SAT solver performance on Sudoku

Bishop, Robert; Graves, Caleb; and Nagisetty, Vineel

July 6, 2018

Abstract

Sudoku is a popular number game played all over the world. In this report, we generate different random instances of varying sized Sudoku boards, reduce to solve using a SAT solver, consider the chance a randomly generated Sudoku board is satisfiable, and build our own Sudoku solver. This report is created for the course COMP 6902. Our code for this project can be viewed at https://github.com/vin-nag/Sudoku-SAT

1 Introduction

1.1 Sudoku Description Problem

Sudoku is a puzzle based on Combinatorics. The traditional Sudoku puzzle is a 9 x 9 board with a number of squares filled in with integers from 1 to 9. The player has to use logic and reasoning to fill in the rest of the squares with integers until the entire board is full. Formally we can define Sudoku as:

Instance:
An N by N Sudoku board, filled with an arbitrary amount of numbers with the symbols $\in [1, N]$

Acceptance Condition:
Accept if the Sudoku board has a valid solution. A Sudoku board is said to have a solution if for all rows, columns and n by n sub-squares, the Sudoku board contains each of the N possible numbers.

We will be converting an instance of a Sudoku board with N randomly generated and placed integers from 1 to 9 in the squares of the board to SAT. From there, we use a SAT solver to verify if the randomly generated Sudoku instance has a solution and, if so, we can convert that back into a solved Sudoku game board.

1.2 Sudoku Rules Primer

The game is played on an $N \times N$ board where N is a square. Here N will be referred to as the order of the board, such that a 4×4 board is a board of order 4. Note that while the canonical version of Sudoku uses a board of order 9, the problem can be abstracted to arbitrarily sized

boards.

For the board to be completed in a winning state, all of the following must apply:

- The value v of any number in a cell on the board is an integer $\in \{1, N\}$
- Each cell of the board must contain a single value
- Every row r of the board must contain one, and only one instance of each number $\in \{1, N\}$
- Every column c of the board must contain one, and only one instance of each number $\in \{1, N\}$
- Every "sub-square" s of the board must contain one, and only one instance of each number $\in \{1, N\}$

When the entire board is filled and all rules are met, the puzzle is solved.

2 Reducing Sudoku to SAT

2.1 Overview

As an NP-Complete problem, the game of Sudoku can be reduced to a SAT solvable state by formalizing the rules into a series of conjunctive statements. We will present the process of this reduction here with the intention of creating a CNF file that may be used in a commercial SAT solver.

The Boolean Satisfiability problem (SAT) is, given a series of Boolean variables, finding if there is a possible variable assignment that produces a satisfied (True) output.

SAT is NP-complete, and any problem that can be reduced to SAT can be shown to also be at most in NP. SAT was the first problem proved to be NP-complete (from the Cook-Levin Theorem) and since then many NP-completeness proofs involve a reduction to SAT. In this instance, we will reduce an instance of a Sudoku puzzle with k randomly generated squares filled in and the rest blank to SAT.

2.2 CNF Interpretation of Sudoku Rules

Here, we will use the notation $x_{rcv}|r, c, v \in [1, N]$ to represent the location and value of a cell. For example: x_{123} represents a value of 3 in the cell at position $(1, 2)$.

Variables: This gives us N^3 possible variables.

Constraints: To formalize the rules we will adopt the nomenclature from Kwon & Jain[GK09] and break the cell, row, column, and subsquare rules into two separate constraints:
definedness: each cell, row, column and subsquare must contain one number
uniqueness: each cell, row, column and subsquare must contain no duplicate numbers

- Definedness Constraints:
  For cells, definedness requires listing each possible permutation of values for each cell:

  $x_{000} \vee x_{001} \vee x_{002} ... \vee x_{00N} \wedge ...$
  $x_{010} \vee x_{011} \vee x_{012} ... \vee x_{01N} \wedge ...$
  ...

  For rows, columns, and subsquares definedness definedness is performed by fixing the values of the other variables and generating each possible permutation. For example, to represent definedness of a row we iterate through each permutation of c, v and define a clause for each cell within the row:

  $x_{000} \vee x_{010} \vee x_{020} ... \vee x_{0N0} \wedge ...$
  $x_{001} \vee x_{011} \vee x_{021} ... \vee x_{0N1} \wedge ...$
  ...

- Uniqueness Constraints:
  Representing uniqueness in CNF is done by conjoining every possible permutation of duplicate values as negations. For cells, for example we use the form $\overline{(rcv_1)} \vee \overline{(rcv_2)}$ for each possible permutation of v_1, v_2:
  $\overline{x_{000}} \vee \overline{x_{001}} \wedge ...$
  $\overline{x_{000}} \vee \overline{x_{002}} \wedge ...$
  ...

  This same approach holds for rows, column and subsquares. Though defining iteration through subsquares is mathematically somewhat more complex, the principle of fixing the other values and iterating through each permutation remains the same.

2.2.1 Constructing the Sudoku Structural CNF

We used a structural CNF that contains our rules structure, encoding our constraints into a series of clauses. These are constructed via a sequence of iterative loops that generate clauses for uniqueness and definedness constraints in DIMACS format. For example, the line:

111 112 113 114 115 116 117 118 119 0

determines that the top left square (1,1) must contain a number from 1 to 9, while the lines:

-110 -120 0
-110 -130 0
...
-170 -190 0
-180 -190 0

determines that if you have a 0 in any two squares in row 1, your Sudoku board is unsolvable (and thus the CNF is unsatisfiable).

2.3 Defining the Playing Field

Typical Sudoku puzzles begin with a board of mostly empty squares with several values already filled in. To represent these pre-defined squares in the above form, you simply append a single (rcv) representing each filled cell. Doing so will allow a traditional SAT solver to determine if the board has a solution (obviously all commercial puzzles have a solution, but the reduction presented here will determine the satisfiability of any arbitrarily filled board).

Using the above method it is fairly simple to iteratively create a CNF definition of the rules of Sudoku such that any boards satisfiability can be determined. It is worth noting that much of the modern literature on the topic is dedicated to finding more efficient implementations as the above method generates much redundancy. In practice, each defined cell effectively reduces the possibility space a great amount, and several of the iteratively generated clauses are redundant with each other.

3 Generating Random Instances of Sudoku

We used a bespoke Monte Carlo distributive small-batch random number generator to fill an empty Sudoku board with n random integers $\in [1, 9]$. If the same square is randomly chosen twice, we simply generate another random number in a new square so our resultant Sudoku board has an actual n filled squares. We then have a script that converts the filled squares to a series of CNF clauses with 1 variable each. Appending these clauses to the structural CNF gives us a CNF that represents a Sudoku board, and a satisfiable solution to the CNF is a winning solution to that board. For example, if we randomly generated:

							7	
								3
	3							
			5		2			

Unsolved Sudoku

Our script would encode the CNF as:

187 0
293 0
523 0
645 0
662 0

This CNF would be appended to the structural CNF and the SAT solver would find that it is unsatisfiable (it violates the uniqueness clause for the top right subsquare, having two squares marked ″7″).

3.1 Satisfiable CNFs to Sudoku Solutions

We can build the solution from a satisfiable output (assuming there is one) using the SAT solver. A valid solution will output N^2 true assignments of the form x_{rcv} which give the row, column and value for each cell of the board. We can take those values and build a valid Sudoku board from them.

4 Experimental Results

4.1 System Specs and Software

The project was executed on a LABNET computer using Ubuntu 16.04 in the Memorial University computer science lab. The computer has a Intel i5-3470 CPU chip-set running at 3.20GHz with 7.8 GB of RAM.

We used python as our main programming language. Our Sudoku class, Generator and Reducer were built using python. We also used Jupyter notebooks to run our experiments, since we can run, plot and present the results easier using this tool.

The code for our project with the experimentation results can be viewed at https://github.com/vinnag/Sudoku-SAT/

4.2 SAT Solver

Minisat is our solver of choice[1]. Minisat is an award winning modern SAT solver. It takes in a DIMACS format file as input, so we can further test our reductions using other SAT solvers with the same input file.

We checked to see if the solver works as intended by giving it a simple input file with 3 variables and 2 clauses. We encoded the formula: $(x \vee \neg z) \wedge (y \vee z \vee \neg x)$. The solver stated this formula is satisfiable, and output values 1 2 -3. This can be interpreted as x, y are true, and z is false, which is a satisfying assignment to this formula.

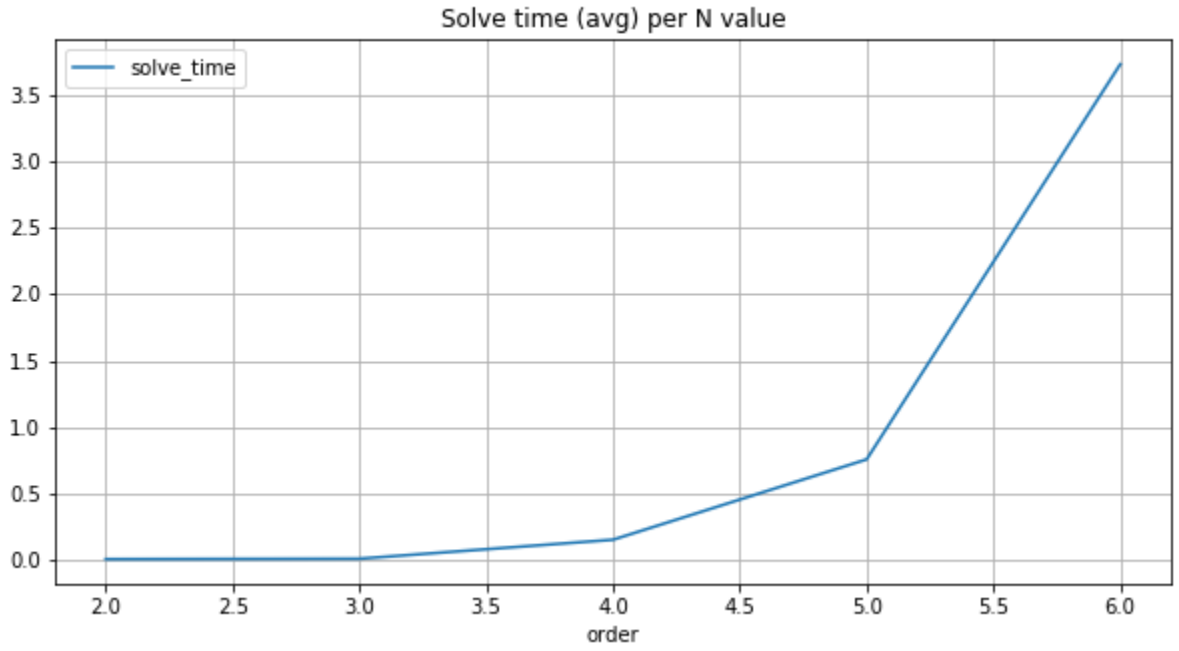
[1] http://minisat.se/

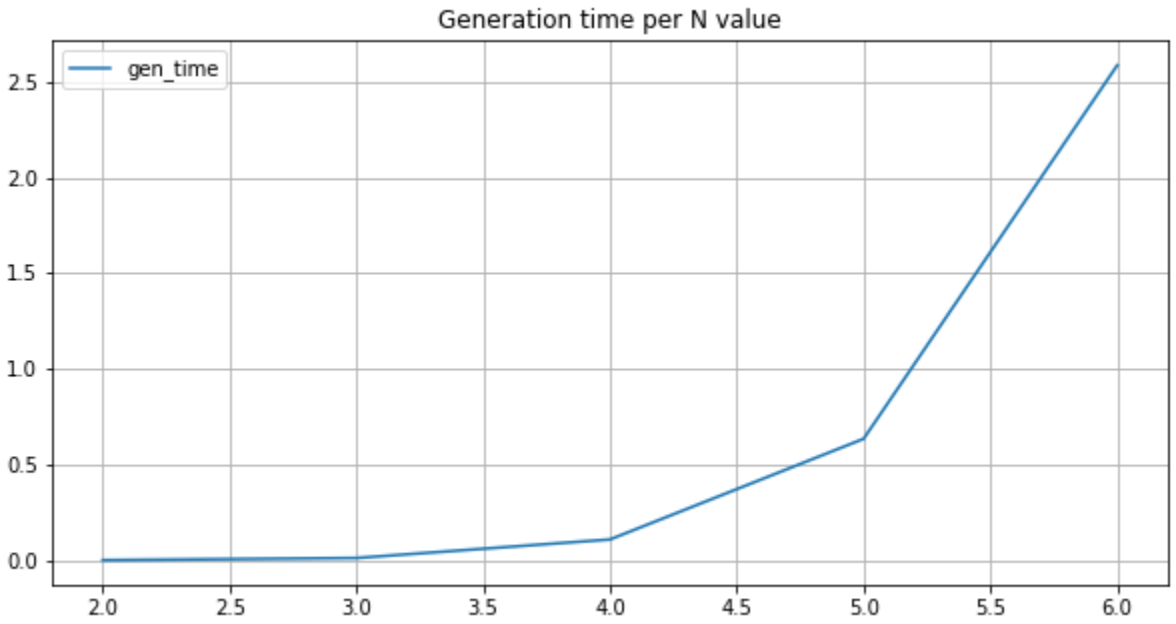
4.3 Experimentation Setup

Our main experiment includes testing the time required to reduce and solve different sized Sudoku boards. A puzzle can only follow the standard sub-square rules if the number of columns and rows is a square of an integer, so we will do additional experiments on boards of order 16, 25 and 36. The number of clauses is dependant on the size of the board, so while a standard Sudoku board of order 9 has 11,988 clauses in the eventual CNF, a board of order 16 generated 123,908 clauses and a board of order 25 generated 752,504 clauses. The number of clauses grows exponentially compared to the size of the board. We check to see if the reduction and solve time are exponential.

4.4 Average Performance of the Solver on Our Instances

Given that the relationship between N and the number of clauses in the completed CNF is exponential, we believed the SAT solver's run time and N would exhibit an exponential relationship. To confirm this, we tested the SAT solver on 10 instances of randomly generated boards from order 4 to 36, each with one random k value appended (ie. one pre-filled square). Each instance was converted into a full CNF and fed into our SAT solver. Below are the graphs for the average time to solve boards of each N value, as well as the time taken to generate the full CNF file:





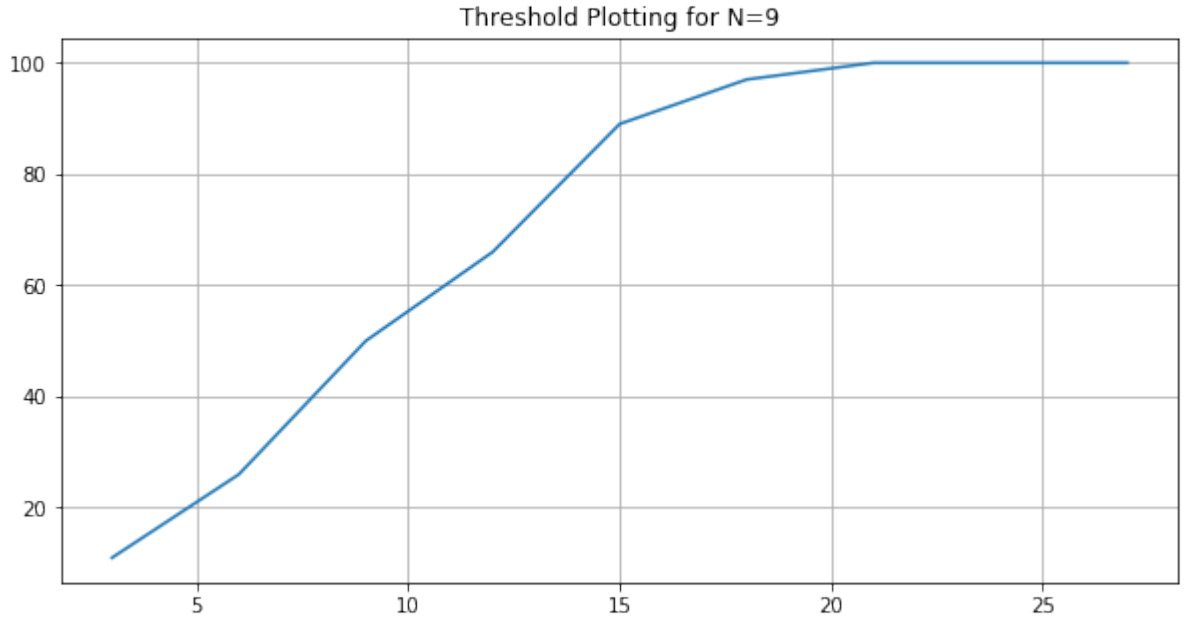
The generation and solve times clearly follow and exponential curve, which conforms to what we know about the exponential growth of the clauses of our CNF in relation to the N value of the Sudoku board.

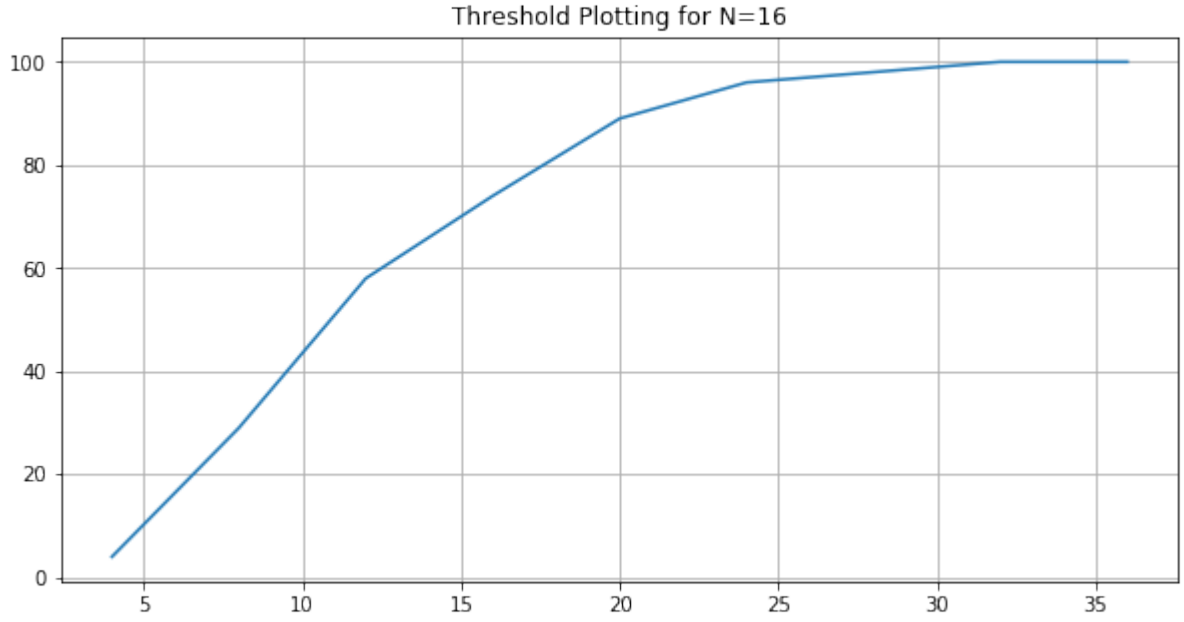
5 Additional experiments

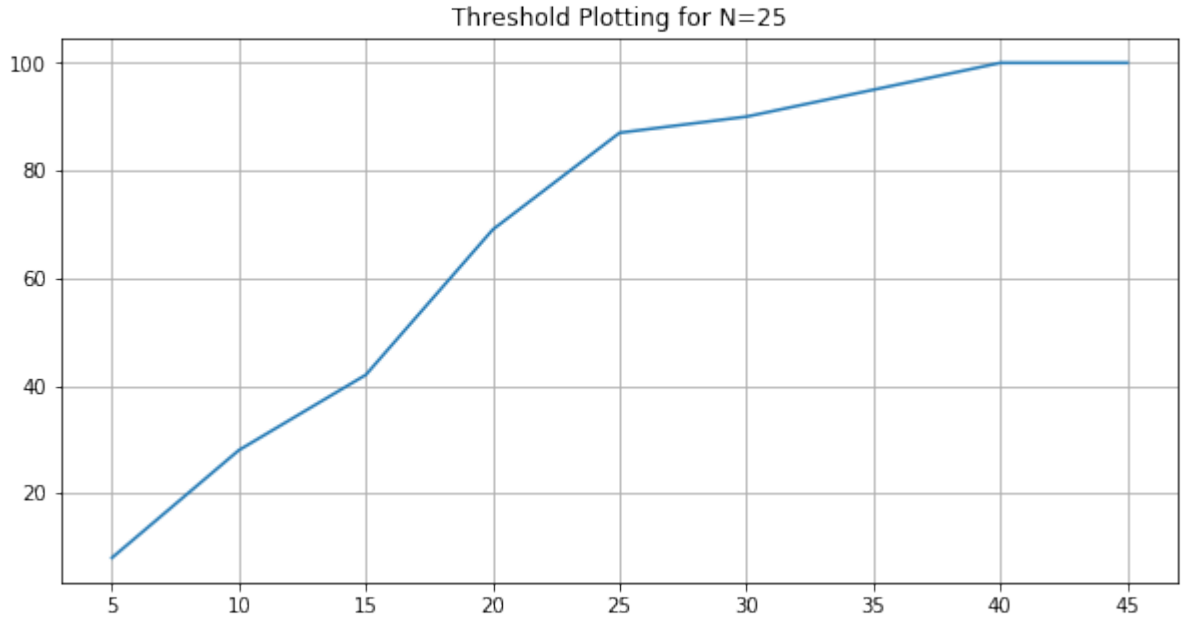
5.1 Determining the Threshold Limit

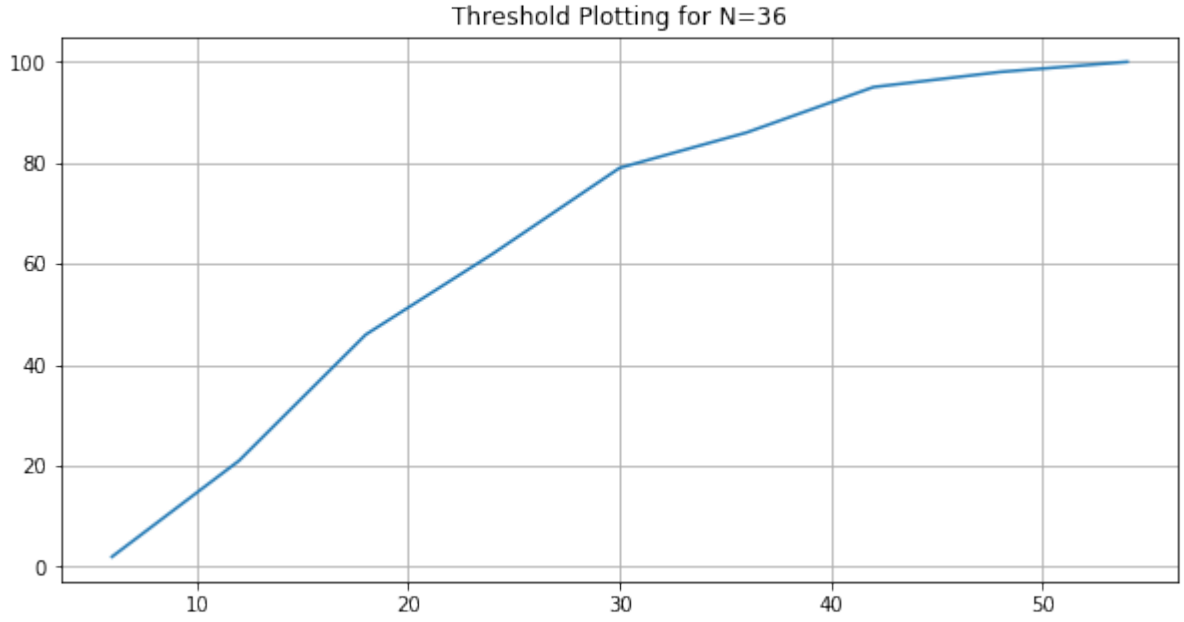
We wanted to see how the number of randomly generated squares affect the unsatisfiability of a Sudoku board. Here, we denote the number of randomly generated values as k. We define the threshold limit as the k value which leads to a 50% unsatisfiability on average for a board of order N. It makes logical sense for a Sudoku board with a higher k value to have a greater chance to be unsatisfiable since these generated values can contradict with the rules of the game. For example, the greater the number of randomly generated values in a particular row, the higher the chance that two values are the same, making the entire board unsatisfiable. The same logic applies to every row, column and sub-square. But how many values is too much? We aim to answer this question.

Since the number of squares in a board are a function of N, it makes sense that a bigger Sudoku board might be satisfiable on average with a higher k value. Therefore, we have made the initial and final size of k a function of N. For a given Sudoku board of size N, our k ranges between $\{\sqrt{N}$ and 10N. We also increment k by $\sqrt{N}$ each time. For each k value, we will be generating 100 random Sudoku boards with k number of filled squares. Our goal is to find out what k value leads to 50 percent unsatisiable board on average. The results from this experiment are plotted below. X-axis represents the k values, while Y-axis represents the percentage of boards which are unsatisfiable for a given k value.


Threshold Plotting for N=9
100
80
60
40
20
5
10
15
20
25


Threshold Plotting for N=16
100
80
60
40
20
0
5
10
15
20
25
30
35





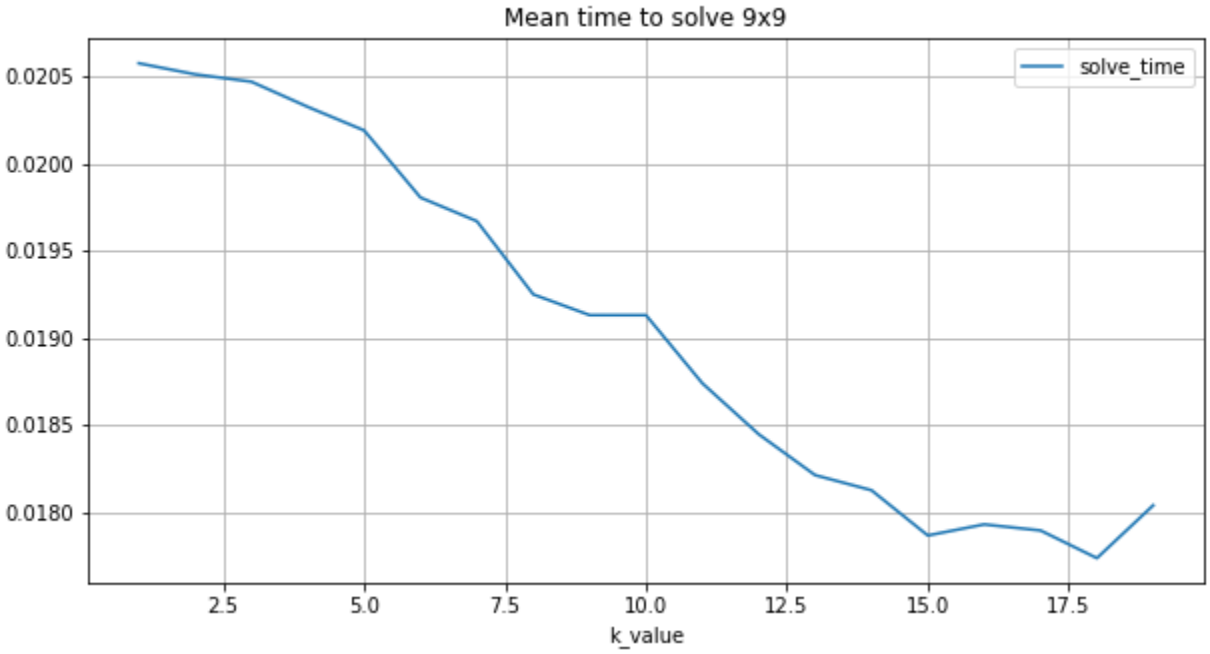
Each of the graphs shows a clear pattern between the percentage unsatisfiable and k for a given board of size N. Our findings suggest that:

Order	**k Value**
9	8
16	12
25	17
36	21

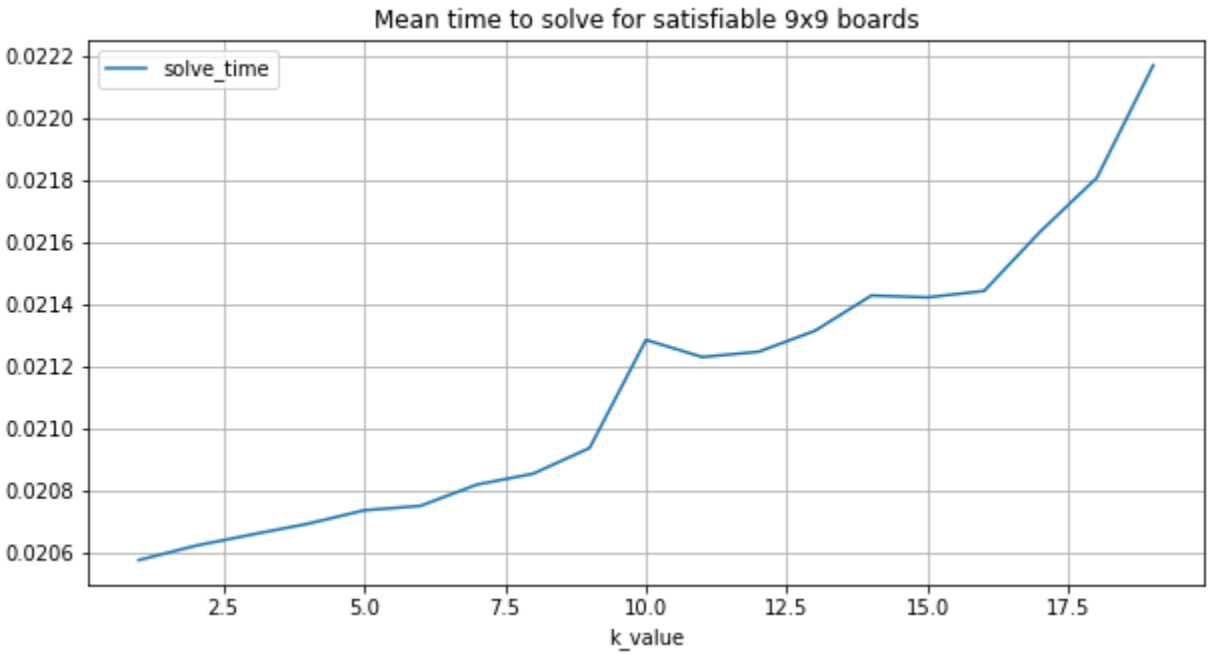
This makes sense because after k=N, we are averaging more than one value in a given row or column and therefore the chance to have two or more values be equal in a given row, column or sub-square increases.

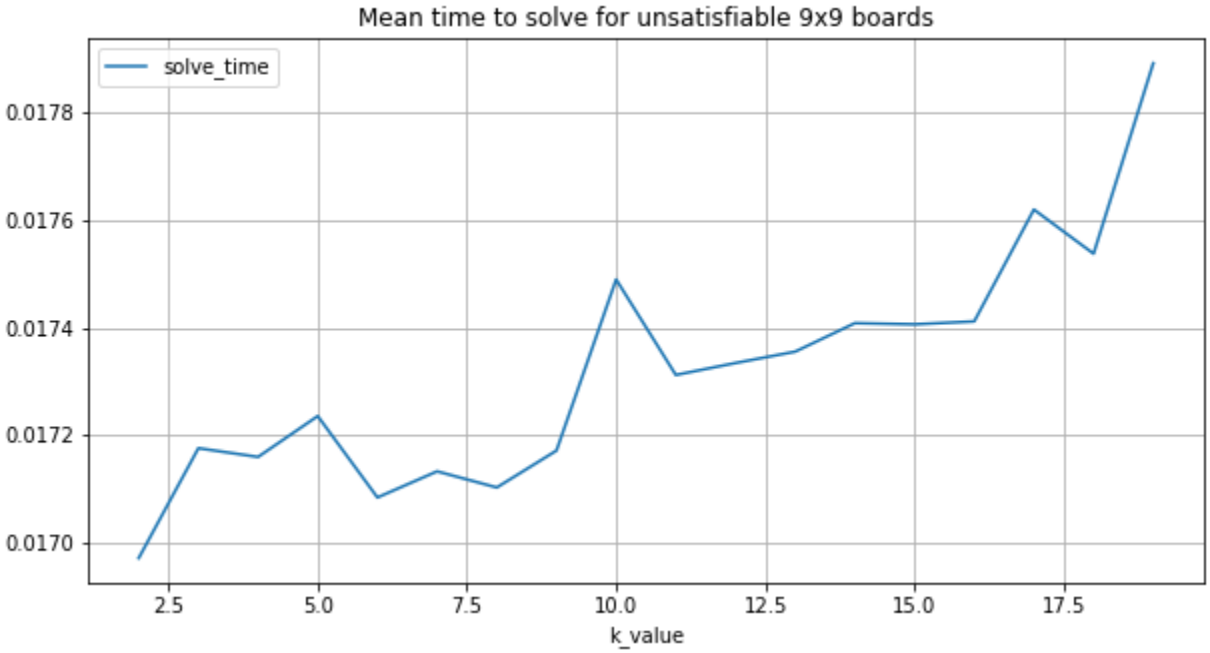
5.2 K Values and Solve Time

Knowing that there is a relationship between the N value of a board and the time to solve, we decided to see if there was a relationship between the k values, and timing. To test this, we fixed our boards N size at 9 and created 500 instances of randomized boards with k values ranging from 1 to 19. The results are displayed below:



It seems as though the boards become progressively easier to solve as the value of k (and as demonstrated above, the overall unsatisfiability) increases, however it is worth examining these two types of boards as separate classes, and seeing the trends within satisfiable and unsatisfiable boards:





Splitting the results into two classes shows us that within each class the difficulty of solving increases as K increases, however as the overall average time to solve an unsatisfiable board is much lower, and their porportion increases as K grows, we see the overall trend of solve time decreasing.

5.3 Building a Sudoku Solver

A satisfiable CNF can be easily converted into a solved Sudoku board by filling the board with numbers based on which variables are true. For example, if the variable 341 is true in the satisfied CNF, then the square (3,4) contains a 1. Due to the structural CNF limitations it is impossible for a satisfiable CNF configuration to violate Sudoku rules. We have a script that easily converts the satisfied CNF to a Sudoku board layout so we can begin with a random board and see a possible valid solution to that board.

Additionally, instead of using the randomly generated Sudoku boards that we were using, we can create our own Sudoku puzzle using values we fill in. In this way we can solve Sudoku puzzles from other sources or simply create our own by hand. We created one using this idea, which is located in our github repository.

6 Conclusion

We were successfully able to determine if a Sudoku board has a solution by reducing it to an instance of a SAT problem, and feeding that to a SAT solver. We then plotted the average time it takes to both generate and solve a board of size N for both Satisfiable and Unsatisfiable bords. We then ran experiments to see the relation between the number of randomly generated values vs the satisfiability for different sized boards. We used this code to build a Sudoku solver which can take a given puzzle as a list and print out a solution for it, if it exists.

6.1 Discussion

The parameters we chose worked well to figure out the threshold value, since we made the number of randomly generated values be a function of the size of the Sudoku board. We found minisat to work really well for us. However, we would have liked to try other solvers, and other heuristics to compare and contrast solve times for Sudoku boards.

We also used Jupyter to run the bash command using a separate function for added ease. Using a bash command through terminal would have improved the solving time.

6.2 Further Research Ideas

We tried creating Sudoku boards of size greater than N=49 but the average run time was significantly higher and the generated CNF file was much bigger in size. For example, running 1000 instances for N=36 took 38 minutes and 41 seconds, and so we didn't investigate running bigger sized boards. Given enough computational resources and time, we would have liked to try out experiments with bigger boards.

References

[GK09] Himanshu Jain Gihwon Kwon. Optimized cnf encoding for sudoku puzzles. 2009.