**Project 1: SAT-based Sudoku Solver**

[***https://github.com/GarethMarriott/CSC322A1***](https://github.com/GarethMarriott/CSC322A1)

Oliver Diaz

Gareth Marriott

Sree Vyshnavi Tejovath

Sadhik Ahamed Shaik

# Introduction

The goal of this project was to construct a SAT solver capable of finding the unique solution to a traditional 9x9 Sudoku grid. Our program is designed to accept an unsolved Sudoku grid and through use of miniSAT solver in conjunction with a built CNF formula will generate the solved puzzle as a text file. Alongside the solved puzzle, *stat.txt* is created containing statistics to be used later in this report. Due to familiarity, we used Python as the language for our translators. In this project we test minimal and efficient encoding. The following report consists of brief descriptions of tasks completed followed by a performance evaluation section where the results of these encodings are compared and further explored.

# Basic Task

The basic task consisted of using minimal encoding in the Python program *sudtosat* to solve the grid. For this task, input was specified as a titled grid with nine lines of nine characters each using 0’s to represent unknown characters. The minimal encoding in this program was completed using multiple nested loops to form a CNF. The result of *sudtosat* is a CNF formula in DIMACS format capable of being read by MiniSAT solver. The MiniSat solver would then process the CNF and a second Python program named *sattosud* is used to format the results into a solved Sudoku puzzle *solution.txt*.

### Shell Scripts

The command ` sh run.sh ` will run the basic task shell script. This script calls the *sudtosat* program inputting the base puzzle from the program specification and outputting the CNF formula in DIMACS. The shell script will then run minisat outputting the stats to a results folder. Finally the script will convert the minisat output to the result sudoku and put it in the results folder.

A similar process can be run with the ` sh run\_puz.sh {./path-to-puzzle.txt} ` command testing any sudoku puzzle that meets the format specified in the project spec.

### Performance monitoring

All statistics about performance are gathered from the MiniSAT statistics outputted to standard out when the program runs. To gather these statistics two additional shell scripts were created *get\_simple\_stats.sh* and *get\_ET1\_stats.sh*.

*get\_simple\_stats.sh* completed all 49 simple puzzles, extracting the key statistics using *extract\_stats.py* and appending them to a file for review.

*get\_ET1\_stats.sh* does the same thing but for the ‘harder’ extended task one puzzles.

# Extended Task 1

For this extended task, the objective was to further test the minimal encoding solution on additional unsolved inputs. These new inputs to be used varied only slightly and were formatted as a single line 81 characters long using .’s to represent unknowns. Due to this, the task was simple and this format of input was already handled by the base program *sudtosat.*

# Extended Task 2

To incorporate the efficient encoding, we need to extend the basic task script with additional constraints that ensure the Sudoku puzzle adheres to the rules more strictly. Specifically, we need to add constraints that ensure each cell contains at most one number. The current setup already encodes cells with specific numbers, so we need to focus on adding constraints that enforce the uniqueness of numbers in each cell, row, column, and subgrid more efficiently.In the script's structure, we added code to generate clauses that ensure no cell can have more than one value. And also Ensured the existing encoding efficiently represents the requirement that each number must appear once in every row, column, and subgrid.

1. Encoding (sud2sat2): It translates each cell's constraints (a cell must have exactly one number from 1 to 9, which doesn't repeat in its row, column, or 3x3 subgrid) into a series of logical statements (CNF). It assigns a unique variable for each possible number in each cell and creates clauses that represent Sudoku rules.

2. SAT Solving: A SAT solver is then used to find a solution to the CNF formula Decoding (sat2sud2): After the SAT solver finds a solution, this function translates the variables that are true back into their corresponding numbers in the Sudoku puzzle. It reconstructs the 9x9 grid with the solved values.

This approach is an efficient way to solve Sudoku puzzles because it leverages the power of SAT solvers, which are highly optimized for solving boolean logic problems. The encoding focuses on minimizing the number of clauses to improve solver performance. The goal is to reduce the size of the problem the solver has to work on and potentially speed up the solution time.

# Performance Evaluation

#### **Basic Task (Minimal Encoding):**

An additional script *get\_simple\_stats.sh* was created to test all 49 inputs and format the statistics provided by miniSAT allowing us to summarize the results as the following. It’s important to note that all these Sudoku instances were deemed satisfiable.

Average-case statistics:

| Restarts | 1 |
| --- | --- |
| Conflicts | 9 |
| Propagations | 965 |
| Conflict Literals | 51 |
| Memory Used (MB) | 11 |
| CPU Time (s) | 0.00367 |

Worst-case statistics:

| Restarts | 2 |
| --- | --- |
| Conflicts | 117 |
| Propagations | 5316 |
| Conflict Literals | 820 |
| Memory Used (MB) | 11 |
| CPU Time (s) | 0.007548 |

#### **Extended Task 1 (Minimal Encoding):**

The script *get\_ET1\_stats.sh* was used to generate the same formatted stats as achieved above. Similarly, all 95 inputs were deemed satisfiable.

Average-case statistics:

| Restarts | 2 |
| --- | --- |
| Conflicts | 155 |
| Propagations | 6751 |
| Conflict Literals | 1968 |
| Memory Used (MB) | 11 |
| CPU Time (s) | 0.00547 |

Worst-case statistics:

| Restarts | 7 |
| --- | --- |
| Conflicts | 1144 |
| Propagations | 47440 |
| Conflict Literals | 17857 |
| Memory Used (MB) | 11 |
| CPU Time (s) | 0.011852 |

When comparing the statistics from the minimal encoding results, it is evident that the puzzles used in Extended Task 1 were much more complex to solve due to less given numbers and more unknowns. This is shown through the much higher average amount of propagations and conflicts illustrating the increased complexity in the CNF formulas created for Extended Task 1.