSAT instances within and outside of this range, the plot where time-outs are not separately tracked shows an even more distinct pattern, but has been excluded from this report for the sake of brevity. In addition to the community modularity being important, we noted that the number of communities was similarly important, arguably in a more predictable manner. We show that there is a peak in average execution time when a random instance is comprised of approximately 130 communities. This peak, shown in Figure 4.5, is not as sharp as the Q transition, but has a more analysable symmetric curve. Though we did note that no instance containing more than 240 communities took longer than one second to solve, despite the fact that some of these (between 240 and 340 communities) were not solvable by simplification.

In Figure 4.6 we show the result of adding “number of communities” score to our model where the score is defined as the number of distance of communities in an instance away from the worst case 130. We see that it has a small, but significant affect on the graph, where we compress the hard instances into a range covering 5% of the graph, when compared to 7.5% of the range for the model containing only Q and CVR. This small change makes sense when you consider Figure 4.7a, showing the number of communities in an instance on the x-axis, and the CVR of the instances on the y-axis, this figure shows there exists a relatively strong overall correlation between CVR and the number of communities, though this correlation does weaken within the range we observed as the peak range of number of communities, between 40 and 240 communities. Hence the improvement we observed.

These results show us that by including the number of communities, Q and CVR we have a model for identifying hard random instances that contains very few false negatives. However, we later discovered that while including the number of communities in this model reduces the false negatives, it increases the false positives — this new model incorrectly identifies many easy instances as hard — this result is discussed further in Chapter 6.

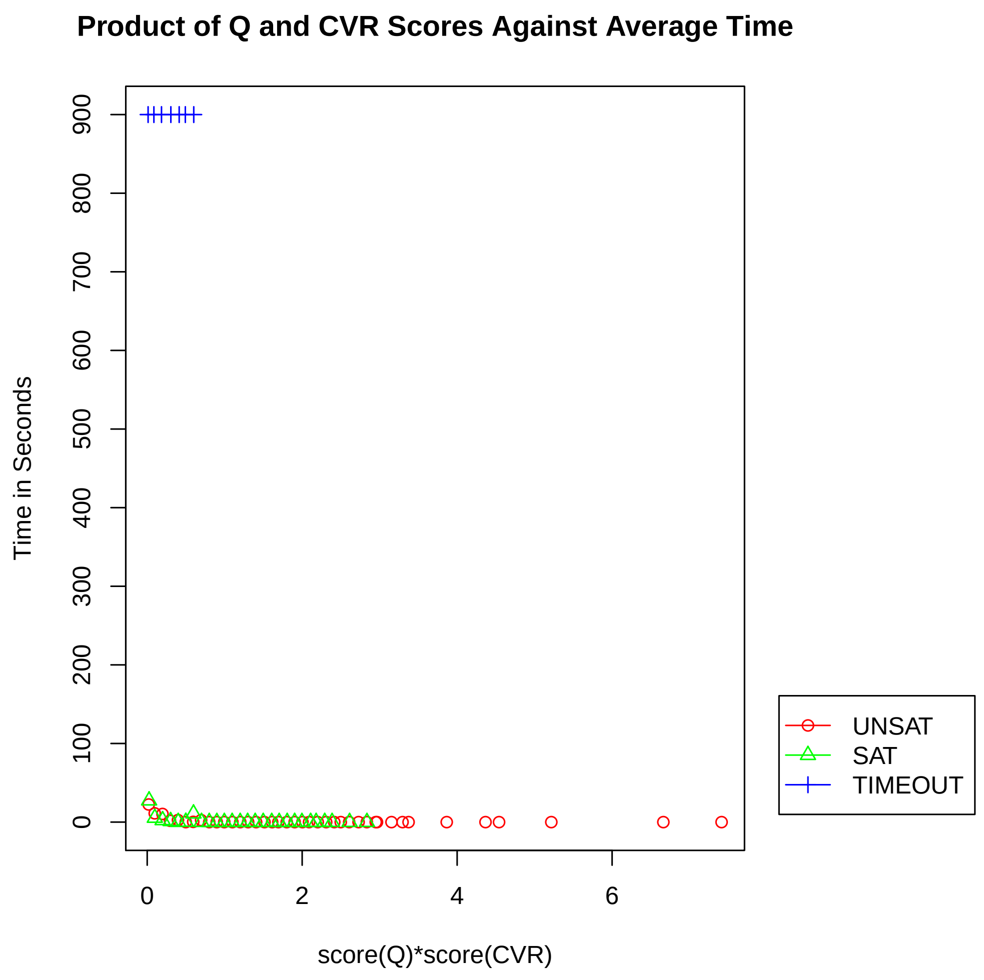


Figure 4.4: The product of Q and CVR scores against average execution time

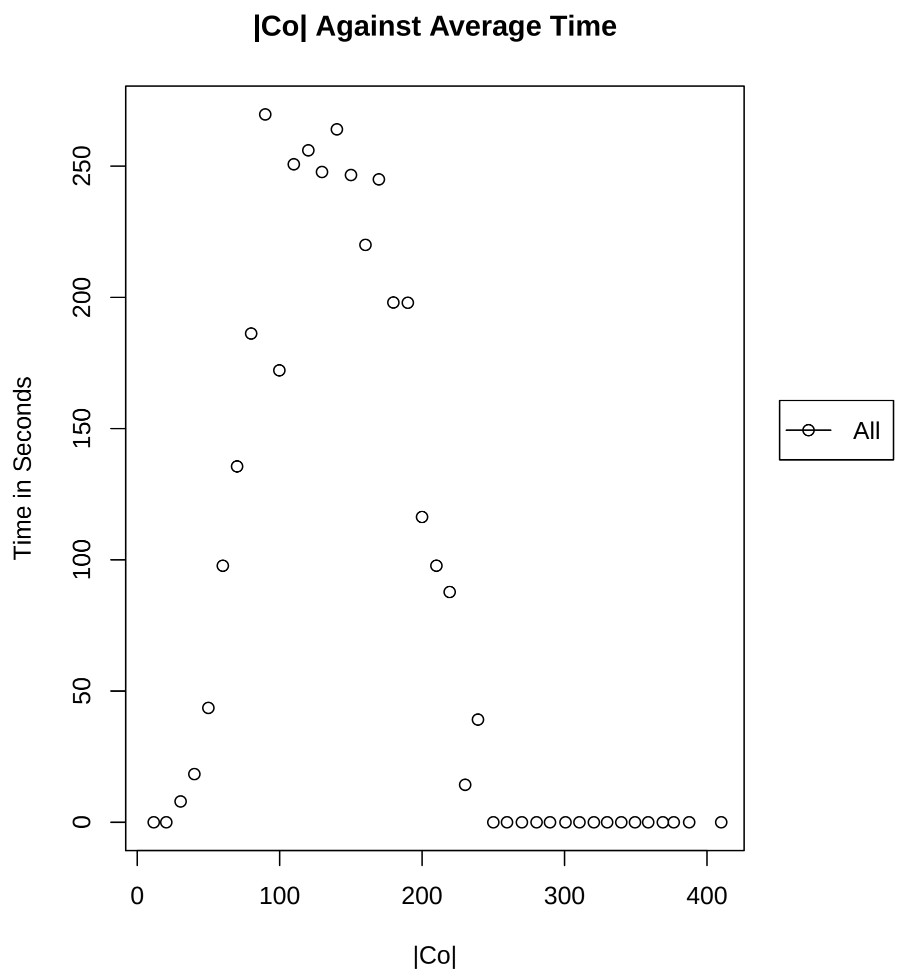


Figure 4.5: The peak of $|Co|$ against time

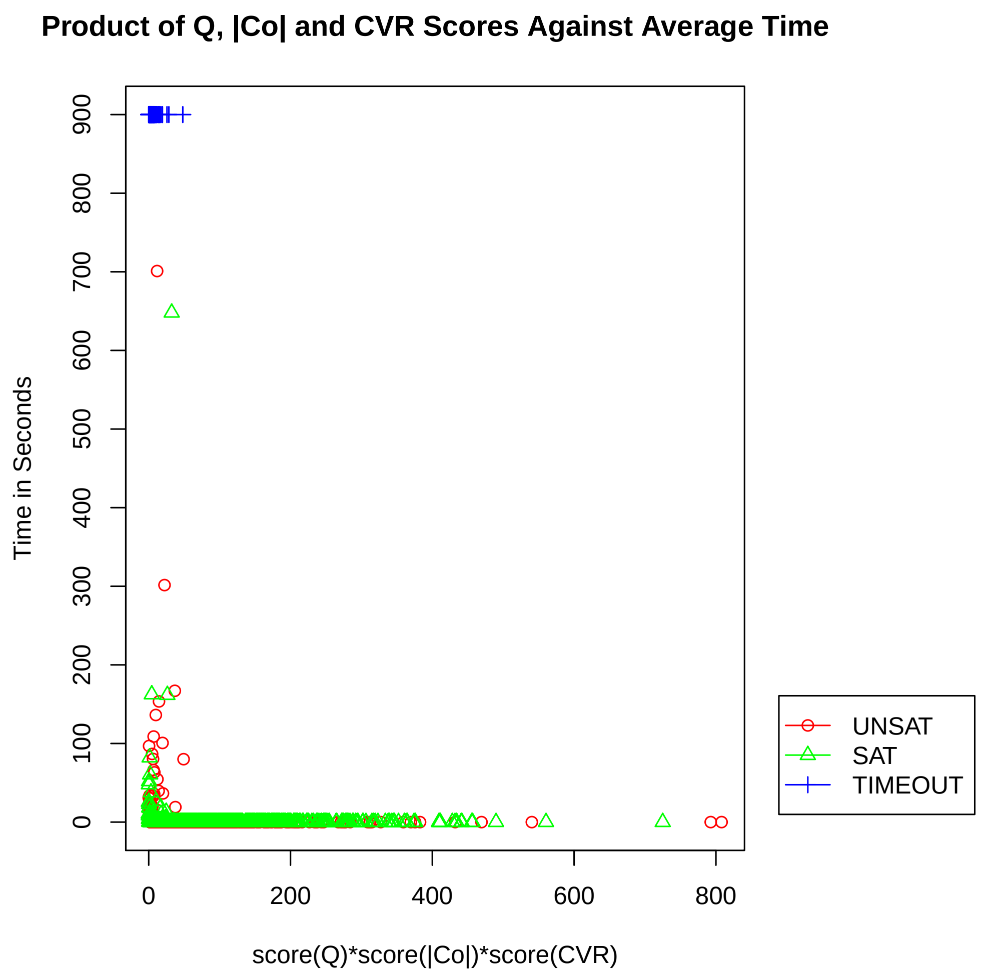
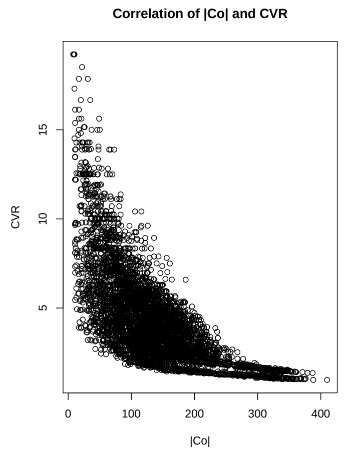
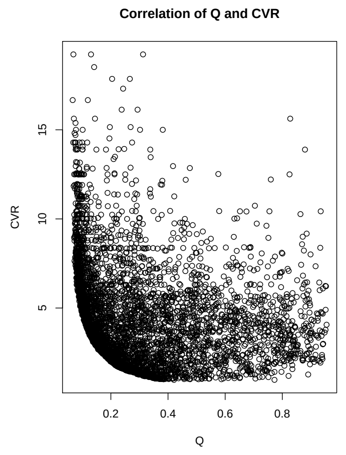


Figure 4.6: The product of Q, $|Co|$ and CVR scores against average execution time



(a) $|Co|$ plotted against CVR



(b) Q plotted against CVR

Figure 4.7: Number of communities plotted against average time for random instances

3 Conclusion and future work

In this chapter we have described the phase transition of random SAT instances with respect to community structure, we have compared the quality of the CVR and community structure based models, and shown that a combined model including community structure, number of communities and CVR has the least false negatives with respect to characterising instances as hard/easy.

Our future plans for this work are to see if the community structure and combined models are better able to characterise instances across different implementations of the CDCL algorithm than the pure CVR model is able to.

Additionally, our discovery that that instances with an average clause length of below 2.9 are easy will also be explored further to determine if this is purely an artefact of our generation technique.

Chapter 5

SATGraf: A tool to view the evolution of a SAT instance during solving

Motivated by these discoveries [6, 7, 4], we built SATGraf, a visualization and evolution tool that displays the structure inherent to Boolean formulae and shows how this structure is morphed by modern CDCL SAT solvers as they solve these formulae. SATGraf takes as input a Boolean formula, constructs the corresponding variable-incidence graph, finds the inherent community structure, and displays it. SATGraf also shows how CDCL solvers morph the input formula, in some cases solving it community by community.

While there are other tools that help us visualize the graph structure of SAT formulae [41], they do not display their community structure. The motivation for us to build SATGraf was to get clues as to how CDCL solvers exploit structure, and indeed it led us to many falsifiable hypotheses that we subsequently tested.

1 Implementation

SATGraf is implemented in three phases:

Phase 1: First, it converts an input Boolean formula (represented in the standard DIMACS format) into its corresponding variable-incidence graph.

Phase 2: Second, it computes the community structure based on user's choice of either the Clauset-Newman-Moore (CNM) algorithm [13] or the online (OL) community algorithm [45]. We modified the CNM algorithm to make it more efficient, and it is the default community discovery algorithm in SATGraf. The original CNM and OL algorithms and our modifications are described in section 3.1.

Phase 3: Finally, SATGraf uses either a modified version of the force-directed graph layout algorithm by Kamada and Kawai called the KK algorithm [22] or Fruchtermon and Reingold algorithm [17] to render the community structure. The original algorithm and our modifications are described in section 3.3. In addition to these two graph layout algorithms, it is possible to view the graph as a grid structure; this is a much more efficient, but less visually appealing way of viewing the community structure. It is possible to extend the drawing behaviour to allow for any arbitrary layout algorithm. For example we have implemented a simple layout algorithm to view round based functions such as MD5 and SHA-1.

Additional As an optimisation for repeated viewing of the same input formula, the user is able to save the viewed formula as a JSON file with community and positional information included. Loading this JSON file is much faster than the associated CNF formula; however the file size is significantly larger.

Figure 5.1 shows the graph generated by SATGraf for two instances from the SAT 2013 competition [1]. Figure 5.1a is an industrial instance, and Figure 5.1b is a randomly-generated instance. Each distinct colour is mapped to a unique community to make it easy to discern the various communities. The colour white is reserved for inter-community edges. As is evident, the industrial instance has many more distinct communities that can be neatly partitioned, while the randomly generated instance has one big community that has lots of edges linking it tightly, with very few discernible communities. SATGraf presents the evolution of the formula by interacting with a modified version of Minipure[42], which periodically generates a new graph of the SAT formula as it is being solved. The period is user specifiable and corresponds to the number of conflicts (n) discovered by Minipure. The output from Minipure after n conflicts is the formula with all satisfied clauses removed.

The remaining clauses do include the learnt clauses generated by the solver. It is worth noting that the tool is not restricted to Minipure.

2 Features

SATGraf not only provides a visualisation of a static representation of a SAT formula, but also the ability to watch it transform whilst it is being solved. We currently have two static examples of this evolution. The first shows the partial evolution of aes_16_10_keyfind_3 [34], unfortunately at this point only a partial evolution is possible due to the large GIF size of the generated file (approximately 70MB). The second example shows the full evolution of the trivial toybox example [33]. There are two separate mechanisms for this, depending on the goal of the user. The first evolution mechanism is accomplished by having the solver dump the state of the formula regularly (every n conflict clauses, where n is user specified) for re-processing. While our experiments all focused on MiniSAT 2.0, it would be possible to implement the required changes on any SAT solver for which source code is available. While currently we don't have an open, published API for this (as the implementation is changing as we decide which features to support) it would be possible for a developer to implement a solver matching the specification found at [30]. This evolution is viewed as a series of slides with the nodes and edges placed and coloured based on the community structure of each of the dumps. This mechanism allows the user to view the way the community structure of the entire formula evolves over time; thus discovering if as conflict clauses are added the formula becomes more or less community clustered. However, as the community structure of the graph is re-calculated at each stage, it is difficult to see how the solver interacts with individual communities.

The second mechanism communicates with the solver via variable assignments. Each time a variable assignment is changed (either by decision, implication or back-jumping) SATGraf updates the graph by either colouring, or adding/removing the node that represents the assigned variable. This is done based on user provided flags. In doing this it is possible to see how the solver interacts with individual communities without the overall structure of the graph changing.

To create an effective tool we decided to build a dynamic tool instead of outputting a static image or series of images representing the community structure and evolution of the formula. This allows the user to hide irrelevant information, based on that user's preferences. A key component of this interface is to allow the hiding of nodes and edges based on various factors, these include the names of the variables if provided. These names, passed in through comments in the DIMACS file can be grouped and a single node may be present in more than one group. We found this a great help when analysing formula representing the MD5 hash collision problem. By grouping variables based on their relevance to specific words of the message and hash we were able to provide the SAT solver with hints as to which intermediate round variables to focus its attention on when solving. This approach is similarly useful to any SAT instance generated from code rather than manually or randomly generated.

In addition to the ability to hide nodes and edges based upon variable naming patterns, the user can hide entire communities, either just the internal edges, external edges, nodes or any combination thereof. This allows the user to clear the visualisation of entire communities where the edges may obstruct the view to important areas that may otherwise go un-noticed. It is also possible to hide specific nodes and edges.

In addition to the ability to hide nodes and edges, the ability to scale the graph in an efficient manner provides users with the benefit of seeing the entire formula displayed, whilst still being able to view in high detail specific areas of the formula.

Finally, to ensure an efficient tool for analysis we provide users with the option of saving the current session to file. While these files are in most cases far larger than the ones containing their original formula it also provides a way of quickly restoring a session, without having the re-run the community detection or placement algorithms.

3 Algorithms used in SATGraf

A number of algorithms are used in SATGraf, for graph layout and community detection. The modular design of SATGraf allows external development of additional algorithms for these tasks, and the simple conversion of existing implementations.

3.1 CNM Algorithm

The purpose of the CNM algorithm [13, 24] is to identify communities in a graph. Initially every node is its own community. For every pair of communities, the pair that maximizes the modularity score is merged into a single community. This process is repeated until the modularity score cannot be maximized any further.

3.2 OL Algorithm

The online community detection algorithm by Zhang et al. [45] has the same purpose as the CNM algorithm and uses the same metric — modularity — to determine the quality of the community structure. However, the difference between the two methods is that the online method has a linear worst-case time complexity in the number of edges, compared to that of the CNM algorithm which is worst-case exponential in the number of edges. For real-world examples this led to 600-fold speed-up [45]. The only drawback to the algorithm is that the modularity found by the algorithm, in our experience, is strictly worse than the modularity found by the CNM algorithm. Consequently, the graphs' communities will not be as strongly connected and may vary each time the algorithm is ran.

3.3 Kamada-Kawai Algorithm

The Kamada-Kawai algorithm [22] is a force-directed graph drawing algorithm for general undirected graphs. It assigns "forces" to edges and nodes based on their position relative to each other and then calculates the *repulsive and attractive forces* between them to reposition them. The KK algorithm's complete explanation can be found in this paper [22]. Several small modifications were made in our implementation of the published algorithm. Firstly, the KK algorithm requires that graphs be fully connected. Unfortunately, this is not always the case for SAT instances. Hence, we implemented a system to create *proxy* edges between nodes for the purpose of the layout. These are removed after the position of each node is determined. The second, and more important modification was to implement a wrapper for the KK algorithm that takes advantage of the community structure present in most large SAT instances to reduce the complexity of the algorithm. Rather than submitting an entire graph to the script for processing, each community is submitted in turn. These communities are then laid out relative to each other using an approximation that replaces each community by a hyper node in the graph. Once each hyper nodes position is determined, the communities are brought back and their positions are scaled relative to each other's size. This results in a decrease in the algorithm's complexity due to the far smaller number of communities versus nodes (usually less than 200 communities are present). This yielded a performance improvement by a factor of three.

3.4 Fruchtermon and Reingold Algorithm

The Fruchtermon and Reingold Algorithm [17] is a force-directed graph drawing algorithm for general undirected graphs. It ensures that nodes which are connected are placed close to one another, and that all nodes (whether connected or not) are not placed too close to one another. It is based on an observation from physics as to the structure of atoms, where when two particles are within one fento-meter (fm) they are strongly attracted to each other, when this distance closes to 0.4fm, the attraction force reverses to repel the particles - stopping the nuclei from collapsing.

A similar approach is used in the FR algorithm where the nodes are atomic particles exerting attractive and repulsive forces on one another. The FR algorithm's complete explanation can be found in this paper [17]. The desired distance between nodes is determined by the number of nodes in the graph and the area selected for the visualisation, as such by varying the size granted it is possible to make a tighter or more diffuse visualisation.

4 Results

SATGraf has been tested on several industrial, hard combinatorial, and randomly-generated formulae from the 2013 SAT competition[2]. The time taken to display the community structure of a single instance grows with the size of the input formula. This is to be expected due to the nature of the community detection and placement algorithms. The resulting images using the CNM community detection and KK layout algorithms, can be seen in Figure 5.1. As mentioned earlier, the right half of the Figure 5.1 is a randomly generated SAT instance from the 2013 SAT competition, whereas the left half of the Figure 5.1 is the graph of an industrial SAT instance from the same competition. The community structure of the industrial instance has much better modularity than the one for the randomly-generated instance. This can be verified both visually and through the modularity measure of quality of the community structure: the industrial instance has a modularity of 0.437, while the randomly-generated one has a modularity of 0.132. Their solve times using Minipure are also different; The industrial instance takes 0.076 seconds to solve compared to the randomly-generated instance which times out after 5000 seconds.

SATGraf's evolution by decision level feature is partly shown in three pictures in Figure 5.2. The SAT instance here is called *aes_16_10_keyfind_3* that can be downloaded from SATGraf website [29]. The entire evolution of *aes_16_10_keyfind_3* can be found at [29]. We chose this SAT formula since it is a good representative of an industrial application of SAT solvers. Furthermore, this instance is small enough so that we can actually show, in a timely manner, how the SAT solver dynamically morphs its graph (the instance and the generated learnt clauses). Figure 5.2b shows the formula after several hundred decisions have been made. Figure 5.2c shows the formula after a few more decisions. The "new" cluster of inter-community edges (coloured in white) represent the learnt clauses that caused this particular back jump. Observing the evolution showed an interesting trend, namely, the removal of entire communities during the solving process. This evolution can be seen when going from the graph of the original SAT formula in Figure 5.2a, to Figure 5.2b. It is easy to see that some of the communities have completely disappeared by the absence of their associated colour, i.e., the corresponding clauses have been satisfied.

5 *SATBench*

During the development of this work we decided to setup an online repository for SAT instances. We chose to do this for a number of reasons, partly due to the large number of instances considered and the diversity of the sources of these formula, as well as the varied experiments performed in this work. But also we found that current repositories such as [1] were not able to provide the level of detail we desired when viewing the formula. Enter *SATBench*, a platform for hosting information about large numbers of SAT instances taken from a variety of sources primarily various SAT competitions, but also custom randomly generated instances and pre-simplified versions of some instances. The site allows for hosting an unlimited number of factors for each instance and currently provides approximately 30 unique factors for each formula stored. These factors can be user specified, and are available publicly. In addition to storing a large number of instances, and factors. We also store information regarding individual executions of an instance on specific versions of solvers (with arguments provided) and on different solving machines. We currently host trials from several different variants of solvers, and approximately 30 different solving machines. It is possible to create custom datasets of formulae, workers and solvers that match the user's specification.

We hope that providing this information publicly will encourage other researchers to provide details about the formula in their experiments and reduce the amount of work necessary to replicate and evaluate published experiments, as well as reduce the work required to expand on current results. For this reason it is possible to "publish" datasets, making the entire set downloadable in CSV format publicly.

In addition to hosting information about a large number of instances, we have also made it possible to view the static community structure of certain instances (all those used in experiments from Chapters 3, 6 and 7). Unfortunately we are not currently able to provide the evolution for these instances online, as it requires a connection to a local instance of MiniSAT (or another appropriately modified solver).

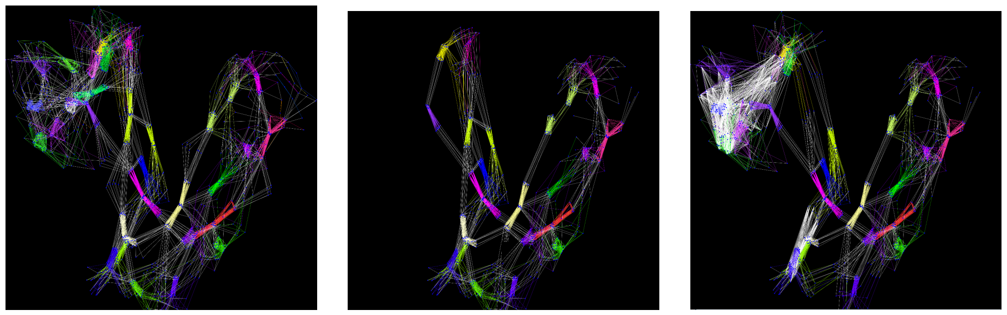

(a) Initial formula state (b) After many decisions (c) After a back jump

Figure 5.2: Partial evolution of the *aes_16_10_keyfind_3* problem

6 Utility and limitations of SATGraf

We have found SATGraf useful in a number of situations, particularly when analysing "hard" SAT instances (whether exhibiting good community structure or not) and have leveraged this tool to provide insight on the utility of variable selection heuristics such as VSIDS. We are also in the process of developing a parallel SAT solver based on the results provided to us by SATGraf.

However, SATGraf does have some limitations. We have found that whenever a SAT formula creates a large number of edges — either from a large clause set or from a single enormous clause — SATGraf struggles to either build the graph or calculate the community structure. Similarly, if the instance takes a particularly long time to be solved under the MiniSAT solver, the evolution may result in large overheads. We have found that instances under 100MB in size typically work well, however some of those approaching this limit that have a small number of particularly large clauses also cause problems.

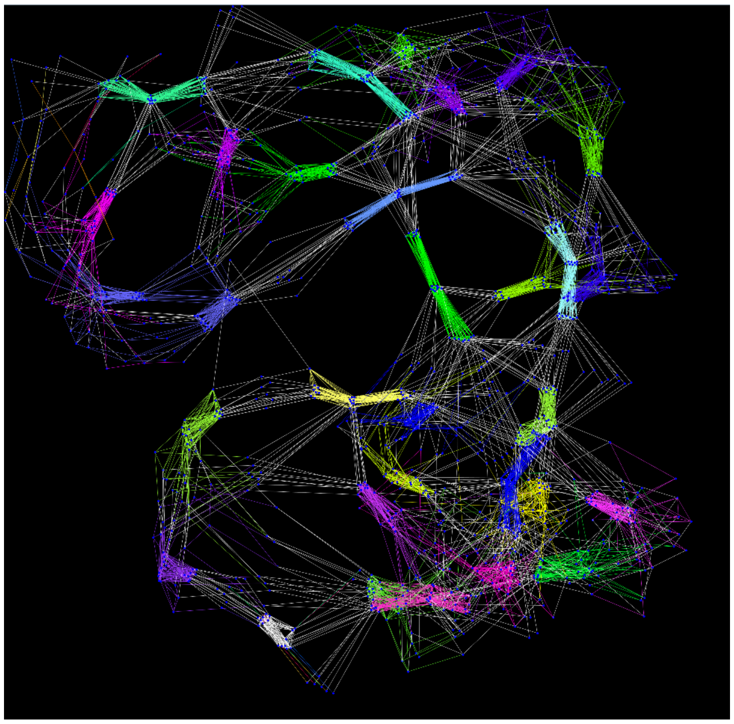
The development of SATGraf provided some interesting challenges. An example of this was separating the visualisation code from structural into separate packages, which had to be re-factored several times. Similarly, developing a system that allows for any community detection or layout algorithm, also proved challenging. Despite these challenges, the overall development process went surprisingly smoothly, development of different interpretations of the graph representation (e.g., the implication graph viewer) went quickly, as did development of the second evolution mechanism.

7 Conclusion and future work

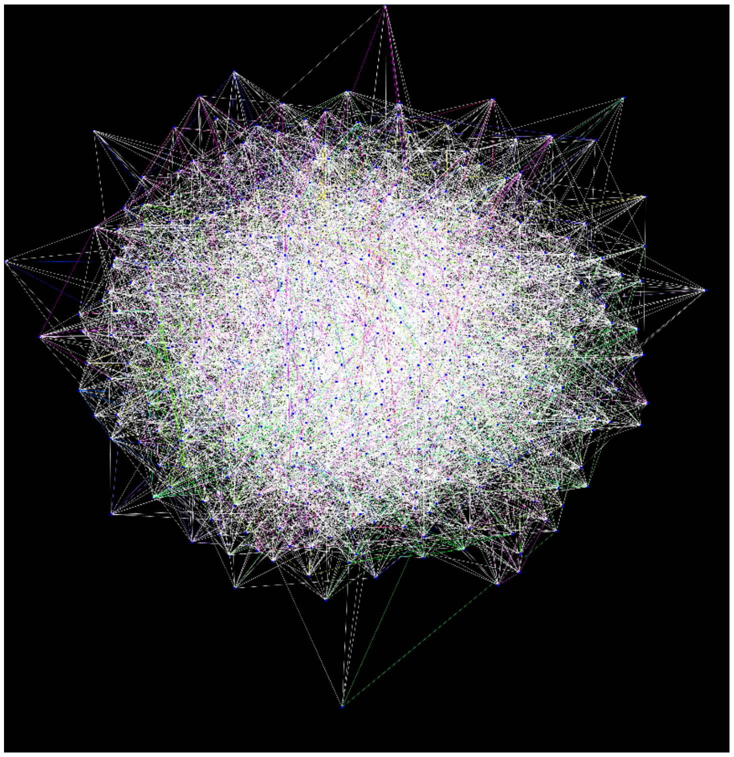
In this chapter we have described SATGraf, a tool for visualising the evolution of the community structure of SAT instances during solution time and SATBench, our online repository for storing SAT instances.

In the future, we hope to integrate more options of solvers into SATGraf, but also more decision heuristics into the solvers we support, so a user may view the evolution with respect to different heuristics.

In addition to this, we are in the process of implementing GPU based layout algorithms to further improve the performance of SATGraf.



(a) Industrial instance *aes_16_10_keyfind_3*



(b) Random instance
unif-k3-r4.267-v421-c1796-S4839562527790587617

Figure 5.1: Community structure of instances from the SAT 2013 Competition.

Chapter 6

Survival analysis of different classes of SAT instances

Survival analysis is used in many disciplines to determine the likelihood of a test completing within a certain time period. It is often used in medical trials to measure the time a patient lives after treatment, or in engineering to measure the time until failure for a component. It is particularly useful where observed trials do not complete in a given time frame. This is the case with SAT solvers as in many cases (particularly those in the SAT competition) the instances time out, or (in a few cases) result in memory errors before completion, this type of data is called right-censored. However, survival analysis is not able to analyse complex models in the same way that a linear regression is; it is best used to compare different segments of data, for example the time taken to find a set of formula SAT vs the time taken to find a set of formula UNSAT.

Formally, the survival analysis used here is a Kaplan-Meyer estimate as described in [23] which uses the function $s(p,t)$ to return the probability that a member of a population p will have a lifetime (in our case solution time) exceeding t. Let the observed times until solution of the N_P sample members be
$t_{11} \leq t_{12} \leq t_{1N_1}$
$t_{21} \leq t_{22} \leq t_{2N_2}$
...
$t_{P1} \leq t_{P2} \leq t_{PN_P}$

Corresponding to each t_{ij} is n_{ij} the number of instances that are likely to be solved ("at risk"). Instances for population i just prior to t_{ij} and d_{ij} the number of instances that were solved at exactly time t_{ij} for population i. This allows for non-uniform distribution of events across time, where a number of formulae may be solved at time $= x_1$, no formula are solved at time $= x_2$, and then several more solved at time $= x_3$ and time $= x_4$.

In more general terms, survival analysis is a technique to help evaluate different measures of structure. It has helped us identify which measures of structure correlates strongly with hardness of solution. While full regression is ideal in many cases for predicting solution time, it — like all analysis techniques — relies on the quality of the data. For regression analysis time-outs are particularly problematic, however survival analysis is designed specifically to use time as the prediction factor and is designed to handle time-outs. In addition to this, the temporal nature of survival analysis allows us to identify not only which classes of instances are hard, but at which point different classes of instance are most frequently solved. While it is not possible to make concrete predictions based on this, it is possible to predict that an instance belonging to class x that doesn't finish within time t will not finish before a pre-determined time-out. This will be discussed in more detail throughout this chapter, with numerous graphical examples.

As an example, consider the data presented in Table 6.1. The column time is the response variable and the columns age and drug are different potential response variables. The censor column states whether an observed reading is right censored (0) meaning that the observed end time was when the subject withdrew from the study (or in our case, the solver timed out). It is then possible to calculate the survival data for this dataset using two simple Equation 6.1. In each equation i is the current event (numerically indexed from 0), where an event is the death of a participant (or solver finding a solution). As such, t_i is the time at event i, d_i is the number of deaths (or solutions) at event i. In the case where there is no right censored data at an event, n_i is the number of participants prior to event i. When right censored data is present n_i is the number of participants minus the number of participants lost due to censoring (time-out or error-out solvers). The result of this equation on the provided dataset is given in Table 6.2. For more details of

time	age	drug	censor
3	30	1	1
5	46	0	1
6	35	1	0
8	30	1	1
22	36	0	1

Table 6.1: Example survival analysis dataset

time	survival
3	0.8
5	0.6
8	0.3
22	0.0

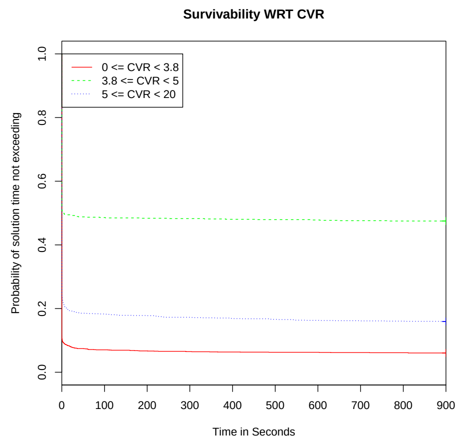
Table 6.2: Example survival probabilities

this example, see [11].

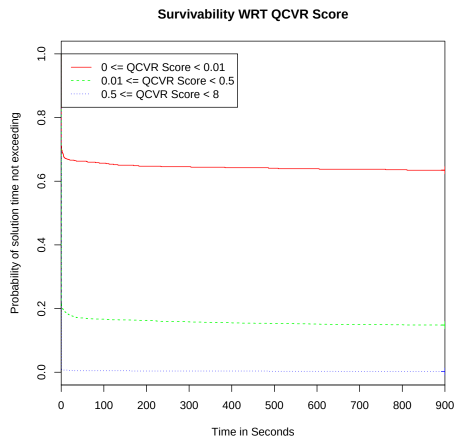
$$\hat{s}(t) = \prod_{t_i < t} \frac{n_i - d_i}{n_i} \tag{6.1}$$

1 Experimental setup

Two sets of data are used in these experiments, the first are the same randomly generated instances (approximately 4887) from Chapter 4. The second set of data is taken from the 2011-2014 SAT competitions in the application specific and crafted/handmade categories. Only instances with a dimacs (comments excluded) file size of < 100MB were used and a time-out of one hour was imposed. The former restriction is due to the memory intensive nature of determining the community structure of SAT instances, the latter is to ensure data was collected on as many instances as possible, rather than spending large amounts of time solving a single instance. We gathered data for 733 instances (377 application specific, 356 handmade/crafted) of which 453 instances (263 application specific, 190 handmade/crafted) finished successfully within the timeout of 5200 seconds on the Minipure solver in the SAT 2013 competition. We then ran 830 trials, where approximately 100 instances were ran twice on the same worker and solver. The purpose of this was to gather the amount of randomness present in the experiment purely from within machine variance. Fortunately in most cases this variance was very low (mean = 8.8 seconds, median = 0.07 seconds), with the exception of a single instance that had an enormous error variance. In the first trial it timed out, in the second it finished in approximately 500 seconds. We are working on the assumption that something triggered an early shut-down of the solver (possibly a memory error), and are considering this single instance to be an outlier. Timing was obtained by running the experiments on MiniSAT 2.0on an AMD FX-8350 Eight-Core Processor with 16GB RAM, running Ubuntu 12.04.



(a) Plotted against distinct CVR ranges



(b) Plotted against distinct Score(Q)*Score(CVR) ranges

Figure 6.1: Survivability plot of randomly generated instances from Chapter 4 highlighting the phase transition range

2 Results

As an initial trial of the effectiveness and validity of survival analysis, we utilised the data and result already presented in Chapter 4. Figure 6.1a shows three curves plotted, one for instances with a clause-variable ratio (CVR) of below 3.8 (where our results suggest instances become hard). The second shows instances with a $3.8 \leq \text{CVR} < 5.0$, and finally those instances with $\text{CVR} \geq 5$. As you can see from the curves plotted, the survival analysis has identified that instances with a CVR within the phase transition range described in [18, 14] have a much lower probability of being solved within the time-out, than otherwise. However, unlike the average plots presented in Chapter 4, the survival analysis has also identified that there are a large number of instances within the phase transition, approximately 50% that are solved in under one second. This will be discussed in more detail when looking at combinations of factors.

If we compare this to Figure 6.1b we see that the model we propose in Chapter 4 utilising the product of the distance of Q from 0.13 and the distance of CVR from 5.0, refereed to as the QCVR Score, we see a similar, but stronger trend where the slowest data $\text{QCVR} < 0.01$ has a lower probability of solving within the 900 second time-out. Additionally, this model includes less than 30% of instances that finish trivially fast within this category; clearly this model is able to better characterise hard instances than considering Q or CVR alone. The final model we propose in Chapter 4 includes the community score — the distance from the worst case 130 communities in a formula. In Figure 6.2 we show that including this factor in a survival analysis weakens the model.

This was a not anticipated prior to performing the analysis, as including the factor in a simple correlation suggested that it improves the model, by sharpening the observed peak and reducing the number of false negatives. In the QCVR score model, the "hard" instances are contained within 7.5% of the observed range. Including the community score in the model reduces this to only 5% of the observed range. However, when we view Figure 6.2 we can see that the slowest range of QCVRCom, instances with a score of below 0.08, includes approximately 31% trivial instances. This is an increased number of false positives. Reducing the range of this curve (for example to 0.07), increased this percentage. What this tells us is the peak of number of communities found in Chapter 4 is not the strongest value when considered from a survival analysis point of view, despite the fact that it was the strongest value from a correlation perspective.

When comparing the survivability of different formula a number of factors were considered, including: the modularity of the formula (Q), the number of communities in the formula $|Co|$ and the clause to variable ratio (CVR). Each of the survivability plots included show the probability of the solver still

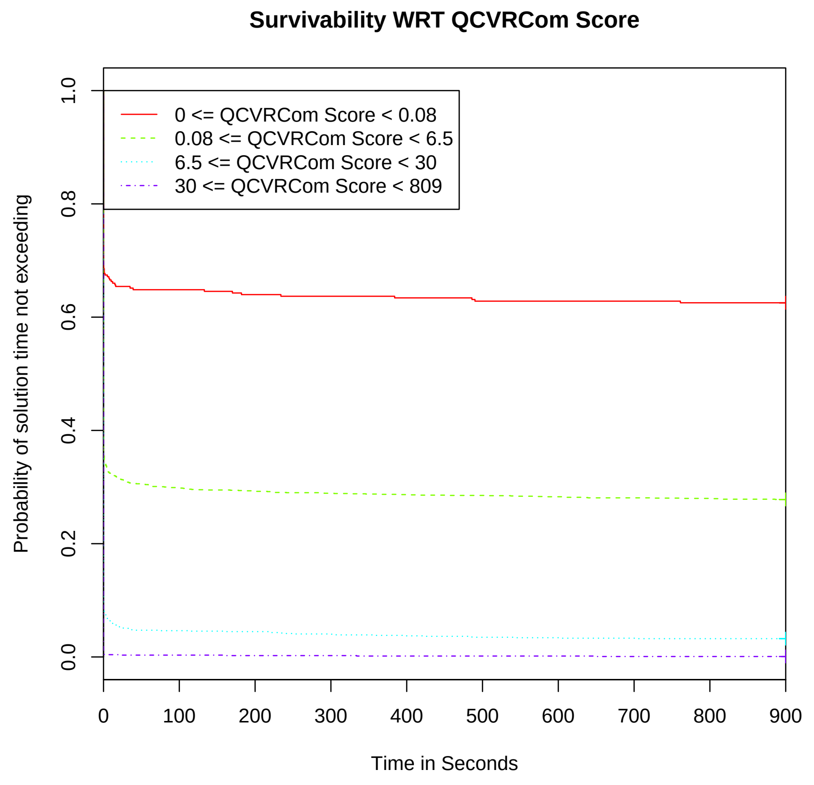


Figure 6.2: Survivability plot of randomly generated instances from Chapter 4 WRT the product of community, Q and CVR scores

running (the y-axis) at a certain point in time (the x-axis) based on formula exhibiting a value within a specific range of the measured factor (plotted as curves).

For each of these results, we looked for interesting patterns in the data through regular analysis, or linear regression and then isolated those into sets. In some cases this meant that only a small number of instances were in specific sets. To ensure this did not create a bias in the analysis, we repeated it with equal sized categories, that isolated similar sets, and saw very similar results.

While they initially appear somewhat complicated, the plots in Figures 6.3, 6.4

and 6.5 show some interesting facts. In each survival plot, the x-axis shows the time in seconds, the y-axis shows the probability of an instance not finishing within that time. Each curve plotted is a separate class of problem, as defined in the legend of the figure. A flatter curve indicates that few non trivial instances finished within the timeout. Similarly, a curve that appears to start low on the y-axis indicates a high number of trivial instances.

Firstly in Figure 6.3 we see that there exists two categories of Q that MiniSAT found hard to solve. This bimodal type distribution is not in keeping with our results from the random instances, and can be explained in part by the differences between application and hard category instances. This can be observed when looking at the same separate plots for application and hand-crafted instances, in Figures 6.4 and 6.5 respectively. From these plots we see that part of this "double peak" can be explained when separating instances into their categories. The crafted instances see a peak in execution time when $0.07 \leq Q < 0.13$, whereas the application instances see a peak in execution time when $0.19 \leq Q < 0.29$. In this case every application instance with a Q within this range timed out. This explains the double peak displayed in Figure 6.3. Unfortunately we still see a somewhat mixed message in application instances where those with a $Q < 0.07$ also have a very slow execution time. At present we are unsure of the reason for this secondary peak, however the presence of instances with a $Q < 0$ are due to a complete lack of community structure within those instances.

It is possible to use survival analysis to determine the probability of a solution being found at a specific point in time. For example, from Figure 6.3 we can see that at 1800 seconds, there is approximately a 72% chance that an instance with a $Q \geq 0.5$ will be solved, regardless of whether it is an industrial, or hand-crafted instance. This can be compared with instances with a $0.07 \leq Q < 0.13$, that have a 23% chance of being solved within this time. Further results are also possible, for example observing Figure 6.5 we can see that instances with a $Q < 0.5$ that don't finish within 2800 seconds, almost all time-out. This may be an interesting feature to invoke in a solver when estimating solution time. In this case it is relatively trivial, as the time-out is only 800 seconds further, however observing the same figure, we can see that instances with $0.07 \leq Q < 0.13$ that do not finish within 500 seconds, have a very small probability of being solved at all, approximately 1.6%

To determine if the clause-variable ratio (CVR) has any effect on so-

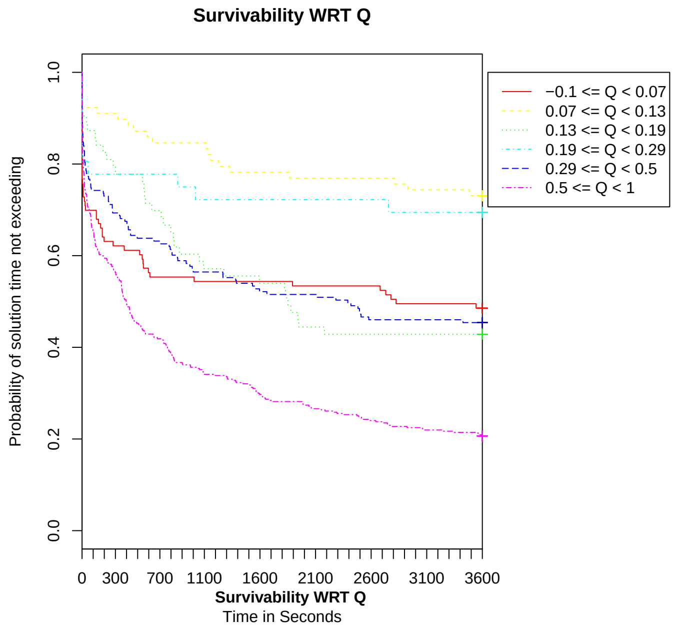


Figure 6.3: Survivability plot of application specific and crafted/handmade 2011-2014 SAT competition instances WRT Q

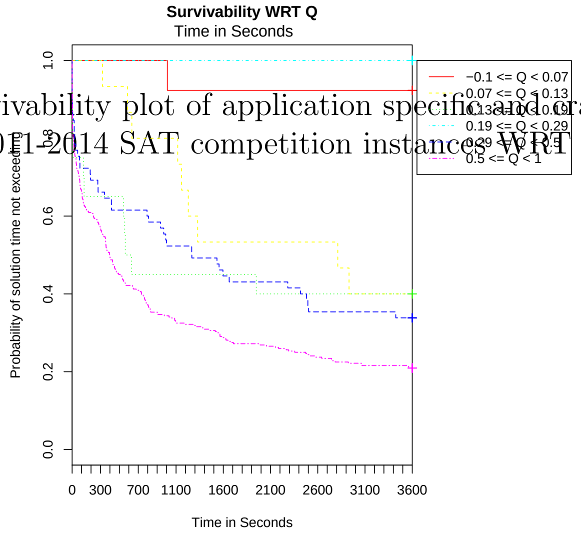


Figure 6.4: Survivability plot of 2011-2014 SAT competition application instances WRT Q

lution time for application or industrial instances, we performed the same survival analysis for CVR, that we did with Q. The results of this analysis are presented in Figure 6.6. From this we can see that while there is

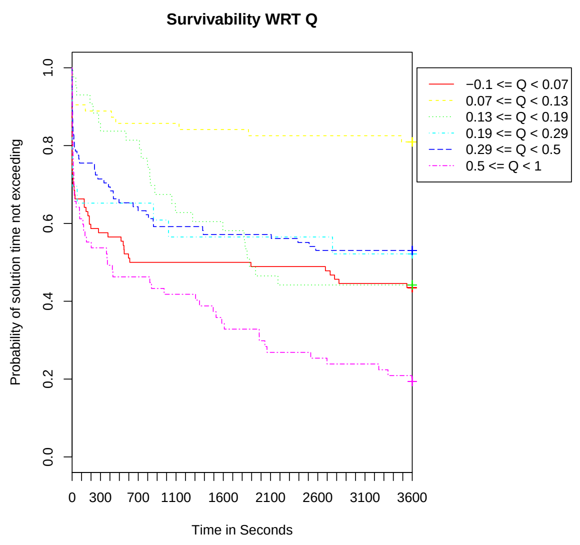


Figure 6.5: Survivability plot of 2011-2014 SAT competition hand-crafted instances WRT Q

no phase transition for application and crafted instances, the performance of these is heavily dependant on the clause-variable ratio. While the upper bound of the final dataset seems very large, it should be noted that only two instances in the dataset exhibit a CVR of greater than 417. Excluding these two instances does not change the result. When we considered the survival plots of the application and hand-crafted instances separately we noticed that application instances showed almost no difference between instances with CVR < 10 and instances with a $10 \leq$ CVR < 50 and those instances with a CVR ≥ 50 showed a relatively small increase in the probability of finding a valid solution. For example at 1800 seconds instances with CVR < 50 had approximately a 65% chance of having been solved, whereas instances exhibiting a CVR ≥ 50 had a 45% chance of having been solved. While this is a statistically significant increase, it is far lower than the increase seen when looking at hand-crafted, or combined instances. In the latter case instances with a CVR < 50 have approximately a 61% probability of being solved within 1800 seconds compared with a 21% probability for instances with CVR ≥ 50.

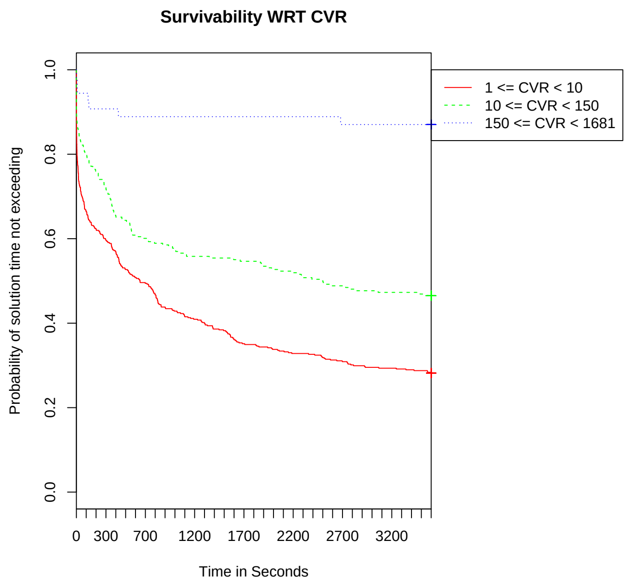


Figure 6.6: Survivability plot of application specific and crafted/handmade 2011-2014 SAT competition instances WRT CVR

3 Conclusion and future work

In this chapter we have describe survival analysis and shown that it is a useful tool for analysing SAT solver performance. We have used it to show that despite our combined CVR/community structure model's reduced number of false negatives, certain variants of it contain a higher number of false-positives. We have also identified different classes of SAT instance that are hard for SAT solvers, described using different input characteristics such as the number of communities and CVR.

In the future we wish to extend our work here to discover complex classes of instances. By which we mean a class that is not based on ranges of a single input factor, but combines multiple input factors in a more comprehensive manner.

Chapter 7

Effect of machine parameters on SAT solver performance

While performing the numerous experiments here, we compared results gathered across several different machines and noticed that performance of the same solver, on the same instance varied greatly across the machines we solved them on. While this in itself is not abnormal, we found that instances solved on machine X were not always faster or slower than the same instances solved on machine Y. Clearly certain aspects of a machine's physical configuration are better suited to solving different types of SAT instances. In this chapter we explore which of these characteristics are the determining factors for different types of instance.

1 Experimental setup

From the outset we were convinced that memory size and speed, CPU clock speed and cache size were going to be significant contributing factors, however as shown in [38] memory layout as well as other factors can have significant impact on any execution times, including that of a SAT solver. For this reason we decided to utilise DataMill [39], a platform that automatically varies system level properties on a heterogeneous set of hardware. By utilising this platform, combined with additional machines with more specific configurations, we were able to generate usage results for the solve time of 26using 5to solve 144. While our previous experiments have been performed exclusively on MiniSAT 2.0we wanted to get a "bigger picture" result with this comprehensive study. As such we opted to not only perform the trials with MiniSAT 2.0, but also with Glucose 3.0, Lingeling , Plingeling and SWDiA5BY 2.3. We chose these as they were the silver and gold meddle winners of the 2013/2014 SAT competition for the application category. Initially we intended to run the experiments with PeneLoPe as well, however we were unable to compile the 2013/2014 SAT competition version on any of our machines. While we would ultimately like to run the trials on more solvers, time did not allow for this. Additionally in the future we intend to run the trials on more machines and more instances. For this set of trials we specifically chose instances exhibiting different characteristics. While this may be seen as a bias of the experiment, it is important to note that we are not trying to show which solver is fastest, only identify which hardware characteristics impact the solution time with these solvers.

The instances were selected from a stratified random sample, from the 2011-2014 SAT competition application categories, across ranges of Q, $|Co|$, $|V|$, $|Cl|$, CVR and solution time. At most three instances were selected for each range of the measured property. The properties were measured on the non-simplified instances on a single trial machine (machine 28 in Appendix Table A). The resulting set contained 144instances after duplicates were removed. These 144instances cover a diverse set of of characteristics and solution times. Selecting instances from a stratified random sample in this manner allowed us to ensure that while no instance was "cherry picked" for any reason, we were guaranteed to see a representative set of instances from the entire population.

Each pair of instance and solver was then run on each of the 26machines. Due to constraints of the DataMill platform, each trial was allowed a maximum of two hours for completion and were ran in batches. Each batch contained no more than 18 instances to keep the batch time (including setup, solving and collection) under two days, this is a requirement of DataMill. 40 batches were ran per machine, the order of trials was randomized prior to batching, as such each batch for each machine contained the same instances, additionally batches were ran in a random order.

Due to the heterogeneous nature of the hardware used, it was not possible to run all solvers on all machines. We found that the ARMV7 machines (machines 22-25 in Appendix Table A) were only able to run Lingeling, all the other solvers (including pLingeling oddly) failed to compile due to a floating point library not being present in the ARM version of the operating system. All machines in the DataMill cluster run Gentoo Linux with kernel version 3.3.8, and GCC 4.5.3, machines 1 and 28 ran Ubuntu 12.04 LTS. Each machine is dedicated to running only the experiment given, and is not virtualised or shared in any way.

To remove the possibility that these timing differences were the result of randomness in the solver, we pre-simplified the instances then turned off simplification on the solvers, this is to resolve the known issue that clause (and variable) ordering has an effect on solution time [12]. In addition to this we set a fixed random seed for all the solvers on all machines (with the exception of pLingeling which did not support this option), such that a single instance/solver pair should have identical performance on an identically specified machine.

The complete set of all combinations of solvers, workers and instances would have created approximately 18500 trials. Unfortunately even with the timeouts and batching we imposed DataMill was unable to complete all of these trials within the required time frame. As such we are limited to analysing those results which were gathered, approximately 15500 of them. Within these results approximately 1600 were unable to run due to a lack of support in the ARM kernel, as mentioned previously. Leaving a total of 13825 instances available for analysis. The majority of our analysis looks at aggregate results, either across the instances, solvers or workers. To ensure that no bias has been introduced by certain combinations of workers/instances not being included we only analyse the complete set of workers and instances, by which we mean that every worker in our final dataset solved every instance at least once. To accomplish this we used an implementation of maximal biclique enumeration algorithm from Alexe et al[3] where the workers are one half of the graph and the instances are the other. The resulting clique included 13648 trials, unless otherwise stated this is the set used in all results presented.

The factors considered in this experiment were CPU speed, CPU cores, CPU Model, cache size (L1, L2 and L3, where available), RAM size (as well as speed and channels, where available) and FSB speed. Modern CPU's express their FSB speed in GT/i whereas older CPUs utilise MHz, since GT/i is considered a more accurate measurement, which describes not only the clock speed of the bus but the data width, we converted all measurements of FSB speed to GT/i. We included CPU Model to determine if CPUs from different models/manufacturers with similar specifications perform differently, a reason for this could be things like cache replacement policies and implementation specific timing characteristics, such as the proportion of integer vs floating point cores within the CPU. The data in this chapter is available online [31].

2 Results

In addition to these static factors, which were available prior to execution, we also measured the number of major/minor page faults and the memory usage peak. We found that while these did change between executions (based on the randomness of the memory layout and scheduling algorithms) they were tightly correlated to RAM size, page size and instance size. In some cases we analyse these as response variables, to replace of time.

The first factor we looked at was the average execution time between each

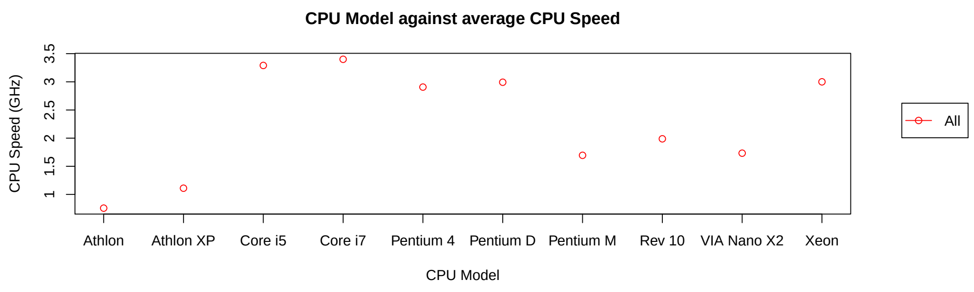


Figure 7.1: CPU Model against average speed for that model

type of CPU. Immediately we saw that each type of CPU solved instances at a different average speed While this is to be expected, as they have differing CPU speeds across the CPU types, we found that the average speed of the machines with a single CPU type — shown in Figure 7.1 — did not always correlate with the average execution time of instances solved on that CPU type — shown in Figure 7.2. One clear example is that of the Pentium M

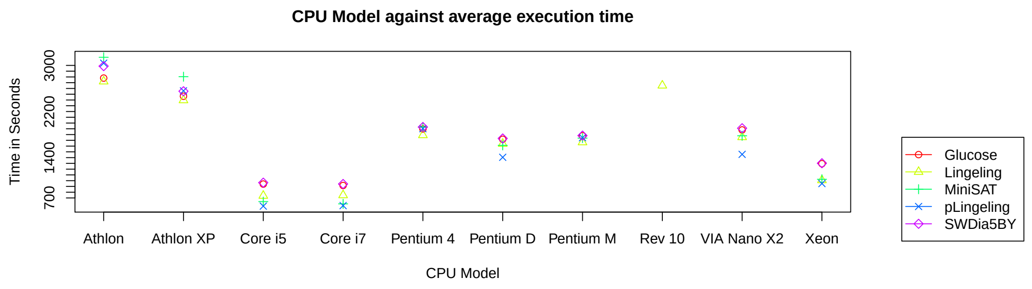


Figure 7.2: CPU Model against average execution time for that model, separated by solver

and Pentium D CPUs. The Pentium M machine used had a CPU speed of 1.7Ghz, compared to the Pentium D, which had a CPU speed of 3Ghz. However, despite this large difference in CPU speed, they presented very similar average execution times for the same set of instances. 1621 seconds for the Pentium D and 1742 seconds for the Pentium M. Clearly factors other than the CPU speed are playing a role here. This case is compounded further by the fact that the Pentium D machine has two CPU cores, versus the single core of the Pentium M. While this should have a minimal effect on the sequential solvers, it would be expected that pLingeling would exploit this additional CPU to improve the average execution time of the entire worker. In the remainder of this document we will explore which factors could be contributing to the unexpected result seen here.

As expected, when comparing the CPU clock speed of a machine to the average execution time there is a clear trend — with a few exceptions — as can be seen in Figure 7.3, however a less obvious result is the difference between the solvers. While almost all solvers performed better on CPUs with a faster clock speed, it is interesting to see that they did not all improve by the same amounts. MiniSAT, one of the slowest solvers on the slowest two CPUs measured (757MHz and 1.1GHz) improved radically to become the fastest overall solver on the faster CPU's measured (3.4GHz). Similarly Lingeling, the fastest solver on the slower CPUs, was relegated to third place on the faster CPUs.

A notable exception to the overall increasing performance as the CPU clock speed increases is that of Lingeling on the ARM machines. Lingeling is the only solver used that can run on these CPUs. Despite their middle ranged CPU speed, Lingeling ran slower on them than any other machine. This result is due to a number of factors, including the cache and RAM amounts, these will be discussed in more detail in subsequent paragraphs.

In addition to the obvious result that increases in CPU clock speed lead to an overall increase in performance, this experiment revealed other less obvious and more interesting results. Figure 7.1 show that the relationship between CPU speed and average time is not a fixed one, there are clearly other affects at work here.

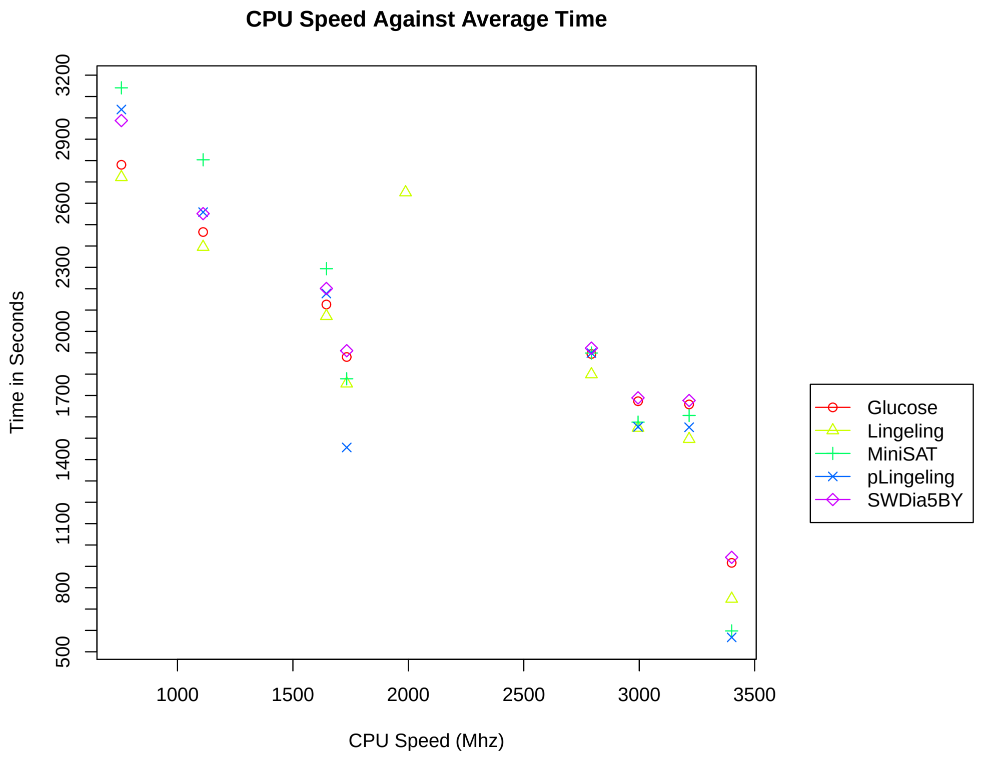


Figure 7.3: Plotting CPU speed against average execution time per solver

A further result observed was that while the majority of solvers perform better under an x86_64 cpu, pLingeling performs best under the i686 architecture. While it is expected that 64 bit machines offer performance increases for most instances due both to the increased width of registers in the CPU and the increased total amount of memory available, it was unexpected that pLingeling should perform better on the lower specified i686 architecture. The likely reason for this is that pLingeling is more bound by the presence of multiple cores than it is by the quantity of RAM available. A further analysis of the results may show that there is a correlation between larger numbers of cores and the i686 architecture.

In addition to determining which solvers performed better under different conditions, we were also interested in determining which characteristics of an input formula causes an instance to perform well on certain machines and less well on others.

Due to the limited quantity of computers this experiment was ran over, it was difficult to produce a single model that estimates the time of an individual instance/solver combination across the machines. Instead of this, our approach was to group solvers together and develop a model that describes the solution time of a specific instance across all solvers and machines. This lead to a far larger sample size per instance and as such allowed us to include more terms in the model without risking over fitting.

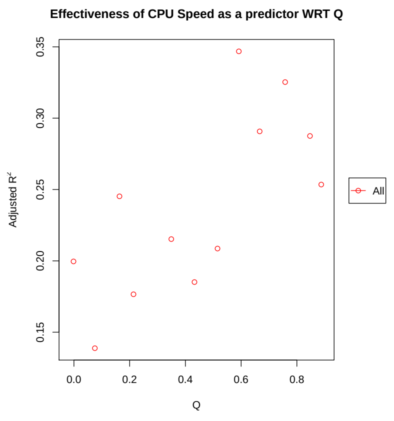
In addition to this, we also considered how important a single machine parameter was to instances based on different input formula characteristics, for example Q or CVR. To describe this we created a model that was of the following form:

$$time \sim factor$$

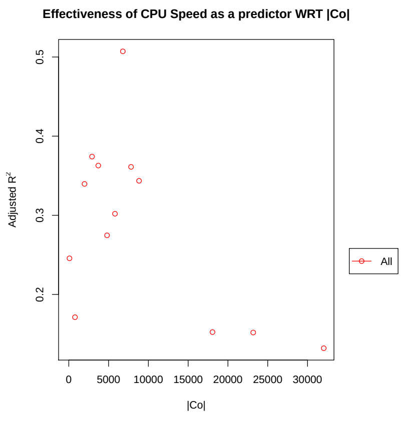
Where factor was one of the machine parameters described earlier. Next we computed the adjusted R^2 for each instance executed across all solvers and machines. We then grouped instances with similar input characteristics and measured the average adjusted R^2. The results of this experiment were quite interesting. We discovered that some instances solution time are tightly bound to the speed of the CPU speed of the worker. However for other instances this was not the case. Consider Figure 7.4a, there exists a strong correlation between the quality of the community structure and the impact of the CPU speed on the solution time.

This would suggest that instances with a strong community structure (high Q) are more CPU bound, by which we mean they spend less time waiting for resources, than their lower Q counterparts. We see a similar pattern, though less strong when looking at the number of communities $|Co|$ in Figure 7.4b. As the number of communities in a formula approaches 8000 the instance becomes more CPU bound, while those instances above this point (particularly those with more than approximately 17500 communities) see a reduced R^2 it is interesting to note that the R^2 for those instances within the "CPU bound" range, is higher, with many values of $|Co|$ exhibiting R^2 of greater than 0.35, and some approaching 0.5. In addition to these two community related trends we found two others with regards to clauses. We found that when the average clause length approaches 7, the instance becomes more CPU bound as shown in Figure 7.4c. Similarly, when then the CVR of an instance approaches 50 the instances also become more CPU bound as shown in Figure 7.4d. We found that neither the number of variables ($|V|$) nor the number of clauses ($|Cl|$) exhibited any particular trend when compared looking at CPU Speed. In the CVR graph Figure 7.4d a set of trials for a single instance was excluded as the instance had a CVR of 1680, this compressed the remaining data in the graph, making the trend more difficult to observe. However it also exhibited the same R^2 as those instances with CVR ~ 290.

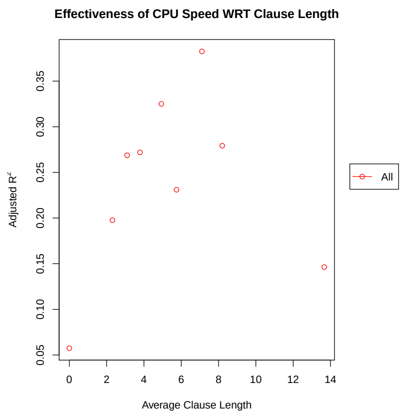
While no discernible pattern was observed that correlates solution time with $|V|$ when considering CPU Speed, we did notice a pattern when looking at RAM size. Figure 7.5a shows this trend, while it is not a particularly relationship, it should be noted that in many cases the RAM size of the solving machine is the main determining feature for instances with a certain sized $|V|$. This is something that requires more detailed analysis to understand fully, as will be discussed in Chapter 9. We saw a less severe, but stronger correlation when observing the $|Co|$ against the effectiveness of RAM size as a predictor as shown in Figure 7.5b. As the number of communities increases, the importance in RAM size also increases, with a single exception. This makes sense, both in regards to $|V|$ and $|Co|$. In the former case as the size of the formula increases, so does the amount of RAM required to store it, before swap space is used. Swapping pages of memory to physical storage is a highly time consuming operation, particularly as a swap out to memory is almost always accompanied by a triggering swap into memory, from storage. In this case the CPU is often unable to continue working while this happens, and the delay can be several milliseconds in some cases, depending on the storage medium and connection used. In the latter case, the $|Co|$ does not directly reflect the size of the input formula, and therefore would not directly rely on the RAM size for performance, however it is likely that those instances with a large number of communities do not exhibit good temporal locality with regards to memory accesses, this will likely trigger a



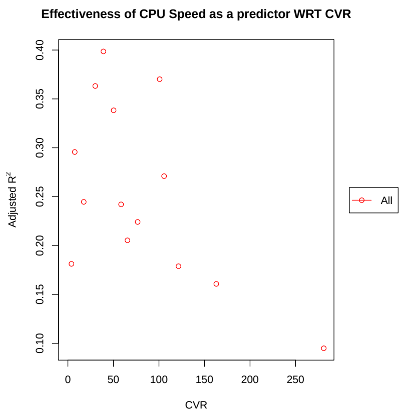
(a) Average adjusted R^2 against Q



(b) Average adjusted R^2 against $|Co|$



(c) Average adjusted R^2 against clause length

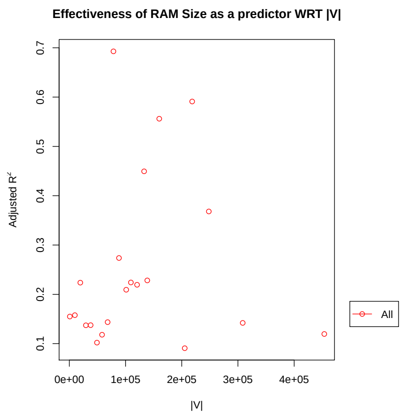


(d) Average adjusted R^2 against CVR

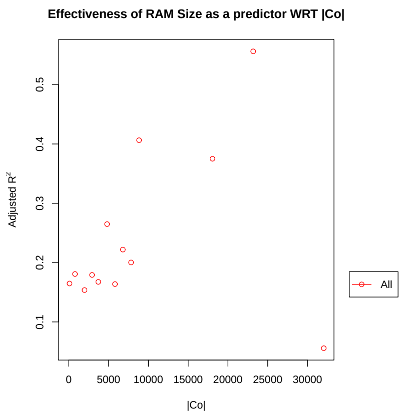
Figure 7.4: Average adjusted R^2 against different formula characteristics for the model of time CPU Speed

large number of page faults. An additional factor not yet discussed in this document is the size of the largest community of the input formula, this is shown in Figure 7.5c, from this figure we can observe that when there are approximately 3000 variables in the largest community of an input formula, the speed of the worker is determined almost entirely by the available RAM.

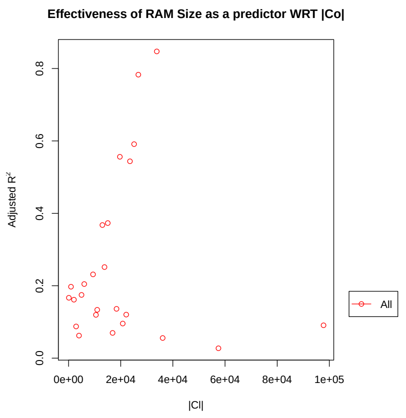
In addition to RAM size and CPU speed, we also looked at the effect of cache size on the performance of individual instances. Unfortunately we



(a) Average adjusted R^2 against $|V|$



(b) Average adjusted R^2 against $|Co|$

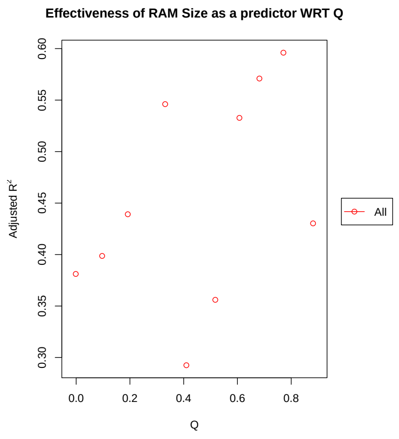


(c) Average adjusted R^2 against max(Co)

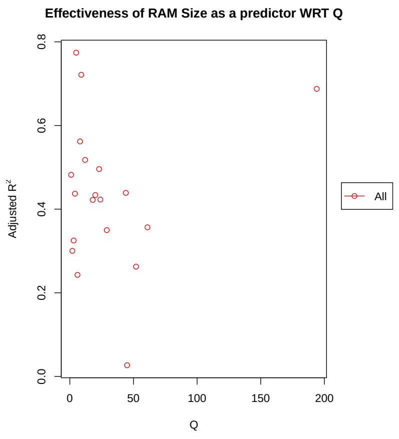
Figure 7.5: Average adjusted R^2 against different formula characteristics for the model of time RAM Size

were unable to gather complete cache sizes for all CPU's in this experiment. However, L1 and L2 cache sizes were mostly available to us. Figure 7.6 shows the impact of cache sizes on the running time of a SAT formula when compared with different ranges of characteristics. Figure 7.6a shows this trend with respect to the community structure (Q), from this we can see that cache size always plays a relatively significant role in the performance of a particular worker, however as Q increases the importance on the cache size becomes, generally speaking, more significant. This was a shock for us as we expected the opposite. This would have supported the belief that as the Q increases, the "locality" of the instance improves. In executions of typical computer programs, the higher the locality (but temporal and spacial) a program is, the less cache it requires to be executed efficiently. This result would suggest that either, as Q increases, the locality of the instance decreases, or that as Q increases the locality of the instance — and thus the size of the cache — becomes more significant to performance. Figure 7.6b shows a negative trend, where the larger the size of the smallest community, the less significant the cache size becomes to performance. This would also suggest that the locality of a formula is affected by its Q and $|Co|$, instances with larger communities cannot fit a single community into cache, meaning that if the solver is working locally, on one community at a time it would need to repeatedly load from variables and clauses RAM, leading to an overall decrease in performance. This is reflected by a decrease in the significance of cache size on those instances with larger, smallest communities. The result shown in Figure 7.6b has a single result removed, an instance where the smallest community size was very large (2.0e+09). This instance followed the same overall trend presented in Figure 7.6b, and had an R^2 of approximately 0.1. It was removed to make the overall trend more clear.

The final parameter we looked at was the number of CPU cores available to the solver. For this result we performed the same analysis as for the other machine parameters, however we separated the dataset not only by formula, but by solver as well. The reason for this is that we would expect there to be significant trends visible on at least some of the formulae when looking at the pLingeling solver (and in the future, any other parallel solver) and less significant trends for all other solvers, as they are sequential and should not be taking advantage of multiple cores. The results for this are shown in Figure 7.7, these results show a number of interesting characteristics. Firstly, we see that Lingeling is unable to take advantage of the number of cores in the solving machine, regardless of the clause-variable ration (CVR). This is to be expected as it is a sequential solver, however it is strange that all other solvers seem to be able to exploit the number of cores to some extent, even though that relationship is unpredictable. A notable exception to this is pLingeling, where we see a weak — but visible — trend where as the CVR increases, the number of cores available becomes less significant. The mixed result regarding other solvers is likely because of some secondary relationship between the number of cores and either the CPU speed, or RAM/cache



(a) Average adjusted R^2 against Q



(b) Average adjusted R^2 against $\min(Co)$

Figure 7.6: Average adjusted R^2 against different formula characteristics for the model of time L1 Data Cache Size $\oplus$ L2 Cache Size

sizes. This is certainly true in certain circumstances, looking at Appendix Table A we see that the majority of machines with more cores have a faster CPU, in the 3.2Ghz-3.4Ghz range (most of the ARM machines were excluded from these trials). However, there is some crossover in CPU speed between those machines with one and two cores. Unfortunately without more data — particularly data where we can test different numbers of CPU cores without changing the CPU speed — it is difficult to determine conclusively whether some sequential solvers perform better on multi core machines rather than single core machines. This is discussed further in Chapter 9

3 Conclusion and future work

In this chapter we have explored the relationship between different classes of instance, and their solution times on differently specified solving machines. We have shown that the solution time for a specific instance varies greatly across different solving machines, in ways that are not completely predictable when considering characteristics such as CPU speed, RAM size and cache size. We have further shown that the impact of each of these factors on the

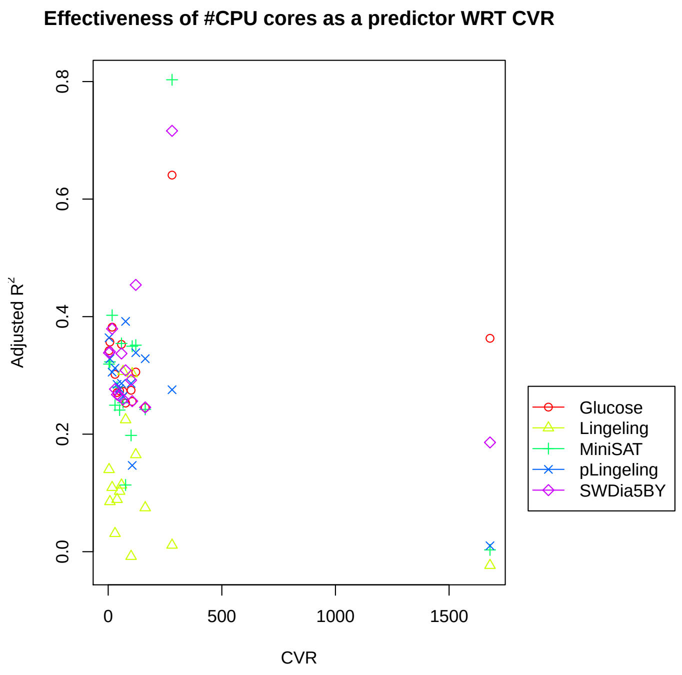


Figure 7.7: Average adjusted R^2 against CVR for the model of time #Cores

solution time of an instance depends on the structure of the instance. In some cases this structure is characterised by graph theoretical concepts, and others use SAT specific concepts such as the clause-variable ratio.

While we have not been able to produce a model that completely explains the variability in solution time when varying the solving machine, we have been able to explain a large amount of it. The remaining variability is likely to be in the factors that were only partially present (e.g. cache sizes, RAM speeds, etc), however we also speculate that cache replacement policies will be a determining factor in solution time of specific instances.

The next steps for this work are to find more details on the machines used, as previously mentioned it was not possible to gather all levels of cache size for all machines, similarly RAM channels and speeds were missing in some cases. We hope this will partially solve the issue of un-defined variation. However, if this does not complete the model we are planning on exploring the relationship between cache replacement policies and performance. We feel this may be what is missing when we compare machines with very different specifications, that perform similarly — for example, the Pentium M and Pentium D machines as described above.

Chapter 8

Related work

In [6] Levy et al introduced the concept of SAT problems having community structure. The paper showed that numerous problems in the SAT 2010 race contained very high modularity compared to graphs of any other nature. It was also suggested in this paper that SAT solvers are able to exploit this *hidden structure* in order to achieve good solve times. However, the paper was unable to explain what characteristics of community structure leads to poor or good solve times. Also, in [7] the authors state that while SAT solvers have shown improvements in solve times for numerous industrial applications, their has been less success in improving the solve time of randomly generated instances. They posit that this is due to the lack of structure present in randomly generated instances. Many algorithms have been proposed to discover the community structure in graphs [16, 13]. Below we give a brief description of other tools that provide visualization and/or evolution capabilities and highlight the differences between these tools and SATGraf.

In [44] Xu et al describe a SAT solver that chooses its algorithms based on 48 features of the input formula. While they did use certain graph theoretic concepts, such as node-degree statistics, they did not consider the concept of communities as a feature of the input. The list of 48 features could be used in a more comprehensive model than the ones used in our regression. In [19] Habet et al present an empirical study of the effect of conflict analysis on solution time in CDCL solvers.

In [5] the authors present a the notion of fractal dimensions in SAT solvers, they have discovered that as the SAT solver progresses the fractal dimension increases when new learnt clauses are added to the formula. They have also discovered that learnt clauses do not connect distant parts of the formula (ones with long shortest paths between nodes), as one would expect. This is interesting when combined with the work we present stating that clauses which are comprised of variables in a small number of communities are more useful to the solver. This means that even when a learnt clause that does connect distant variable in the formula is added, it is not as useful as a clause that connects locally occurring variables.

In [14] the authors discuss the impact of the previous CVR result [18] on a more diverse set of input formula and solvers. They state that the relationship between CVR and solution time in random instances is different depending on the algorithm and implementation chosen, and that the CVR = 4.26 result is not conclusive.

SATGraf is the only tool that we know of that has both visualization capabilities to view the community structure of SAT instances, and evolution feature that shows how the community structure is morphed by a CDCL SAT solver as it solves an input instance. While other tools [41, 40, 37, 10] have visualization and/or evolution capabilities, they do not focus on the community structure of SAT instances nor do they show how the solver morphs the community structure of input SAT instances. Instead these tools allow the user to view the SAT instance as a graph without any community structure information. The following Table 8.1 highlights the differences between the various visualization tools that we found. Those differences range across a handful of categories such as interactive (ability to set a value to a variable on the graph), evolution (ability to see the evolution of the SAT formula), community (ability to display the community structure), 3D (three dimensional capability), and implication (can generate the implication graph).

DPViz [41], probably the closest to SATGraf in terms of features, is a graphing tool designed to expose how a CDCL solver morphs a SAT instance as it is being solved. It offers a number of features such as multiple layout algorithms, zooming into the graph to show more detail, the ability to set specific values on literals displayed in the graph, and performing unit propagation.

Each tool presented above has different strengths and weaknesses. However, the only tool that can accomplish visualizing community structure of a SAT formula, both in its original state and while being solved by a SAT solver, is SATGraf.

Tool	Interactive	Evolution	Community	3D	Implication
DPViz[41]	✓	✓	✗	✗	✓
GraphInsight[37]	✓	✗	✗	✓	✗
iSat[40]	✗	✓	✗	✗	✗
GraphViz[10]	✗	✗	✗	✗	✗
SATGraf	✓	✓	✓	✗	✓

Table 8.1: Comparison of Tools

Chapter 9

Future work

The results presented in this work are far from the last word on the subject of community structure and SAT solver performance. Before beginning this work we knew it would not cover every angle, leaving a plethora of subjects available to study in the future. While working on this document that number increased further as we found hitherto unconsidered factors.

In regards to the linear regression we have a number of avenues for future work. Firstly we intend to consider a larger set of data, enabling us to exclude time-outs and focus on specific categories of SAT instances, i.e. random, industrial and hand-crafted. This will allow us to utilise the same (or a similar) model to gather a more accurate result by excluding our right censored (time-out) data. It will also allow us to present more accurate category specific results which, as discussed in Chapter 3 will require separate models for each category.

In addition to this, we are exploring different regression techniques more suited to non-normally distributed data. At present gamma regression is looking the most likely candidate. We are also looking into the utility of bootstrapping this experiment to estimate confidence intervals for our estimates, which is not currently possible.

Finally, we are exploring new input factors for this model, such as the size of the smallest, and largest communities within the formula, as well as non community related factors, such as the total number of edges, and the number of times a variable is reused within a formula.

In regards to the random data experiment, we observed an interesting trend when considering the average clause length of the input formula. Due to the random generation technique used, a clause in a formula contains exactly three literals, but some clauses may contain the same literal more than once. Modern SAT solvers immediately remove these duplicate variables during simplification, we noticed that when the average clause length was below 2.8, the formulae were all trivially easy, and an exponential curve was present between 2.8 and 3.0.

In addition to this, we want to explore different SAT solvers, the work in [14] states that different algorithms present different patterns in this area, it will be interesting to see how different implementations of the same algorithm (for example different clause deletion, or decision heuristics) affect this result.

In regards to SATGraf, there are a number of potential new features, the least of which is including new drawing and community detection algorithms as options. Additionally, supporting different solvers, or at least different decision and clause deletion heuristics in the supported MiniSAT solver is a priority. In addition to this, we are working on better SATGraf support for our web platform SATBench.

In regards to our work on survival analysis, there are numerous other experiments that can be run, primarily we intend to focus on results that show more than just a simple peak in execution time — we found that the more interesting results revolve around observations that define when certain instances will be solved, rather than their overall probability of solution.

In Chapter 7 we looked into which machine parameters were determining factors in the speed of different solvers for different classes of instances. While we did confirm that CPU speed, cores and RAM amounts were significant factors, as well as other less significant (and less obvious) factors, we have further experiments planned regarding these results.

Firstly, we would like to study how closely the solution time of different instances is correlated with the percentage of cache misses at different levels, while it is known that this will affect performance to some degree, it will be interesting to determine how much this one factor affects solve time, and to see if this changes based on the community structure of the instance being solved. In addition we would like to use this new trial as an opportunity to analyse the different cache replacement policies, and their effectiveness on different classes of instances under different solvers. we feel this may be the cause of the differing results for similar specified computers from different manufacturers.

Additionally, while we have explored the relationship between a number of machine based factors and performance of SAT solvers on different instances, we are also interested in exploring the relationship between advertised speed of a CPU vs the speed it actually runs at (due to over/under clocking), we are not certain if a CPU that is over clocked will run with the same performance as an identically specified machine with a natively faster CPU, reasons for this include insufficient memory access channels or cache configurations, as well as potential issues with overheating.

In addition to these chapters specific areas of future research, we are also in the process of repeating these results with different graph representations, such as the clause-incidence graph. In addition to this, we have found that slight differences in the implementation of the OL or CNM algorithms can have huge impact on the utility of the models utilised, for this reason we will be exploring other community detection algorithms, particularly ones that focus on weighted edges.

In addition to this, we are looking at the differences between graph width, back-door size and community based models. It is possible that some combination of these will lead to stronger results.

Chapter 10

Conclusion

This concludes the thesis undertaken for my Master of Applied Science. The work has focussed primarily on the performance of SAT solvers, particularly when considered from the perspective of graph theoretical concepts such as community modularity.

This work has covered several areas. It contains an initial and revised version of a predictive model for the solution time of the MiniSAT solver. In addition to this, it contains an analysis of the effect of machine parameters on SAT solver performance for a number of state of the art solvers. We have expanded on our work on the relationship between the clause-variable ratio and community modularity (Q) of a SAT instance. Furthermore, we have considered for the first time the use of survival analysis techniques to analyse the survivability of SAT instances with respect to different characteristics.

From the above work, we can establish that not only does the community structure of a SAT instance have a significant impact on the solution time of a SAT solver, but that this solution time is affected by different physical machine attributes depending on the class of problem. We have shown that this is linked closely with the community structure of the instance. In addition we have shown that different solvers also respond differently to these physical attributes.

We have presented several resources with this work. Firstly, SATGraf a tool for visualising the community structure and evolution of this structure in SAT instances while they are being solved. Secondly, SATBench an online repository tracking solution time for a number of solvers and machines against numerous instances. This archive is searchable and exportable as well as offering basic analytical tools for comparing trends of different sets of data. An API is also available for this such that other researchers can provide data to this archive. It is our hope that this system become a central location for storing instances utilised not only in SAT competitions but whenever an experiment is ran that looks at the solution time of SAT solvers, so further research may be carried out without necessarily re-running experiments that already exist. It also provides a central location where researchers may publish their data so their findings may be corroborated.

Finally, we have introduced survival analysis to the field of SAT solver performance and shown that its ability to include right censored data — such as memory out and time-out errors — enables it to show trends that would otherwise be difficult to analyse. We have further shown that survival analysis is able to describe more than simple peaks in execution time, by providing insight into when instances of a certain class are solved.

References

[1] 2013 sat competition. http://satcompetition.org/2013/. Accessed: 2014-12-01.

[2] 2014 sat competition. http://satcompetition.org/2014/. Accessed: 2014-12-01.

[3] et al Alexe. Maximal biclique enumeration implementation. http://genome.cs.iastate.edu/supertree/download/biclique/README.html, 2004. Accessed: 2015-01-10.

[4] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. The fractal dimension of SAT formulas. *CoRR*, abs/1308.5046, 2013.

[5] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. The fractal dimension of sat formulas. *arXiv preprint arXiv:1308.5046*, 2013.

[6] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 410–423. Springer, 2012.

[7] Carlos Ansótegui and Jordi Levy. On the modularity of industrial sat instances. In *CCIA*, pages 11–20, 2011.

[8] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. 2009.

[9] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS Press, 2009.

[10] A. Bilgin, J. Ellson, E. Gansner, O. Smyrna, Y. Hu, and S. North. Graphviz - graph visualization software. http://www.graphviz.org/. Accessed: 2015-01-10.

[11] J. Bruin. R textbook examples, applied survival analysis,chapter 2: Descriptive methods for survival data. http://www.ats.ucla.edu/stat/r/examples/asa/asa_ch2_r.htm, 2011. Accessed: 2015-01-13.

[12] Rodrigo Castaño and José M Castaño. Propositional satisfiability (sat) as a language problem. In *XVII Congreso Argentino de Ciencias de la Computación*, 2011.

[13] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[14] Cristian Coarfa, Demetrios D Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, and Moshe Y Vardi. Random 3-sat: The plot thickens. In *Principles and Practice of Constraint Programming–CP 2000*, pages 143–159. Springer, 2000.

[15] Niklas Een and Niklas Sörensson. Minisat: A SAT solver with conflict-clause minimization. *SAT*, 5, 2005.

[16] S. Fortunato, V. Latora, and M. Marchiori. Method to find community structures based on information centrality. http://www.w3.org/People/Massimo/papers/2004/community$_$pre$_$04.pdf, 2004. Accessed: 2015-01-10.

[17] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[18] Ian P Gent and Toby Walsh. The sat phase transition. In *ECAI*, pages 105–109. PITMAN, 1994.

[19] Djamal Habet and Donia Toumi. Empirical study of the behavior of conflict analysis in cdcl solvers. In *Principles and Practice of Constraint Programming*, pages 678–693. Springer, 2013.

[20] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 215–224. IEEE, 2010.

[21] David S Johnson and Michael R Garey. Computers and intractability: A guide to the theory of np-completeness. *Freeman&Co, San Francisco*, page 32, 1979.

[22] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Trans. Graph.*, 10(1):1–39, January 1991.

[23] Edward L Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 53(282):457–481, 1958.

[24] J. Leskovec. Snap system. http://snap.stanford.edu/snap/index.html. Accessed: 2015-01-10.

[25] Inês Lynce and Joao Marques-Silva. Building state-of-the-art sat solvers. In *ECAI*, pages 166–170, 2002.

[26] Haralambos Mouratidis and Paolo Giorgini. Security attack testing (sat)testing the security of information systems at design time. *Information systems*, 32(8):1166–1183, 2007.

[27] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[28] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[29] Z. Newsham, W. Lindsay, J. Liang, K. Czarnecki, S. Fischmeister, and V. Ganesh. Satgraf: Results. http://ece.uwaterloo.ca/$\sim$vganesh/SATGraf/Results.html, 2014. Accessed: 2015-01-10.

[30] Z. Newsham, W. Lindsay, J. Liang, K. Czarnecki, S. Fischmeister, and V. Ganesh. Satgraf: Source code. http://bitbucket.org/znewsham/satgraf, 2014. Accessed: 2015-01-10.

[31] Zack Newsham. Maching parameters experimental data. http://satbench.uwaterloo.ca/download/39/1, 2014. Accessed: 2015-01-10.

[32] Zack Newsham. Sat 2013 competition minipure timing, corrected data. http://satbench.uwaterloo.ca/download/43/1, 2014. Accessed: 2015-01-10.

[33] Zack Newsham. Evolution of toybox. http://satbench.uwaterloo.ca/evo/toybox.gif, 2015. Accessed: 2015-01-10.

[34] Zack Newsham. Partial evolution of aes_16_10_keyfind_3. http://satbench.uwaterloo.ca/evo/aes_16_10_keyfind_3.gif, 2015. Accessed: 2015-01-10.

[35] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Community Structure of SAT Instances Webpage with Data and Code. https://ece.uwaterloo.ca/~vganesh/satcommunitystructure.html. Accessed: 2015-01-10.

[36] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2014. best student paper award.

[37] Carlo Nicolini and Michele Dallachiesa. Graphinsight: An interactive visualization system for graph data exploration. http://www.graphinsight.com. Accessed: 2015-01-10.

[38] Augusto Oliveira, Jean-Christophe Petkovich, and Sebastian Fischmeister. How Much Does Memory Layout Impact Performance? A Wide Study. In *Proceedings of the International Workshop on Reproducible Research Methodologies (REPRODUCE)*, page 23–28, Orlando, USA, Febuary 2014.

[39] Augusto Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 137–149, Prague, Czech Republic, April 2013.

[40] Ezequiel Orbe, Carlos Areces, and Gabriel Infante-López. isat: structure visualization for SAT problems. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 335–342. Springer, 2012.

[41] Carsten Sinz and Edda-Maria Dieringer. DPvis–a tool to visualize the structure of SAT instances. In *Theory and Applications of Satisfiability Testing*, pages 257–268. Springer, 2005.

[42] T. Taiwan and H. Wang. Minipure. http://satcompetition.org/edacc/SATCompetition2013/experiment/25/solver-configurations/859, 2013. Accessed: 2015-01-10.

[43] Craig A Tovey. A simplified np-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85–89, 1984.

[44] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res.(JAIR)*, 32:565–606, 2008.

[45] Wangsheng Zhang, Gang Pan, Zhaohui Wu, and Shijian Li. Online community detection for large complex networks. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1903–1909. AAAI Press, 2013.

Appendices

A Benchmarking machines

A list of all machines used in the various experiments in this thesis.

Number	CPU	Cores	CPU speed	Cache (L1 i/d + L2 + L3)	RAM amt	Ram speed
1	Intel Core i7 i686	4	3400	32/32 + 256 + 8192	8266580	0
2	Intel Pentium M i686	1	1695	0/0 + 256 + 0	902287	0
3	Intel Pentium 4 i686	2	2992	0/16 + 2048 + 0	894177	533
4	VIA Nano X2 i686	2	1733	128/128 + 2048 + 0	1814036	1066
5	Intel Pentium 4 i686	2	3200	0/0 + 512 + 0	1000263	0
6	Intel Pentium 4 i686	1	1595	0/0 + 256 + 0	254781	0
7	Intel Pentium 4 i686	2	2998	0/16 + 1024 + 0	893347	0
8	Intel Pentium 4 i686	1	1595	0/0 + 256 + 0	514119	133
9	Intel Pentium 4 i686	2	2992	0/0 + 1024 + 0	894269	0
10	Intel Pentium 4 i686	2	3200	0/0 + 512 + 0	1000540	0
11	Intel Pentium 4 i686	2	2793	64/64 + 2048 + 0	902461	0
12	Intel Pentium 4 i686	2	1614	0/0 + 256 + 0	894269	0
13	Intel Pentium 4 i686	2	1600	0/0 + 256 + 0	242851	0
14	Intel Pentium 4 i686	2	3198	0/0 + 512 + 0	894269	0
15	AMD Athlon XP i686	1	1111	64/64 + 256 + 0	514199	0
16	Intel Pentium D i686	2	2993	0/0 + 1024 + 0	2076180	0
17	Intel Pentium 4 i686	2	3200	0/16 + 512 +0	894269	0
18	Intel Pentium 4 i686	2	3192	0/0 + 512 + 0	505661	0
19	Intel Xeon x86_64	2	3000	0/0 + 4098 + 0	2831155	0
20	Intel Pentium 4 i686	2	3200	0/0 + 512 + 0	2076180	0
21	Intel Core i7 x86_64	8	3401	128/128 + 1024 + 0	8095006	0
22	ARM Rev 10 armv71	4	1988	0/0 + 1024 + 0	896563	0
23	ARM Rev 10 armv71	4	1988	0/0 + 1024 + 0	896563	0
24	ARM Rev 10 armv71	4	1988	0/0 + 1024 + 0	896563	0
25	ARM Rev 10 armv71	4	1988	0/0 + 1024 + 0	896563	0
26	Intel Core i5 x86_64	4	3291	128/128 + 1024 + 0	8095006	0
27	Intel Core i7 64	4	3400	32/32 + 256 + 8192	8388608	0
28	AMD Athlon	1	757	64/64 + 256 + 0	773079	0

List of machine specifications used for the varied parameters trial.
Machines 2-28 were ran through the DataMill platform

B Benchmarking instances

A list of all instances used in Chapter 7.

Instance name	Q	$\lvert Co\rvert$	$\lvert Cl\rvert$	$\lvert V\rvert$	CVR
005	0.090262	64	55861	3254	17.17
289-sat-6x20	-0.001587	3	11760	360	32.67
6pipe_6_ooo.shuffled-as.sat03-413	0.133085	10	539799	14948	36.11
6s137	0.466398	8468	1104820	218561	5.05
6s139	0.436815	4964	727544	134368	5.41
7cnf20_90000_90000_7.shuffled	0.017496	1	1532	20	76.60
9pipe_k	0.402002	9	2302466	45830	50.24
aaai10-planning-ipc5-pathways-13-step17	0.40244	1213	123637	19174	6.45
aaai10-planning-ipc5-pipesworld-12-step15	0.841376	250	931750	56496	16.49
aaai10-planning-ipc5-pipesworld-12-step16	0.843843	259	1016489	61519	16.52
aaai10-planning-ipc5-TPP-30-step11	0.863159	2359	2910796	308480	9.44
ACG-10-10p0	0.465775	5887	422890	85306	4.96
aes_32_3_keyfind_1	0.34408	24	2204	450	4.90
AProVE11-06	0.351294	22535	778280	159983	4.86
AProVE11-10	0.283643	18194	749349	143647	5.22
AProVE11-11	0.384989	6663	225071	49526	4.54
AProVE11-13	0.319356	30663	1294039	249832	5.18
AProVE11-15	0.307996	4833	172762	35192	4.91
AProVE11-16	0.377515	5440	192596	41165	4.68
battleship-10-17-sat	0.055898	9	865	170	5.09
battleship-10-18-sat	0.067677	11	910	180	5.06
battleship-14-26-sat	0.087463	15	2562	364	7.04
battleship-16-31-sat	0.053327	17	3976	496	8.02
battleship-5-8-unsat	0.013009	5	105	40	2.62
bc57-sensors-1-k303-unsat.shuffled-as.sat03-406	0.541139	8775	600244	110933	5.41
bob12m04	0.461189	975	149496	35038	4.27
bob12s04	0.313953	6176	322010	90321	3.57
b_unsat	0.793949	92	1009915	112707	8.96
connm-ue-csp-sat-n1200-d-0.02-s405595518.used-as.sat04-950	0.125351	44	13008	843	15.43
connm-ue-csp-sat-n800-d-0.02-s1542454144.sat05-533.reshuffled-07	0.120872	35	8688	552	15.74
connm-ue-csp-sat-n800-d0.02-s925928766.sat05-538.reshuffled-07	0.124302	30	8592	525	16.37

Instance name	Q	$\|Co\|$	$\|Cl\|$	$\|V\|$	CVR
ctl_4291_567_12_unsat_pre	0.701089	31	259225	15232	17.02
ctl_4291_567_2_unsat	0.749368	35	149117	17742	8.40
cube-11-h14-sat	0.706586	656	828022	131729	6.29
dated-10-13-u	0.548559	3000	266796	43356	6.15
dated-5-11-u	0.588522	2010	175711	31545	5.57
driverlog3_v01a.renamed-as.sat05-3963	0.087007	3	163	28	5.82
E02F17	0.433218	77	63775	5283	12.07
E02F20	0.472681	175	392787	9892	39.71
E02F22	0.04212	197	1263700	12542	100.76
E04F19	0.538132	197	293680	8025	36.60
E04F20	0.382924	141	465947	9532	48.88
E04N18	0.466961	87	86799	5803	14.96
E05F20	0.561251	144	470330	9796	48.01
E05X15	0.521341	120	39904	4043	9.87
em_11_3_4_exp	0.495151	118	379659	8709	43.59
em_12_2_4_exp	0.49548	141	673048	12536	53.69
em_7_3_6_cmp	0.500984	46	11213	1465	7.65
em_7_4_9_fbc	0.509187	45	28463	1673	17.01
em_8_4_5_all	0.503798	61	61604	2416	25.50
gensys-icl007.shuffled-as.sat05-3133	0.225554	12	11487	666	17.25
gss-16-s100	0.770504	655	55745	13554	4.11
gss-20-s100	0.765423	735	57320	13940	4.11
gus-md5-08	0.15498	4392	163004	28424	5.73
gus-md5-11	0.154939	4369	163299	28472	5.74
hard-6-U-7061	0.474487	1340	407829	68144	5.98
hid-uns-enc-6-1-0-0-0-0-30856	0.519168	32	5492	1664	3.30
hitag2-8-60-0-0x880693399044612-25-SAT	0.23415	19	25259	2174	11.62
hwmcc10-timeframe-expansion-k45-pdtswvqis8x8p2-tseitin	0.714523	317	193587	27042	7.16
hwmcc10-timeframe-expansion-k45-pdtviseisenberg1-tseitin	0.784991	180	143866	23404	6.15
ibm-2002-21r-k95	0.53606	1073	406418	56624	7.18
ibm-2002-30r-k85	0.518531	2937	655165	85912	7.63
itox_vc1033	0.327283	5994	126273	28172	4.48
itox_vc1130	0.302241	7537	193003	40087	4.81
k2fix_gr_rcs_w9.shuffled	0.16935	15	303750	9929	30.59
korf-17	0.411141	152	88786	5924	14.99

Instance name	Q	$\|Co\|$	$\|Cl\|$	$\|V\|$	CVR
LABS_n067_goal001	0.633233	959	262103	76128	3.44
LABS_n068_goal001	0.644354	939	277430	80853	3.43
LABS_n070_goal001	0.643218	1043	299142	87292	3.43
lksat-n900-m6174-k4-l4-s819398222.used-as.sat04-929	0.129821	96	3667	719	5.10
manol-pipe-c10nidw	0.444685	1568	577387	93075	6.20
manol-pipe-f7idw	0.416056	804	330538	52766	6.26
marg3x3add8ch.shuffled-as.sat03-1448	0.098309	5	272	41	6.63
md5_48_1	0.493783	467	157214	26665	5.90
md5_48_4	0.494696	467	157348	26691	5.90
minandmaxor016	0.391523	65	6844	1393	4.91
mizh-sha0-36-2	0.393503	495	120186	20535	5.85
mizh-sha0-36-4	0.393658	487	120280	20555	5.85
mod2-rand3bip-sat-270-1.shuffled-as.sat05-2248	0.162143	54	1080	270	4.00
mrpp_4x4#10_16	0.40342	85	16177	1660	9.75
mrpp_4x4#10_20	0.39835	100	20879	2123	9.83
MUS-v300-3	0.139001	37	1024	283	3.62
ndhf_xits_19_UNKNOWN	0.216174	11	456820	2803	162.98
ndist.b.26487	0	0	0	0	0
openstacks-sequencedstrips-...-p30_3.085-SAT	0.217335	22	1331173	205326	6.48
partial-10-15-s	0.616535	5779	686037	120599	5.69
pb_200_03_lb_02	0.598218	228	236382	47916	4.93
pb_200_05_lb_00	0.647451	245	226038	45649	4.95
pb_300_02_lb_07	0.678614	179	527489	106836	4.94
pb_300_06_lb_02	0.676937	214	510059	103115	4.95
Q3inK10	0.000606	1	75600	45	1680.00
q_query_3_L150_coli.sat	0.343241	1680	936264	91073	10.28
q_query_3_L70_coli.sat	0.351663	808	298118	28568	10.44
rand_net60-40-10.shuffled	0.266552	581	10755	2888	3.72
rbsat-v760c43649g3	0.187684	12	40775	730	55.86
rbsat-v760c43649g7	0.140115	13	43098	731	58.96
rbsat-v945c61409g5	0.138853	45	58491	945	61.90
rnd_100_28_s	0.319278	10	5462	861	6.34
SAT_dat.k80	0.431933	598	120758	17924	6.74
SGI_30_50_30_20_1-log.shuffled-as.sat03-107	0.02662	1	42147	150	280.98
shift1add.10997	0.528773	1347	42917	14621	2.94
shift1add.19970	0.54556	2489	80140	27195	2.95

Instance name	Q	$\lvert Co \rvert$	$\lvert Cl \rvert$	$\lvert V \rvert$	CVR
shift1add.23958	0.527967	2790	87551	29833	2.93
shift1add.28943	0.548842	3295	107510	36465	2.95
slp-synthesis-aes-top25	0.460695	595	129117	25928	4.98
slp-synthesis-aes-top26	0.449641	612	139149	27953	4.98
slp-synthesis-aes-top27	0.450997	624	150017	30189	4.97
slp-synthesis-aes-top28	0.444864	668	160659	32468	4.95
slp-synthesis-aes-top29	0.473257	688	172347	34832	4.95
smtlib-qfbv-aigs-bin_libsmbclient_vc1228502-tseitin	0.431906	316	154597	33775	4.58
smtlib-qfbv-aigs-bin_libsmbsharemodes_vc5759-tseitin	0.501419	888	139574	30089	4.64
smtlib-qfbv-aigs-lfsr_004_127_112-tseitin	0.292715	58	74579	6252	11.93
smtlib-qfbv-aigs-lfsr_008_063_080-tseitin	0.295903	130	64816	6360	10.19
smtlib-qfbv-aigs-lfsr_008_079_112-tseitin	0.325816	158	105644	9868	10.71
sokoban-sequential-p145-microban-sequential.030-NOTKNOWN	0.259026	3	487093	35876	13.58
sokoban-sequential-p145-microban-sequential.070-NOTKNOWN	0.223978	3	1504693	107716	13.97
srhd-sgi-m27-q225-n25-p15-s58217873	0.107995	24	35446	542	65.40
stable-300-0.1-20-98765432130020	0.028899	1	17097	300	56.99
total-10-13-u	0.533613	3808	336030	53321	6.30
toughsat_factoring_958s	0.299018	276	9284	1742	5.33
tph6	-0.011324	5	1716	65	26.40
traffic_pcb_unknown	0.638722	55	698392	32426	21.54
traffic_r_sat	0.586552	88	3158889	97506	32.40
traffic_r_uc_sat	0.767679	619	4453116	453652	9.82
UCG-15-10p0	0.428974	7674	635885	101289	6.28
UCG-15-10p1	0.426842	7686	649235	101968	6.37
UCG-20-10p0	0.436047	10101	915516	136805	6.69
UCG-20-10p1	0.435672	10143	931387	137282	6.78
UCG-20-5p1	0.443108	8932	815936	120706	6.76
UR-15-10p0	0.430162	7661	635846	101272	6.28
UR-20-5p0	0.437544	8927	802143	120287	6.67
urquhart3_25bis.shuffled	0.127052	16	264	70	3.77
UTI-10-5t1	0.483595	2276	220437	36086	6.11
UTI-15-10p1	0.471582	6382	649363	102107	6.36
UTI-20-10p0	0.470823	8322	917592	137619	6.67
UTI-20-10p1	0.462407	9112	933889	137991	6.77
valves-gates-1-k617-unsat.shuffled-as.sat03-412	0.561996	17543	1344664	246554	5.45
VanDerWaerden_pd_2-3-20_388	0	1	20393	194	105.12
VanDerWaerden_pd_2-3-22_443	0	1	26201	222	118.02
VanDerWaerden_pd_2-3-22_462	0	1	28738	231	124.41
velev-pipe-o-uns-1.0-7	0.337154	4	703071	21498	32.70
velev-vliw-uns-4.0-9	0.716108	27	1786538	85471	20.90
vmpc_29	0.180482	16	89294	841	106.18
vmpc_33	0.192585	17	132531	1087	121.92
N	0.138853	45	58492	945	61.90

List of all instances used in the machine parameters experiment

C Regression co-efficients

Factor	Estimate	Std. Error	t value	$Pr(>\|t\|)$	Sig
$\|CO\|$	-1.237e+00	3.202e-01	-3.864	0.000121	***
$\|CL\| \odot Q \odot QCOR$	-4.226e+02	1.207e+02	-3.500	0.000492	***
$\|CL\| \odot Q$	-2.137e+02	6.136e+01	-3.483	0.000523	***
$\|CL\| \odot Q \odot \|CO\| \odot QCOR \odot CVLR$	-1.177e+03	3.461e+02	-3.402	0.000702	***
$\|CL\| \odot Q \odot \|CO\|$	-6.024e+02	1.774e+02	-3.396	0.000719	***
$Q \odot QCOR$	3.415e+02	1.023e+02	3.339	0.000881	***
Q	1.726e+02	5.200e+01	3.318	0.000947	***
$Q \odot \|CO\| \odot QCOR$	9.451e+02	2.927e+02	3.229	0.001292	**
$Q \odot \|CO\|$	4.839e+02	1.503e+02	3.220	0.001335	**
$\|V\| \odot QCOR$	-3.177e+01	1.004e+01	-3.164	0.001617	**
$\|CL\| \odot \|V\| \odot VCLR$	-1.263e+01	4.503e+00	-2.805	0.005163	**
$\|CL\| \odot \|V\| \odot QCOR \odot CVLR$	-2.521e+01	9.008e+00	-2.798	0.005263	**
$\|V\|$	-1.376e+01	4.947e+00	-2.782	0.005526	**
$QCOR$	-1.057e+01	3.912e+00	-2.701	0.007065	**
$\|CL\| \odot \|V\| \odot QCOR$	2.096e+01	7.894e+00	2.656	0.008073	**
$(Intercept)$	-4.949e+00	1.950e+00	-2.538	0.011327	*
$\|CL\| \odot QCOR$	9.486e+00	3.792e+00	2.502	0.012556	*
$\|CL\| \odot \|V\|$	9.641e+00	3.933e+00	2.451	0.014456	*
$QCOR \odot VCLR$	9.035e+00	3.789e+00	2.385	0.017323	*
$VCLR$	4.452e+00	1.892e+00	2.353	0.018845	*
$\|CL\|$	4.299e+00	1.894e+00	2.270	0.023507	*
$\|V\| \odot QCOR \odot CVLR$	1.700e+01	7.556e+00	2.250	0.024755	*
$\|V\| \odot VCLR$	8.059e+00	3.811e+00	2.115	0.034769	*
$\|CL\| \odot \|V\| \odot Q \odot QCOR$	-4.680e+02	2.298e+02	-2.036	0.042060	*
$\|CL\| \odot \|V\| \odot Q$	-2.373e+02	1.167e+02	-2.034	0.042268	*
$\|CL\| \odot \|V\| \odot Q \odot \|CO\|$	-6.594e+02	3.315e+02	-1.989	0.047042	*
$\|CL\| \odot \|V\| \odot Q \odot \|CO\| \odot QCOR$	-1.286e+03	6.469e+02	-1.988	0.047160	*

List of all significant effects, three stars indicates the highest level of confidence that the effect is important. $\odot$ indicates an interaction between two or more factors and *Sig* stands for Significance