

# What Makes a Problem Hard

## Part 1

Jonathan Scott

David Anuta

In an effort to understand theoretical computer science *hardness* we explore a set of problems represented by 3-SAT, which is known to be NP-Complete. The NP-Complete domain, however, contains certain instances of problems which certainly are not NP-Complete *hard* to solve. We use the open-source SAT solver MiniSat[?] and *scripting friends* to empirically test the differing *hardness* distributions within the NP-Complete domain of 3-SAT problems.

## Introduction

Boolean satisfiability problems (SAT problems) are boolean formulas evaluated as to whether or not a possible assignment of all variables in the problem can make the formula TRUE. SAT problems are known to be NP-Complete as all NP problems can be reduced to SAT. Although worst-case runtime is exponential, empirically there are many instances of SAT problems which can be solved in much better time. The implications for understanding SAT problems and solvers are quite important as many world problems require optimization and are solvable by a SAT solver. Therefore, it is of interest to explore the different distributions of problem instances within this NP-Complete domain.

3-SAT is a subset of SAT problems which contain three variables per clause and  $C$  joined clauses. The problem is satisfiable if there is an assignment of variables which makes the conjunction of clauses TRUE. We implement a random generator which selects, for each clause of  $C$  clauses generated, three variables out of a set of  $V$  variables, without replacement. To get the empirical data on problem *hardness*, we hypothesize that the ratio  $R = \frac{C}{V}$  and *hardness* have a direct relationship. Therefore, we run  $T$  trials through the open-source SAT solver MiniSat with various  $R$  and record the Satisfiability of the formula and runtime statistics.

## Implementation

### MiniSat

The Minisat executable reads from standard input or a text file with the form

```
p cnf NumVARIABLES NumCLAUSES
LITERAL LITERAL LITERAL 0
LITERAL LITERAL LITERAL 0
...
```

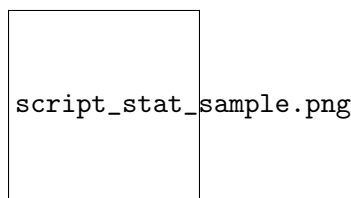
The output of the MiniSat Solver is in the form

### Random 3-SAT Generator

We scripted/built our 3-SAT generator in Python 3. The script (SATgen.py) must be passed two arguments when run: the number of variables to use ( $V$ ) and the number of clauses to generate ( $C$ ). Using these numbers, it then generates a list of  $V$  numbers, ranging from 1 to  $V$  to represent the input variables. It then starts to generate random clauses by sampling 3 random numbers from the generated list of numbers, none of which are equivalent to each other. In other words, our 3-SAT generator employs random sampling without replacement when picking the 3 variables for a clause. Despite this, the 3 variables randomly picked are replaced after each clause is generated. With each of the variables sampled, the script also randomly determines whether the sampled number (which represents a variable in the 3-SAT problem being generated) should be negative or positive. This will represent the truth value of a variable, meaning that positive numbers (variables) are true and negative numbers (variables) are false. Once all 3 variables have been randomly picked and assigned random truth values, our generator moves on to the next clause. It does this until it has generated a list of  $C$  clauses, each having 3 variables selected randomly from a list of  $V$  variables. The output of this script is resulting 3-SAT problem in the text file format designated as input for the MiniSat solver.

### Scripting with Bash Friends

We compiled/built/scripted several different useful scripts in both Bash and Python 3 to aggregate and present all of our data. The first two of these scripts (script.sh and run\_tests.sh) were Bash scripts which simply aggregated 3-SAT problem data that we could use. Script.sh generated, solved, and collected data from a given number of 3-SAT problems. The script took in two arguments when run: the number of clauses ( $C$ ) and the number of 3-SAT problems to generate and collect data from (this is also referred to as the number of tests to run or  $T$ ). This script then generated and collected data from  $T$  3-SAT problems by using the Python generator described above  $T$  times, passing it  $C$  clauses and 100 variables (as specified in the project guidelines). The output is fed into MiniSat to solve. During the process the solutions generated by MiniSat were each temporarily written to a text file and the 'awk' Bash command was used to get obtain both the number of decisions and satisfiability of each problem. After having generated and tested  $T$  3-SAT problems for satisfiability, the percentage of problems satisfiable, ratio of clauses to variables ( $R = \frac{C}{V}$ ), and several other stats pertaining to the number of decisions made overall were calculated and output to a text file. An example of such a text file is shown below.



The second script mentioned (run\_tests.sh) was a simple extension of the first script which took in a range of clauses ( $C_1$  to  $C_2$ ) and a number of tests ( $T$ ) and ran script.sh using each number of clauses within the range  $C_1$  to  $C_2$ , using  $T$  tests each time. This script was used in conjunction with the 'nohup' Bash command in order to run 1000 tests of satisfiability for problems with 300 to 500 clauses as a background process. These two scripts were what we used to collect data directly from MiniSat solutions. The rest


of our scripts either manipulated the data into a more presentable format, or analyzed the data to gather more information.

With regards to analyzing our data, the two scripts `hypothesis_test.sh` and `binom_test.py` (Bash and Python 3 scripts respectively) were used to analyze the data for statistical significance. For each of the MiniSat results, corresponding to a summary of  $T$  tests per ratio  $R$ , the `hypothesis_test` script employed a single sample proportion test using the python script `binom_test.py`. The hypothesis took the form

$$H_0 : \text{Sample Proportion} = \text{True Proportion (satisfiability probability)}$$

$$H_a : \text{Sample Proportion} \neq \text{True Proportion}$$

for each of the True Proportion targets (90%, 50%, 10%). Each hypothesis test was appended as a new line to a text file. Each line contained the hypothesis data and all data which was in the corresponding results file. An example of part of the text file obtained is shown below.



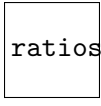
`p_value_sample.png`

For our purposes, we were interested in only the ratios where the null hypothesis was not rejected. That is to say, we could not statistically prove that the sample proportion differed from the true proportion. The `gather_sigratios.sh` used the 'grep' command to collect all results from the output of the hypothesis tests which read "Reject Null Hypothesis: False."

After analyzing all this data, we created several scripts to present the data. The `plot_ratios.py` and `plot_hardness.py` scripts (created in Python 3) were created and used to read in data from the text file you see above and generate the graphs seen later on in this report using the Python libraries MathPlotLib and Pandas.

## Results

### Satisfiability



`ratiosVSpercentages.png`

The problem statement asked to find the ratios ( $R$ ) of Clauses to Variables which had a Satisfiability rate of 90%, 50%, and 10%. While the empirical results of 1000 trials per  $R$  have a clear trend, exact matching percentages were not obtained. We believe that as the number of trials increases, the percentages will become more precise. The time requirement on this is too severe for our purposes. Instead, we decided to run binomial hypothesis tests on the results to determine which ratio's percent satisfiability could be considered different from the target percentages of 90%, 50%, and 10% at the 0.05 significance level. We obtained the following results using a simple binomial single sample proportion hypothesis test.

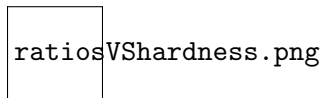
Null Hypothesis: Sample Satisfiability Proportion = True Satisfiability Proportion

The ratios with the closest percent satisfiable to the target proportions were

90%: 4.02 (90.5%)  
 50%: 4.28 (51.3%)  
 10%: 4.6 (10.2%)

## Hardness

With these ratios came the complexity/hardness of the problem, which is usually measured using the run time of a program. For SAT problems this can also be quantified by the number of decisions that had to be made to determine whether the problem was solvable or not. Using MiniSat, the number of decisions that were made when determining the satisfiability of a problem has a direct correlation to the run time. We use this correlation to get a basic idea of what the run time of each ratio looks like. The results for both the average number of decisions and median number of decisions for the 1000 trials of each ratio we compiled are shown below.



## Conclusion

**What hypotheses do these results suggest about the relationship between the clause to variable ratio and the percentage of SAT instances?**

The results we found suggest that there is a direct relationship between the satisfiability of the problem and ratio of clauses to variables. That being said, this relationship is only true within a certain range of ratios. From our data it would seem that from 3.80 to 5.00, the relationship is direct. One would expect a similar trend after a ratio of 5.00 was reached, however, our data is limited and thus we cannot conclusively say at what ratio the upper cut off of problem satisfiability would occur. It seems evident from the data, however, that at some point, on average 0% of all problems will be satisfiable and this will continue to be the case as  $R$  increases. Above this point (roughly 5.00) and below the lower cutoff point (roughly 3.80), the direct relationship between the ratio and the run time no longer holds.

**What hypotheses do these results suggest about the relationship between the percentage of SAT instances and the average problem hardness?**

Our results actually suggest that average problem hardness peaks around when the ratio of clauses to variables is 4.5. At this point in our data, the average, median, and total number of decisions all clearly decline. This indicates that though the run time of the problem starts getting longer as the problems get harder, at a certain point, the immense amount of disproportionality between the clauses and the variables causes many boolean conflicts within the formulas. Therefore, MiniSat will make fewer decisions to determine that a formula is unsatisfiable when the ratio of clauses to literals is much higher. This may mean that problems with very high clause to variable ratios (such as 10.00 or greater) actually have similar run times to problem with relatively low clause to variable ratios. With our limited data, we do not know whether this is a fully accurate assumption since there may be an upper ratio where run time

stops decreasing and simply levels off. Due to the nature of our data we cannot predict when something, or if something like this would occur since we only have average and median run times for decisions up to the 5.00 clause to variable ratio.

**What hypotheses are suggested by the observed differences between the median and mean of the run times?**

Interestingly enough, the median run time (i.e. median number of decisions) for each ratio, is fairly consistently less than the average run time (i.e. number of decisions). This would suggest that the average run time is being pulled upwards by some large outliers meaning that most ratios had some very bad worst case timings. Basically, within each of our batches of 1000 tests, MiniSat consistently got held up; taking longer than, on average, should be expected. The higher average could be due to the fact that MiniSat has to do a tree search through the solution space for possible satisfiable assignments. This means, that if MiniSat picks a bad assignment it has to backtrack back up to a point where it can pick a correct assignment. So one possible cause of this higher average run time, could be the fact that there are times when MiniSat accidentally picks the wrong assignment early on, leading to a longer run time because it has to explore farther down the tree and then backtrack back up to a correct assignment.

## **Appendix**