

School of Computing

FACULTY OF ENGINEERING AND PHYSICAL SCIENCE



UNIVERSITY OF LEEDS

Final Report

AI for Solving Puzzles

Sylvain Hu

**Submitted in accordance with the requirements for the degree of
BSc Computer Science with Artificial Intelligence**

2020/2021

40 credits

The candidate confirms that the following have been submitted:

Items	Format	Recipient(s) and Date
<i>Final Report</i>	<i>PDF</i>	<i>Minerva (10/05/21)</i>
<i>Software Source Code</i>	<i>GitLab Access</i> <i>URL in Appendix B</i>	<i>Supervisor, assessor (10/05/21)</i>
<i>Software Instructions</i>	<i>GitLab Access</i> <i>URL in Appendix B</i>	<i>Supervisor, assessor (10/05/21)</i>

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Sylvain Hu

Summary

Sudoku is one of the most famous puzzles in the world, it has variants such as Hyper Sudoku. These puzzles can be solved by different AI algorithms. This project consists of a solver that can solve standard Sudokus and Hyper Sudokus using different computer algorithms and compares the performance of these methods. The solver has a command-line interface, and allows the user to type his own Sudoku or solve a Sudoku in the database using one of the algorithms.

Acknowledgements

I would like to thank my supervisor Dr Brandon Bennet for his guidance and support throughout the project. I would also like to thank my assessor Dr David Head for the useful feedback and advice.

Table of Contents

Summary.....	iii
Acknowledgements	iv
Table of Contents.....	v
Chapter 1 Introduction.....	1
1.1 Project aim and objectives	1
1.2 Deliverables	1
Chapter 2 Project planning and management	2
2.1 Initial plan.....	2
2.2 Changes in project aims and objectives	2
2.3 Project methodology and risk mitigation.....	2
2.4 Version Control	3
Chapter 3 Background Research.....	4
3.1 Sudoku Puzzles	4
3.1.1 History of Sudoku	4
3.1.2 Standard Sudoku Puzzle	4
3.1.3 Hyper Sudoku	5
3.1.4 Mini Sudoku.....	6
3.2 AI algorithms.....	6
3.2.1 Backtracking	6
3.2.2 Constraint Programming	7
3.2.3 Exact Cover	9
3.2.4 Other algorithms	13
CHAPTER 4 Design and Implementation	14
4.1 Programming Language	14
4.2 Code Repository	14
4.3 Code Structure	14
4.4 Implementation	15

4.4.1 Backtracking	15
4.4.2 Constraint Satisfaction Problem.....	16
4.4.3 Exact Cover	18
4.4.4 Hyper Sudoku variants.....	19
4.4.5 Command Line Interface	20
CHAPTER 5 Testing and Analysis	24
5.1 Testing	24
5.2 Hardware Resources.....	24
5.3 Analysis for standard Sudokus	24
5.3.1 Backtracking	24
5.3.2 Constraint Programming	27
5.3.3 Exact Cover	29
5.3.4 Overall Analysis for Standard Sudokus.....	31
5.4 Analysis of Hyper Sudokus	32
5.4.1 Backtracking for Hyper Sudoku	32
5.4.2 CSP for Hyper Sudoku	34
5.4.3 Overall Analysis for Hyper Sudokus.....	36
CHAPTER 6 Legal, Social, Ethical and Professional Issues	38
3.1 Legal issues	38
3.2 Social issues	38
3.3 Ethical issues.....	38
3.4 Professional issues	38
CHAPTER 7 Conclusion	39
7.1 Summary.....	39
7.2 Future work	39
7.3 Personal Reflection.....	40
List of References.....	41
Appendix A External Materials.....	43
Software Tools.....	43
Appendix B	44
Source Code.....	44

Chapter 1 Introduction

1.1 Project aim and objectives

The aim of the project is to design a solver for different variants of Sudoku puzzles using different types of AI algorithms and compare its performance. The objectives of the project are:

- Solve a standard Sudoku with backtracking
- Solve a standard Sudoku with constraint programming
- Solve a standard Sudoku with exact cover
- Using these algorithms to solve a hyper Sudoku
- Using these algorithms to solve a mini Sudoku
- Create a command-line interface for the solver
- Create a database to store puzzles of different difficulties for the user to test
- Let the user type his own Sudoku
- Analyse and explain the performance of the different algorithms

1.2 Deliverables

The deliverables for this project are:

- Final report describing the project
- Source code for the solver stored on GitLab.
- Software Instructions in a README file stored on GitLab

Chapter 2 Project planning and management

2.1 Initial plan

The project has started since the allocation, on the 17th October 2020. It consists of two main parts, a research part and a development part. The first part takes place till the end of the year, and is mainly background research. Development should start in January or February 2021. The plan is not very structured due to the unstable situation caused by the global pandemic. The planning consists of multiple segments:

1. Express the algorithms in code
2. Find a data structure for Sudokus so that it can be used for each algorithm
3. Adapt the algorithms so that it can solve hyper Sudokus and mini Sudokus
4. Create a command-line interface for the solver
5. Compare performances of the different algorithms when solving Sudokus
6. Write the final report

2.2 Changes in project aims and objectives

Project aim and objectives changed during development as some difficulties were met. The principal aim of the project remains the same, create a solver that can solve Sudoku and some variants using different algorithms. Some objectives were abandoned due to a lack of time, such as solving the variant mini Sudoku, and the hyper Sudoku version of the method exact cover.

2.3 Project methodology and risk mitigation

For an exploratory software project, it is very difficult to have a very well-specified and strict planning, therefore an agile project methodology is adopted, so that parts of the project can be developed independently, and adapt the planning following the outputs of different stages of the project development.

There are no external dependencies in this project. Therefore the only risks for this project are difficulties coming from the developing phase, abandoning the development of some components of the Sudoku solver may be considered for risk mitigation. Source code is stored in GitLab's cloud-based storage, and the report is stored in local machine and cloud-based Google Docs.

2.4 Version Control

Version control is an essential part of software engineering nowadays. It keeps a backup of the source code and serves as a safety net to protect the source code from unwanted harm. If a problem is encountered, the system reverts changes to a previous functioning version of the code. This project uses GitLab for version control, one of the most used source code management platform. GitLab allows an easy way to keep track of the software's evolution, and to share the source code.

Chapter 3 Background Research

3.1 Sudoku Puzzles

3.1.1 History of Sudoku

The Sudoku puzzles find its roots back to the 18th century in Switzerland, with mathematician Leonhard Euler's game "Latin Squares". Modern Sudoku puzzles were invented by Howard Garns and first published in 1979 in the USA and were known as "Number Place". The game was then popularized in Japan and given the name "Sudoku", which means digit-single, in the 80s. The puzzle has become a world phenomenon since 2004, thanks to the efforts of Wayne Gould, who developed a computer program that could generate Sudoku puzzles. [1]

3.1.2 Standard Sudoku Puzzle

		5		9				1
					2		7	3
7	6				8	2		
	1	2			9			4
			2		3			
3			1			9	6	
		1	9				5	8
9	7		5					
5				3		7		

Copyright 2005 M. Feenstra, Den Haag

Figure 3.1: A standard sudoku puzzle. Taken from Sudoku.ws [2]

A standard Sudoku puzzle contains 81 cells, with 9 rows and 9 columns, and nine 3x3 boxes which can be distinguished by the bold lines. Each cell may contain one digit between 1-9, the goal is to fill all the blank cells with a digit, where each row, each column and each box contains all the digits from 1-9, no repeated digit is allowed. The given numbers by the puzzle setter are called the clues. A well-made sudoku puzzle has a single solution.

3.1.3 Hyper Sudoku

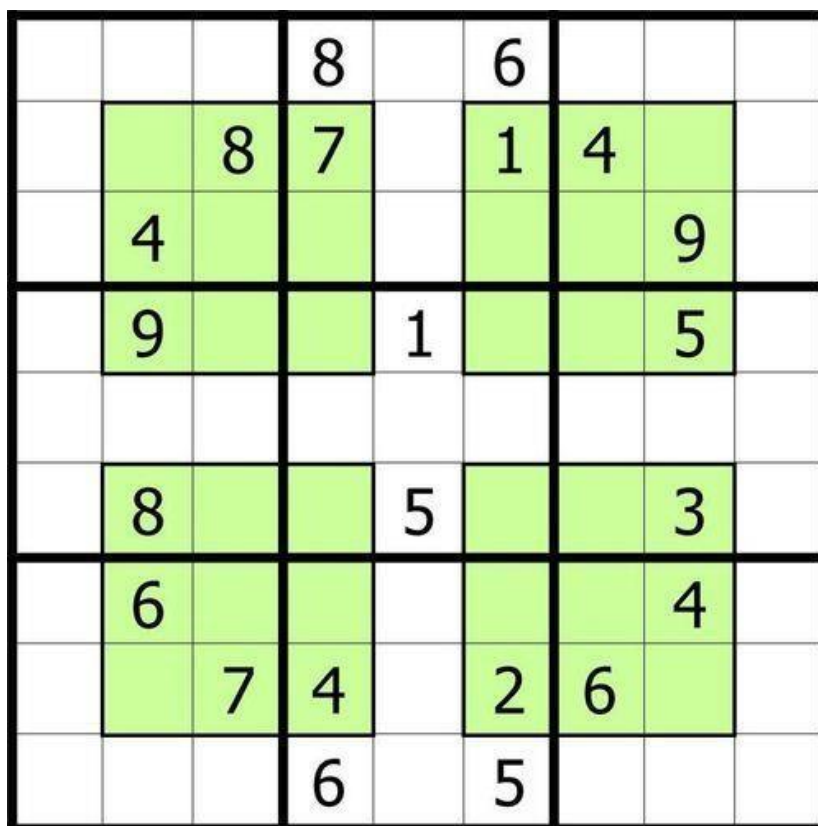


Figure 3.2: A hyper sudoku puzzle. Taken from Educmat.fr 0

Hyper Sudoku or Windoku is a variant of the sudoku puzzle, the rules are the same as the standard sudoku puzzle, except this variant has an extra constraint, there are four 3x3 interior boxes (coloured in green see figure 3.2) in which the numbers 1-9 can appear only once. Therefore, there are thirteen 3x3 boxes to consider in this variant.

3.1.4 Mini Sudoku

			5		
		4			1
		5			6
	2				
	1		3	2	5

Figure 3.3: A mini sudoku puzzle. Taken from Sudoku.cool [4]

A simpler variant of the standard sudoku puzzle, this variant has 36 cells which can be filled with digits between 1-6, with six 3x2 boxes. The rule is the same as the standard sudoku puzzle, only the dimension of the grid changed.

3.2 AI algorithms

3.2.1 Backtracking

The backtracking algorithm is a brute force and a depth-first search, in the case of a standard sudoku puzzle, this algorithm fills every empty cell sequentially to find the solution, and if a path is tested and no solution is found, it backtracks to test another path.

The algorithm fills the first empty cell with “1”, and check if there is any violation of the 3 constraints. If the answer is no, then the algorithm moves to the next empty cell, and puts “1” in that cell, if a violation is detected, it puts “2” instead, and checks again for violations. If it

advances to a cell where all the 9 digits have a violation, the algorithm leaves the cell blank and goes back to the previous cell and increments it by one, this is backtracking.

The algorithm repeats (by calling itself) until a solution is found, it is a recursive algorithm. And if there is no solution, it returns false.

The solving time of this algorithm may be longer than other algorithms, as there is no optimization, it is a brute force approach. [5]

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 3.4: A standard Sudoku being solved by backtracking. Taken from Wikipedia 0

3.2.2 Constraint Programming

Formal Definition: A CSP (constraint satisfaction problem) is a way to formulate a mathematical problem, and it is defined by 3 sets:

$X = \{X_1, X_2, \dots, X_m\}$ is a set of variables

$D = \{D_1, D_2, \dots, D_m\}$ is a set of domains

$C = \{C_1, C_2, \dots, C_m\}$ is a set of constraints

Each variable X_i in the set X , has a domain D_i in the set D , and can take any value of its domain D_i . A constraint C_j in the set C , is defined by the pair $\langle t_j, R_j \rangle$ where $t_j \subset X$ is a subset of k variables and R_j is an k -ary relation on the corresponding subset of domains D_j [7]. A constraint C_j is said satisfied, if the values assigned to the variables t_j by the evaluation v satisfy the relation R_j .

An evaluation is consistent if no constraints were violated. It is complete if all variables are assigned. An evaluation is a solution if it is both consistent and complete, such evaluation is said to solve the CSP.

In this project, the sudoku puzzle can be modelled as a CSP, and it has a set of variables, a set of domains and a set of constraints just like every other CSP:

Variables: Rows are represented by letters from A to I, columns by integers from 1 to 9, therefore the set $V = \{A1, A2, \dots, I8, I9\}$. Each variable represents a cell, for example, the variable C4 represents the cell in the third row and the fourth column. For clues, the variable is the number in the cell.

Domains: Each cell can be filled with digits from 1 to 9, therefore the set $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For clues, the domain of the cell is the number in that cell.

Constraints: There are three types of constraints, one with the columns, one with the rows and one with the boxes. Variables in each type of these constraints need to differ.

a Column Constraints: $\text{AllDiff}(A1, B1, C1, D1, E1, F1, G1, H1, I1), \dots, \text{AllDiff}(A9, B9, C9, D9, E9, F9, G9, H9, I9)$. There are nine column constraints as there are nine columns.

b Row Constraints: $\text{AllDiff}(A1, A2, A3, A4, A5, A6, A7, A8, A9), \dots, \text{AllDiff}(I1, I2, I3, I4, I5, I6, I7, I8, I9)$. There are nine row constraints as there are nine rows.

c Box Constraints: $\text{AllDiff}(A1, A2, A3, B1, B2, B3, C1, C2, C3), \dots, \text{AllDiff}(G7, G8, G9, H7, H8, H9, I7, I8, I9)$. There are nine box constraints as there are nine boxes.

By taking account of these factors, constraint propagation is implemented, when doing an assignment for a variable, domains of unassigned variables are modified too. There are many forms of constraint propagation, forward checking is a good example. When a value is assigned to a variable, the algorithm calculates each unassigned variable that neighbours that variable, and deletes the assigned value from the domains of these neighbours.

For example, the cell A1 has column constraint {A1, B1, C1, D1, E1, F1, G1, H1, I1}, row constraint { A1, A2, A3, A4, A5, A6, A7, A8, A9} and box constraint { A1, A2, A3, B1, B2 , B3, C1, C2, C3}. If the cell A1 is assigned the value 1, the algorithm calculates the unassigned neighbours (also called peers), which are the elements in the constraint excluding A1, which corresponds to A2-A9 for the row, B1-I1 for the column, and B2, B3, C2, C3 for the box. Then the algorithm deletes the value 1 from the domain of these neighbours. [8]

3.2.3 Exact Cover

Formal Definition: Given a collection S of subsets of a set X , an exact cover of X is a subcollection S^* of S that satisfies two conditions :

- The intersection of any two distinct subsets in S^* is empty. That is, each element of X should be contained in at most one subset of S^* .
- Union of all subsets in S^* is X . That means union should contain all the elements in set X . So we can say that S^* covers X . 0

For example:

Let $S = \{ A, B, C, D, E \}$ and $X = \{ 1, 2, 3, 4, 5, 6, 7 \}$ such that :

- $A = \{ 1, 3, 5 \}$
- $B = \{ 1, 3 \}$
- $C = \{ 3, 5, 7 \}$
- $D = \{ 4, 5, 6 \}$
- $E = \{ 2, 4, 6, 7 \}$

Then $S^* = \{ A, E \}$ is an exact cover, because each element in X is contained exactly once in subsets $\{ A, E \}$. The union of these two subsets corresponds to all the elements of the set X .

In this project, for a standard sudoku puzzle, there are 81 cells, and each cell is assigned a digit between 1 and 9, there are $81 \times 9 = 729$ possibilities. A possible assignment of a number in a cell can be labelled R1C1#1 for example, for an assignment of the digit 1 in the cell of the first row and first column. There are four constraints sets in the exact cover representation of sudoku:

- **Cell:** Each intersection of a row and column, therefore each cell must contain exactly one digit. For example, there are 9 possibilities for the cell of the first row and first column, the constraint set is: $R1C1 = \{ R1C1\#1, R1C1\#2, \dots, R1C1\#8, R1C1\#9 \}$. There is a total of 81 cell constraint sets.
- **Row:** Each row must contain each digit (1-9) exactly once. For example, the first row can have the digit 2 in 9 different cells, the constraint set is: $R1\#2 = \{ R1C1\#2, R1C2\#2, \dots, R1C8\#2, R1C9\#2 \}$. There is a total of 81 row constraint sets.
- **Column:** Each column must contain each digit (1-9) exactly once. For example, the first column can have the digit 3 in 9 different cells, the constraint set is: $C1\#3 = \{ R1C1\#3, R2C1\#3, \dots, R8C1\#3, R9C1\#3 \}$. There is a total of 81 column constraint sets.
- **Box:** There are a total of 9 boxes in a standard sudoku puzzle, and each box must contain each digit (1-9) exactly once. For example, for the top-left corner box, the digit 4 can be placed in 9 of the box's cells, the constraint set is: $B1\#4 = \{ R1C1\#4, R1C2\#4, \dots, R3C2\#4, R3C3\#4 \}$. There is a total of 81 box constraint sets.

There is a total of $4 \times 81 = 324$ constraint sets in a standard sudoku puzzle, and as in the previous paragraph, there is a total of 729 possibilities in this puzzle, therefore it can be represented as a 729×324 matrix. [10]

After modelling the sudoku puzzle as an exact cover problem, Knuth's Algorithm X can be used to solve the puzzle.

Formal Definition: For a given matrix A of 0s and 1s.

If A is empty, the problem is solved; terminate successfully.

Otherwise choose a column, c (deterministically).

Choose a row, r, such that $A[r, c] = 1$ (non-deterministically).

Include r in the partial solution.

For each column j such that $A[r, j] = 1$,

 For each row i such that $A[i, j] = 1$,

 Delete row i from matrix A

 Delete column j from matrix A

Repeat this algorithm recursively on the reduced matrix A [11]. For the example in the formal definition of exact cover, the problem is represented by the matrix:

	1	2	3	4	5	6	7
A	1	0	1	0	1	0	0
B	1	0	1	0	0	0	0
C	0	0	1	0	1	0	1
D	0	0	0	1	1	1	0
E	0	1	0	1	0	1	1

Figure 3.5: Matrix representation of an exact cover problem

Level 0

Step 1 – The matrix is not empty, the algorithm proceeds

Step 2 – The lowest number of 1s in any column is 1. Column 2 is the first column with one 1 and thus is selected (deterministically)

Step 3 – Row E has a 1 in column 2, thus is selected (non-deterministically).

The algorithm moves to first branch at level 1.

Level 1 Select Row E

Step 4 – Row E is included in the partial answer

Step 5 – Row E has a 1 in columns 2, 4, 6 and 7: column 2 has a 1 in row E, column 4 has a 1 in rows D and E, column 6 has a 1 in rows D and E, column 7 has 1 in rows C and E. Thus rows C, D and E are to be removed and also columns 2, 4, 6 and 7:

	1	3	5
A	1	1	1
B	1	1	0

Figure 2.6: Matrix representation of an exact cover problem being solved by Knuth's Algorithm X

Rows A and B remain, and columns 1, 3 and 5 remain.

Step 1 – The matrix is not empty, the algorithm proceeds

Step 2 - The lowest number of 1s in any column is 1. Column 5 is the first column with one 1 and thus is selected (deterministically)

Step 3 – Row A has a 1 in column 5, thus is selected (non-deterministically).

The algorithm moves to first branch at level 2.

Level 2 Select Row A

Step 4 – Row A is included in the partial answer

Step 5 – Row a has a 1 in columns 1, 3 and 5: column 1 has a 1 in rows A and B, column 3 has a 1 in rows A and B, column 5 has a 1 in row A. Thus rows A and B are to be removed and also columns 1, 3 and 5.

Step 1 – The matrix is empty, thus this branch of the algorithm terminates successfully.

As rows A and E are selected, the final solution is:

	1	2	3	4	5	6	7
A	1	0	1	0	1	0	0
E	0	1	0	1	0	1	1

Figure 3.7: Matrix representation of the solution for the example

The subcollection {A, E } is an exact cover, because each element in X is contained exactly once in subsets { A, E }. The union of these two subsets corresponds to all the elements of the set X.

There are no more selected rows at level 2, thus the algorithm returns to the next branch at level 1.

There are no more selected rows at level 1, thus the algorithm returns to the next branch at level 0.

There are no more selected rows at level 0, thus the algorithm terminates.

This is an example of how Knuth's Algorithm X works on a 5 x 7 matrix. For a standard Sudoku, it is a 729 x 324 matrix.

3.2.4 Other algorithms

There are more AI algorithms to solve a Sudoku puzzle. A stochastic optimization method is a random-based algorithm. Multiple search techniques exist such as Cultural Genetic Algorithm, or Quantum Simulated Annealing, but it is not considered in this project. [12]

CHAPTER 4 Design and Implementation

4.1 Programming Language

This project uses Python to implement the Sudoku solver. As developing features is the main goal of this project, Python is chosen because of its high productivity. Due to its simplicity, more features can be done while using less code than other languages such as C or C++. Python's portability is also a reason why this project opted for it, the will to be able to use the solver on different platforms is very important. Other languages offer better efficiency than Python in performing some task, which can result in a higher speed. However, Python is still opted because of its high flexibility and portability, analysis of the performance of the different algorithms should not change too much, as the time complexity depends on the algorithms and not the programming language used.

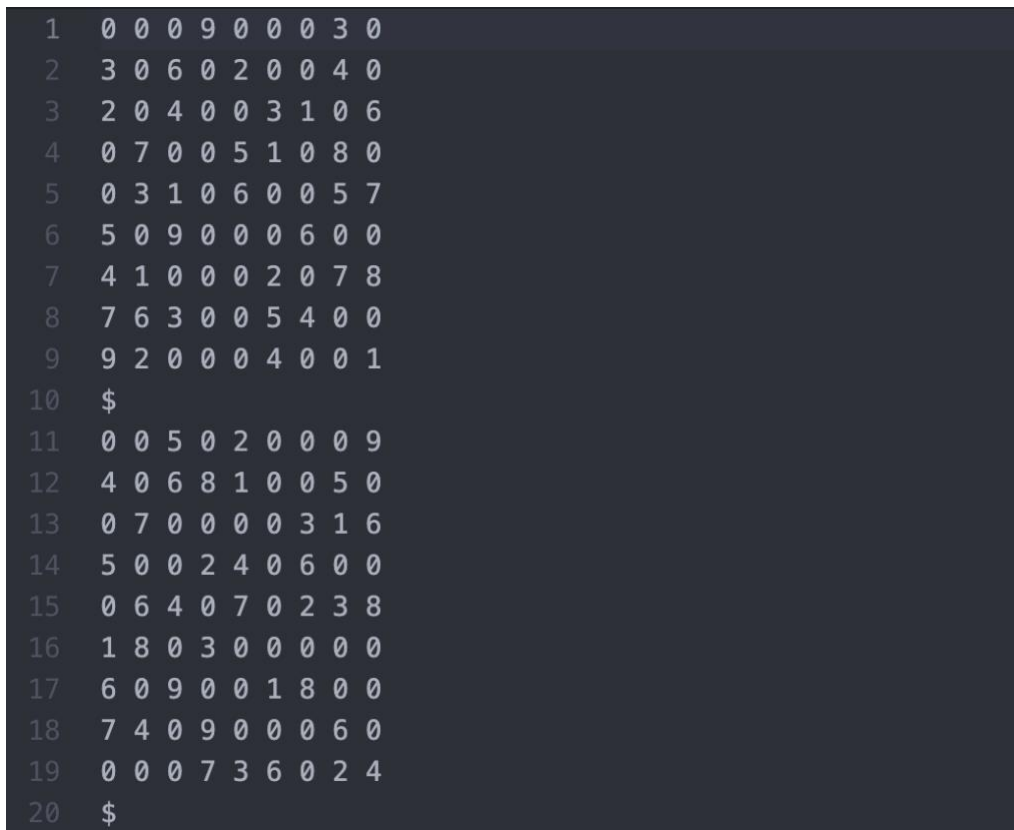
The Sudoku solver is run through the command line. The code is written in Python3 using the editor Atom. No additional program is required to run the solver.

4.2 Code Repository

The code repository is on GitLab. All development was made on the 'master' branch, as no major issue was encountered. Every change to the source code was tested before committing to the 'master' branch, to avoid fatal errors.

4.3 Code Structure

A 2D list data structure is opted to represent Sudoku puzzles. This data structure is easy to access, as each row is a row of the puzzle and each column is a column of the puzzle, it is very intuitive. As the puzzles are all the same size, 9 rows and 9 columns, the length of the 2D list remains the same as well, with 9 rows of 9 elements. Empty cells of a puzzle are filled with 0s. Grids are stored in different text files, each file represents a different difficulty of Sudoku puzzles or Hyper Sudoku puzzles. Each text file has 10 grids, each grid is separated by a \$, and there is a new line between each row, and a space between each cell for better readability. Standard Sudoku puzzles are from sudoku.com 0, Hyper Sudoku puzzles are from e-sudoku.fr 0. These websites are open source.



1	0	0	0	9	0	0	0	3	0
2	3	0	6	0	2	0	0	4	0
3	2	0	4	0	0	3	1	0	6
4	0	7	0	0	5	1	0	8	0
5	0	3	1	0	6	0	0	5	7
6	5	0	9	0	0	0	6	0	0
7	4	1	0	0	0	2	0	7	8
8	7	6	3	0	0	5	4	0	0
9	9	2	0	0	0	4	0	0	1
10	\$								
11	0	0	5	0	2	0	0	0	9
12	4	0	6	8	1	0	0	5	0
13	0	7	0	0	0	0	3	1	6
14	5	0	0	2	4	0	6	0	0
15	0	6	4	0	7	0	2	3	8
16	1	8	0	3	0	0	0	0	0
17	6	0	9	0	0	1	8	0	0
18	7	4	0	9	0	0	0	6	0
19	0	0	0	7	3	6	0	2	4
20	\$								

Figure 4.1: Database Sudoku Grids

Each algorithm is developed in an independent python file for better clarity. One final function wraps up every file, and these functions are the only ones that are called in the main file, the main file also provides the command-line interface.

A utility file is created, for functions that are used in every algorithm, such as a function named *printing* that displays the grid in a more readable way, another function named *readFile* that reads the database and creates a 2D list, and one final function named *own* that takes user input and creates a 2D list.

4.4 Implementation

4.4.1 Backtracking

4 functions constituted the *backtracking.py* file.

The first one is the *valid* function, this function takes 4 arguments: *x* for row number, *y* for column number, *n* for inserted cell value and *board* as the whole Sudoku puzzle. This function checks if the number *n* is a valid value in a given position determined by *x* and *y* in

the given *board*, if the inserted number *n* is already in the row, or the column or the box, then the function returns *False*. If it passes all the steps, then the function returns *True*.

The second function is *find_empty_cell*, which takes argument *board* that corresponds to the puzzle in a 2D list structure. This function loops through the whole board and returns row number *i* and column number *j* of an empty cell, empty cells are 0s in the list. If no empty cell is found, the function returns the tuple (-1, -1).

The third function is named *brute* as backtracking is a brute force approach of solving the puzzles and it contains the backtracking part of the file. First, it checks if there are any empty cells in the puzzle, if there is, a tuple (x, y) takes the values *i* and *j* returned by the *find_empty_cell* function, if there is no empty cells, the tuple's value is (-1, -1), the *brute* function returns the puzzle. If there are empty cell, the function tries to assign values from 1 to 9 to the detected empty cell, and checks if the assigned value is valid using the *valid* function. If the value is valid, the function puts the value in the cell. Then by calling this function recursively, it tries to fill the whole board, the backtracking happens when none of the 9 values are valid for a cell, the function resets the cell to 0 to find another path. If the board is solved, the function returns the board as a 2D list.

The last function of the *backtracking.py* file is named *brute_solve* and takes *board* as argument and prints the unsolved puzzle, and the puzzle solved by the *brute* function using the *printing* function from *utility.py* file, this function formats the output for a more readable result. The *brute_solve* function also prints out the time used for solving a puzzle with the *brute* function in seconds for later analysis.

4.4.2 Constraint Satisfaction Problem

The constraint satisfaction problem algorithm is in *csp.py* file. This file is based on Peter Norvig's python2 Sudoku solver and is correctly referenced in source code 0. This file contains 10 functions.

As this algorithm expresses the puzzle as a CSP, with sets of variables, domains and constraints, a new way to access the Sudoku grid is needed. 2 strings are created, one called *digits* represents '123456789' and another one called *rows* represents 'ABCDEFGHI'. The digits are used to express column number and letters are used to express row number. Therefore the cell C1 corresponds to the cell of the third row and first column. This notation for each cell is the same as the one in section 3.2.2 Constraint Programming. A *cross* function that takes 2 arguments *A* and *B* does the string concatenation of *rows* and *digits*, it is then stored in a list called *squares*. Thus, there are 81 elements in the list which corresponds to the 81 cells in a Sudoku puzzle. There are 3 sets of constraints in a Sudoku

puzzle: row constraints, column constraints and box constraints, each set has 9 constraints which makes a total of 27 constraints in the whole board. These constraints are stored in *unitlist* in the code. Each cell in the Sudoku grid has 3 corresponding constraints, constraints of a specific cell can be accessed with *units*, a python dictionary in the code. In this dictionary, key is the cell and value is the list of constraints. When called, this dictionary returns the row constraint, column constraint and box constraint where the given cell is also in the constraints by searching in *unitlist*. For example, *units['A2']* returns *['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'], ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9'], ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']*. To find the neighbours of a given cell, a dictionary named *peers* is created. This dictionary loops through the values returned by the *units* dictionary, and returns all values without the given cell, these values are unique using python set. For the example above, *peers['A2']* returns *{'A7', 'D2', 'B3', 'A5', 'H2', 'C1', 'G2', 'B1', 'I2', 'C2', 'A8', 'A1', 'C3', 'F2', 'A9', 'A4', 'B2', 'E2', 'A3', 'A6'}* in an unordered way.

This algorithm needs to read Sudoku grids that are in 2D list data structure in the database, the function *grid_values* takes Sudoku grids as an argument. It stores all the cell values of the grid in a list called *grid_chars*, and associates the cell name with the stored value in a dictionary. Therefore, the key of the dictionary is the cell name such as 'A2' for example, and its corresponding value is a digit between 0 and 9, 0 means that the cell is empty as explained in 4.3 Code Structure.

The domain of each cell is $D = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$, therefore each cell has possible values ranging from 1 to 9 at the start of the puzzle. These possible values are stored in the dictionary *values* in the *parse_grid* function. The function uses *grid_values* to extract the given grid, and calls the *assign* function to assign the value to the cell with initial digit value while eliminating the value from the cell's peers. If it fails to assign, *parse_grid* returns *False*, otherwise it returns the *values* dictionary.

The *assign* function is used in the previous function, it takes 3 arguments. A dictionary called *values*, which corresponds to the one returned by *parse_grid* function in this case, cell name *s* and a digit *d*. The *assign* function updates the values dictionary of a given cell *s*, by eliminating the other values than *d* with the function *eliminate*. If the *eliminate* function fails to do its job, *assign* returns *False*, otherwise it returns the *values* dictionary.

The *eliminate* function does the constraint propagation part of the code, it takes 3 arguments, a dictionary *values*, cell name *s* and digit value *d*. It removes the given value *d* from the dictionary *values* for a given cell *s* if *d* is not in the domain of the cell. If there are no more values left in *values[s]*, which means that a cell has no potential value, the function returns *False*. If a cell has only one potential value, the function eliminates this potential value from all the cell's peers. If the given digit *d* has no place elsewhere in the puzzle, the function returns *False*. If there is only one place for digit *d*, then it puts *d* in that cell, and

removes *d* from all that cell's peers. If the function has no failures, it returns the dictionary *values*.

The *search* function takes one argument, a dictionary *values*. If the domain of each cell in the puzzle has exactly one value, which means that for all cells *s* in the grid, *values[s]* has exactly one element, then the puzzle is solved, the function returns the dictionary *values*. If the puzzle is not solved, the algorithm uses variable ordering by looking for minimum remaining values for a cell, a common heuristic method. *Search* chooses a cell with the minimum number of potential values, and calls *assign* to try to eliminate potential values from the peers, this step is repeated by calling the *search* function recursively.

The *some* function returns some elements if the last step of the *search* function succeeded.

The *solve* function parses the grid with *parse_grid* and calls *search* to solve the puzzle.

The *display* function takes argument the dictionary *values*, and prints out the grid in a more readable way. Adjustments have been made to obtain the same output as the function *printing* in *utility.py*.

The last function of the *csp.py* file is named *csp_solve* and takes *board* as argument and prints the unsolved puzzle, and the puzzle solved by the *solve* function using the *display* function. The *csp_solve* function also prints out the time used for solving a puzzle with the *solve* function in seconds for later analysis.

4.4.3 Exact Cover

The exact cover algorithm is in *exact.py* file. This file is based on Ali Assaf's algorithm X and Sudoku solver, and is correctly referenced in the source code 0. The code has a GNU General Public License, which grants open-source permission for users to make changes to the software, download it or to redistribute it 0.

The first function is named *solve_sudoku*, it takes argument the whole Sudoku puzzle. Then, it creates a list of 4 dictionaries, each corresponding to one of the 4 constraints sets as in section 3.2.3 Exact Cover: cell constraints with key "rc" in code, row constraints with key "rn", column constraints with key "cn" and box constraints with key "bn". For example, cell constraint for the cell in the first row and first column of the puzzle R1C1 is labelled ('rc', (0, 0)) in the code. Row constraint of the first row with digit 9 R1#9 is labelled ('rn', (0, 9)). Column constraint of first column with digit 1 C1#1 is labelled ('cn', (0, 1)). Box constraint of top left box with digit 2 B1#2 is labelled ('bn', (0, 2)). Row, column and box numbers all start by 0 in source code. A dictionary *Y* is created, to store the possibilities. For example, a possible assignment of a number in a cell labelled R1C1#1 corresponds to an assignment of

the digit 1 in the cell of the first row and first column, this possibility can be called with $Y[(0, 0, 1)]$ in the code. Values of this key are the corresponding constraints, in this example, $Y[(0, 0, 1)]$ returns $[('rc', (0, 0)), ('rn', (0, 1)), ('cn', (0, 1)), ('bn', (0, 1))]$, note that these values can be found in the X list. Using the *exact_cover* function, which takes input X and Y and keeps only unique values using set, calling a constraint now returns all the related possibilities. For example, standard representation of cell constraint $R1C1 = (\{R1C1\#1\}, \{R1C1\#2\}, \dots, \{R1C1\#8\}, \{R1C1\#9\})$, can now be called using $('rc', (0, 0))$ as the dictionary key in the code, and it returns $(0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 0, 5), (0, 0, 6), (0, 0, 7), (0, 0, 8), (0, 0, 9)$ that are originally in the Y dictionary.

Now that the Sudoku puzzle is represented as an exact cover problem, so as a matrix (some constraints in X have been discarded, for example, if the digit one is already in the first row, there is no $R1\#1$, so no $('rn', (0, 1))$ in X), Knuth's Algorithm X can be applied to the problem to solve it, by using the functions *select* and *deselect*, the steps are the same as the ones used in section 3.2.3 Exact Cover. Possibilities are discarded and once the puzzle is solved, only 81 possibilities are left, in a (x, y, z) format with x and y ranging from 0 to 8 representing the row and column number respectively and z ranging from 1 to 9 representing the digit value in the cell.

The last function of the *exact.py* file is named *exact_solve* and takes *board* as argument and prints the unsolved puzzle, and the puzzle solved by the *solve* function using the *printing* function. The *exact_solve* function also prints out the time used for solving a puzzle with the *solve* function in seconds for later analysis.

4.4.4 Hyper Sudoku variants

Two files are for solving the Hyper Sudoku variants: *hypersudoku_backtracking.py* and *hypersudoku_csp.py*. As Hyper Sudoku differs from a standard Sudoku puzzle with 4 extra box constraints, modifications in the *backtracking.py* and *csp.py* files made it possible to solve the variants.

In *hypersudoku_backtracking.py*, modifying the *valid* function by checking the extra 4 boxes, therefore *valid* checks a total of 13 boxes in addition to the standard row and column check. The algorithm backtracks to the previous cell if some constraints are violated and all possible digits are invalid for the current cell.

In *hypersudoku_csp.py*, modifying the unitlist, which now has 4 sets of constraints sets compared to the previous 3 sets. The new set is the Hyper Sudoku box constraint which is composed of 4 constraints as there are 4 extra boxes. This brings the total number of constraints to 31 in the whole Hyper Sudoku puzzle while solving it as a CSP.

These modifications to the algorithm's constraints are enough to solve the variant Hyper Sudoku puzzle.

4.4.5 Command-Line Interface

The command-line interface is provided in the *main.py* file. The solver is run by executing the main file, and a help screen shows (see figure 4.2).

```
Sudoku Solver

Options:
  Type your own Sudoku, please type 1
  Solve a written Sudoku, please type 2
  Exit the solver, please type 3

Please enter a number: █
```

Figure 4.2: Solver screen

This screen offers the users some options and asks the user to input a number between 1 and 3 to use the solver. Option 1 lets the user type his own Sudoku puzzle, option 2 lets the user solve a puzzle in the database, option 3 exits the solver. If the user chooses either option 1 or option 2, another screen displaying the available algorithms shows (see figure 4.3).

```
Sudoku Algorithms

Options:
  For Bruteforce Standard Sudoku, please type 1
  For Constraint Propagation Standard Sudoku, please type 2
  For Exact Cover Standard Sudoku, please type 3
  For Bruteforce Hyper Sudoku, please type 4
  For Constraint Propagation Hyper Sudoku, please type 5
Please enter a number:
```

Figure 4.3: Solver algorithm screen

After choosing the correct algorithm, if the user's input is 1 in the first screen, the solver prompts the user to enter his own Sudoku line by line with each cell separated by a space (see figure 4.4). This functionality is provided by *own* in *utility.py*.

```
Please enter line by line your grid.  
For example: 1 2 3 4 5 6 7 8 9  
  
Enter space separated elements of row 0
```

Figure 4.4: User's own Sudoku

If the user chooses option 2 in the first screen, and has chosen the algorithm, the solver asks the user what difficulty of Sudoku puzzle does it need to solve (see figure 4.5).

```
Sudoku Difficulty  
  
Options:  
    For easy, please type 1  
    For hard, please type 2
```

Figure 4.5: Database Sudoku difficulty

Once the user inputted Sudoku puzzle valid, or the database Sudoku puzzle difficulty chosen, the output format of the answer remains the same (see figure 4.6).

```
0 0 0 | 9 0 0 | 0 3 0
3 0 6 | 0 2 0 | 0 4 0
2 0 4 | 0 0 3 | 1 0 6
- - - - - - - - - - -
0 7 0 | 0 5 1 | 0 8 0
0 3 1 | 0 6 0 | 0 5 7
5 0 9 | 0 0 0 | 6 0 0
- - - - - - - - - - -
4 1 0 | 0 0 2 | 0 7 8
7 6 3 | 0 0 5 | 4 0 0
9 2 0 | 0 0 4 | 0 0 1
-----
1 5 7 | 9 4 6 | 8 3 2
3 9 6 | 1 2 8 | 7 4 5
2 8 4 | 5 7 3 | 1 9 6
- - - - - - - - - - -
6 7 2 | 3 5 1 | 9 8 4
8 3 1 | 4 6 9 | 2 5 7
5 4 9 | 2 8 7 | 6 1 3
- - - - - - - - - - -
4 1 5 | 6 9 2 | 3 7 8
7 6 3 | 8 1 5 | 4 2 9
9 2 8 | 7 3 4 | 5 6 1
-----
--- 0.0016460418701171875 seconds ---
```

Figure 4.6: Initial Sudoku and solved Sudoku with used time

If option 2 is opted at the first screen, when a puzzle is solved, the solver asks the user if he wants to run another grid from the database. This screen repeats after each solved puzzle until there are no more grids in the corresponding txt file (see figure 4.7)

```
Next Grid?  
  
Options:  
    Next grid, please type 1  
    Exit, please type 2
```

Figure 4.7: Solver asking the user for another puzzle

If the input is 1, the solver outputs another Sudoku puzzle and solves it like Figure 4.6. If the input is 2, the solver exits.

This sums up the functionalities of the solver.

CHAPTER 5 Testing and Analysis

5.1 Testing

As described in section 3.1.2 Standard Sudoku Puzzle, a well-made Sudoku puzzle has only one solution, every puzzle in the database has only one solution possible and this project assumes that when users type their own Sudokus, these puzzles respect the same criteria. There are 4 txt files in the database, each file contains 10 grids which makes a total of 40 puzzles. Solutions for the 20 standard Sudoku puzzles returned by backtracking, csp and exact cover, all have been validated by hand. It is also the case for the 20 Hyper Sudoku puzzles. No unit testing is made due to a lack of time, but the solver does solve all the puzzles in the database using different algorithms and displays the time used by each algorithm, which is the aim of the project.

Each grid is run 3 times with each algorithm to get an average time for more precision.

5.2 Hardware Resources

Solver is run on local PC, therefore the performance of each algorithm is impacted by the hardware. The specification of the PC are:

- OS: macOS Big Sur 11.2.3
- Processor: 2.2GHz 6-core Intel Core i7
- Graphics: Intel UHD Graphics 630
- RAM: 16GB DDR4 2400 MHz

All tests are performed using the same PC and software, with only essential programs running to ensure consistency of the result.

5.3 Analysis for standard Sudokus

5.3.1 Backtracking

There is a total of 20 grids in the database to be solved by this algorithm. 10 graded as easy and 10 graded as expert difficulty by sudoku.com. Backtracking is a brute-force approach, its performance is expected to be slow compared to other algorithms.

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.0013401508331298828	0.0016400814056396484	0.001753091812133789	1.57777
2	0.00209808349609375	0.002438068389892578	0.0020029544830322266	2.17970
3	0.0029020309448242188	0.003604888916015625	0.003555774688720703	3.35423
4	0.0029644171396891275	0.002849102020263672	0.0030410289764404297	2.96441
5	0.0031659603118896484	0.004679203033447266	0.004530906677246094	4.12535
6	0.005460977554321289	0.006062984466552734	0.0054128170013427734	5.64559
7	0.009733915328979492	0.009826183319091797	0.00984501838684082	9.80171
8	0.0021219253540039062	0.0019459724426269531	0.0021390914916992188	2.06900
9	0.0043141841888427734	0.004699230194091797	0.004530906677246094	4.51477
10	0.010593175888061523	0.010345935821533203	0.010307073593139648	10.4154

Figure 5.1: Backtracking table for easy puzzles

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	1.285672903060913	1.2860898971557617	1.288867712020874	1286.88
2	0.43306398391723633	0.4369959831237793	0.43694376945495605	435.668
3	5.172250986099243	5.191348075866699	5.203246831893921	5188.95

4	2.31778883934021	2.307745933532715	2.3244497776031494	2316.67
5	1.3387858867645264	1.3408401012420654	1.3291070461273193	1336.24
6	0.03474783897399902	0.0340123176574707	0.03425312042236328	34.3378
7	0.43198585510253906	0.42772889137268066	0.43154096603393555	430.419
8	0.10492801666259766	0.10414910316467285	0.10429000854492188	104.456
9	5.454643964767456	5.439525127410889	5.467653036117554	5453.94
10	0.6015429496765137	0.6028859615325928	0.602114200592041	602.181

Figure 5.2: Backtracking table for hard puzzles

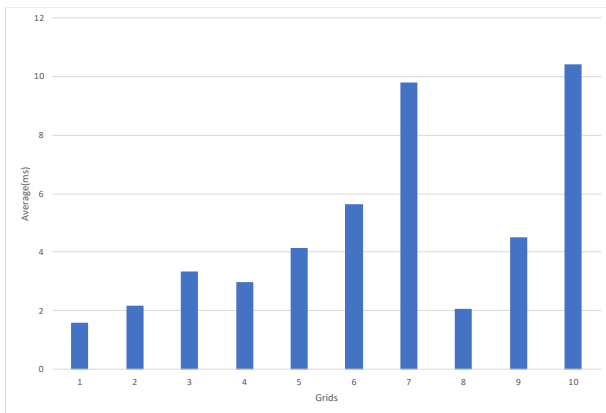


Figure 5.3: Average time for easy puzzles.

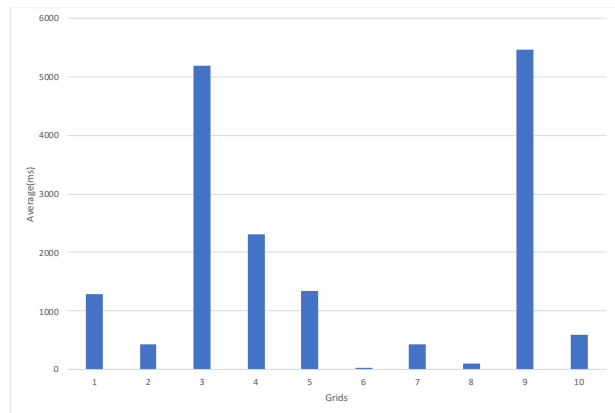


Figure 5.4: Average time for hard puzzles

A significant increase in time can be observed between the two difficulties of the puzzle. Where for easy puzzles, the backtracking method never exceeds 11 ms to solve a puzzle, for hard puzzles, even the fastest solution takes 34 ms and the slowest one takes up to 5 seconds. This difference can be explained by the fact that there are generally more clues given in the easy grids than the hard grids. Another factor is that backtracking in this project has no heuristic method, therefore it always go through the grid the same way no matter the

clues, the algorithm starts by the first empty cell of the grid and tries digits from 1-9. The more backtrack needed in solving a puzzle, the more time the algorithm takes.

5.3.2 Constraint Programming

Constraint programming algorithm solves puzzles that are in the same database as backtracking. There are two types of Sudokus in the database, 10 easy grids and 10 hard grids. This algorithm is expected to be more efficient than backtracking in solving hard puzzles, as there is a heuristic method in this algorithm.

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.004914045333862305	0.005265235900878906	0.00545191764831543	5.21040
2	0.0050351619720458984	0.004868984222412109	0.004681110382080078	4.86175
3	0.005506038665771484	0.004932880401611328	0.005099058151245117	5.17933
4	0.004841804504394531	0.004723072052001953	0.004853010177612305	4.80596
5	0.0045931339263916016	0.004900932312011719	0.0051190853118896484	4.87105
6	0.004940032958984375	0.004991054534912109	0.005122184753417969	5.01776
7	0.004858970642089844	0.00483393669128418	0.005101919174194336	4.93161
8	0.004905223846435547	0.0047528743743896484	0.005002021789550781	4.88671
9	0.004810333251953125	0.0048370361328125	0.005059242248535156	4.90220
10	0.004591941833496094	0.004785299301147461	0.0049631595611572266	4.78013

Figure 5.5: CSP table for easy puzzles

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.006618022918701172	0.0082900524139404	0.0065348148345947266	7.14763
2	0.00565791130065918	0.0063190460205078125	0.006063222885131836	6.01339
3	0.020637989044189453	0.016963720321655273	0.02107524871826172	19.5590
4	0.009830236434936523	0.008533000946044922	0.008785247802734375	9.04950
5	0.004937410354614258	0.004970073699951172	0.0047762393951416016	4.89457
6	0.00789499282836914	0.007922887802124023	0.008070945739746094	7.96294
7	0.0049228668212890625	0.004624128341674805	0.0053827762603759766	4.97659
8	0.006520271301269531	0.00640416145324707	0.005944013595581055	6.28948
9	0.007712125778198242	0.007925987243652344	0.009093046188354492	8.24372
10	0.009009122848510742	0.008620977401733398	0.009122133255004883	8.91741

Figure 5.6: CSP table for hard puzzles

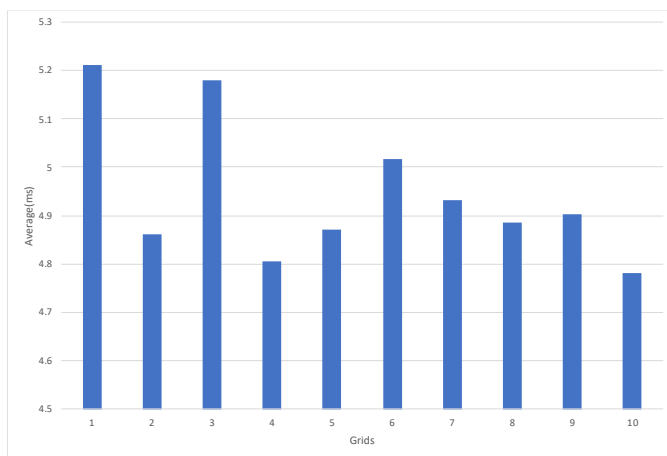


Figure 5.7: Average CSP time for easy puzzles

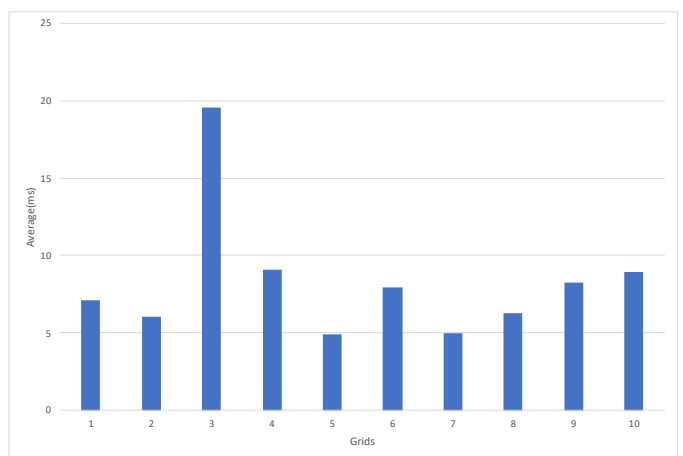


Figure 5.8: Average CSP time for hard puzzles

CSP has a minimum performance of more than 4 ms, which is more than the minimum of backtracking, this can be explained by the fact that this algorithm is implemented with more code, therefore more elements to compile for the computer. There is only a very small increase in time for harder puzzles, this small difference can be explained by the efficiency of the algorithm, which uses heuristic method to choose which cell to solve after each step (explained in section 4.4.2 Constraint Satisfaction Problem).

5.3.3 Exact Cover

The exact cover algorithm solves the same puzzles from the database as the other two methods, 10 easy standard Sudokus and 10 hard standard Sudokus. By expressing the Sudoku grid as an exact cover problem, the algorithm only needs to solve a matrix, therefore this algorithm is also expected to be more efficient than backtracking for hard puzzles, but slower than CSP.

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.006553173065185547	0.006176948547363281	0.0063381195068359375	6.35608
2	0.00515294075012207	0.004959821701049805	0.0049610137939453125	5.02459
3	0.005212068557739258	0.0055789947509765625	0.004514932632446289	5.10200
4	0.00491023063659668	0.004759073257446289	0.004873037338256836	4.84745
5	0.005141019821166992	0.0051267147064208984	0.005028247833251953	5.09866
6	0.004945993423461914	0.004947185516357422	0.004903316497802734	4.93217
7	0.005109071731567383	0.004599094390869141	0.005480766296386719	5.06298
8	0.005065202713012695	0.00460505485534668	0.005120992660522461	4.93042
9	0.004892826080322266	0.005012989044189453	0.004586935043334961	4.83092

10	0.005204916000366211	0.005285739898681641	0.005400896072387695	5.29718
----	----------------------	----------------------	----------------------	---------

Figure 5.9: Exact Cover table for easy puzzles

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.008473873138427734	0.008497953414916992	0.008559942245483398	8.51059
2	0.009385108947753906	0.008426904678344727	0.009485721588134766	9.09925
3	0.03735661506652832	0.03766584396362305	0.03725433349609375	37.4256
4	0.019131898880004883	0.019995927810668945	0.01898193359375	19.3699
5	0.00792694091796875	0.007883071899414062	0.008410930633544922	8.07365
6	0.015487194061279297	0.015547752380371094	0.01468801498413086	15.2410
7	0.006564140319824219	0.0061872005462646484	0.0056591033935546875	6.13681
8	0.009937047958374023	0.010026931762695312	0.010331869125366211	10.0986
9	0.013519763946533203	0.012639999389648438	0.01890707015991211	15.0223
10	0.013720035552978516	0.013689041137695312	0.013419866561889648	13.6096

Figure 5.10: Exact Cover table for hard puzzles

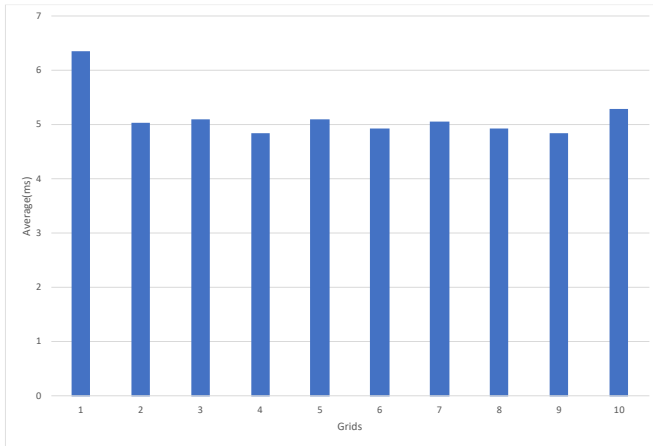


Figure 5.11: Average EC time for easy puzzles

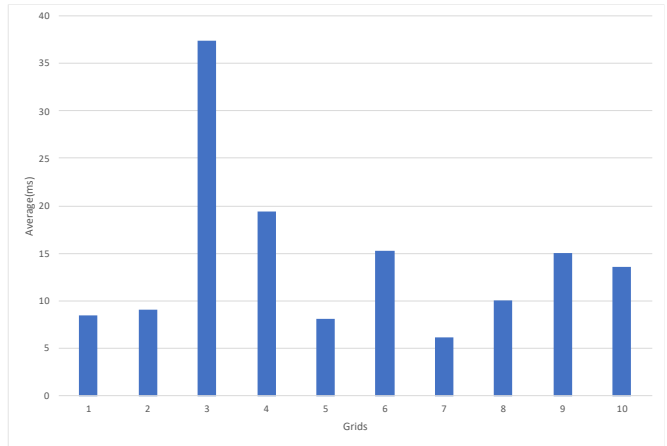


Figure 5.12: Average EC time for hard puzzles

As CSP, exact cover algorithm has a very stable performance when solving easy Sudokus. Averaging around 5 ms per easy grid, which is slower than backtracking, this can be explained just as CSP, a lengthier code makes this algorithm slower to compile. When solving hard puzzles, this algorithm performs better than backtracking and is slower than CSP just as expected. This algorithm does not include a major heuristic method like CSP, but by expressing the Sudokus as an exact cover problem simplifies the solving of the puzzle, which makes this algorithm faster than backtracking.

5.3.4 Overall Analysis for Standard Sudokus

Overall, for easy Sudokus, backtracking has the best performance with around 4.7 ms average, this can be explained as its code is the shortest, an easy puzzle does not solicit many methods from the different algorithms, solving of these puzzles is very straightforward. CSP and Exact Cover are less efficient but the difference is minimal, with around 4.9 ms average and 5.1 ms average respectively. For hard Sudokus with fewer clues, CSP performs the best, with around 8 ms average for the given puzzles in the database, then follows Exact Cover with around 14 ms average and finally backtracking which is far behind with 1718 ms average. This huge difference between backtracking and the other two algorithms can be explained by the lack of clues in harder puzzles, therefore more backtracking is performed in the algorithm which causes a lack of efficiency. Whereas, the other two algorithms still express the puzzle as a CSP and as an exact cover problem respectively, which does not alter their performance so much. CSP's heuristic method in its *search* function makes the algorithm faster than exact cover. The association of expressing a Sudoku puzzle as an exact cover problem and the use of Knuth's Algorithm X to solve that problem, makes exact cover a more efficient algorithm than backtracking in solving hard sudokus.

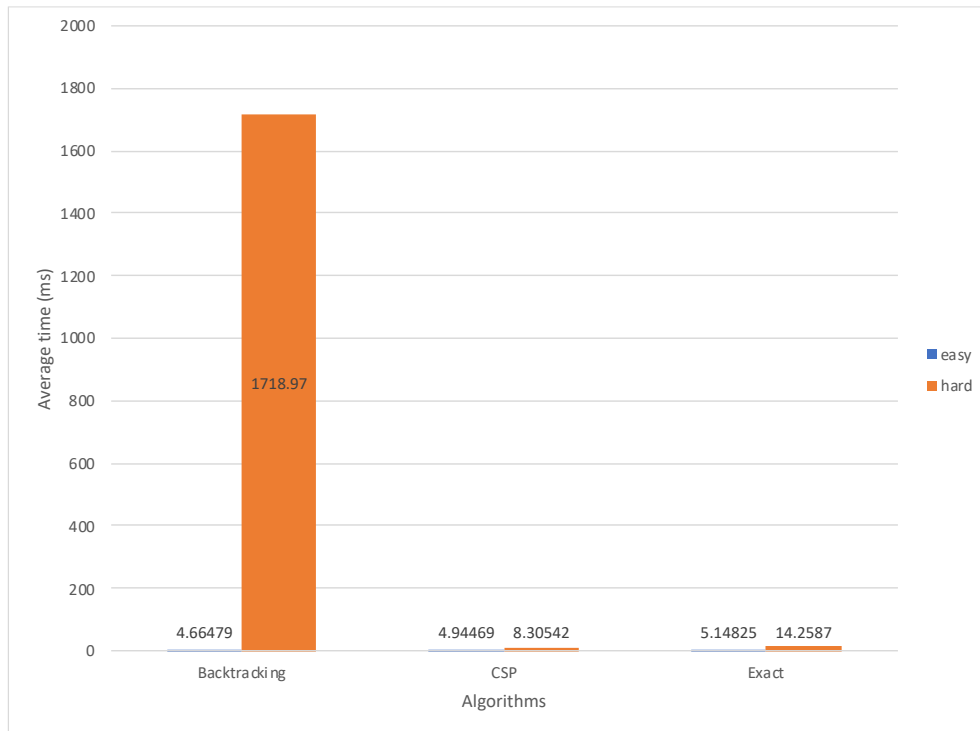


Figure 5.13: Average time of all algorithms for Standard Sudoku

5.4 Analysis of Hyper Sudokus

5.4.1 Backtracking for Hyper Sudoku

Just as standard Sudokus in the database, there is a total of 20 hyper Sudoku puzzles too. 10 are labelled as easy and another 10 labelled as hard by e-sudoku.fr. Backtracking is expected to perform fairly well in easy hyper Sudokus and very poorly in hard hyper Sudokus.

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.0036611557006835938	0.0035338401794433594	0.003064870834350586	3.41996
2	0.0020182132720947266	0.002010822296142578	0.0018510818481445312	1.96004
3	0.0010728836059570312	0.0009009838104248047	0.0010912418365478516	1.02170
4	0.013570070266723633	0.015076160430908203	0.014076948165893555	14.2411

5	0.037506103515625	0.03620004653930664	0.03625178337097168	36.6526
6	0.03377032279968262	0.03060293197631836	0.030639171600341797	31.6708
7	0.009909868240356445	0.008951187133789062	0.009302139282226562	9.38773
8	0.0012218952178955078	0.0012178421020507812	0.001194000244140625	1.21125
9	0.008750677108764648	0.00870823860168457	0.008104801177978516	8.52124
10	0.016927003860473633	0.016437053680419922	0.01613593101501465	16.5000

Figure 5.14: Backtracking table for easy hyper Sudoku puzzles

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.34116387367248535	0.34184718132019043	0.34281492233276367	341.942
2	15.707941055297852	16.26479411125183	15.899603843688965	15957.4
3	4.9730119705200195	5.1021058559417725	4.971346378326416	5015.49
4	0.5106329917907715	0.5145809650421143	0.5171787738800049	514.131
5	38.899807929992676	39.91359305381775	40.225334882736206	39679.6
6	0.6702980995178223	0.694188117980957	0.7058260440826416	690.104
7	128.3936402797699	133.47421193122864	127.89539408683777	129921
8	1.2595899105072021	1.2988879680633545	1.298346996307373	1285.61
9	1.3026628494262695	1.350059986114502	1.3667981624603271	1339.84
10	7.905869960784912	8.163571119308472	8.072489023208618	8047.31

Figure 5.15: Backtracking table for hard hyper Sudoku puzzles

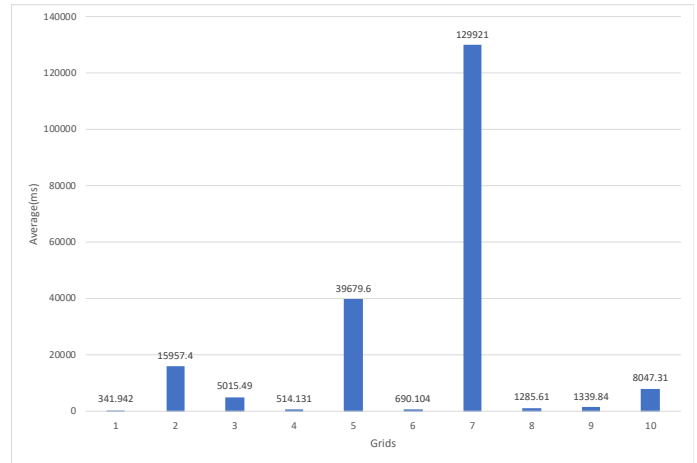
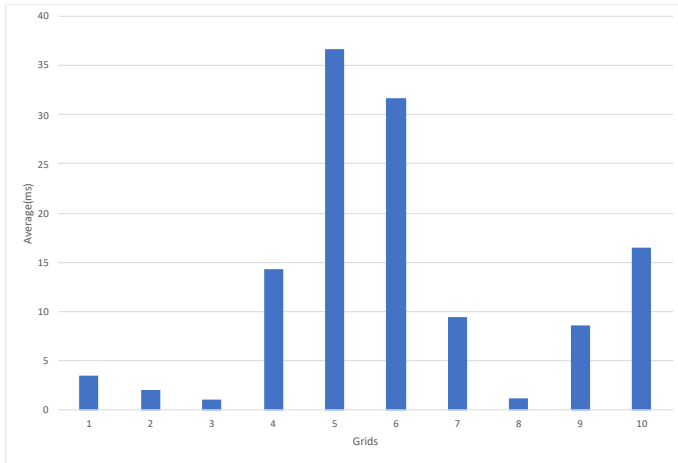


Figure 5.16: Average backtracking time for easy

Figure 5.17: Average backtracking time for hard

A huge increase in time and very long solving time can be observed when solving hard puzzles. This can be explained by the fact that there are more constraints which corresponds to the 4 extra hyper Sudoku boxes, thus backtracking backtracks more when a constraint is violated.

5.4.2 CSP for Hyper Sudoku

This modified CSP algorithm has one more constraint set which corresponds to the hyper sudoku boxes. This algorithm solves 20 grids, the same as backtracking for hyper Sudokus.

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.005585193634033203	0.0056209564208984375	0.00583195686340332	5.67937
2	0.005600929260253906	0.005882740020751953	0.0055348873138427734	5.67285
3	0.0051119327545166016	0.005438804626464844	0.005181074142456055	5.24394
4	0.0057048797607421875	0.005597114562988281	0.005638837814331055	5.64694
5	0.0055389404296875	0.0055370330810546875	0.00564885139465332	5.57494

6	0.0055921077728271484	0.004848957061767578	0.005402088165283203	5.28105
7	0.0054819583892822266	0.005773067474365234	0.005354881286621094	5.53664
8	0.0055620670318603516	0.005478858947753906	0.006025075912475586	5.68867
9	0.005601167678833008	0.005458831787109375	0.005386829376220703	5.48228
10	0.005657672882080078	0.0050182342529296875	0.0055391788482666016	5.40503

Figure 5.18: CSP table for easy hyper Sudoku puzzles

Grids	Test 1 (s)	Test 2 (s)	Test 3 (s)	Average (rounded in ms)
1	0.01194000244140625	0.012660026550292969	0.012367010116577148	12.3223
2	0.006752967834472656	0.0064220428466796875	0.006289005279541016	6.48801
3	0.0542597770690918	0.051853179931640625	0.05097079277038574	52.3612
4	0.007296085357666016	0.0076329708099365234	0.0074748992919921875	7.46799
5	0.012808084487915039	0.013007164001464844	0.011523008346557617	12.4461
6	0.01134490966796875	0.011215925216674805	0.011272907257080078	11.2779
7	0.008751869201660156	0.010045051574707031	0.008953094482421875	9.25001
8	0.0061070919036865234	0.0061299800872802734	0.006109952926635742	6.11567
9	0.02385711669921875	0.023035287857055664	0.02202606201171875	22.9728
10	0.01013803482055664	0.009792089462280273	0.00984811782836914	9.92608

Figure 5.19: CSP table for hard hyper Sudoku puzzles

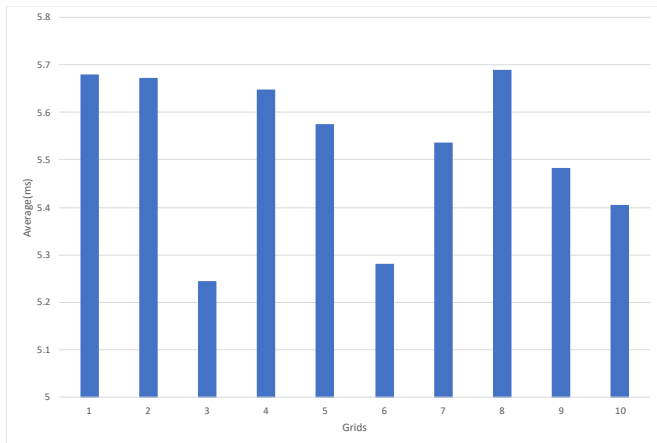


Figure 5.20: Average CSP time for easy

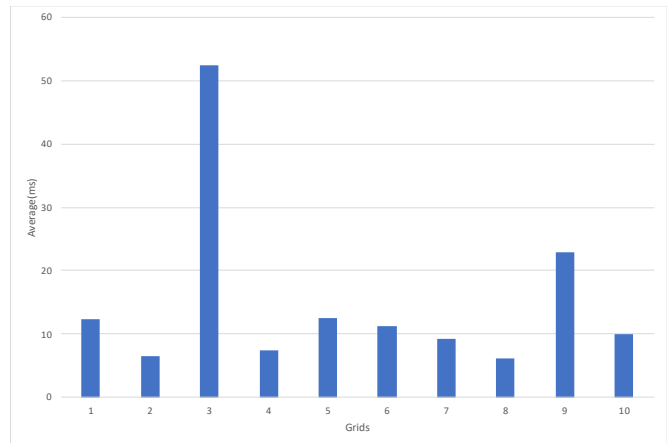


Figure 5.21: Average CSP time for hard

Average time used to solve easy hyper Sudokus with algorithm CSP is very stable, it always gets the solution between 5 and 6 ms. This stability can be explained by the fact that the grid is expressed in a CSP problem, therefore there is no direct relation between Sudoku difficulty and CSP difficulty. Average time for hard puzzles ranges from 6 ms to 52 ms, compared to backtracking, this average time can be called stable too.

5.4.3 Overall Analysis for Hyper Sudokus

Overall, for both easy and hard Hyper Sudokus, CSP has the best performance with around 5.5 ms average for easy puzzles and around 15.1 ms for hard puzzles. Backtracking on the other hand has around 12ms average for easy puzzles and an astronomical around 20279 ms (around 20 seconds average) for hard puzzles, this is due to the higher number of constraints that forces the algorithm to backtrack more often to satisfy all the constraints. CSP is not that much impacted because it has a heuristic *search* function which makes the method more intelligent. The environment of the CSP is also simpler as it works with sets.

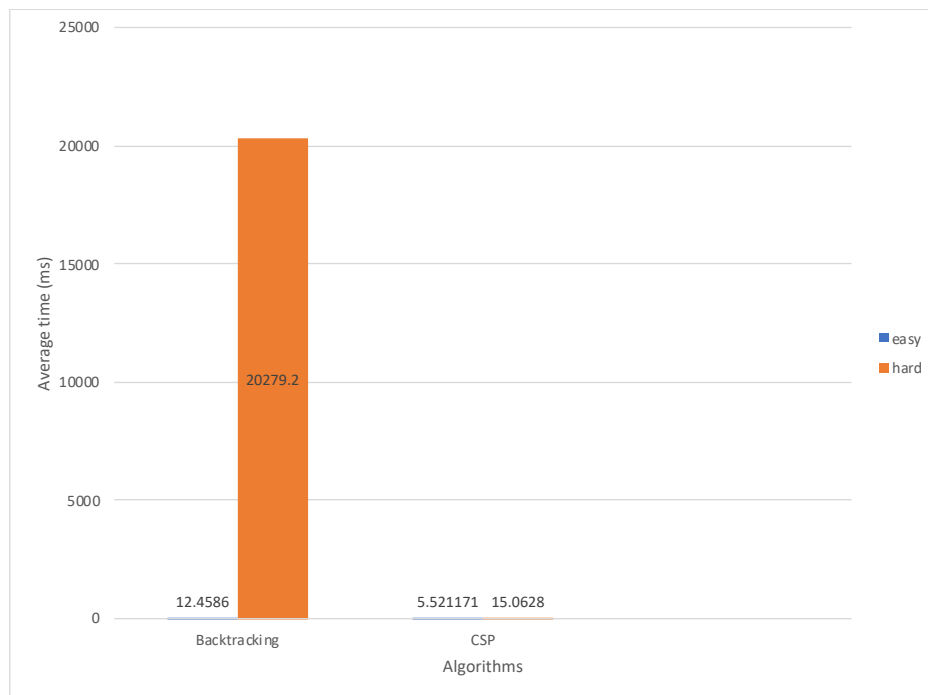


Figure 5.22: Average time for all algorithms for Hyper Sudokus

CHAPTER 6 Legal, Social, Ethical and Professional Issues

3.1 Legal issues

In this project, there are no major legal issues. Minor legal issues are addressed, as modification and inclusion of other's source code are correctly referenced in the source code and the report. The use of data is also referenced in the report. All of these are assured to be open source and licensed under GNU General Public License or allow free non-commercial use.

3.2 Social issues

Creating a solver to solve sudokus and its variants has no social implication, therefore there are no related social issues.

3.3 Ethical issues

No ethical issues arose during the development of this project. The solver may be used to cheat in competition for example, but it is very unlikely as the results of this project are not revolutionary.

3.4 Professional issues

No external client or any other shareholder is needed for this project, and only one person developed it. Use of other's work is correctly referenced and assured to be open source. Testing was made by hand. Back-ups of the source code are stored on a GitLab repository and local computer.

CHAPTER 7 Conclusion

7.1 Summary

This project aimed to create a solver that can solve Sudoku puzzles and its variants using different AI algorithms, and also compare the performance of these algorithms. The aim has been achieved to some extent. A solver has been created, with the ability to solve standard Sudoku puzzles with 3 different AI algorithms: Backtracking, Constraint Programming and Exact Cover. It can also solve Sudoku variant Hyper Sudoku with 2 different AI algorithms: Backtracking and Constraint Programming. When displaying a solution, the solver also shows the time used to solve the puzzle, thus letting the user compare performances between different algorithms.

Some of the objectives from section 1.1 Project aim and objectives were abandoned due to a poor time management. Features such as the solving of the Sudoku variant Mini Sudoku, or solving of Hyper Sudoku with the exact cover algorithm were not implemented. The project instead focused on improving and testing existing features.

Overall, the project's main goal is achieved, but the solver does not have as many functionalities as wanted at the start of the project.

7.2 Future work

As mentioned in the previous section, many functionalities ideas were abandoned due to lack of time, and these features could be implemented in the future. Many improvements can be made to upgrade the solver.

First, the implementation of the exact cover algorithm, that can solve Hyper Sudokus, can complete the solver. Implementation of the already working algorithms that can solve more Sudoku variants, such as Mini Sudokus, Killer Sudokus or Twin Sudokus, can be very interesting.

Next, improvements in the command line interface can be made. Features such as return to the previous menu, or switching to another txt file when all puzzles in the current txt file are solved, can be implemented. All these features do not exist in the current version of the solver and if implemented, it will provide a better user comfort.

Finally, more AI algorithms can be implemented, such as stochastic optimization using different search techniques as mentioned in section 3.2.4 Other algorithms.

7.3 Personal Reflection

Due to a lack of experience and knowledge about individual large-scale project planning, development was poorly executed in semester 2. The initial plan was to start developing software either in January or February, but due to the current situation with COVID-19 and personal matters, development only started in March, which resulted in unfinished objectives. Luckily, the main goal has been achieved and I'm satisfied with the current solver.

Through this project, I have discovered new AI algorithms, and the use of different methods to solve Sudoku puzzles was very exciting. Surprisingly, I did not know that some algorithm like backtracking was that easy to implement and that it can solve that many Sudoku puzzles. Reading through some research papers talking about different AI algorithms, I found them very overwhelming as many methods explained are very difficult to understand and to implement, whereas other research papers were very interesting, and I found myself often surprised by the ingenuity of the author. While implementing CSP and exact cover algorithms using Python, I have strengthened my Python coding skill and learned how to use the dictionary structure.

Another thing I learned in this project, is to spread the workload equally, setting objectives with planning, and respecting that planning, are all essential parts to accomplish a good project. Finding motivation while being lockdown due to the current pandemic was very challenging, but yet, the sensation to have achieved some goals remains very joyful. Keeping that will of completing goals is key to find motivation. Aside from working, taking reasonable time to relax and rest is primordial and makes us even more productive. When facing difficulties such as bugs, dealing with the frustration can be very hard, therefore being patient and calm is very important to overcome these challenges.

To conclude, I am happy with the current project, although it still has many things to improve. And I am satisfied with all the experiences I have gained.

List of References

- [1]. History of Sudoku:
<https://sudoku.com/how-to-play/the-history-of-sudoku/>
- [2]. Standard Sudoku puzzle example:
<https://www.sudoku.ws/standard-1.htm>
- [3]. Hyper Sudoku puzzle example:
https://www.educmat.fr/categories/jeux_reflexion/fiches_jeux/hypersudoku/index.php
- [4]. Mini Sudoku puzzle example:
<https://sudoku.cool/mini-sudoku.php>
- [5]. Dhanya Job and Varghese Paul. **Recursive Backtracking for Solving 9*9 Sudoku Puzzle**. Bonfring International Journal of Data Mining, 2016, Vol.6, No.1
- [6]. Backtracking algorithm on a Sudoku puzzle:
https://upload.wikimedia.org/wikipedia/commons/8/8c/Sudoku_solved_by_bactrackin_g.gif
- [7]. Jyoti, Tarun Dalal. **Constraint Satisfaction Problem: A case study**. International Journal of Computer Science and Mobile Computing, 2015, Vol.4
- [8]. Helmut Simonis. **Sudoku as a Constraint Problem**. CP Workshop on modelling and reformulating Constraint Satisfaction Problems, 2005, Vol.12
- [9]. Exact Cover Problem:
<https://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-1/>
- [10]. The 729 x 324 matrix for a standard Sudoku puzzle.
<https://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm>
- [11]. Donald E. Knuth. **Dancing Links**, 2000

- [12]. Meir Perez, Tshilidzi Marwala. **Stochastic Optimization Approaches for Solving Sudoku**, 2008

- [13]. Standard Sudoku puzzles:
<https://sudoku.com>

- [14]. Hyper Sudoku puzzles:
<https://www.e-sudoku.fr/windoku.php>

- [15]. Peter Norvig's python Sudoku solver:
<http://norvig.com/sudoku.html>

- [16]. Ali Assaf's algorithm X:
https://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html

- [17]. GNU General Public License:
<https://snyk.io/learn/what-is-gpl-license-gplv3-explained/>

Appendix A

External Materials

Software Tools

The software used for development of source code :

- Atom

Appendix B

Project Code

Source Code

The GitLab repository for source code:

<https://gitlab.com/sc18s3h/fyp>