Sudoku Solver
(using Propositional Logic)

Shashank Vij

Degree Programme: Meng (Hons) Computer Science
Project Supervisor: Dr. Konstantin Korovin
Date: April 2016
Project Title: Sudoku Solver (using Propositional Logic)

Student: Shashank Vij
Supervisor: Dr. Konstantin Korovin

Abstract

The aim of the project was to create a tool that could solve the well known Sudoku puzzles. Since the puzzle contains almost 17 million different combinations, the challenge was to solve it achieve efficiency along with the right solutions. The puzzle consists of a 9 by 9 grid of cells, in such a manner that every row, column and sub grid consists of every number from 1 to 9. In the past, techniques such as crosshatching and pencilling in have been used to solve the puzzle manually, and algorithms such as backtracking have been implemented for automated solving of the puzzle.

The task was approached as a satisfiability problem and the report illustrates how the propositional formulae were generated and implemented. A satisfiability solving application programming interface (API) was used in order to generate a model for the clauses, and the solution was extracted from this model

The report also outlines the types of tests that were undertaken and how the results were evaluated. It is concluded with a reflection on the development process, knowledge gained and the achievements of the project.
Acknowledgements

I would like to thank my supervisor, Dr. Konstantin Korovin, for his guidance and feedback. His suggestions played a key role in the development of the application.

I would also like to thank my family and friends, for their support and suggestions throughout my life. I cannot begin to describe how important all of you are to me.
# Table of Contents

**CHAPTER 1 - INTRODUCTION**

1.1 BACKGROUND ........................................................................................................... 1  
1.2 PROJECT PROPOSAL AND OBJECTIVES ............................................................... 2  
1.3 EXISTING SOLVERS ................................................................................................. 2  
1.4 APPROACHING THE PROBLEM ............................................................................. 3  
1.5 REPORT OVERVIEW ................................................................................................. 3  

**CHAPTER 2 – TECHNICAL BACKGROUND** .................................................................. 4  

2.1 PROPOSITIONAL LOGIC .......................................................................................... 4  
2.2 SATISFIABILITY PROBLEM .................................................................................... 6  
2.3 METHODS OF SOLVING SAT ............................................................................... 6  
  2.3.1 Truth Tables ....................................................................................................... 6  
  2.3.2 Size of Truth Tables ......................................................................................... 7  
  2.3.3 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm ................................... 7  
  2.3.4 DPLL Complexity ............................................................................................ 8  

**CHAPTER 3 - DEVELOPMENT** ................................................................................... 9  

3.1 DESIGN .................................................................................................................... 9  
  3.1.1 Functional Requirements .................................................................................. 9  
  3.1.2 Non-functional Requirements ......................................................................... 10  
  3.1.3 System Architecture ....................................................................................... 10  
  3.1.4 Flow of Program ............................................................................................. 12  
  3.1.5 Sudoku Solver ................................................................................................ 13  
  3.1.6 Sudoku Game ................................................................................................ 14  
3.2 IMPLEMENTATION ................................................................................................. 15  

Student ID: 9138137
3.2.1 Propositional Formulae for Sudoku Puzzles ................................................................. 15
3.2.2 Sudoku Solver .................................................................................................................. 16
3.2.3 Sudoku Game .................................................................................................................... 18

CHAPTER 4 – TESTING AND EVALUATION ......................................................................... 21
4.1 Unit Testing .......................................................................................................................... 21
4.2 Correctness Testing .............................................................................................................. 21
4.3 User Testing ........................................................................................................................ 22
4.4 Performance Evaluation ...................................................................................................... 22

CHAPTER 5 - CONCLUSION ................................................................................................... 25
5.1 Changes to Original Milestones ......................................................................................... 25
5.2 Knowledge Gained .............................................................................................................. 25
5.3 Achievements ....................................................................................................................... 26

BIBLIOGRAPHY ..................................................................................................................... 27

APPENDICES .......................................................................................................................... 29
APPENDIX A: Conjunctive Normal Form (CNF) Formula ...................................................... 29
APPENDIX B: Satisfiability (SAT) Solver ................................................................................. 29
APPENDIX C: DIMACS Format ................................................................................................. 29
APPENDIX D: User Testing Form ............................................................................................ 31
List of Figures

FIGURE 1.1 – UNSOLVED SUDOKU PUZZLE .................................................................1
FIGURE 3.1 – HIGH LEVEL OVERVIEW ........................................................................11
FIGURE 3.2 – EXPECTED FLOW OF PROGRAM ..............................................................12
FIGURE 3.3 – BASIC CLASS DIAGRAM FOR SOLVER .....................................................13
FIGURE 3.4 – BASIC CLASS DIAGRAM INCLUDING GAME .............................................14
FIGURE 3.5 – SUDOKU SOLVER’S DOMAIN CLASS DIAGRAM .....................................17
FIGURE 3.6 – DOMAIN CLASS DIAGRAM INCLUDING GAME .......................................19
FIGURE 4.1 – PERFORMANCE GRAPH ...........................................................................23
FIGURE 4.2 – BENCHMARK .............................................................................................24
FIGURE A.1 USER TESTING FORM ................................................................................31
List of Tables

TABLE 2.1 – TRUTH TABLE FOR \((A \land B) \lor C\) ........................................................................................................6

TABLE 3.1 – FORMALAE TO ENCAPSULATE CONSTRAINTS OF A SUDOKU PUZZLE .........................................................................................................................16

TABLE 3.5 – EXPLANATION OF THE FUNCTIONALITIES PROVIDED BY CLASSES OF THE SOLVER ........................................................................................................18

TABLE 3.6 – EXPLANATION OF THE FUNCTIONALITIES PROVIDED BY CLASSES OF THE GAME ........................................................................................................20
Chapter 1

Introduction

This chapter gives a brief background of Sudoku puzzles, followed by the project proposal and objectives. The research done on existing Sudoku Solvers has also been mentioned here followed by a description of how the problem was approached. The chapter is concluded with a concise overview of the report.

1.1 Background

Sudoku is a logic-based combinatorial number placement puzzle (Murphy, 2011). The puzzle likely originated from the 18th-century Swiss mathematician Leonhard Euler’s game called “Latin Squares”. (The Economist, 2005) It first gained popularity in Japan, which is also where the name ‘Sudoku’ was given to it. (Sudoku Dragon, 2015)

![Unsolved Sudoku Puzzle](EducationWorld, 2006)

The puzzle consists of a 9 X 9 grid of 81 cells, and the objective is to fill the grid in such a manner that each row, column and 3 X 3 sub-grid contain all the digits from 1 to 9. (Murphy, 2011) Players start with an initial puzzle that consists of pre-filled cells, as shown in Figure 1.1. (Education World, 2006) Since the puzzle is based on mathematical logic rather than guessing, a well set puzzle should have only one solution.
There are two main techniques of solving Sudoku puzzles, namely; Crosshatching and Pencilling in. All Sudoku puzzles require an iterative approach, with players having to visit each cell multiple times in order to fill it in. (Murphy, 2011) In crosshatching, players try to find the right cells to hold the number, rather than trying to find the right number to go in a cell. Pencilling in is used for puzzles with higher difficulty levels, wherein the player writes down all the numbers that could go in a cell, and then proceed to cancel out the possible values as they get filled into another cell. (Stephens, 2005)

1.2 Project Proposal and Objectives

The project proposal was to create a tool which uses propositional logic to solve Sudoku puzzles. The expectation was that since Sudoku is a logic based puzzle, it could be broken down into a set of propositional constraints and these constraints could be used to find its solution by utilizing the power of modern processors; thus helping players to check their solution to a given Sudoku puzzle. Users would be required to spend some time to input the puzzle into the solver, but this would amount to a matter of seconds as compared several minutes (or hours) spent on actually solving it.

On further discussion with Dr. Korovin (project supervisor), the objective of the project was extended to a Sudoku game, rather than only a solver. This would give the users an opportunity to solve different puzzles within the application and improve their solving skills. The application would assist novice users by providing solutions to certain cells in order for them to continue solving the puzzle without examining the entire solution.

From the above proposal, the objectives of the project could be described as follows:

I. The solver should use propositional logic to encapsulate the constraints of a Sudoku puzzle and be able to solve a puzzle of any difficulty level. The solution should be presented in a manner that is understandable easily by both novice and experienced users.

II. The game should include puzzles of all difficulty levels and provide support to users by checking their solutions and displaying the solution to certain cells or the entire puzzle, as requested by the user.

1.3 Existing Solvers

Sudoku solving has been a major topic of research over the years, with pre-existing solvers utilizing a range of techniques in order to make the algorithms faster and more efficient. In order to write an algorithm, the Sudoku problem is often reduced to a graph colouring problem. A more common approach is to consider the problem as an exact cover problem and then solve it using
Knuth’s Dancing Links algorithm. (Attractive Chaos Blog, 2011) Solvers that utilize Knuth’s algorithm are usually fast, but faster implementations of the Sudoku solving algorithm have been seen in programs that use the generic backtracking method. Backtracking systematically searches for the solution by assuming it to be represented by vector values and traversing through them in a depth first manner until a solution is found. (Ohio State Lecture, n.d.) The solvers accomplish this by selecting a cell with the minimum choices, fixing one number to it and moving forward. When there is no choice left for a cell, the algorithm goes back one step and selects the next allowed number. (Attractive Chaos Blog, 2011) Even though this is a brute force approach, it works well when combined with techniques such as naked single and hidden single, which fill in the more intuitive cells first and reduce the number of times the algorithm needs to backtrack.

The speed of backtracking algorithms has significantly improved over the years by combining more approaches to find the best possible match for a cell such as the more complicated locked candidate approach. It has been identified that the fastest implementations of the Sudoku solving algorithm in every programming language use the backtracking approach, such as JSolve in C and Kudoku in Java. (Attractive Chaos Blog, 2011)

1.4 Approaching the Problem

As outlined in section 1.3, there are a number of ways of approaching the problem of writing a Sudoku solving algorithm. But none of the solvers that were examined were seen as utilizing propositional logic, which was one of the core aims of the project.

After a conversation with Dr. Korovin and some brief research, it was decided that the best way to approach the problem would be to consider it as a satisfiability problem (SAT) and use a Satisfiability Solver API to generate a model, which can then be used to extract the solution to the given puzzle.

Satisfiability problems and solvers have been described in more detail in Chapter 2.

1.5 Report Overview

The report follows a structured argument and analysis of the application created, along with a thorough breakdown of the tests undertaken and the algorithm implemented.

Chapter 2 outlines the technical background to support the concepts explained during the report. Chapter 3 focuses on the development process, which includes the design and implementation of the application. Chapter 4 illustrates the testing and evaluation performed, and Chapter 5 concludes the report.
Chapter 2

Technical Background

This chapter provides the technical background required to understand the concepts and development process described in the report. Section 2.1 consists of a brief explanation to propositional logic. Section 2.2 outlines satisfiability problems and Section 2.3 describes the methods of solving satisfiability problems and their complexities.

2.1 Propositional Logic

Propositional Logic (PL) is a branch of mathematical logic used to reason the truth and falsehood of logical expressions. (Ullman, Chapter 12: Propositional Logic, 2015) A propositional formula is made up of other propositions and the truth value of the formula is defined by the truth value of the propositions that it is made up of. (Pelletier, 2007)

Example 1 Consider the two premises below:
Premise 1: If it is snowing, then it is cold.
Premise 2: It is snowing
If we know that both the above premises are true, then we can conclude that it is cold outside.

Example 1 shows how it is intuitive to predict the weather if the above two conditions are known. In PL, ‘It is snowing’ and ‘It is cold’ are considered propositions and by knowing their truth value, the truth or falsehood of the weather being cold can be predicted.

Propositional formulae are made up of propositions and connectives. Propositions are referred to by letters, such as P, Q, and R (also known as literals) and connectives are used to join multiple propositions. There are three basic connectives in PL, along with two higher level connective which can be broken down into the basic connectives. These are:

I. **CONJUNCTION (AND)** This connective is used to represent that both the conditions that it is joining need to be satisfied in order for the formula to be true. It is represented by the symbol ∧.

II. **DISJUNCTION (OR)** The OR connective is used to represent that if either of the propositions is true, the the formula is true. It is represented by the symbol ∨.
III. **NEGATION (NOT)** While AND and OR are used to join two propositions, NOT is used for a single proposition. It is used to express that a given proposition is not true and is represented by the symbol $\neg$.

IV. **IMPLICATION (IF … THEN)** It is used to join two propositions (say p and q) to represent “If p, then q”. The symbol $\rightarrow$ is used to express this connective.

V. **BI-IMPLICATION (IF AND ONLY IF)** When two propositions p and q are joined by the bi-implication connective, it means “p if and only if q”. In other words, p takes place if and only if q is true. It is represented by the symbol $\leftrightarrow$.

A single propositional formula such as $\neg p \lor q$ is referred to as a clause. The Implication and Bi-implication connectives can be converted to be represented by AND, OR and NOT. For the rest of the report, the above-mentioned connectives will be represented by their mentioned symbols.

Example 2 illustrates how Example 1 can be represented by using propositional formulae.

**Example 2** Consider the propositions P and Q, where P can be interpreted as “It is snowing” and Q can be interpreted as “It is cold”. Now consider the premises below:

*Premise 1: $P \rightarrow Q$*
*Premise 2: P*

**Conclusion:** $Q$

In the example, it can be seen that if both the premises are true, then Q has to be true. This example is the same as Example 1, but is represented using propositions and connectives instead.

As seen in Example 2, a propositional clause can be used to encapsulate constraints and the truth and false values of the literals under which the given clauses evaluate to true can be found. These truth values of the literals are known as a model of the given set of clauses. Finding the model is not always intuitive and therefore, a method of solving the given formula and finding the model is required. Example 3 below illustrates the truth table method of finding a model for a given clause.

Truth Tables are explained in further detail in section 2.3.1

**Example 3** Consider a propositional formula $(A \land B) \lor C$. The model for this formula can be found by drawing a truth table as given in Table 2.1 below:
From Table 2.1, it can be seen that the models for \((A \land B) \lor C\) are \((A = True, B = True, C = False)\), \((A = True, B = True, C = False)\), \((A = True, B = False, C = True)\), \((A = False, B = True, C = True)\), \((A = False, B = True, C = True)\). This illustrates that a propositional formula can have more than one model.

### Table 2.1 – Truth Table for \((A \land B) \lor C\)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>((A \land B) \lor C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

2.2 Satisfiability Problem (SAT)

A satisfiability problem is the problem of determining if there exists a set of true and false values for the literals in a given set of clauses, for which the the clauses evaluate to true. (Ullman, 2010) Therefore, instead of trying to find all the models of the given clauses, SAT concentrates on finding only one model. If at least one model is found, then the given clauses are said to be satisfiable, and if not, then they are said to be unsatisfiable.

Satisfiability Problems have many applications, ranging from model checking to Artificial Intelligence to Software Verification. SAT was also the first non-deterministic polynomial complete (NP-complete) problem and is used by mathematicians to prove other problems as being NP-complete. (Lee, Roychowdhury, & Seshia, 2011)

2.3 Methods of Solving SAT

This section outlines two widely used methods of solving the satisfiability problem, namely, Truth Tables and the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. It also analyses the size of truth tables and the complexity of the DPLL algorithm.

2.3.1 Truth Tables

A truth table is a device used to determine the model for a given propositional formula. (El Paso Community College, 2012) In a truth table, every combination of truth and false values for literals is substitutes into the formula and the combination for which the formula results in true is given as the model, as illustrated in Example 3. It is considered to be one of the simplest methods of
finding the model for a propositional formula. This in turn can be used to solve the satisfiability
problem. If there is at least one truth value in the final column of the truth table, then the formula
is said to be satisfiable, otherwise, it is considered unsatisfiable.

2.3.2 Size of Truth Tables

If the given propositional formula has \( k \) literals, then the truth assignment for it would consist of a
list of \( k \) elements, with each being either True or False. This would result in the number of rows
in the truth table to increase by a power of two of the number of elements, producing a total of \( 2^k \)
rows. Therefore, for a Sudoku puzzle, which consists of 729 elements, the truth table would consist
of \( 2.824e+219 \) rows. This makes it an unrealistic choice to solve the Sudoku satisfiability problem.

2.3.3 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

The DPLL Algorithm builds on the backtracking algorithm, which assigns a truth value to literals
iteratively and then backtracks and makes a different choice if the initial choice results in the
formula becoming false. (Liberatore, 2012) It makes a choice and then goes down that path in a
depth first manner till the formula becomes false before backtracking and making the alternative
choice on the literal, which is known as splitting on a literal. When it finds a path which results in
the formula being true, it means that the formula is satisfiable. Its advantage though, lies in the
unit propagation and pure literal rules.

The unit propagation rule states that when a clause contains only a single literal, then all the clauses
containing that literal in a Conjunctive Normal Form (CNF) formula (Appendix A) should be
removed, as the clause with a single literal has to be true for the formula to be true. The negation
of proposition is also removed from the remaining clauses, as it will always be False. (Kreitz &
Constable, 2009) This is illustrated by Example 4.

**Example 4** Consider the following three clauses:

\[
A \lor B \\
\neg A \lor B \\
A
\]

By applying unit propagation to these three clauses with the unit clause being \( A \), the unit clause
gets removed, along with \( A \lor B \), as it also contains the unit clause and \( \neg A \) also gets deleted from
\( \neg A \lor B \). Thus, we are left with: \( B \)

This makes it a lot easier to find the truth value which would satisfy the given clauses.
The pure literal rule states that all the clauses containing a pure literal can be removed from the formula. (i.e., we assign the truth value to the pure literal that satisfies the clauses) Pure literals are identified those which occur only as either positive or negative literals in the given CNF formula, as illustrated in Example 5. (Larrosa, Lynce, & Joao, 2010)

**Example 5** Consider the following set of clauses:

\[ A \lor B \]
\[ A \lor C \]
\[ \neg B \lor \neg C \]

\( A \) is identified as the pure literal in the above set of clauses, therefore, clauses \( A \lor B \) and \( A \lor C \) can be removed from the set of clauses by assigning the truth value True to \( A \). Thus, the only clause remaining would be: \( \neg B \lor \neg C \)

### 2.3.4 DPLL Algorithm Complexity

The DPLL Algorithm essentially uses the backtracking algorithm, with a little bit of pruning by using the unit propagation and pure literal rules. Therefore, the worst case of the algorithm can be assumed to be one wherein it is not able to apply either of the two rules and the given formula is unsatisfiable. In this case, it would have to split on each variable and go through each and every path trying to find one which satisfies the given formula. If there are \( k \) elements in the formula, this would result in the algorithm splitting \( k^2 \) times, which would make its complexity \( O(k^2) \). (Bhat, 2013)

Even though the worst case complexity of the algorithm is not ideal, it only applies in an extreme case, which more often than not does not occur. This is why the DPLL Algorithm is considered to be one of the best algorithms to solve the satisfiability problem.
Chapter 3

Development

This chapter focuses on the development process of the application. The development process was split into two halves; design and implementation. Section 3.1 covers the design of the application, which ranges from the identification of requirements to the design decisions that were made. Section 3.2 gives an overview of the implementation of the application. An effort to present a high level view of the implementation has been made, with an in-depth analysis of how the propositional formulae were created and implemented in order to solve Sudoku puzzles.

3.1 Design

This section of the report outlines the design process of the application. Since the design of the application covers a range of topics, it has been broken up into sections with relations between sections stated when necessary.

Sections 3.1.1 and 3.1.2 aim to cover the functional and non-functional requirements that were identified for the application. Even though the requirements of the Sudoku game were identified after the solver had been developed, they are mentioned together in the report. Section 3.1.3 gives a very brief high level overview of the system. The expected flow of the program that was constructed after taking into account the requirements and architecture is outlined by Section 3.1.4. Lastly, Sections 3.1.5 and 3.1.6 state any other design decisions that were made before implementing the Sudoku Solver and the Sudoku Game respectively.

3.1.1 Functional Requirements

The functional requirements that were identified for the application were as follows:

- The solver should facilitate the SAT4J API by generating formulae in a format that is accepted by the API.
- The solver should be able to understand the model generated by SAT4J and convert the model into the solution for the given Sudoku puzzle.
• The game should be able to check the solution entered by the user by generating clauses for the solution and inputting them into SAT4J to check if they are satisfiable. This should then be reported to the user.

• The game should be able to present the solution for the Sudoku puzzle if the user requests for it.

• The game should be able to present the solution for unfilled cells, one at a time, in order to help the user move forward with the puzzle.

3.1.2 Non-functional Requirements

The non-functional requirements for the application were identified as follows:

• The interface of the application should be intuitive, and give a clear indication of how the user should proceed without having to read any documentation.

• The interface of the solver should present the answer to the user in a manner which is easy to understand and give the user an option of inputting a new puzzle without having to restart the application.

• The interface of the solver should provide clear error messages to the user if the input is not acceptable.

• The game should report the correctness of a user’s solution to the given Sudoku puzzle clearly and unambiguously.

• If the user has solved a puzzle incorrectly, then the game should give the user another opportunity to attempt the puzzle and solve it correctly.

• The game should present the user with the option of selecting the level of difficulty of the puzzle they would like to solve, to accommodate users of all experience levels.

• The game should contain a large number of puzzles, to avoid becoming mundane and obsolete after a few runs.

3.1.3 System Architecture

This section describes the basic architectural design that was constructed to help with the design and development process. There were a few changes made to the architecture through the course of the project, and only the final version has been presented in the report.
Figure 3.1 – High level overview

The architecture gives a high level layout of the system and a structure of its working. The architecture represents the user’s interaction with the system, and the system’s internal interaction. It helps to draw clear boundaries between the major parts of the system, and their dependencies. Figure 3.1 represents the architecture constructed for the application, which shows the above-mentioned interaction, and a basic flow of data through the system as well.
3.1.4 Flow of Program

The expected flow of program is illustrated by Figure 3.2

Figure 3.2 – Expected flow of program
3.1.5 Sudoku Solver

As mentioned in Section 1.2, the initial objective of the project was to create a system that could solve Sudoku puzzles, therefore, this section talks about the design of the system that was created keeping only the solver in mind.

It was decided that the best approach for designing the system would be to build upon the architecture mentioned in Section 3.1.3.

After analysing the requirements, it was concluded that the system should be built in an Object Oriented manner, as this would help separate the functionalities and reduce their dependence on each other. The SAT Solver (Appendix B) was to be implemented in a separate class, which would help keep its interference with the rest of the system to a minimum and also develop cleaner code.

The next design decision was determining the classes to be implemented and outlining their interaction, which is shown in Figure 3.3.

![Figure 3.3 – Basic class diagram for solver](image)

Each class to be implemented had a clear purpose. The main class was to bind all the other classes together and it was decided that the interface would not interact with anything other than the main class. Their implementation has been described in detail in Section 3.2.2.
3.1.6 Sudoku Game

The designing process of the game commenced only after implementing the solver completely, however, the structure of this report pertains to a logical chronology of implementation post design. An attempt to adapt the pre-existing classes and functions was made during the design process of the game.

In the design of the game, it was decided that the main class would remain the same as the one that existed in the solver application. It was also concluded that methods that checked for satisfiability for the solver would be adapted to be used for the game as well.

It was determined that the easiest approach at the time would be to store the various pre-set puzzles for the game in a text file rather than a database. This decision was made because the time remaining for the completion of the project was limited, and designing and setting up a database would take a long time. Another class was to be added to the application, which could read the Sudoku puzzles from the text file and select one of them randomly. Taking into consideration the non-functional requirement of having puzzles of different difficulty levels, it was established that the new class to read puzzles should also be able to provide this functionality.

After taking into consideration the changes to the existing application and new functionalities, the class structure of the application was changes to the ones shown in the Figure 3.4.

![Figure 3.4 – Basic class diagram including game](image-url)
The attempt to add game functionality to the application while making minimal changes to the existing application is made apparent by comparing Figure 3.3 and Figure 3.4. The changes that were made to the actual code of the existing classes and the additional class for selecting puzzles have been described in the implementation of the Sudoku Game in Section 3.2.3.

3.2 Implementation

This section outlines the implementation process of the application. Section 3.2.1 elaborates on the most challenging and interesting part of the development process, i.e., the formulation and implementation of propositional formulae in order to encapsulate the constraints of the Sudoku puzzle. Sections 3.2.2 and 3.2.3 give the high level implementation of the Sudoku Solver and the Sudoku Game respectively. Low level detail of the code has been avoided as far as possible and only the functionalities provided by the different classes have been described in these sections.

3.2.1 Propositional Formulae for Sudoku Puzzles

In order to be able to solve Sudoku puzzles using propositional logic, there was need to first encapsulate all the constraints of a Sudoku puzzle using propositional formulae. This section concentrates on the development of these formulae and how these were programmed into the application.

Before the formulae could be developed, a variable that would be used in the formulae was required. The variable $K(r, c, n)$ was selected for this purpose. In the variable, $r$ stands for row, $c$ for column and $n$ for the number. Therefore, the variable $K(1, 1, 1)$ would depict that row = 1, column = 1 of the Sudoku board consists of the number 1.

The constraints that were required to be encapsulated by the formulae were identified as the number constraint, row constraint, column constraint, duplicate constraint and the sub-grid constraint.

The row constraint, column constraint, and sub-grid constraints make sure that every row, column and sub-grid of the Sudoku puzzle should consist of every number from 1 to 9. The number constraint establishes that every cell on the board should contain some number from 1 to 9. And the duplicate constraint verifies that in the solution, no cell contains more than one number.

Table 3.1 shows the developed formulae. The table only mentions the sub-grid constraint for the top left sub-grid, as the ones for the others are exactly the same, but with different limits.
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Formula</th>
<th>Formula No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Constraint</td>
<td>$\bigwedge_{r=1}^{9} \left( \bigwedge_{c=1}^{9} \left( \bigvee_{n=1}^{9} K(r,c,n) \right) \right)$</td>
<td>(1)</td>
</tr>
<tr>
<td>Row Constraint</td>
<td>$\bigwedge_{r=1}^{9} \left( \bigwedge_{n=1}^{9} \left( \bigvee_{c=1}^{9} K(r,c,n) \right) \right)$</td>
<td>(2)</td>
</tr>
<tr>
<td>Column Constraint</td>
<td>$\bigwedge_{c=1}^{9} \left( \bigwedge_{n=1}^{9} \left( \bigvee_{r=1}^{9} K(r,c,n) \right) \right)$</td>
<td>(3)</td>
</tr>
<tr>
<td>Duplicate Constraint</td>
<td>$\bigwedge_{r=1}^{9} \left( \bigwedge_{c=1}^{9} \left( \bigvee_{n=1}^{9} \left( \bigwedge_{n'=n+1}^{9} K(r,c,n) \rightarrow \neg K(r,c,n') \right) \right) \right)$</td>
<td>(4)</td>
</tr>
<tr>
<td>Top left sub-grid</td>
<td>$\bigwedge_{n=1}^{9} \left( \bigwedge_{r=1}^{9} \left( \bigvee_{c=1}^{9} K(r,c,n) \right) \right)$</td>
<td>(5)</td>
</tr>
</tbody>
</table>

Table 3.1 – Formulae to encapsulate the constraints of a Sudoku puzzle

It is not necessary to understand every equation in Table 3.1 in order to grasp its implementation in the application.

The formulae were implemented in the ClauseGenerator class of the application. Since SAT4J accepts the clauses only in DIMACS format, this is the format in which the clauses were generated in. The DIMACS format has been explained in Appendix C. Without going into the low level detail of the code, the clause generation algorithm can be described as a series of nested for loops that generate the clauses according to the formula that they are implementing. On an average, there are more than 3000 clauses generated for every Sudoku puzzle that needs to be solved.

3.2.2 Sudoku Solver

The decisions taken and code written during the period of development of the solver have been explained in this section.

The most important decision that had to be taken before beginning the implementation for the Solver was to decide what programming language to use. The restrictions while selecting a language were the availability of a SAT Solving API in the language and the condition that the language should be Object Oriented. After some research, it was concluded that Java would be an acceptable choice for the development of the application, due to my prior experience and comfort with the language along with the availability of a SAT Solving API.
The SAT Solving API that was seen fit to be used in the application was the SAT4J API. SAT4J is the Java implementation of MiniSAT, which is one of the most famous SAT Solvers. The reason for selecting this API was that while respecting the algorithm of MiniSAT, SAT4J adapts to Java practices to increase its efficiency in the language. (SAT4J, 2016)

The short iterations approach was used during the implementation process of the application. Due to short deadline, this approach helped make sure that the workload was managed by breaking down the application into smaller functionalities, and that development was done regularly and not left to the end. Another extremely helpful feature of this approach was the regular testing, as it made sure that there was always a working model of the application to present to my supervisor. Testing has been explained in more detail in Chapter 4.

As described by Figure 3.2, there were four main classes that were identified in order to implement the solver, one of which was the Graphical User Interface. The high level class diagram was evolved to generate a more details domain class diagram, shown in Figure 3.5.

![Figure 3.5 – Sudoku Solver’s domain class diagram](image)

Without going into too much low level technical detail, the main functionalities that the other three classes provided are describes in Table 3.2 below.
<table>
<thead>
<tr>
<th>Class</th>
<th>Functionalities Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve</td>
<td>The Solve class is the main class of the project. Even though the functionalities it provides are minimal, its main purpose is to call methods from other classes to perform the required calculations and operations and interact with the GUI. One key function of this class is to extract the solution for the given Sudoku puzzle from the model generated by the SAT Solver.</td>
</tr>
<tr>
<td>ClauseGenerator</td>
<td>As the name suggests, the function of the ClauseGenerator class is to generate the general and Sudoku specific constraints and save them to a file, so that they can be inputted into the SATSolver class.</td>
</tr>
<tr>
<td>SATSolver</td>
<td>The SATSolver class implements the SAT4J API. The class takes input in the form of a text file that includes all the clauses to be checked for satisfiability in DIMACS format. The class returns a boolean variable which reflects the satisfiability of the inputted clauses. If the clauses are satisfiable, it stores the model in an ArrayList.</td>
</tr>
</tbody>
</table>

Table 3.2 – Explanation of the functionalities provided by classes of the solver

3.2.3 Sudoku Game

The decisions taken and the code written during the period of development of the game have been explained in this section.

During the implementation of the game, the development methodology was kept the same as what it had been during the development of the solver, i.e., short iterations. This was due to the fact that short iterations had proven to be beneficial while implementing the solver, and also because the regular testing would make sure that the functionality of the solver wasn’t compromised at any point while developing the game.
Figure 3.6 illustrates the detailed class diagram that was designed, while maintaining the class structures and relations as in Figure 3.4.
Some changes were made to the *Solve* class and the *readPuzzle* class was added to include the functionalities for the game, and these have been outlined in Table 3.3.

<table>
<thead>
<tr>
<th>Class</th>
<th>Functionalities Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve</td>
<td>The first major change that was made to the <em>solve</em> class was the addition of a function that could check the validity of the puzzle displayed to the user. Functionalities were also added to check the solution entered by the user by generating clauses for it using the ClauseGenerator class and inputting them into SAT4J to check if they are satisfiable. Other checks such as checking if the whole puzzle had been solved were also added. If the user requests for the solution to the entire puzzle, then the solver is used to solve the puzzle and present the answer. Further to this, it was also made sure that the application could help the user by displaying the solution to cells one at a time if he/she gets stuck.</td>
</tr>
<tr>
<td>readPuzzle</td>
<td>The readPuzzle class was added to interact with the text file that stores the various puzzles for the game. Its functionality was extended beyond just reading puzzles from a file and displaying them to being able to select puzzles of different difficulty levels as well.</td>
</tr>
<tr>
<td>Interface</td>
<td>The GUI also had to be changed in order to accommodate the game. There was an option added to select the game or the solver within the same application. The puzzle board was also programed to lock all the pre-filled cells in the game so that the user can’t change them.</td>
</tr>
</tbody>
</table>

*Table 3.3 - Explanation of the functionalities provided by classes for the game*
Chapter 4

Testing and Evaluation

This chapter describes the testing that was carried out in order to check that the application worked as expected and was of good quality. This is done by describing the different types of tests that were performed and reason for performing them. The chapter also outlines the evaluation of the performance of the application.

4.1 Unit Testing

Since the application was developed using the short iterations approach, unit tests provide a way of testing the functions developed during previous iterations. Along with running tests for pre-existing functionalities, unit tests were also written for functions developed during an ongoing iteration for immediate testing, as well as future testing.

Unit tests proved most useful for testing the functions that affected the generation of the propositional clauses. Since the functions that actually generated the clauses were not changed in any way, more focus was placed on the functions providing the input instead.

During the development of the application, unit tests were often combined into integration tests in order to test functionalities rather than only the methods, especially for the Sudoku Game. This was due to the fact that the game consisted of a wide range of functionalities, which were dependant on each other.

4.2 Correctness Testing

The correctness testing began during the design process, where a number of Sudoku puzzles of different difficulty levels and their solutions were identified as test cases. The major functionality to be tested in the game was checking if the solution inputted by the user was being evaluated correctly. The test cases also included a number of incorrect puzzles, to check that the application ability to handle these cleanly and without crashing. These tests were performed on the completion of the Sudoku solver and game.
4.3 User Testing

One of the main objectives of testing is to make sure that the identified requirements have been met. User testing was thus used to check if the non-functional requirements outlined had been met. These were done after the unit tests and correctness tests had been performed.

User testing was done through users actually using the application and filling a form to rate it on various criteria. The form has been included in Appendix D.

These tests proved quite useful and a number of changes were made due to the feedback given. These changes included the addition of a back button for users to go back to the puzzle and try solving it again if they have gotten the solution wrong and the inclusion of the option to provide users with the solutions to some cells if they got stuck during the solving process.

4.4 Performance Evaluation

The evaluation of the application was conducted in the two stages. The first stage was to measure the time taken by the application to solve puzzles of different difficulty levels. The second stage was to compare the performance of the solver to the benchmarks present for pre-existing Sudoku solvers.

The speed of the solver was evaluated by inputting 100 puzzles of each of the three difficulty levels, as well as 100 puzzles with 1 input, 2 inputs, 3 inputs, 4 inputs and 5 inputs each. The performance measured through these tests has been depicted in Figure 4.1
As evidenced by Figure 4.1, the performance decreases drastically as the number of inputs decrease. The reason for this decline can be understood by analysing the concepts mentioned in Section 2.3.4. The lesser number of inputs there are, the lesser number of truth values are known to SAT Solver, which in turn decreases the optimizations such as the pure literal rule and unit propagation rule that the SAT Solver can implement. This results in the solver taking far longer in providing a model for the given set of clauses, which increases the time taken by the application to solve the puzzle.

The other performance evaluation was performed through a comparison of the application’s performance against the performance of Sudoku solver benchmarks. The benchmark compared the performances of existing solvers for 20 very hard puzzles that were solved 50 times repeatedly, thus amounting to a 1000 tests. (Attractive Chaos Blog, 2011) The result of this evaluation have been depicted in Figure 4.2.
The same 20 puzzles were tested the same number of times using the developed application and the average time taken to solve the puzzles was found to be 1.827 seconds. On comparing this to the performances of the solvers in Figure 4.2, the developed application rates as the fastest Java based solver. It has to be taken into account that the processors used for the benchmark were not mentioned, and that the most recent benchmark found was from 2011. But even after taking these factors into account, the developed application ranks quite high in terms of performance.
Chapter 5

Conclusion

This chapter concludes the report by analysing how the application and its features evolved over the course of the development cycle, what technical and non-technical knowledge was gained by doing the project and what was achieved.

5.1 Changes to Original Milestones

As mentioned throughout the report, the milestones changed quite a bit during the development of the project. The change to add the game has been covered in depth in the report, but the milestone I would like to cover here was one that was not completed due to insufficient time. When it was decided that the game would be a part of the application, a feature that I wanted to add was to give the application the ability to generate its own Sudoku puzzles, so that the game never runs out of possible puzzles to provide to the user. The challenge here, was to create puzzles in such a way that they had a unique solution, otherwise they would not be human solvable. After conducting research on how this would be possible, it was decided that it would take far too long to think about and generate the formulae to implement this, and thus had to be left out.

Another change was brought about due to the change in milestones. Since the functionalities of the application were outlined before the start of the development process, I had initially decided to use a waterfall approach. In a meeting with my supervisor prior to the actual implementation process, it was discussed that the game functionality may be something I could add if the solver was completed ahead of time. Due to the uncertainty of the specifics of the application at that point, I decided to change the development methodology to agile. This turned out to be a big benefit, as it provided me with a number of other advantages as mentioned in the report.

5.2 Knowledge Gained

The project has been a learning experience for me. In context of the project itself, I gained invaluable knowledge of propositional logic and satisfiability, and how these could be used to solve complex problems. The insights given by my supervisor, Dr. Konstantin Korovin were invaluable in improving my understanding of proposition formulae.
Along with this, I was also able to work on my programming and problem solving skills. My time management got a lot better by the end of the project, and I think this is something that would hugely benefit me in industry.

5.3 Achievements

Towards the start, the problem I was presented with seemed challenging. With time, as my research of the topic area strengthened, so did my confidence – enabling me to achieve milestones set out at the start and more.

I would like to focus on the speed of the solver produced. During the evaluation, I was astonished by the results when the solver was compared against the Sudoku Solver benchmarks, as mentioned in Section 4.4. To me, the ability to produce an application that could even be mentioned with the best ones available marked my biggest achievement. I acknowledge that there is a massive scope for improvement, but the existing results have motivated me to keep working on it furthermore.
Bibliography


Appendices

Appendix A: Conjunctive Normal Form (CNF) Formula

A formula is known as a CNF formula if it is composed of a conjunction of clauses, wherein, the clauses are made up of a disjunction of literals. Example A.1 illustrates this concept.

**Example A.1** Consider two propositional formulae

*Formula 1*: \((A \lor B \lor C) \land (B \lor \neg C) \land (C \lor \neg A)\)

*Formula 2*: \((A \land B \lor C) \lor (\neg A \land C)\)

*Formula 1* is in CNF and *formula 2* isn’t. This is because, every clause in *formula 1* is made up of literals that are connected using ORs (disjunctions) and these clauses have been joined together using ANDs (conjunctions). Since this is not the case in *formula 2*, it is not in CNF.

CNF is an important concept in propositional logic, and most solvers accept equations only in this format.

Appendix B: Satisfiability (SAT) Solver

SAT Solvers are used to solve satisfiability problems and generating a model for the given satisfiability problem. They use the DPLL algorithm in order to perform this function.

Over time, SAT Solvers have been evolved to use techniques beyond the unit propagation rule and pure literal rule, to eliminate as many variables as possible and increase their efficiency.

The most well known SAT Solving API is MiniSAT, which is written in C.

Appendix C: DIMACS Format

The DIMACS format is the format in which the SAT4J SAT Solver accepts its clauses. The format can be illustrated by considering Example C.1.

**Example A.2** Consider a propositional Variable \(K (r,c,n)\), where \(r\), \(c\) and \(n\) are representations of the row, column and the number. Therefore, \(K (1, 1, 1)\) would mean that row 1, column 1 contains the number 1. In the DIMACS format, this variable would be represented simple by the number 111. Whereas, \(K (7, 2, 3)\) would be represented as 723. In the DIMACS format, the number
'0' is used as a delimiter to mark the end of a clause. And the negation symbol $\neg$ is represented by the minus '-' sign. To clause $K (1, 1, 1) \lor K (7, 2, 3)$ would be represented in the DIMACS format as '111 723 0'.

The DIMACS format is not only used as input by the SAT Solver, but also as the output format. Since it consists of only numbers, the numbers can be considered as integers, which makes it a lot easier to extract the solution from them.
Appendix D: User Testing Form

The form used for user testing is presented in Figure A.1.

![User Testing Form](image)

**3rd Year Project User Test**
**Sudoku Solver and Game**

**Name (Not mandatory):**

**Rating Criteria**
1 - Strongly Disagree
2 - Disagree
3 - Satisfiable
4 - Agree
5 - Strongly Agree

<table>
<thead>
<tr>
<th>Usable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuitive</td>
<td></td>
</tr>
<tr>
<td>Useful</td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td></td>
</tr>
<tr>
<td>Range of Functionality</td>
<td></td>
</tr>
<tr>
<td>Will you use it in the future? (Yes/No)</td>
<td></td>
</tr>
</tbody>
</table>

**Comment:**