The University of Manchester

Third Year Project

Final Report

Sudoku Solver Using Propositional Logic

Author:
Razvan F. Vasile
Degree Program:
BSc (Hons) Computer Science and Mathematics

Supervisor:
Dr. Konstantin Korovin

May 1, 2018

Contents

List of Figures

List of Tables

Abstract

This project aims to replicate a handful of algorithms and techniques used to visualize the structure of difficult Sudoku puzzles.

The approach is to implement an efficient CDCL SAT Solver, involving well known procedures such as Backjumping, Conflict Driven Clause Learning and two decision heuristics: Variable State Independent Decaying Sum (VSIDS) and Largest Individual Sum (LIS).

Moreover, the Sudoku puzzle is encoded into SAT and the visualization is done by interactively displaying the different actions made by the solver, involving decided, propagated and conflict literals at the time they occur.

Finally, the evaluation of the SAT Solver is another area of prime importance. This involves profiling the application and analysing the structure of hard SAT and Sudoku instances.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Konstantin Korovin for all his support, guidance and patience through all the stages of the report.

Moreover, I would like to thank my family for always being so supportive and helping me throughout my whole life.

Chapter 1

Introduction

This project implements a particular SAT solving algorithm that will be used to efficiently compute the solution for a range of hard instances of Sudoku puzzles. The project attempts to replicate a range of state-of-the-art techniques that aim to drastically improve the performance of the solver. The most important features are the following:

- The SAT solver enhancements include: non-chronological Backjumping, Conflict-Driven Clause Learning and the Two-Watch literal scheme for unit propagation. Moreover, two decision heuristics are implemented: VSIDS and LIS.
- A graphical user interface enclosing the Sudoku puzzle grid and being able to visualize the actions of the solver while it is looking for the solution.
- Evaluation phase that involves studying the structure of SAT and Sudoku instances. Measurement of the performance of the SAT Solver and different Sudoku encodings.

Alternatively, we aim to implement an efficient CDCL SAT solver that would allow us to formally reason about the complexity of the Sudoku game and come up with a solution. Some other extensions are added that allow interactive control over the Sudoku board such as giving a particular value for a focused entry, showing visual feedback for easy detection of mistakes and being able to choose from different puzzle difficulties.

1.1 Motivation

The Satisfiability Problem (SAT) was the first problem proven to be NP-Complete [1]. This means that any other problem in the complexity class NP, which includes a wide range of optimization and decision problems are at most as difficult to solve as SAT. Furthermore, the importance of the SAT problem has grown considerably in the recent years. This is due to better implementation techniques such as the two watch literal scheme and conceptual enhancements like non-chronological backtracking. For this reason, SAT algorithms are able to solve problems involving tens of thousands of variables [2], which is sufficient for many problems in areas as diverse as hardware verification, test pattern generation and problems from algebra.

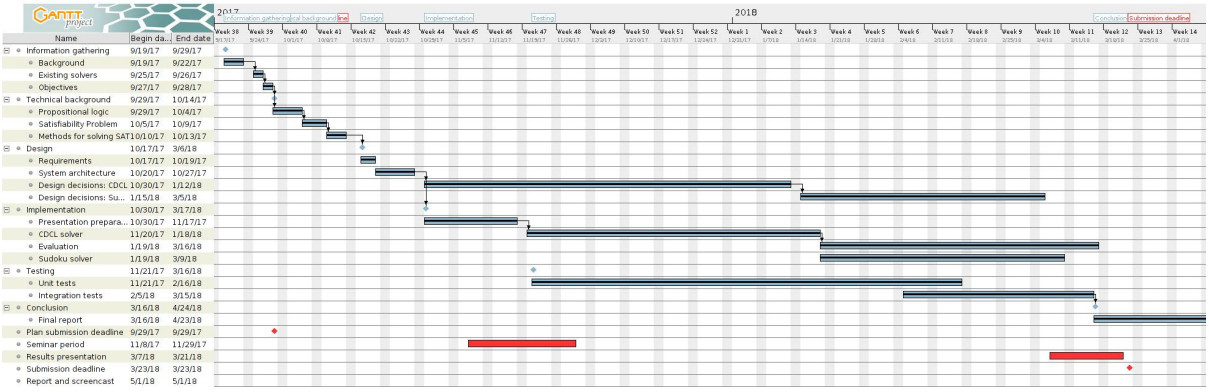

Figure 1.1: Gantt chart with the main phases of the project.

1.2 Goals

In broad terms, the completion of the project involved following this path:

- Study of different technologies: SatGraf, MiniSAT, profiling tools (e.g. Kcachegrind and Hotspot).
- Implementation of the CDCL algorithm. The choice of which decision heuristics to develop.
- Evaluation phase studying the structure of SAT and Sudoku instances using SatGraf. Profiling performance using aforementioned technologies.
- Integration with the Sudoku game and implementation of the *Interactive Mode*.

In the initial phase, I read some of the literature that reviewed how the CDCL algorithm works [3; 4; 5] and provided good examples for each technique that was to be implemented. The summary of this research is shown in Chapter 2. Moreover, we discovered SatGraf [6], a tool used to visualize the evolution of SAT formulas and which would later be used to derive conclusions on the experimental results.

In the second phase, we implemented the SAT solver and created unit tests[1] to ascertain that further modifications would not introduce unexpected bugs in the application. At this time, we also decided to use Gitlab to store the project [7], in order to facilitate working on different features simultaneously. The implementation approach is explained in Chapter 3.

Thirdly, the evaluation phase involved using the Gorilla problem generator [8] to assess the SAT Solver against problems of different difficulties. The results are shown in Chapter 4.

Finally, the last phase of the project involved implementing the Sudoku GUI and the integration with the SAT Solver. The most significant feature represents the Interactive Mode, which allows the user to visualize the steps made by the solver while looking for a solution. The implementation approach is further described in Chapter 3.

[1] A screenshot of the unit tests that were created is available in the appendix.

Chapter 2

Preliminaries

2.1 Propositional Logic Definitions

In order to help us reason about the ideas that will be introduced, we need to formally define the notions that we are talking about. The following definitions represent an introduction to propositional logic and enable the reader to understand the ideas behind the CDCL SAT Solver.

Definition 2.1. Propositional Language. A *propositional language* is a non-empty set L of symbols, called *propositional variables* or just *variables*, usually denoted s, p, q, l_1, l_2.... The set of *propositional terms* on L, denoted SL, is defined by induction:

- $S_0L = L$
- $S_{n+1}L = S_nL \cup \{(\neg s), (s \vee t), (s \wedge t) \colon s, t \in S_nL\}$

And finally define

$$SL = \bigcup_{n \geq 0} S_nL.$$

Note that instead of the negation symbol, $\neg s$, we use $\bar{s}$, as this will ease the creation of figures and the illustration of some examples later on.

Example 2.2. For instance $L = \{l_1, l_2, l_3\}$, then the propositional terms are: $(l_1 \vee l_2 \vee l_3) \in S_1L$, $(\overline{l_2}) \in S_1L$, $(l_1 \wedge \overline{l_2}) \in S_2L$ and $((\overline{l_1} \vee l_2) \wedge l_3) \in S_3L$.

Definition 2.3. Valuation. Let L be a propositional language and SL the corresponding propositional terms. A *valuation* on the set of propositional terms is a function $v \colon SL \to \{\top, \bot\}$ such that for all terms $s, t \in SL$ we have

1. $v(\bar{s}) = \top$ if and only if $v(s) = \bot$
2. $v(s \vee t) = \top$ if and only if $v(s) = \top$ or $v(t) = \top$
3. $v(s \wedge t) = \top$ if and only if $v(s) = \top$ and $v(t) = \top$.

Definition 2.4. Literal. Fix a propositional language L. We call a propositional variable p or its negation $\overline{p}$ a literal. For a set of variables L, define the set of literals Σ over L by

$$\Sigma \coloneqq L \cup \{\overline{p} \mid p \in L\}.$$

We say that p is a positive literal and $\overline{p}$ is a negative literal. Lastly, p and $\overline{p}$ are called *complementary literals*.

Definition 2.5. Clause. Let Σ be a set of literals. We define an **empty clause** to be a clause consisting of no literals, that is $C_0(\Sigma) \coloneqq \square$. A clause is a disjunction of literals and is defined as

$$C_k(\Sigma) \coloneqq \{l_1 \vee l_2 \vee ... \vee l_k \mid l_1, l_2, ..., l_k \in \Sigma\}$$

to be a clause of length k.

Moreover, the set of all clauses over the set of literals Σ is defined as:

$$C(\Sigma) \coloneqq \bigcup_{i=0}^{\infty} C_i(\Sigma).$$

The **empty clause** is by definition unsatisfiable (UNSAT), meaning that it has no solution.

A **unit clause** is a clause of length one.

From now on, we will consider the set of literals to be understood and will write C instead of $C(\Sigma)$.

Definition 2.6. Let $c \in C$ be a clause and $l \in \Sigma$ a literal. If c contains multiple occurrences of l then l is called a **duplicate literal** in c.

Definition 2.7. Let $c \in C$ be a clause and $l \in \Sigma$ a literal. If c contains both l and $\bar{l}$ then c is called a **tautological** clause. A tautological clause is always true.

Definition 2.8. CNF formula. A formula in Clausal Normal Form (CNF) is a conjunction of one or more clauses. A formula containing no clauses is said to contain *an empty set of clauses* and is defined as $F_0(C) \coloneqq \{\emptyset\}$.

Moreover, a formula of length $k \in \mathbb{N}^+$ is defined by:

$$F_k(C) \coloneqq \{c_1 \wedge c_2 \wedge ... \wedge c_k \mid c_1, c_2, ..., c_k \in C, k \in \mathbb{N}^+\},$$

and the set of all possible formulas follows:

$$F(C) \coloneqq \bigcup_{i=0}^{\infty} F_i(C).$$

Observation. In general, formulas in CNF that are resolved by SAT solvers do not involve characters to represent literals. Instead, literals are represented by integers. For instance, a clause that consists of 3 literals could be $(1\ \overline{2}\ 3)$.

Example 2.9. A formula F defined on the set of variables $L = \{1, 2, 3, 4, 5\}$ is given by:

$$F = (1 \vee 3 \vee \overline{2} \vee \overline{4}) \wedge (\overline{1} \vee 4 \vee 5).$$

Formula F contains two clauses: $(1 \vee 3 \vee \overline{2} \vee \overline{4})$ and $(\overline{1} \vee 4 \vee 5)$ respectively.

Definition 2.10. Total/Partial truth assignment. A (total) truth assignment is a valuation v such that every literal p in Σ is assigned a value by v. Moreover, v is defined as $v : \Sigma \mapsto \{\top, \bot\}$ such that

$$v(p) = \top \iff v(\overline{p}) = \bot, \text{ for all } p \in \Sigma.$$

Intuitively, by saying that $v(\overline{p}) = \bot$, I think of $\overline{p}$ as not being defined in v. This helps to reason about assertion trails, as a literal and its negation cannot belong to the trail at the same time.

Moreover, a partial truth assignment is a truth assignment u where only a subset $M \subsetneq \Sigma$ is defined by u. That is

$$u : M \mapsto \{\top, \bot\}, \quad M \subsetneq \Sigma.$$

Definition 2.11. Assertion Trail. During the solving process, the solver needs to keep track of the current truth assignment. A truth assignment is called an **assertion trail** where some literals contained in it are called **decision literals**. The decision literals are annotated like so l^d. All other literals in the trail that are not decision literals are called **implied literals**.

Definition 2.12. Let p be a literal. A trail T for which $p \in T$ and $\overline{p} \in T$ is said to be in a **conflicting state**.

Example 2.13. Example of trails:

- 1
- $\overline{1}$
- 1 3 $\overline{2}$
- 1 $\overline{1}$ 2 - in a conflicting state

Definition 2.14. Let M be a trail of the form

$$(M_0\ l_1^d\ M_1\ ...\ l_k^d\ M_k),$$

where l_i are the decision literals in M. M is considered to be at decision level k, since there are k decision literal in M. Literal of the form $l_i^d\ M_i$ belong to decision level i.

Example 2.15. A trail could be of the form

$$M := [4, 5^d, 3, \overline{2}, 1^d, 6^d, \overline{7}]$$

.

Decision literals are $5, 1$ and 6. Literal 4 belongs to decision level 0, 5^d to decision level 1 and $\overline{7}$ to decision level 3. All non-decision literals are implied literals.

Definition 2.16. SAT Problem. The Boolean Satisfiability problem (SAT) is the problem of determining if there exists a total truth assignment that makes true a given *Boolean formula* in Conjunctive Normal Form.

Definition 2.17. Types of literals. A literal l is **true** in a valuation v, denoted $v \models l$, iff $v(l) = \top$.

The literal l is **false** in v, denoted $v \models \bar{l}$, iff $v(\bar{l}) = \top$.

If l is neither false nor true in v then it is **undefined** in v. A literal is **defined** in v if it is either true or false in v.

A literal l for which its negation $\bar{l}$, is not true in the partial assignment $v1$ is called an **unfalsified literal**.

Definition 2.18. Types of clauses. A clause c is **true** in a valuation v, denoted $v \models c$, iff any of its literals is true in v.

Clause c is **false** or a **conflicting clause** in v, denoted $v \models \neg c$ iff all of its literals are false in v.

If c is neither true nor conflicting then it is **undefined**.

Definition 2.19. A formula F is **true** in a valuation v, denoted $v \models F$, iff all of its clauses are true in v. If this is the case then the trail M containing the valuation is a **model** for F.

F is **false** in v, denoted $v \models \neg F$, iff one of its clauses is conflicting in v.

Example 2.20. Suppose M is a trail and F a formula. Define

$$M := 1\ 2\ \overline{3}\ \overline{4} \text{ and } F := c_1 \wedge c_2 = (1 \vee \overline{2}) \wedge (3 \vee 4)$$

.

The clause c_1 is *true* in M because $1 \in M$. Moreover, c_2 is a *conflicting clause* in M, because $\{\overline{3}, \overline{4}\} \in F$.

If M were of the form [1 2 $\overline{3}$ 4], then both c_1 and c_2 would be *satisfied* in M and M would become a model for F, denoted $M \models F$.

Definition 2.21. SAT Basics Summary

Variable	natural number
Literal	either a positive variable (v) or a negated variable ($\overline{v}$)
Clause	of length s is a disjunction of s literals $l_1 \vee l_2 \vee ... \vee l_s$
Formula	of length t is a conjunction of t clauses $c_1 \wedge c_2 \wedge ... \wedge c_t$
Trail	A list of literals

2.2 Graph Theory

Graphs will play an important role in Chapter 4, for the evaluation purposes and so they need to be formally defined. Specifically, graphs will be used to visualize the structure of SAT instances, either by using SatGraf [6] or in order to determine a special type of clauses, called conflict clauses, which are used to resolve conflicts occurring in a formula.

Definition 2.22. Graphs. A graph is a finite, nonempty set V, the vertex set, along with a set E the edge set whose elements $e \in E$ are pairs $e = (a, b)$ with $a, b \in V$.

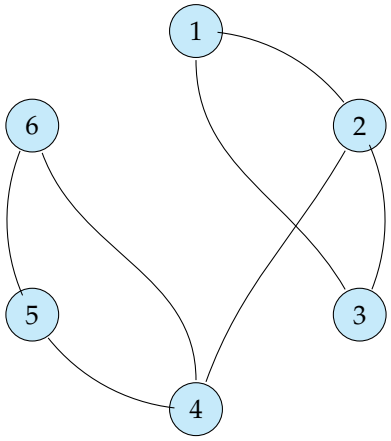

Figure 2.1: Variable Incidence Graph

Definition 2.23. Directed and Undirected graphs An undirected graph is a graph in which the edge set consists of unordered pairs. A directed graph is a graph in which the edge set consists of ordered pairs.

Definition 2.24. Variable Incidence Graph. Given a SAT instance, we can construct an undirected graph called the Variable Incidence Graph model (VIG for short), where vertices represent variables, and edges represent the existence of a clause relating two variables. A clause $l_1 \vee l_2 \vee ... \vee l_n$ results into $\binom{n}{2}$ edges, one for each pair of variables.

Example 2.25. The corresponding VIG for the clauses $(1, 2, 3), (4, 5, 6), (2, 4)$ is shown in Figure 2.1.

Note. A special type of directed graph is the *Implication Graph*, introduced in Section 3.2.5. Moreover, the VIG is used to present results using Satgraf in Section 4.2.

2.3 Sudoku

In this section, we explain the rules of the Sudoku game and some of its more important properties.

Definition 2.26. The game involves a 9x9 grid made up of 3x3 subgrids called "regions". In the beginning of the game various cells are filled up and the aim is to enter a digit from 1 to 9 in each cell of the grid so that each row, column and region contains only one instance of each digit.

Figure 2.2 represents a Sudoku puzzle on the left with a solution on the right.

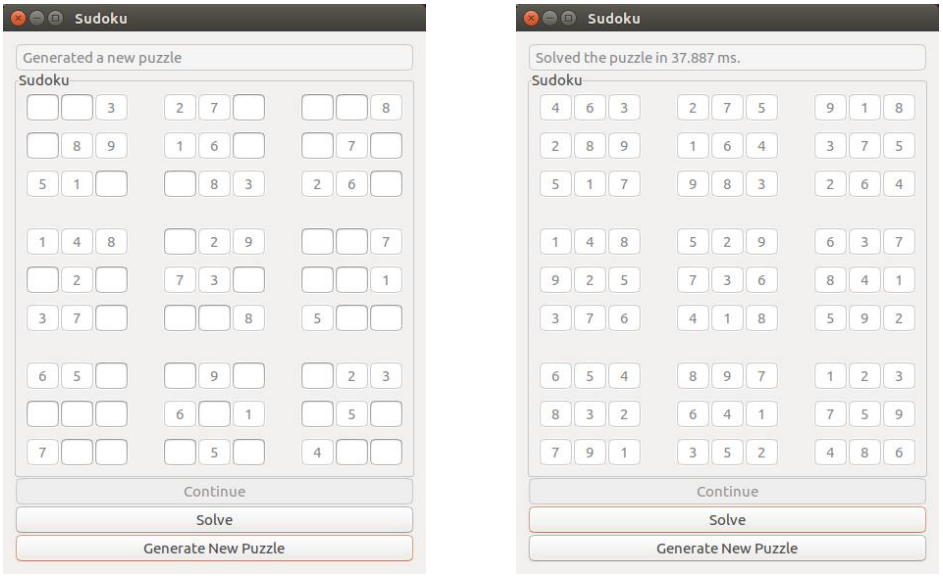

Figure 2.2: Solving a Sudoku puzzle.

The main properties of the game are the following:

- The puzzle contains only one solution.
- The puzzle can be solved mainly by reasoning, with no search.

The first property means that all the numbers that are preassigned are *necessary* assignments. The second property requires, that at any stage in the course of solving the puzzle, the non-empty cells can be merely identified by considering the set of non-blank entries.

Chapter 3

Implementation

This chapter outlines in greater detail the techniques used by the solver, as well as the main features of the Sudoku game.

In order to familiarize the reader with the general constructs of the DPLL algorithm, the recursive implementation is introduced first. Later, in successive increments, the main extensions of the CDCL algorithm are presented, including: the two watch literals scheme, the decision heuristics and finally first Unique Implication Point learning scheme. These techniques optimize the performance of the solver.

Furthermore, we revise some of the main components of the Sudoku interface including: the different encodings used and the most importantly, the Interactive Mode which allows the user to visualize the process through which the solution is found.

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

The approach used to implement the CDCL solver follows closely [3].

The original DPLL SAT solver uses a backtracking-based search algorithm, for deciding the satisfiability of formulas in CNF. The algorithm works by choosing a literal (called a decision literal), assigning it a truth value, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable, otherwise it must backtrack and assume the opposite truth value for the decided literal. When the formula contains an *empty clause*, then it is unsatisfiable. On the other hand, if the formula contains an *empty set of clauses* then it is satisfiable.

We now briefly introduce the main rules that a typical DPLL implementation can do:

- **Decision**: represents a case split. An undefined literal l is given a truth value and added to the trail M. This literal is annotated as a *decision literal*. If a conflict occurs then the opposite of l has to be considered and this is done by the Backtrack rule.
- **Unit Propagation**: this rule is applied whenever there exists a clause with all literals but one false in M. Whenever this happens then M must be extended to make l true.

- **Fail**. This is applied whenever there occurs a conflict and there are no decision literals in M. If this is the case then F is reported to be unsatisfiable.
- **Backtrack**: if a conflicting clause is detected and Fail does not apply then the rule backtracks one decision level, by replacing the most recent decision literal, l^d, by its negation. Note that $\bar{l}$ is not a decision literal.

The recursive implementation is shown in Algorithm 3.1. The algorithm has the benefit of being much easier to implement, while still containing the main rules presented above. However, it is more difficult to add extensions and for this reason we make the transition to the iterative implementation in Section 3.2.

Algorithm 3.1 Recursive DPLL

```
   Input: A formula F
   Output: Whether F is satisfiable or unsatisfiable
 1 F ← Propagate(F)
 2 if F contains empty set of clauses then return true
 3 if F contains □ then return false
 4 L = DecideLiteral(F)
 5 if DPLL(F ∪ {L}) = true then
 6   return true
 7 else
 8   return DPLL(F ∪ {L̄})
```

Example 3.1. Below is an example illustrating the main rules used by the solver.

$$
\begin{array}{lllllll}
\emptyset & \bar{1} \vee 2 & 3 \vee \bar{4} & 5 \vee 6 & 5 \vee \bar{6} \vee 3 & \Longrightarrow & (\mathit{Decide}: \bar{1}) \\
1^d & \cancel{\bar{1}} \vee 2 & 3 \vee \bar{4} & 5 \vee 6 & 5 \vee \bar{6} \vee 3 & \Longrightarrow & (\mathit{UP}: 2) \\
1^d 2 & \cancel{\bar{1} \vee 2} & 3 \vee \bar{4} & 5 \vee 6 & 5 \vee \bar{6} \vee 3 & \Longrightarrow & (\mathit{Decide}: \bar{3}) \\
1^d 2 \overline{3^d} & \cancel{\bar{1} \vee 2} & \cancel{3} \vee \bar{4} & 5 \vee 6 & 5 \vee \bar{6} \vee \cancel{3} & \Longrightarrow & (\mathit{UP}: \bar{4}) \\
1^d 2 \overline{3^d 4} & \cancel{\bar{1} \vee 2} & \cancel{3 \vee \bar{4}} & 5 \vee 6 & 5 \vee \bar{6} \vee \cancel{3} & \Longrightarrow & (\mathit{Decide}: \bar{5}) \\
1^d 2 \overline{3^d 4 5^d} & \cancel{\bar{1} \vee 2} & \cancel{3 \vee \bar{4}} & \cancel{5} \vee 6 & \cancel{5} \vee \bar{6} \vee \cancel{3} & \Longrightarrow & (\mathit{Backtrack}) \\
1^d 2 \overline{3^d} 45 & \cancel{\bar{1} \vee 2} & \cancel{3 \vee \bar{4}} & \cancel{5 \vee 6} & \cancel{5 \vee \bar{6} \vee 3} & \Longrightarrow &
\end{array}
$$

Figure 3.1: DPLL example using basic rules

3.2 Non-recursive DPLL: CDCL Algorithm

We describe the main phases of the algorithm. There are three main processes in the CDCL implementation as Figure 3.2 suggests. These are the unit propagate phase, the decision phase and the conflict resolution phase respectively.

During the unit propagation phase, the CDCL algorithm detects any unit clauses that arise and propagates them in a way that causes a minimal amount of overhead on the overall performance. This is the purpose of the two-watch literal scheme. It provides a lightweight mechanism for detecting and dealing with unit clauses and it is further described in Section 3.2.3.

There are two outcomes during the second phase: when a conflict does not occur and when one does occur. In the formal case, the problem is potentially satisfiable, if the number of variables in the trail is equal to the number of variables in the formula. If the two are not equal, then a decision is made and this process is further described in Section 3.2.4.

In the later case, when a conflict does occur, the formula might turn out to be unsatisfiable. This happens when a conflict occurs at decision level 0. If the decision level is greater than 0, then the conflict resolution process starts and a backjumping clause is generated, using the Unique Implication Point.The learning scheme is related to what kind of information is inferred from the backjumping clause. This process is further described in Section 3.2.5.

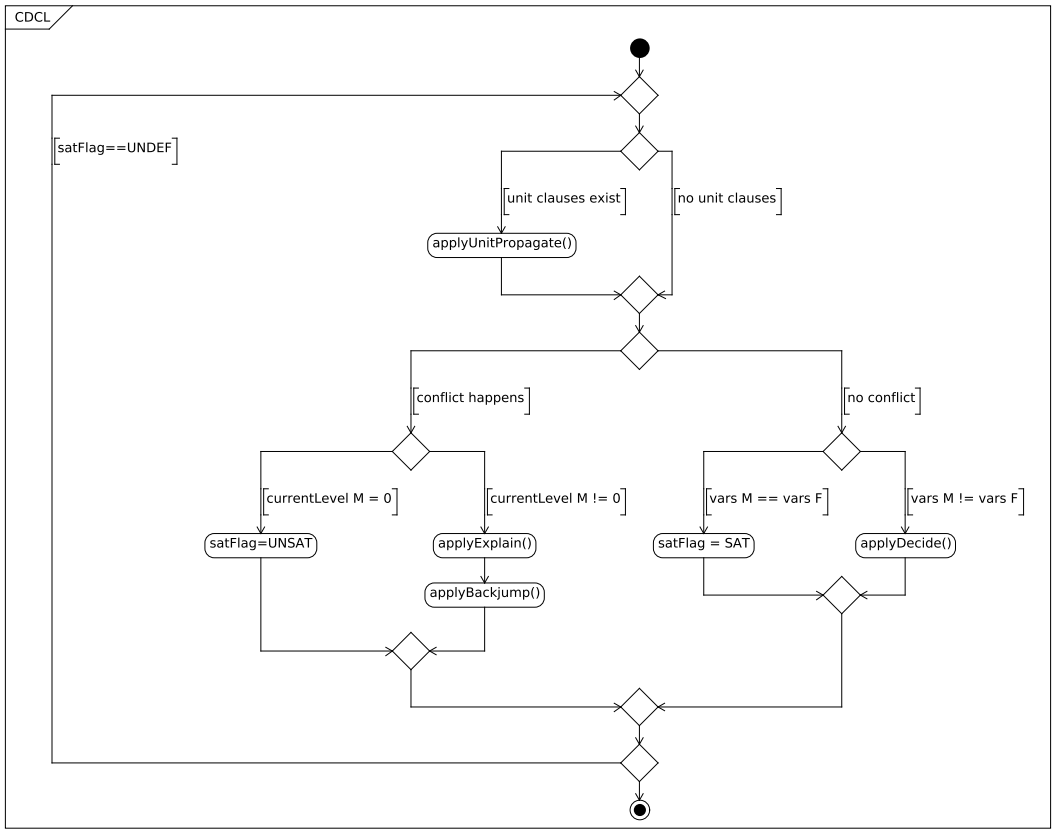

Figure 3.2: The main phases of the CDCL algorithm.

3.2.1 Application Programming Interface

In this subsection, we explore the external programming interface that can be used to specify and solve SAT-problems. The class diagram shown in Figure 3.3 reveals various classes and their relationship with respect to the main *Solver* class, used for solving the SAT instance. The approach for simplifying the initial formula and for implementing the decision heuristics follows the one explained in Minisat [5].

The classes: *Literal*, *Clause* and *Formula* are used to represent literals, clauses and formulas respectively. The *Vec* class is used to implement the trail, which keeps track

of the current partial assignment of the problem. It behaves in many respects like a stack and is similar to the *std::vector* class, found in the standard library. One similarity is that it stores an array that will dynamically readjust its size whenever a new element is added and its current size is equal to its maximum capacity [1]. Furthermore, the class supports push and pop operations which allow to add and remove elements from the top of stack. Its main advantage, over using the standard library, is that it allows to use specialized methods such as displaying the contents of the array and checking whether a literal is already satisfied by the trail.

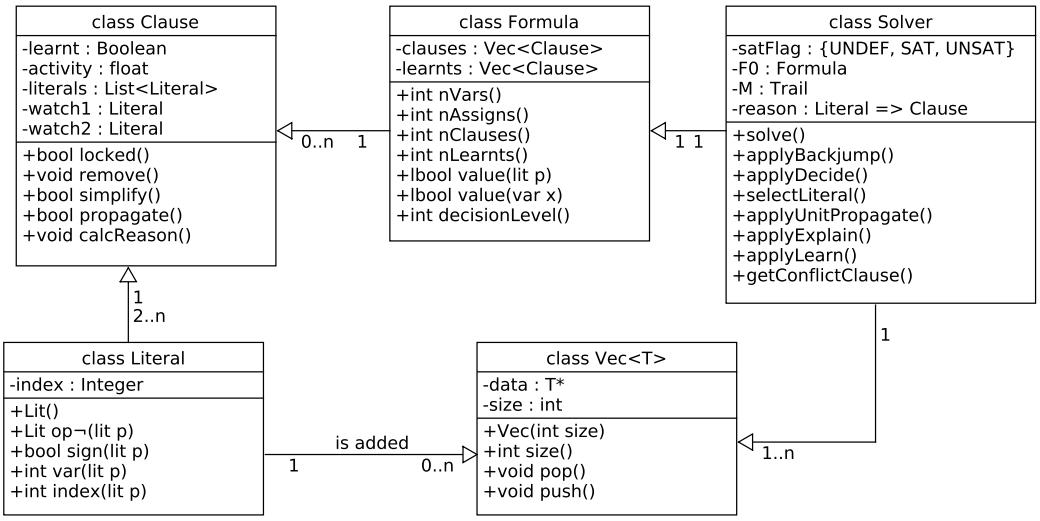

Figure 3.3: Application Programming Interface of the SAT solver.

3.2.2 CDCL Pseudocode

We start with an informal description of the solver and suggest the central ideas in Algorithm 3.2. Formula F is checked for satisfiability. The state of the solver is initially *UNDEFINED* and the current partial valuation is stored in trail, M. The variable *state* will be either *satisfiable* or *unsatisfiable* at the end of the algorithm.

If there exists a clause, say c, that is false in M, or $M \models \neg c$, we say that the solver is in a conflicting state and the flag $isConflicting$ is assigned to true. A conflict might arise whenever a new literal is added to the trail. If a conflict arises at decision level 0 then the formula is determined to be unsatisfiable. However, if the conflict arises at a decision level greater than 0 then a special literal called the Unique Implication Point has its value flipped and a backjumping clause is generated (both concepts are explained in Section 3.2.5). This case is handled by line 7 below.

The operation *decideLiteral* from line 12 is used to select a literal that is non currently assigned a truth value and push it to the trail (the decision heuristics are explained in Section 3.2.4). This procedure is usually referred to as the *decision* procedure. When a new decision is made, say literal d, a new copy of that literal is pushed to the trail and

[1]The capacity represents the size of the storage space currently allocated for the vector. To contrast this with the size of the vector which represents the number of elements currently stored.

the corresponding clauses in which d is a watch literal are updated (the Two-watch literal scheme is explained in Section 3.2.3). Any unit clauses resulted in this process are stored in a *propagation queue* and will be propagated once *exhaustiveUnitPropagate* is reached. The propagation queue is a simple queue that supports first in first out type operations.

Algorithm 3.2 Non-recursive DPLL

```
   Input: A formula F
   Output: Whether F is satisfiable or unsatisfiable
 1 while (state = UNDEFINED)
 2   exhaustiveUnitPropagate()
 3   if(isConflicting)
 4     if(decisionLevel() == 0)
 5       state = UNSAT
 6     else
 7       resolveConflicts()
 8   else
 9     if(trail.size() == F.vars())
10       state = SAT
11     else
12       decideLiteral()
```

When the number of variables in the trail is equal to the number of variables in F, then the formula is determined to be satisfiable.

The operation *exhaustiveUnitPropagate()* on line 2 is used to propagate the unit clauses accumulated in the propagation queue. Unit clauses may be generated when new literals are added to the trail. There are two ways that can stop the execution of the method. One way is when all the unit clauses have been propagated by the solver. The second way is when a conflict arises in-between the propagations and the solver must resolve the conflict first before continuing (which is related to the learning scheme from Section 3.2.5).

3.2.3 Two watch literal scheme

The scheme was first introduced in [9]. The number of variables can reach up to 1 or 2 million in many industrial problems and so it quickly becomes intractable to loop over all the clauses in the formula to check whether new unit clauses emerge. For this reason, there is a need to find unit clauses in a way that yields minimum overhead on the overall performance. For example, if a clause c contains N literals, we are interested in this clause only when the number of false literals become from $N-2$ to $N-1$. At this point, c becomes a unit clause and should be propagated.

The central idea behind the two watch literal scheme is illustrated in Figure 3.4 and can be described as it follows. Let C be the clause shown in the figure. Choose two literal for each clause, literals that are not defined in the trail, and use them as watch literals. In the beginning of the process none of the literals are false in the trail. Note that $\overline{V2}$ and $V3$ are the watch literals illustrated by the yellow box. After making literal $\overline{V3}$ true, the solver updates each clause where the negation of the propagated literal,

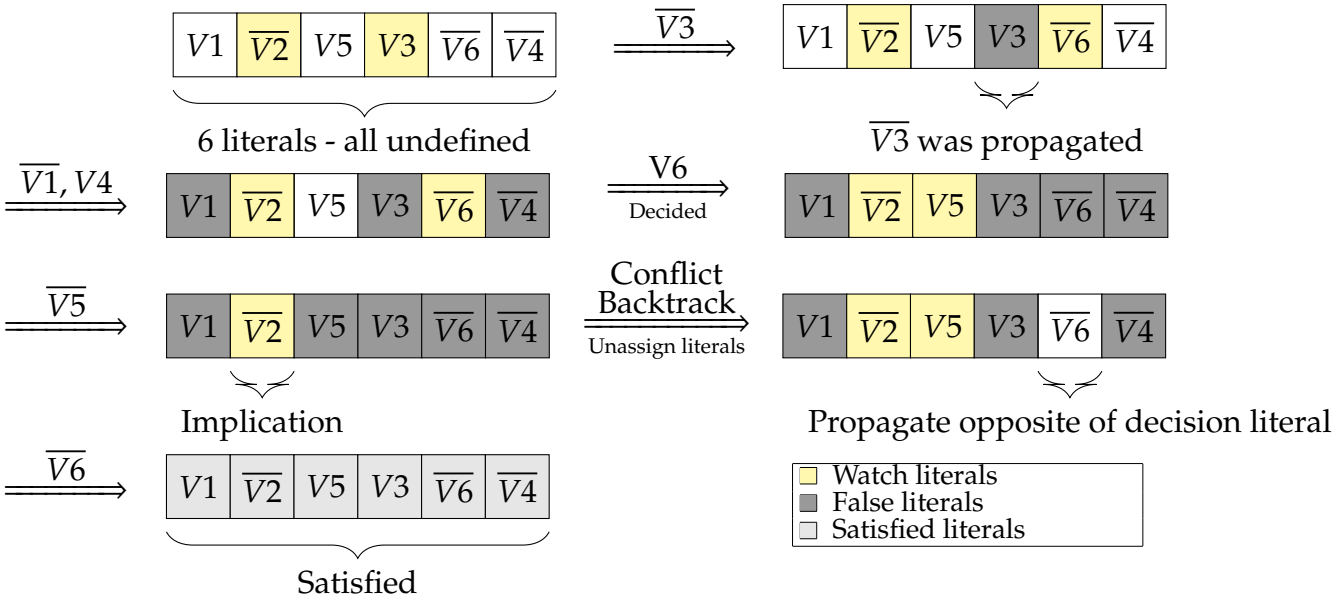

Figure 3.4: The Two Watch Literal scheme.

in this case $V3$, is used as a watch literal. Since the negation of one of the watches in C is propagated, C must be assigned a new watch literal before continuing. It is possible to find a new watch literal that is unfalsified in the trail, in this case choose $\overline{V6}$, and proceed.

In the third image, $\overline{V1}$ and $V4$ become true. Since none of these literals are watches for C, the solver does not check whether C becomes a unit clause. This is impossible since the two watch literals are still not assigned a truth value. Next, literal $V6$ is decided and the solver checks whether there exist any other literals that have undefined value. $V5$ becomes the new watch literal in the fourth figure.

The next action is to propagate $\overline{V5}$. At this point C becomes a unit clause because there do not exist two literals that are undefined in the trail and as a result the only undefined literal,$\overline{V2}$, is pushed to the propagation queue. However, suppose that by the time $\overline{V2}$ would be propagated, $V2$ would have already been propagated by an implication caused by another unit clause in the formula. Clause C would become a conflict clause and the conflict would need to be resolved before continuing. At least one decision level would be backtracked. The literal $V5$ becomes once again unassigned in the trail and is a suitable candidate to be a watch literal. The clause would be assigned two watch literals and the solver continues looking for a solution.

3.2.4 Two decision heuristics

Two decision heuristics are implemented, namely Variable State Independent Decaying Sum (VSIDS) and Largest Individual Sum (LIS). The formal performed better in our experiments and the implementation technique follows closely the one implemented in Minisat 2.2. [5].

The term VSIDS refers to a family of heuristics that are widely used in modern SAT solvers and rank the variables of a boolean formula according to their activity in

conflict clauses. The central idea is to collect information about the learnt clauses to guide the direction of the search, where more recent conflicts are favored. The core characteristics are described below:

- Each literal has a counter initialized to 0.
- Literals that occur in a conflict have the activity increased by a constant, called the *bump factor*. The literals in the conflicting clause as well as the literals in the resolvent are bumped. There are variations to this strategy.
- The unassigned literal with the highest counter is chosen at each decision.
- The bump factor is initialized with 1. Periodically it is multiplied by a constant (i.e. in our implementation 1.05). This results in the activity of literals that occur in more recent conflicts to be incremented by a larger number.

Notice that the last item has the effect that literals that occur in more recent conflicts are favored over literals that occur in older conflicts.

On the other hand, the LIS heuristic counts, at the beginning of the search, the number of clauses in which a given variable appears as a positive literal, C_P, and a negative literal, C_N. The values C_P and C_N are considered separately and we select the variable with largest individual value, and assign it to value true, if $C_P \geq C_N$ and value false, if $C_P \leq C_N$.

Example 3.2. Let $c_1 = (1, \overline{2}, \overline{3})$, $c_2 = (1, 3, \overline{4})$ and $c_3 = (1, 2, \overline{3})$. In this case: $C_p = (3, 1, 1, 0)$ and $C_n = (0, 1, 2, 1)$. Since the greatest individual value is assigned to $C_p(1) = 3$, literal 1 would made true.

The performance of the two implemented heuristics is evaluated in Section 4.1.2.

3.2.5 Unique implication points and Learning Schemes

In the following section, it remains to show how the backjumping clause is constructed in order to resolve a conflict. The following learning scheme is also used by MiniSAT [10]. We introduce another definition.

Definition 3.3. Suppose the implication graph (which is defined below) of a conflict is given. An *Unique Implication Point* (UIP) represents a node N from the current decision level such that any path from the decision literal to the conflict node must pass through N.

Remember that a conflict clause, C, has the property that $M \models \neg C$, or all the literals of C are false in the trail M. It is intuitive to think that the way of obtaining the backjumping clause is through a series of manipulations to the conflict clause, in a way that could relate the literals from the current decision level, with the literals from earlier decision levels that had a role in the conflict. By doing this we could 'resolve' the conflict by changing the value of one literal, from the current decision level, that caused the conflict in the first place. This special literal is called an Unique Implication Point (UIP).

Example 3.4. Another, perhaps more useful way of thinking about this process is through an implication graph. Consider formula F where, among other clauses, it contains:

$$(\overline{1}, \overline{8}) \quad (8, \overline{1}, 4) \quad (\overline{2}, 3) \quad (5, \overline{4}, 7) \quad (\overline{7}, 9) \quad (\overline{7}, \overline{9}, \overline{6}) \quad (\overline{4}, \overline{7}, 6, \overline{9})$$

among with the trail M which contains: $...1^d\ \overline{8}\ 4\ 2^d\ 3\ \overline{5^d}\ 7\ 9\ \overline{6}$.

The final state in M can be reached by a series of decisions and propagations as illustrated in Figure 3.5. This type of graph is called an *Implication Graph* and is created as follows.

The nodes that correspond to the conflicting clause are shown in gray, while the node that appears in yellow is the node that was next to be propagated if it were not for the conflict. The graph was created by following the path from the conflict node back to the decision literal.

The first line in the figure corresponds to the literals that belong to the decision level 3. Further on, by application of the decision rule and a propagation, the literals from decision level 2 are drawn. Finally, the literals corresponding to the decision level 1 are drawn. For example, literal $\overline{8}$ and 4 are propagated due to the *reason clauses* $(\overline{1}, \overline{8})$ and $(8, \overline{1}, 4)$ respectively. Moreover, literal 3 is propagated due to the reason clause $(\overline{2}, 3)$. The remaining literals are propagated in a similar way. Due to the propagation of literal $\overline{6}$, clause $C := (\overline{4}, \overline{7}, 6, \overline{9})$ becomes a conflict clause.

A typical CDCL implementation would keep track of the reason clauses for each propagated literal. Using the saved information one could inquire the UIP and by extension the backjumping clause which are used to resolve the conflict. For instance, the solver can reason that the propagation of $\overline{6}$ was implied by 7 and 9 due to the reason clause $(\overline{7}, \overline{9}, \overline{6})$. Similarly, the propagation of 9 was implied by 7 due to $(\overline{7}, 9)$.

Both 7 and $\overline{5}$ are UIPs as both satisfy Definition 3.3 (in fact the decision literal is always an UIP, however there might exist more than one). The idea is to choose the UIP as close to the conflict as possible, the notion behind which is that hopefully the knowledge gained is as relevant to the conflict as possible.

Notice that the literals 2 and 3 that belong to decision level 2 do not play a role in the conflict. Therefore, exploring the computation when the truth value of literal 2 is flipped would not matter for the purpose of finding the solution to the problem. The CDCL implementation is able to make this observation, due to how the backjumping clause is constructed. However, the classical DPLL algorithm is not equipped to make this kind of judgement and would waste computation time exploring this redundant branch.

In practice however, it is not necessary to find the UIP by building the implication graph, because the UIP arises naturally in the process when we substitute each literal from the conflicting clause, by the set of literals from the reason clause that implied the propagation.

This process is called the *backwards conflict resolution process* and is carried out the following way. In our example, the conflicting clause is $(\overline{4}, \overline{7}, 6, \overline{9})$. By two successive substitutions one can derive the backjumping clause $(4, \overline{7})$, by resolving away the literals 6 and $\overline{9}$ with the respective reason clauses that caused the propagations. The

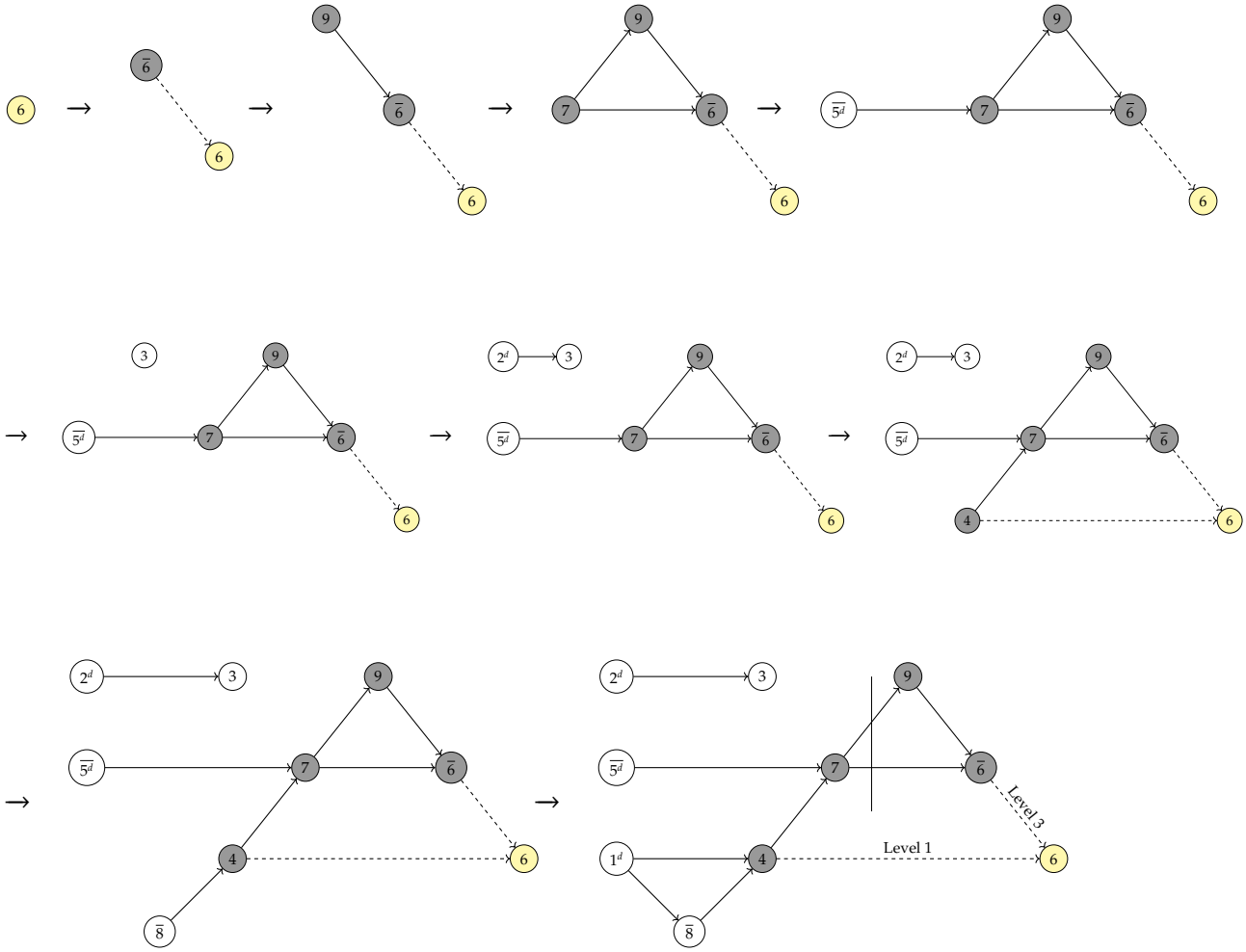

Figure 3.5: Implication graph for Example 3.4

$$\dfrac{\dfrac{\overline{7} \vee \overline{9} \vee \overline{\mathbf{6}} \qquad \overline{4} \vee \overline{7} \vee \mathbf{6} \vee \overline{9}}{\overline{7} \vee \mathbf{9} \qquad \overline{4} \vee \overline{7} \vee \overline{\mathbf{9}}}}{\overline{4} \vee \overline{7}}$$

Figure 3.6: Conflict resolution process to derive Backjumping Clause

derivation is shown in Figure 3.6 with the literal that is substituted shown in bold. On the right-hand side is the conflicting clause together with the resulted clauses after the substitution and on the left-hand side are the reason clauses.

3.3 Sudoku implementation

This section outlines the implementation decisions for the Sudoku GUI. It describes the Sudoku encodings into SAT and most importantly the interaction between the SAT Solver and the Sudoku game to display backtracking events, implied and decided literals in interactive mode. Due to the hint functionality being a trivial extension of actually solving the Sudoku puzzle, it is not covered in this report.

3.3.1 The Interaction between Sudoku and the SAT Solver

The interaction between the SAT solver and the Sudoku GUI is illustrated in Figure 3.7 and can be described as follows:

- User requests a new puzzle. The puzzle is read from the disk and is returned to the main class as an array of integers.
- Main class sends a solving request to the SAT solver. The previously generated array is encoded into SAT and a file is written on the disk.
- The SAT solver reads the encoded formula and attempts to solve the problem. The result is returned to the main class.
- If the result is satisfiable then the formula is decoded back into an array of integers and the GUI is updated with the result.
- If the result is unsatisfiable the GUI is updated with the appropriate message.

3.3.2 Extended and Minimal encodings

The encodings used in this project follow closely [11]. The Sudoku rules that are encoded into SAT are related to the cells, columns, rows, blocks and the pre-assigned numbers as illustrated in Figure 3.8. The total number of propositional variables is $9 \times 9 \times 9 = 729$. Each entry is associated 9 variables and are referred to using a triple of the form $v_{(r,c,v)}$. A variable of the form $v_{(r,c,v)}$ is true if and only if the entry in row r and column c is equal to v (i.e. $[r, c] = v$). Similarly $\neg v_{(r,c,v)}$ is true if and only if $[r, c] \neq v$.

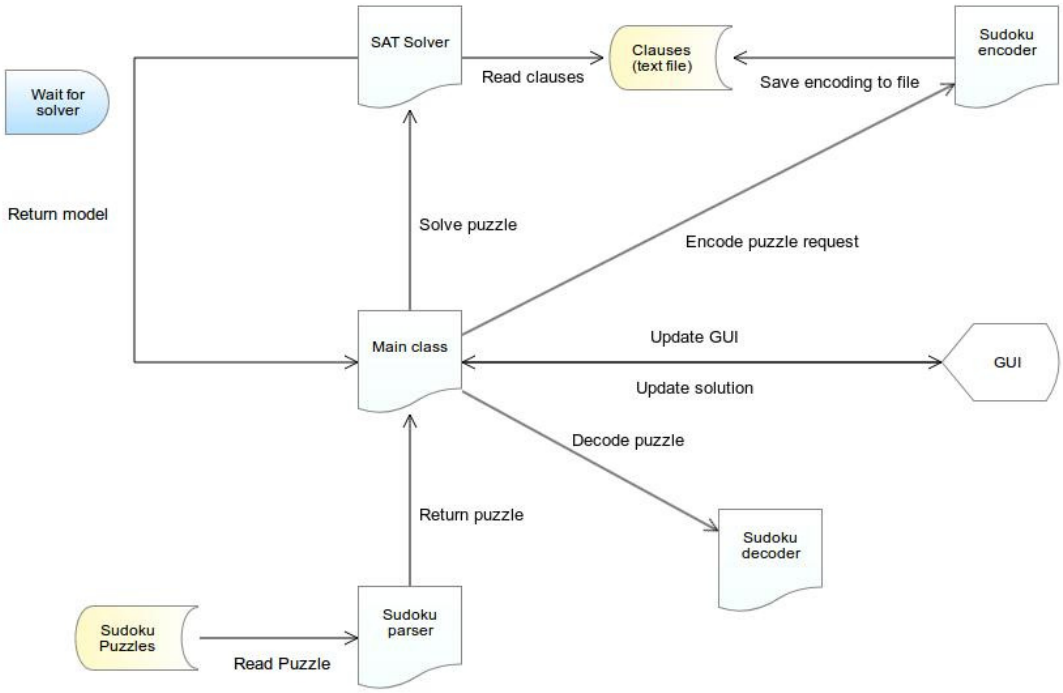

Figure 3.7: The interaction SAT Solver-Sudoku.

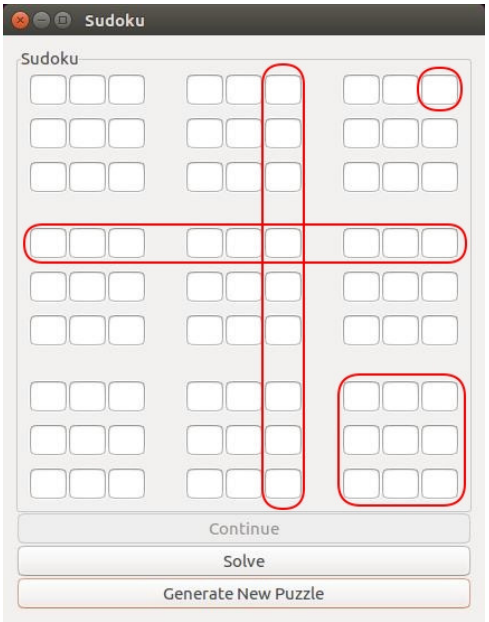

Figure 3.8: Related encodings into SAT

The subindices in the following rules signify uniqueness and definedness respectively. The minimal encoding is defined as follows:

- At least one number appears in each cell:
  $Cell_d = \wedge_{r=1}^{9} \wedge_{c=1}^{9} \vee_{v=1}^{9} (r, c, v)$
- At most one number appears in each row:
  $Row_u = \wedge_{r=1}^{9} \wedge_{v=1}^{9} \wedge_{c_i=1}^{8} \wedge_{c_j=c_i+1}^{9} \neg(r, c_i, v) \vee \neg(r, c_j, v)$
- At most one number appears in each column:
  $Col_u = \wedge_{c=1}^{9} \wedge_{v=1}^{9} \wedge_{r_i=1}^{8} \wedge_{r_j=r_i+1}^{9} \neg(r_i, c, v) \vee \neg(r_j, c, v)$
- At most one number appears in each 3x3 sub-grid:
  $Block_d = \wedge_{r_{offs}=1}^{3} \wedge_{c_{offs}=1}^{3} \wedge_{v=1}^{9} \wedge_{r=1}^{9} \vee_{c=r+1}^{9}$
  $\neg(r_{offs} * 3 + (\text{r mod } 3), c_{offs} * 3 + (\text{r mod } 3), v) \vee$
  $\neg(r_{offs} * 3 + (\text{c mod } 3), c_{offs} * 3 + (\text{c mod } 3), v)$
- A pre-assigned fact is represented as a unit clause:
  $Assigned = \wedge_{i=1}^{k} \{(r, c, a) \mid \exists_{1 \leq a \leq 9}[r, c] = a\}$

There are nine-ary and binary clauses that are produced by these rules. The nine-ary clauses are produced by the at-least-one rules and the binary clauses are produced by the at-most-one rules.

The extended encoding adds more constraints to the formula that explicitly assert that each entry in the grid contains *at most* one number. Similarly, each row, column and 3x3 block must contain *at least* one number. By adding additional constraints, the solver can predict ahead of time whether a conflict will occur. The reason for this is that a clause that asserts definedness for a row might become false before the corresponding clause that asserts uniqueness for the same row.

- At most one number appears in each cell:
  $Cell_u = \wedge_{r=1}^{9} \wedge_{c=1}^{9} \wedge_{v_i=1}^{8} \wedge_{v_j=v_i+1}^{9} \neg(r, c, v_i) \vee \neg(r, c, v_j)$
- At least one number appears in each row:
  $Row_d = \wedge_{r=1}^{9} \wedge_{v=1}^{9} \vee_{c=1}^{9} (r, c, v)$
- At least one number appears in each column:
  $Col_d = \wedge_{c=1}^{9} \wedge_{v=1}^{9} \vee_{r=1}^{9} (r, c, v)$
- At least one number appears in each square:
  $Block_d = \wedge_{r_{offs}=1}^{3} \wedge_{c_{offs}=1}^{3} \wedge_{v=1}^{9} \vee_{r=1}^{3} \vee_{c=1}^{3} (r_{offs} * 3 + r, c_{offs} * 3 + c, v)$

The Minimal Encoding is composed of the following rules:

$$Cell_d \cup Row_u \cup Col_u \cup Block_u \cup Assigned.$$

Similarly, the rules for the extended encoding are:

$$Cell_d \cup Cell_u \cup Row_d \cup Row_u \cup Col_d \cup Col_u \cup Block_d \cup Block_u \cup Assigned.$$

The key difference between the two encodings is that the minimal encoding is a subset of the extended one.

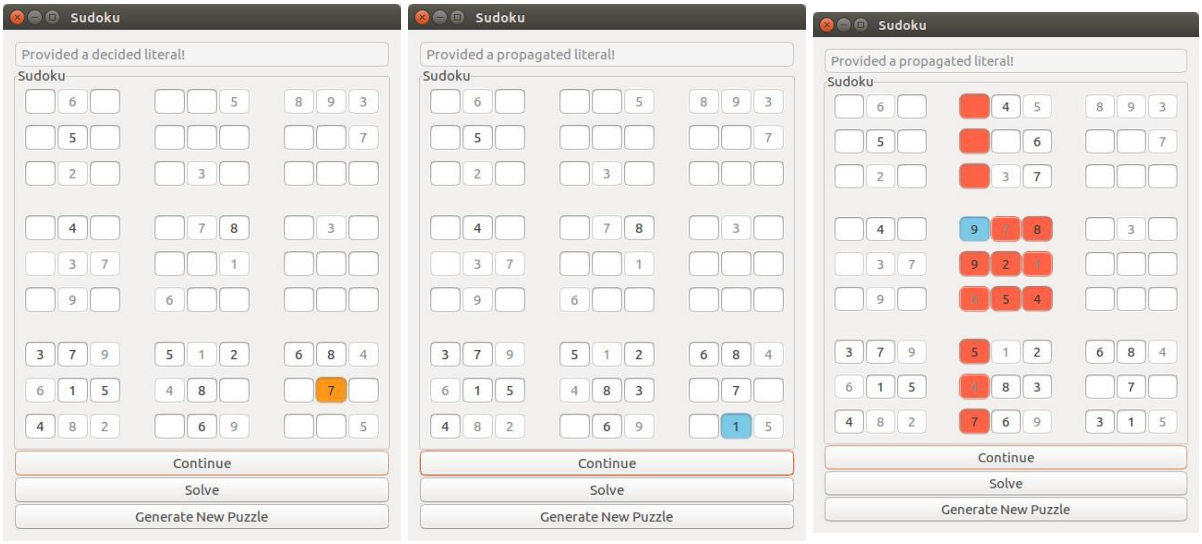

Figure 3.9: Decided, Propagated and Conflict Literals in interactive mode.

Decoding the output of the solver

When the formula is satisfiable, the output of the solver is a list of 999 variables between -999 and 999 not including 0. Of all these variables only some are relevant and should be translated back into a format that can be used by the Sudoku game.

Since the variables were encoded in the form (r, c, v), such a literal is valid and should be translated back if and only if

$$(r, c, v) \text{ is a positive natural number with digits between 1 and 9.}$$

A variable that is negative should not be translated back as it holds the meaning that a number should not be placed in a specific entry. For example, $-(9, 6, 8)$ suggests that row 9, column 6 should not be assigned the value 8, while $(7, 3, 4)$ suggests that row 7, column 3 should be assigned value 4.

3.3.3 Interactive mode

Interactive mode allows the user to visualize the different steps followed by the solver in order to generate the solution. For example, one can see when literals are decided, propagated and when conflicts occur what happens during the backwards conflict resolution process to generate the backjumping clause. Each action is assigned a special color. The process is illustrated in Figure 3.9.

The feature was implemented using multiple threads. The GUI main thread receives a signal from the main class to create a worker thread that computes the solution to the problem. Whenever a valid literal is decided or propagated, a signal is sent back to the main thread that updates the GUI with the correct value.

The central ideas are illustrated in Algorithm 3.3. The CDCL Procedure is the second thread that signals the main thread to update the GUI at appropriate times.

Algorithm 3.3 Interactive Mode

```
1 Solver solver
2 worker = thread(&Solver::solve_problem, solver)
3
4 {
5   ready = true
6   print ‘Signal data is ready to be processed‘
7 }
8 cv.notify_one() // signal the worker to start
9 ...
10 if processed
11   worker.join()
```

Algorithm 3.4 CDCL Procedure

```
1 unique_lock<mutex> lock(m)
2 cv.wait(lock, []{return ready;}) // wait here
3
4 resolveProblem() // among other things update GUI
5 processed = true
6 print ‘Data has been processed‘
```

The Interactive Mode procedure is the one that joins both threads together so that the application can continue execution, once all the work is finished.

Chapter 4

Evaluation

In this section we explore ways to optimize the performance of the solver from three different perspectives. Firstly, we show the profiling results and suggest areas that could be optimized. Moreover, successful ways to optimize performance are presented. Secondly, we introduce the idea of a community graph for a SAT instance, which is used later in Section 4.3 to understand why the VSIDS solver did not perform as well as we had expected. Thirdly, profiling results for the Sudoku problems are shown in Section 4.4.

4.1 Performance analysis

In this section we analyse the performance of the SAT Solver using Flamegraphs and Callgraphs. Moreover, effective optimizations are later introduced. Unit tests are also created to ensure that no bugs are occur as we add more functionality. A portion of the created unit tests can be seen in the Appendix.

4.1.1 Bottlenecks of the VSIDS heuristic

Flame Graphs can be used to determine which areas of an application should be focused in order to improve performance. Consider the graph shown in Figure 4.1 which analyzes performance of the VSIDS solver. The x axis represents the stack methods sorted in alphabetical order, while on the y axis there is the stack depth. In this graph, it is possible to see which methods are consuming most CPU cycles by looking at the large areas at the bottom and finding the widest sample. Then, following the corresponding samples to the top it is possible to find independent methods that can be more easily optimized. Widths are proportional to presence in samples and so a wider length represents an area more utilized by the CPU.

It becomes clear that the area that is utilized most intensely by the CPU is *applyExplainUIP*. Moreover, we see that the solver spends most of the running time resolving conflicts and in particular updating the activity queue which is used to rank literals. One area to optimize is related to the implementation details for the solver. Specifically, instead of using the standard library, one could implement specialized data structures, that apply less overhead, like the ones used in Minisat [5].

Figure 4.1: Flame graph illustrating the VSIDS heuristic on a hard SAT instance

4.1.2 Call graph analysis

The two decisions heuristics are compared side by side to compare running time in a difficult random SAT instance, process which is illustrated in Fig.4.2. The LIS heuristic is shown in the left side of the tree and VSIDS on the right side. There is a significant performance gap between the two, LIS spending most of its running time in the backjumping phase, while VSIDS spends most time while trying to find the UIP. This suggests that the bottleneck for the first heuristic is related to the implementation technique and could be optimized in the future. However, for VSIDS most computation time is spent during the conflict resolution process and this is largely what we'd expect. Therefore, possible optimizations would be more difficult to apply effectively.

4.1.3 Attempts to optimize performance

In light of the above results, we tried to optimize the performance of both heuristics. The experimental outcome is shown in Table 4.1. The problems are randomly generated using Gorilla [8]. The approach is to allow a maximum computation time of 20 seconds for each problem, after which if a solution is not found the solver times out and the result is considered unknown.

Stats	VSIDS	LIS
Satisfiable	61 problems	57 problems
Average	1.2213s	3.503s
Unsatisfiable	68 problems	52 problems
Average	2.312s	3.833s
Unknown	2 problems	22 problems
Total SAT	61 problems	
Total UNSAT	70 problems	

Table 4.1: Benchmarks results for the two heuristics

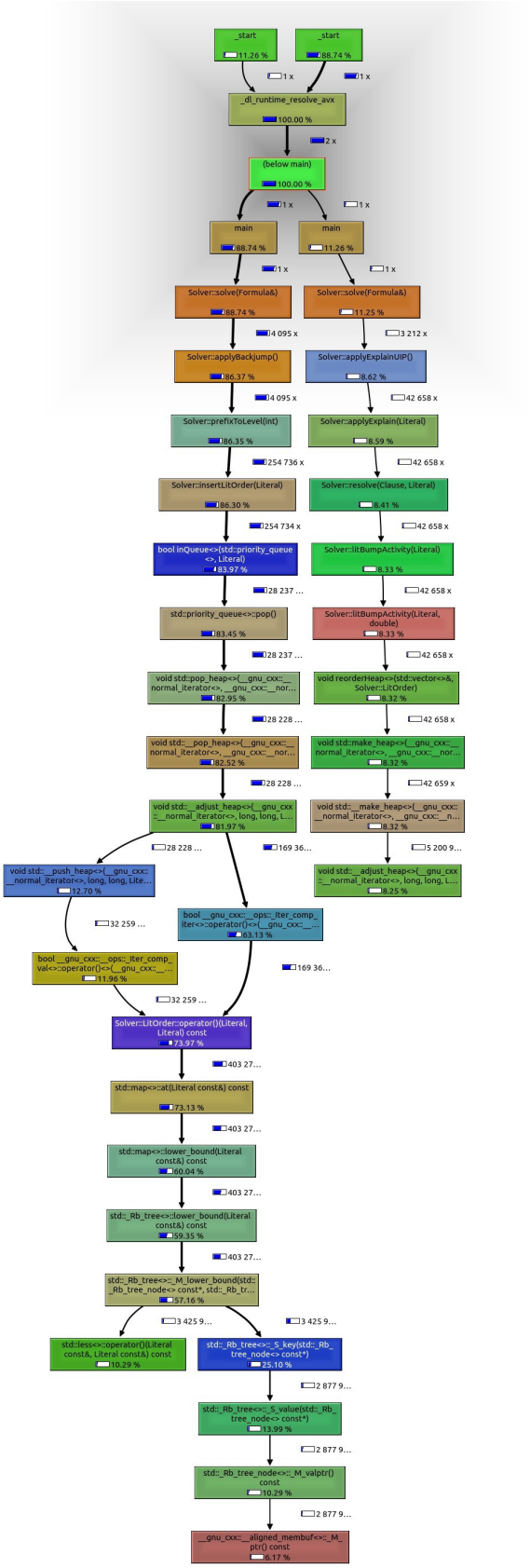

Figure 4.2: Call graph with the running time percentage of each function

131 Hard SAT Problems			
Time	Initial	Improved	Optimized
Real	14m 58.739s	14m 0.884s	4m 43.623s
User	14m 58.380s	14m 0.568s	4m 42.763s
System	0m 0.256s	0m 0.235s	0m 0.724s

Table 4.2: VSIDS solving times for 131 hard problems

Out of 131 hard problems evaluated,the VSIDS solver times out for 2 of them. However, the LIS solver is significantly less efficient timing out for 22 problems.

Table 4.2 shows the resulting solving times after certain types of optimization attempts for the VSIDS heuristic.

The difference between two variations shown in the *Initial* and *Improved* columns is related to how the activity of literals is bumped and decayed. The second column involves the heuristic explained in Section 3.2.4. The Initial version is less efficient by using a different literal decay function. Instead of increasing the bump factor each time the decay function is called, the solver would loop over all the literals in the formula to reduce their activity. This results in more overhead and the solving time is less efficient by approximately one minute. The Initial version is explained in [5], however the improved version is used in the current implementation of Minisat 2.2.

The third optimization involves compiler related optimizations, namely compiling with g++ -O4 flag. This yields the most significant performance increase overall, of almost 10 minutes.

4.2 VSIDS Heuristic and Communities

Many experiments have been conducted on the effectiveness of different branching heuristics [12] and it is generally agreed that VSIDS is one of the more effective ones. The VSIDS strategy is highly coupled with the learning procedure. Because the activity (priority) queue is modified only when a conflict occurs, it has the advantage of incurring low overhead. The following definitions are useful in reasoning about the performance of the VSIDS heuristic introduced in this section.

Definition 4.1. Community Structure. Let G be a VIG. G is said to exhibit community structure if it is possible to partition G into groups of vertices such that each group has more internal edges than outgoing edges.

Definition 4.2. Bridge Variables. A bridge variable is a variable that connects distinct communities together.

Recent research suggests that the VSIDS heuristic can take advantage of good community structure in SAT instances leading to faster solving time [13]. One explanation for this is that the solver can focus its attention on the bridge variables and by assigning them appropriate values this can lead to divide-and-conquer sort of strategies, so the communities can be tackled in turn. Once enough bridge variables are assigned the correct values the remaining connected components can be solved without interference

from each other. Even if the graph cannot be completely separated, it may prove useful to cut the bridge variables because the remaining communities can be solved almost independently.

The program used to visualize the structure of various communities is called SAT-Graf [6]. In the following figure, the image on the left is a random problem generated using Gorilla [8] and the one on the right is a Sudoku instance.

The randomly generated problem does not have a good community structure, as it is illustrated by the white edges. The vertices connecting the white edges represent bridge variables that link distinct communities together, namely the purple and the green communities. However, these communities do not have a high number of inter-community edges and as a consequence the VSIDS heuristic is not able to exploit any special properties of the problem.

On the other hand, on the right is shown the Sudoku problem. It quickly becomes apparent the green community that contains a high number of inter-community edges and the various symmetries in the problem are clearly visible in the VIG. As a result, we expect the VSIDS to exploit such properties and to solve the problem efficiently.

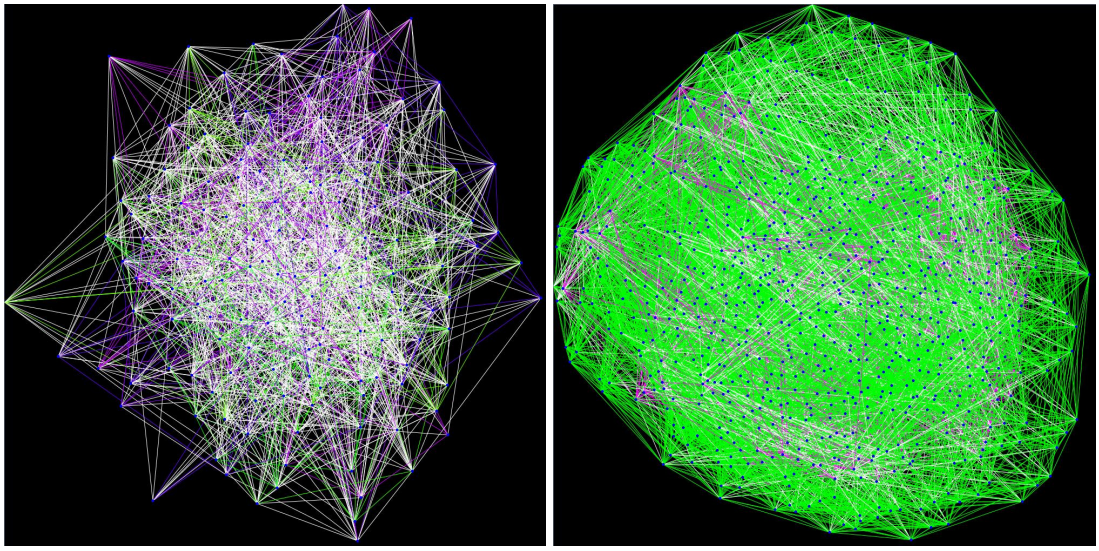

Figure 4.3: Random Hard-SAT instance and Sudoku community graphs.

4.3 DLIS vs VSIDS performance

More experiments are made to evaluate both decision heuristics against a set of medium and hard SAT problems.

In Figure 4.4, 106 SAT difficult SAT problems evaluated. The maximum time allowed to compute a problem is 20 seconds after which the solver would time out and the result would be considered unknown. The VSIDS heuristic solves significantly more problems, however it does on average more decisions than the LIS solver. Partly, the reason for this is because the problems were randomly generated and therefore **lack good community structure**.

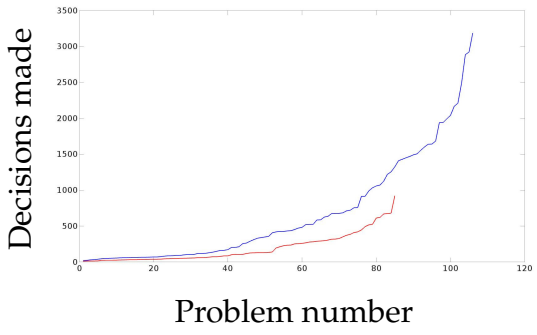

Figure 4.4: Hard 106 SAT problems

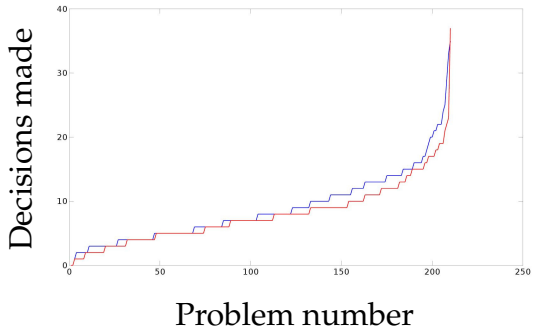

Figure 4.5: Medium 210 SAT problems

In Figure 4.5, the medium instances were solved in a matter of seconds by both solvers. However, VSIDS still makes more decisions and the reason for this is related to the community structure as well. Since they are randomly generated, VSIDS is unable to exploit some of the symmetries which are exhibited for instance, by Sudoku problems.

4.4 Minimal against Extended encoding

The minimal encoding constraints are a subset of the extended encoding constraints and in this section we analyse how significant is the performance gap between the two. The results are shown in Fig. 4.6.

Sudoku problems have better community structure as illustrated in Fig.4.3 which means that there exist symmetries that the VSIDS solver can exploit to find the solution in a shorter time.

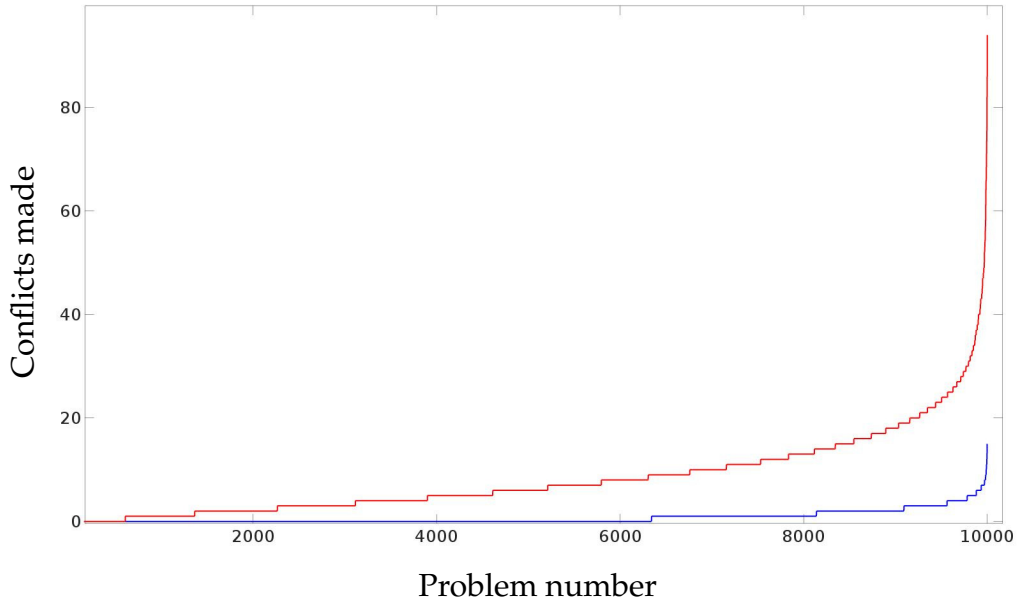

Figure 4.6: Minimal versus Extended encodings

The computation time for both type of problems is in the order of milliseconds and for this reason a more meaningful way of evaluating the performance has to be used. The decision was made to evaluate instead, in terms of the number of conflicts made.

The problem set consists of 10000 difficult Sudoku puzzles, each containing 25 preassigned entries and the VSIDS heuristic is used. One important property of the puzzles is preserved across all instances, namely the uniqueness of solution.

The performance gap turns out to be significant, as the maximum number of conflicts made for the Minimal Encoding reaches almost 100 conflicts for the most difficult problems. The Extended Encoding is significantly more efficient as the maximum number of conflicts reached is 20 for the most difficult problems.

Due to the solver being able to find the solution so quickly, the performance gap is best visualized in *Interactive Mode*. By using the interactive mode, it is possible to see the difference between the two encodings and so to predict when conflicts are about to occur in the problem.

Chapter 5

Conclusion

In this report, we described some of the techniques that many well known SAT solvers are currently using. Also, we explored the structure of random SAT and Sudoku instances and found ways to optimize performance. Lastly, an algorithm was implemented that allows to visualize the process through which a Sudoku solution is found.

5.1 Personal development

Overall, the project was a fantastic learning experience. It provided the opportunity to become more experienced with C++ and allowed me to study more about a subject I am deeply interested in, namely Logic and its Applications. Moreover, it was a good experience to learn how a significant piece of software can be built over the course of one year.

5.2 Future applications

The tool could potentially be used by anyone interested in logic, to gain knowledge of how the techniques in the CDCL algorithm work. The solver works both independently and in conjunction with the Sudoku application. Used by itself, it can give solutions to many difficult SAT problems and used together, one could investigate the output of the solver in *Interactive Mode* and do further analysis on the properties exhibited by a specific problem.

We note some improvements that that could enhance the performance. The solver does not support a *restart* scheme which would allow to restart the search whenever it is not doing enough progress according to some measure. This has been proved this to be an important factor to consider in many competition and industrial problems [14]. Also, Minisat is able to solve the experiments much quicker, part of the reason being the specialized data structures which are used, instead of relying on the standard library. This would be a good area to optimize in the future.

Moreover, other enhancements include being able to expand the Sudoku grid and to allow puzzles of different sizes. Also, the Interactive Mode could be improved to allow more features such as being able to go backwards in the conflict analysis process.

Appendices

A Unit tests

```
[ RUN      ] Clause.doesContain
[       OK ] Clause.doesContain (0 ms)
[ RUN      ] Clause.doesNotContain
[       OK ] Clause.doesNotContain (0 ms)
[ RUN      ] Clause.removeLiteral
[       OK ] Clause.removeLiteral (0 ms)
[ RUN      ] Clause.addLiteral
[       OK ] Clause.addLiteral (0 ms)
[ RUN      ] Clause.getNLiterals
[       OK ] Clause.getNLiterals (0 ms)
[ RUN      ] Clause.clear
[       OK ] Clause.clear (0 ms)
[ RUN      ] Clause.containsLiteral
[       OK ] Clause.containsLiteral (0 ms)
[ RUN      ] Clause.doesNotcontainLiteral
[       OK ] Clause.doesNotcontainLiteral (0 ms)
[ RUN      ] Clause.removeLiteralNonExistantLit
[       OK ] Clause.removeLiteralNonExistantLit (0 ms)
[ RUN      ] Clause.getWatch1
[       OK ] Clause.getWatch1 (0 ms)
[ RUN      ] Clause.getWatch2
[       OK ] Clause.getWatch2 (0 ms)
[ RUN      ] Clause.getNonsetWatch1
[       OK ] Clause.getNonsetWatch1 (0 ms)
[ RUN      ] Clause.getNonsetWatch2
[       OK ] Clause.getNonsetWatch2 (0 ms)
[----------] 13 tests from Clause (0 ms total)

[----------] 6 tests from ClauseTest
[ RUN      ] ClauseTest.getWatch1Pos
[       OK ] ClauseTest.getWatch1Pos (0 ms)
[ RUN      ] ClauseTest.getWatch1PosT2
[       OK ] ClauseTest.getWatch1PosT2 (0 ms)
[ RUN      ] ClauseTest.getWatch2Pos
[       OK ] ClauseTest.getWatch2Pos (0 ms)
[ RUN      ] ClauseTest.getWatch2PosT2
[       OK ] ClauseTest.getWatch2PosT2 (0 ms)
[ RUN      ] ClauseTest.switchWatch1Pos
[       OK ] ClauseTest.switchWatch1Pos (0 ms)
[ RUN      ] ClauseTest.switchWatch2Pos
[       OK ] ClauseTest.switchWatch2Pos (0 ms)
[----------] 6 tests from ClauseTest (0 ms total)

[----------] 3 tests from Formula
[ RUN      ] Formula.setNVars
[       OK ] Formula.setNVars (0 ms)
[ RUN      ] Formula.setNClauses
[       OK ] Formula.setNClauses (0 ms)
[ RUN      ] Formula.getClauses
[       OK ] Formula.getClauses (0 ms)
[----------] 3 tests from Formula (0 ms total)

[----------] 2 tests from Vec
[ RUN      ] Vec.contains
[       OK ] Vec.contains (0 ms)
[ RUN      ] Vec.doesNotContain
[       OK ] Vec.doesNotContain (0 ms)
[----------] 2 tests from Vec (0 ms total)

[----------] Global test environment tear-down
[==========] 32 tests from 5 test cases ran. (1 ms total)
[  PASSED  ] 32 tests.
```

Bibliography

[1] S. A. Cook, "The complexity of theorem-proving procedures," *Journal of Automated Reasoning*, pp. 151–158, 1971.

[2] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation = lazy clause generation," in *Principles and Practice of Constraint Programming – CP 2007* (C. Bessière, ed.), (Berlin, Heidelberg), pp. 544–558, Springer Berlin Heidelberg, 2007.

[3] F. Maric, "Formalization and implementation of modern sat solvers," *Journal of Automated Reasoning*, vol. 43, pp. 81–119, 2009.

[4] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: from an abstract davis-putman-logemann-loveland procedure to dpll(t)," *Journal of ACM (JACM)*, vol. 53, pp. 937–977, 2006.

[5] N. Eèn and N. Sórensson, "An extensible sat-solver," *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518, 2003.

[6] Z. Newsham, W. Lindsay, V. Ganesh, J. H. Liang, S. Fischmeister, and K. Czarnecki, "Satgraf: Visualizing the evolution of sat formula structure in solvers," in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 62–70, Springer International Publishing, 2015.

[7] R. Vasile, "Solving sudoku using propositional logic." `https://gitlab.cs.man.ac.uk/razvan.vasile/solving_sudoku_using_propositional_logic`, 2018. Accessed: 2018-05-01.

[8] K. Korovin and A. Voronkov, "Gorilla generator of random problems for linear/rational arithmetic and propositional logic." `http://www.cs.man.ac.uk/~korovink/rpg/index.html`, 2008. Accessed: 2018-03-20.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, 2001.

[10] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," *Transactions on Computer-Aided Design of Integrated Circuits and System*, pp. 279–285, 2001.

[11] I. Lynce and J. Ouaknine, "Sudoku as a sat problem," in *Proceedings of the 9 th international symposium on artificial intelligence and mathematics, aimath 2006, fort lauderdale*, Springer, 2006.

[12] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva, "Empirical study of the anatomy of modern sat solvers," in *Theory and Applications of Satisfiability Testing - SAT 2011* (K. A. Sakallah and L. Simon, eds.), (Berlin, Heidelberg), pp. 343–356, Springer Berlin Heidelberg, 2011.

[13] J. H. J. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers," *CoRR*, vol. abs/1506.08905, 2015.

[14] A. Biere and A. Fröhlich, "Evaluating cdcl restart schemes," 2015.