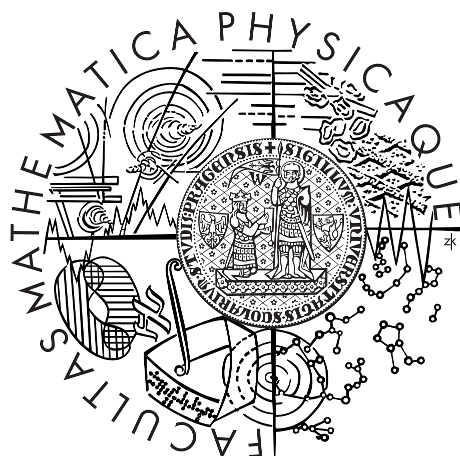
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Samuel Bartoš

Effective encoding of the Hidato and Numbrix puzzles to their CNF representation

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	RNDr. Tomáš Balyo
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2014

None of this would be possible without the guidance and patience of my thesis supervisor RNDr. Tomáš Balyo. I would also like to thank my family for their support. Finally, I want to express my gratitude to those who went above an beyond to help me, including Bc. Natália Tyrpáková, Zuzana Kovaľová and Bc. Juraj Citorík.

I declare that I carried out this bachelor thesis independently and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... date .... signature of the author

Název práce: Efektivní zakódovaní hlavolamů Hidato a Numbrix do jejich CNF reprezentace

Autor: Samuel Bartoš

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Tomáš Balyo, Katedra teoretické informatiky a matematické logiky

Abstrakt: I když je problém booleovské splnitelnosti NP-úplný, díky pokroku výkonnosti SAT solverů, může být mnoho jiných problémů řešeno efektivně, když je zakódujeme do booleovkých formulí. V této práci je tato metoda použita k nalezení řešení hlavolamů Hidato a Numbrix. Ukázáno je několik kódovacích technik spolu s implementací aplikace, která je sestrojuje. Zkoumáme také efektivitu jednotlivých kódování i jejich vliv na výkon různých SAT solverů. Výsledky experimentů ukazují, že efektivita kódování se u různých solverů může lišit a také že výběr nejlepší kombinace solver-kódovaní závisí na velikosti a typu hlavolamu.

Klíčová slova: SAT, kódování, Hidato, Numbrix,

Title: Effective encoding of the Hidato and Numbrix puzzles to their CNF representation

Author: Samuel Bartoš

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Tomáš Balyo, Department of Theoretical Computer Science and Mathematical Logic

Abstract: Although Boolean satisfiability problem (SAT) is NP-complete, thanks to the progress in the performance of SAT solvers many other problems can be solved efficiently by encoding them into Boolean formulas. In this thesis this method is applied to find the solution to Hidato and Numbrix puzzles. We present various encoding techniques and describe an implementation of an application designed to construct them. The efficiency of individual encodings together with their effect on the performance of several SAT solvers is examined. The results of conducted experiments indicate that the efficiency of encoding can change from solver to solver and that the best choice of encoding-solver combination depends heavily on the size and type of puzzle.

Keywords: SAT, encoding, Hidato, Numbrix,

Contents

Introduction **3**

1 Definitions **4**
1.1 Basic concepts from propositional logic . . . 4
1.1.1 Formulas and conjuctive normal form (CNF) . . . 4
1.1.2 Interpretations . . . 5
1.1.3 Set representation of CNF formulas . . . 6
1.1.4 Boolean satisfiability problem (SAT) . . . 6
1.1.5 Basic properties of interpretations . . . 6
1.2 Basic concepts from set theory . . . 7
1.2.1 Functions . . . 7
1.2.2 Transversals . . . 7
1.3 Hidato and Numbrix puzzles . . . 7
1.3.1 Numbering on cells . . . 8
1.3.2 Neighbourhood of cell . . . 8
1.3.3 Solutions . . . 10
1.3.4 Problem transformation . . . 10

2 Unary encodings **12**
2.1 Unary encoding . . . 12
2.1.1 Variables . . . 12
2.1.2 Clauses . . . 12
2.1.3 Validity . . . 12
2.1.4 Number of variables and clauses . . . 14
2.2 Reduced encoding . . . 15
2.2.1 Variables and clauses . . . 15
2.2.2 Validity . . . 15
2.2.3 Number of variables and clauses . . . 16

3 Binary and Tseitin encodings **17**
3.1 Binary encoding . . . 17
3.1.1 Variables . . . 17
3.1.2 Clauses . . . 18
3.1.3 Validity . . . 18
3.1.4 Number of variables and clauses . . . 20
3.2 Tseitin encoding . . . 20
3.2.1 Tseitin transformation . . . 21
3.2.2 Variables . . . 21
3.2.3 Clauses . . . 21
3.2.4 Validity . . . 22
3.2.5 Number of variables and clauses . . . 22

4 Implementation **23**
4.1 Overview . . . 23
4.2 Input and output . . . 23
4.2.1 DIMACS format . . . 23
4.2.2 Puzzle format . . . 24
4.3 Class `Puzzle` . . . 25
4.4 Class `Encoding` . . . 25
4.4.1 Unary encoding . . . 26
4.4.2 Reduced encoding . . . 28
4.4.3 Tseitin encoding . . . 29

5 Experiments **33**
5.1 Comparison criteria . . . 33
5.2 Methodology . . . 33
5.3 Puzzles with unique solutions . . . 34
5.3.1 Number of variables . . . 35
5.3.2 Number of clauses . . . 35
5.3.3 Size of encoding . . . 35
5.3.4 Time and spatial complexity . . . 36
5.4 Puzzles with multiple solutions . . . 38
5.4.1 Number of variables and clauses, size of formula and spatial complexity . . . 39
5.4.2 Time complexity . . . 40
5.5 Discusion . . . 40
5.5.1 Number of variables, number of clauses, size of encoding and spatial complexity . . . 40
5.5.2 Time complexity . . . 41
5.6 Summary . . . 41

Conclusion **46**
Future work . . . 46

Bibliography **48**

Introduction

Boolean satisfiability problem is the most important problem in complexity theory both from theoretical and practical point of view. Currently, there is no known algorithm that solves all instances of SAT in polynomial time and the general consensus is that no such algorithm can exist. However, thanks to the recent advancements in propositional solving, modern state-of-the-art SAT solvers are able do so with a large enough subset of SAT instances to be useful in a wide range of practical areas such as artificial intelligence [6], circuit design [7] and software verification [8]. This is done by representing problems arising in those areas as instances of SAT.

Two of such problems are Hidato and Numbrix. To solve Hidato, one must fill the grid with consecutive numbers that connect horizontally, vertically or diagonally. In Numbrix the goal is the same, only diagonal connections are not allowed.

This thesis focuses on exploring different techniques of encoding these problems into SAT instances in order to find the most efficient one with regard to the performance of SAT solver. The foundation of every encoding is the choice of variables and their representation, dividing encodings in this paper into two main groups: unary and binary. We also describe some perspective improvements of encodings in each group and examine their impact on the performance of various SAT solvers.

The structure of this thesis is following. In the first chapter we establish basic concepts used throughout the rest of the paper. The second chapter introduces unary encodings with possible enhancements and proves their validity. Chapter number 3 does the same with binary encodings. Next chapter describes one of the possible implementation of algorithm which puts encoding techniques into practice. Chapter 5 compares the efficiency of encodings using several criteria and supports its conclusions with results of conducted experiments.

1. Definitions

The focus of this chapter is a formal definition of basic concepts used throughout this thesis including CNF formulas, SAT solvers, Hidato and Numbrix puzzles.

1.1 Basic concepts from propositional logic

1.1.1 Formulas and conjuctive normal form (CNF)

Let $\mathcal{V}$ be an unbounded set of symbols called *variables* or *atoms*. Variables will be denoted using lower case letters possibly with subscripts.

Let F be an expression formed using variables from $\mathcal{V}$, operators (or connectives) $\land$, $\lor$, $\neg$, and complementary symbols (and). We say that F is a *formula* if it satisfies one of the following:

- $F = v$, where $v \in \mathcal{V}$;
- $F = \neg G$, where G is a formula;
- $F = (G_1 \land G_2)$, where G_1 and G_2 are both formulas;
- $F = (G_1 \lor G_2)$, where G_1 and G_2 are both formulas.

Due to the proliferation of parentheses, a complex formula may become very difficult to read and work with. To alleviate this phenomenon we introduce precedence rules akin to the standard mathematical order of operations:

1. $\neg$
2. $\land$
3. $\lor$

Example: The formula $(((v_1 \land v_2) \land (\neg v_3)) \lor ((v_4 \lor v_5) \land v_6))$ may be abbreviated to $v_1 \land v_2 \land \neg v_3 \lor (v_4 \lor v_5) \land v_6$.

Formulas will be represented by upper case letters sometimes accompanied by subscripts.

A *negation* of a formula F is formula a $\neg F$. A *conjunction* of formulas $F_1, F_2, \ldots, F_n$ is a formula $F_1 \land F_2 \land \cdots \land F_n$. A *disjunction* of formulas $F_1, F_2, \ldots, F_n$ is a formula $F_1 \lor F_2 \lor \cdots \lor F_n$. A *literal* is a variable or its negation. A *clause* is a disjunction of literals.

We say that a formula is in its *conjunctive normal form*, or CNF for short, if it consists of a conjunction of clauses.

Example: The formula $F = (v_1 \lor \neg v_2 \lor v_3) \land (\neg v_3 \lor v_4) \land \neg v_4$ is in CNF.
$(v_1 \lor \neg v_2 \lor v_3), (\neg v_3 \lor v_4), \neg v_4$ are clauses, $v_1, \neg v_2, v_3, \neg v_3, v_4, \neg v_4$ are literals and $var(F) = \{v_1, v_2, v_3, v_4\}$ are variables.

The notion of *subformula* is defined inductively as follows:

1. every formula F is a subformula of itself;
2. formula F is a subformula of formula $\neg F$;
3. formulas $F_1, F_2, \ldots, F_n$ are subformulas of formula $F_1 \wedge F_2 \wedge \cdots \wedge F_n$;
4. formulas $F_1, F_2, \ldots, F_n$ are subformulas of formula $F_1 \vee F_2 \vee \cdots \vee F_n$;
5. if F_1 is a subformula of F_2 and F_2 is a subformula of F_3, then F_1 is a subformula of F_3.

An *atomic formula* is a subformula that is an atom. Otherwise it is a *non-atomic subformula.*

1.1.2 Interpretations

We say that I is an *interpretation* if it is a set of literals such that $l \in I$ implies $\neg l \notin I$. An interpretation I *satisfies a formula* F if one of the following conditions holds:

1. F is a variable v and $v \in I$;
2. F is a negation of formula G and I does not satisfy G;
3. F is a conjunction $G_1 \wedge \cdots \wedge G_n$ and I satisfies all of $G_1, \ldots, G_n$;
4. F is a disjunction $G_1 \vee \cdots \vee G_n$ and I satisfies at least one of $G_1, \ldots, G_n$.

Otherwise we say that I *does not satisfy* F.

A formula F is *satisfiable* if there exist an interpretation satisfying F. Otherwise F is *unsatisfiable.* Two formulas are *equivalent* if an interpretation satisfies the first if and only if it satisfies the second. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Since every formula can be converted to an equivalent CNF formula using Boolean algebra, throughout this thesis we will limit ourselves to use only formulas in this form.

We say that an interpretation I is an *interpretation of a formula* F, denoted $I(F)$, if it contains only literals in the form of v or $\neg v$, where $v \in var(F)$. From this it immediately follows that $|var(F)| = |I(F)|$.

Example: The formula $F = (v_1 \vee \neg v_2)$ is satisfiable ($I(F) = \{v_1, \neg v_2\}$). Formulas F and $G = (\neg v_1 \wedge v_2)$ are equivalent and formulas F and $H = (v_1 \vee v_3) \wedge (\neg v_3 \vee \neg v_2)$ are equisatisfiable.

1.1.3 Set representation of CNF formulas

It is often convenient to think of a clause as a set of literals and of CNF formula as a set of sets of literals, or in other words a set of clauses. This convention will enable us to use the standard set notation and set operations when working with CNF formulas. Sets differ from formulas in that they are unordered and cannot contain duplicate elements. However, in the context of this thesis both of these distinctions are negligible, hence we will treat all CNF formulas as sets unless specified otherwise.

Example: The formula $F = (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_3 \vee v_4) \wedge \neg v_4$ expressed in set representation is $\{\{v_1, \neg v_2, v_3\}, \{\neg v_3, v_4\}, \{\neg v_4\}\}$ with clauses $\{v_1, \neg v_2, v_3\}$, $\{\neg v_3, v_4\}$ and $\{\neg v_4\}$.

1.1.4 Boolean satisfiability problem (SAT)

Boolean satisfiability problem, often abbreviated SAT, is the problem of determining whether a given formula is satisfiable. SAT is generally an NP-complete problem [4].

An algorithm designed to solve SAT is called *SAT solver*. Input for SAT solver is a formula in CNF. Output, provided it exists, is a satisfying interpretation. For the first part of this thesis, SAT solver will be thought of as a hypothetical black box with CNF formulas coming in and satisfying interpretations coming out. Distinction in the underlying algorithms of different SAT solvers will not be made until chapter 5.

1.1.5 Basic properties of interpretations

According to the definition, an interpretation $I(F)$ satisfies CNF formula F only if it satisfies every clause $C \in F$. Analogically, clause C is satisfied by $I(F)$ only if $I(F)$ satisfies at least one literal $l \in C$. This means that in order for $I(F)$ to satisfy F, $C \cap I(F) \neq \emptyset$ for every clause $C \in F$.

Let us summarize this observation in lemma 1.

Lemma 1: An interpretation $I(F)$ satisfies CNF formula F if and only if for all clauses $C \in F$ the fact $C \cap I(F) \neq \emptyset$ is true.

Proof: Follows immediately from the definition of CNF and satisfying interpretation. $\square$

Example: Let $F = \{\{v_1, \neg v_2, v_3\}, \{\neg v_3, v_4\}, \{\neg v_4\}\}$ be the input for a SAT solver. Then one of possible outputs is the interpretation $I(F) = \{v_1, v_2, \neg v_3, \neg v_4\}$. In addition $\{v_1, \neg v_2, v_3\} \cap I(F) = \{v_1\}$, $\{\neg v_3, v_4\} \cap I(F) = \{\neg v_3\}$, $\{\neg v_4\} \cap I(F) = \{\neg v_4\}$.

1.2 Basic concepts from set theory

1.2.1 Functions

A *Cartesian product* of sets A and B is a set containing all ordered pairs (a, b), where $a \in A$ and $b \in B$. It is usually denoted as $A \times B$.

A *(binary) relation* between A and B is any of the subsets of $A \times B$. A *domain* of a relation R is a set of those $a \in A$ that $(a, b) \in R$ for some $b \in B$. Analogically, a *range* of a relation R is a set of those $b \in B$ that $(a, b) \in R$ for some $a \in A$. To refer to the domain and the range of a relation R we use $dom(R)$ and $rng(R)$ respectively.

A *function* from A to B, denoted $f \colon A \to B$, is a relation between A and B with the following properties:

- $dom(f) = A$;
- $rng(f) \subseteq B$;
- $(a, b_1), (a, b_2) \in f$ implies $b_1 = b_2$.

Function f is *injective* if $(a_1, b), (a_2, b) \in f$ implies $a_1 = a_2$.
Function $f \colon A \to B$ is *surjective* if $rng(f) = B$. A function both injective and surjective is said to be a *bijection*.

1.2.2 Transversals

Given a collection of mutually disjoint sets, a *transversal* is a set containing exactly one element from each member of the collection.

Example: Let C be a collection of sets A_1, A_2 and A_3, where $A_1 = \{a, b, c\}$, $A_2 = \{d, e, f\}$ and $A_3 = \{g, h\}$. Then $\{a, d, g\}$ is a transversal on C. Other possible transversals are for instance $\{c, f, h\}$, $\{b, e, g\}$, and $\{b, e, h\}$.

1.3 Hidato and Numbrix puzzles

Hidato (from Hebrew word Hida, meaning riddle) is a logical puzzle invented by Izraeli mathematician Dr. Gyora Benedek [11]. The goal is to fill each cell of a grid with a unique number so that every pair of consecutive numbers is connected horizontally, vertically or diagonally. The grid is usually square or rectangular, but it can also assume a more irregular shape, like a skull or a heart. It can also contain one or more inner holes provided it stays continuous. Initially, the first and the last number are filled in. There may be some other numbers filled in too, so as to ensure the puzzle has a unique solution.

Numbrix, created by American columnist Marilyn vos Savant [11], is similar to Hidato. The only difference is that in Numbrix a diagonal connection of numbers is forbidden.

For an example of both Hidato and Numbrix puzzles see figure 1.

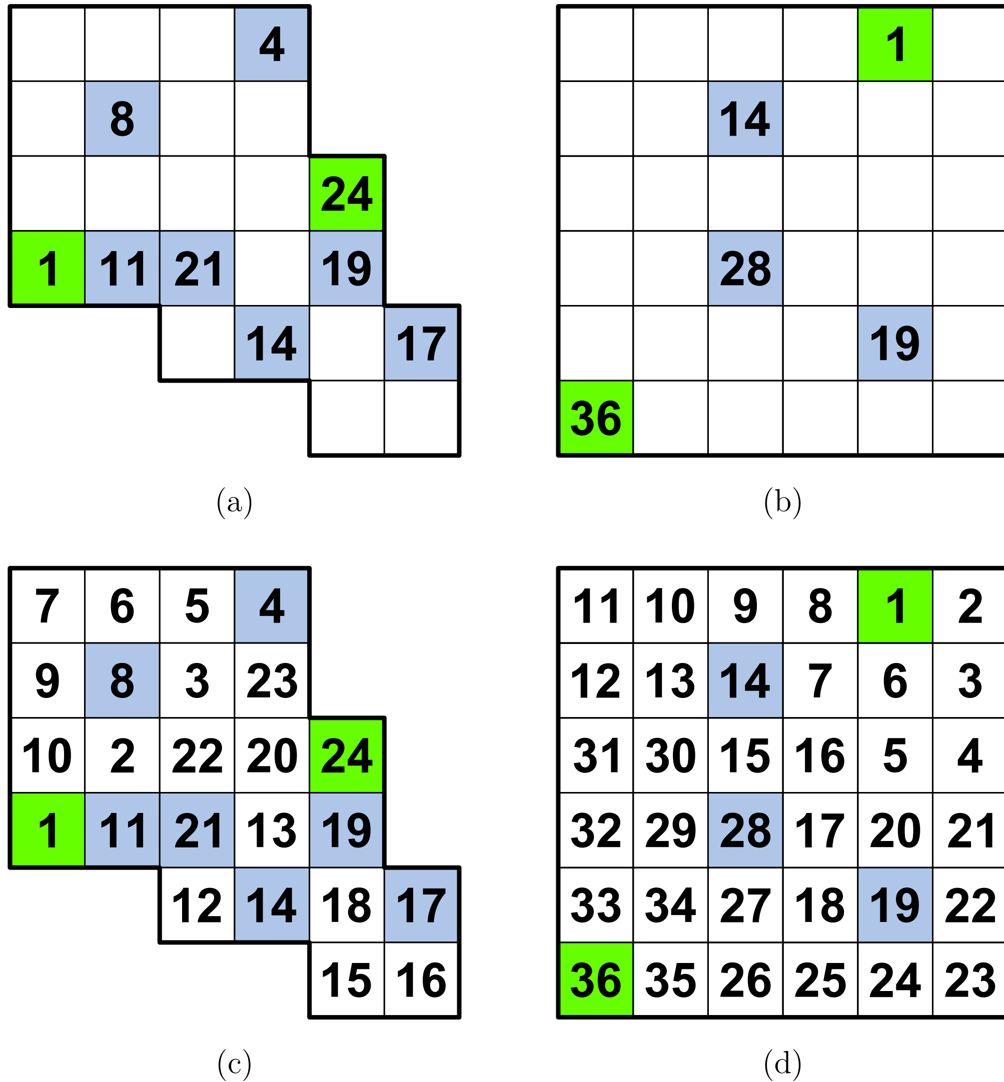


Figure 1: An example of Hidato (a) and Numbrix (b) puzzles together with their solutions (c) and (d) respectively

1.3.1 Numbering on cells

In this and the following chapters, we will need to refer to individual cells of Hidato and Numbrix puzzles. To avoid confusion, let us introduce a numbering on the set of all cells of a puzzle. Without any loss of generality, let the leftmost cell in the uppermost row of a Hidato puzzle be the first, the rightmost cell in the lowermost row be the last and let the numbering increase first from left to right and then from top to bottom. Let c_i denote the i-th cell. The situation is illustrated in figure 2.

1.3.2 Neighbourhood of cell

Let $\mathcal{H}$ denote a Hidato puzzle and $\mathcal{N}$ denote a Numbrix puzzle. Let $C(\mathcal{H})$ and $C(\mathcal{N})$ be the set containing all the cells of $\mathcal{H}$ and $\mathcal{N}$ respectively. Letter n will be used to refer to the total number of cells: the value of $|C(\mathcal{H})|$ or $|C(\mathcal{N})|$. Let $Const(\mathcal{H}) \subseteq C(\mathcal{H}) \times \{1, 2, \ldots, n\}$ be a set of pairs (c_i, x), where c_i is a cell with pre-filled number x. $Const(\mathcal{N})$ is defined analogically.

7	6	5	4		
9	8	3	23		
10	2	22	20	24	
1	11	21	13	19	
		12	14	18	17
				15	16

(c)

11	10	9	8	1	2
12	13	14	7	6	3
31	30	15	16	5	4
32	29	28	17	20	21
33	34	27	18	19	22
36	35	26	25	24	23

(d)

c_1	c_2	c_3	c_4		
c_5	c_6	c_7	c_8		
c_9	c_{10}	c_{11}	c_{12}	c_{13}	
c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	
		c_{19}	c_{20}	c_{21}	c_{22}
				c_{23}	c_{24}

(a)

c_1	c_2	c_3	c_4	c_5	c_6
c_7	c_8	c_9	c_{10}	c_{11}	c_{12}
c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}
c_{19}	c_{20}	c_{21}	c_{22}	c_{23}	c_{24}
c_{25}	c_{26}	c_{27}	c_{28}	c_{29}	c_{30}
c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	c_{36}

(b)

Figure 2: Numbering on Hidato (a),(c) and Numbrix (b),(d) puzzles

Definition: *A* neighbourhood *of a cell $c_i \in C(\mathcal{H})$, denoted $N(c_i)$, is the subset of $C(\mathcal{H})$ containing all the cells in $C(\mathcal{H})$ that are adjacent to (share an edge or a corner with) c_i. We say that c_i and $c_j \in N(c_i)$ are* neighbouring cells*.*
A successor *of cell a c_i is a neighbouring cell containing a consecutive number, if such a cell exists.*

After a closer look at the description of Hidato and Numbrix at the beginning of this chapter, it is apparent that both are very similar. The only difference is the presence of diagonal constriction (or the absence of diagonal rule) in Numbrix. Informally, the set of constrictions of Hidato is a proper subset of the set of constrictions of Numbrix. This means that if a (more constricted) solution to a Numbrix puzzle is found, it effectively solves a Hidato puzzle with the same grid and number placement. Hence, we will consider Numbrix a special case of Hidato, differing from the latter only in the definition of the neighbourhood of a cell.

Definition: *A* neighbourhood *of a cell $c_i \in C(\mathcal{N})$, denoted $N(c_i)$, is a subset of $C(\mathcal{N})$ containing all the cells in $C(\mathcal{N})$ sharing an edge with c_i.*

Example: In figure 1, $C(\mathcal{H}) = \{c_1, c_2, \dots, c_{24}\}$, $C(\mathcal{N}) = \{c_1, c_2, \dots, c_{36}\}$ and

$$\begin{aligned} Const(\mathcal{H}) =& \{(c_4, 4), (c_6, 8), (c_{13}, 24), (c_{14}, 1), (c_{15}, 11), \\ & (c_{16}, 21), (c_{18}, 19), (c_{20}, 14), (c_{22}, 17)\}; \\ Const(\mathcal{N}) =& \{(c_5, 1), (c_9, 14), (c_{21}, 28), (c_{29}, 19), (c_{31}, 36)\}. \end{aligned}$$

In $\mathcal{H}$ neighbouring cells of c_{11} are $N(c_{11}) = \{c_6, c_7, c_8, c_{10}, c_{12}, c_{15}, c_{16}, c_{17}\}$ and the successor of c_{11} is c_8.

In $\mathcal{N}$ neighbouring cells of c_{11} are $N(c_{11}) = \{c_5, c_{10}, c_{12}, c_{17}\}$ and the successor of c_{11} is c_{10}.

For the above mentioned reasons we will focus only on Hidato unless specified otherwise.

1.3.3 Solutions

From the nature of the problem presented we can draw the conclusion that every solution to Hidato puzzle must satisfy these five conditions:

Condition 1: Every cell of the grid must be filled with a natural number ranging from 1 to n;

Condition 2: No cell has two numbers;

Condition 3: No two cells have the same number;

Condition 4: Every cell except the one with number n does have a successor;

Condition 5: Pre-filled numbers are unchanged.

Therefore, if we construct a bijection from $C(\mathcal{H})$ to $\{1, 2, \dots, n\}$ obeying condition 4 and respecting condition 5, the problem of finding a solution to $\mathcal{H}$ will be transformed into the problem of finding this bijection. This is reflected in the following definition.

Definition: *A bijection $s\colon C(\mathcal{H}) \to \{1, 2, \dots, n\}$ is a* solution *to $\mathcal{H}$ if for every $c_i \in C(\mathcal{H})$ either $s(c_i) = n$ or $s(c_j) = s(c_i) + 1$, where $c_j \in N(c_i)$, and $Const(\mathcal{H}) \subseteq s$.*

1.3.4 Problem transformation

In the following chapters we are trying to find the most efficient way of solving Hidato by transforming it into CNF formulas. Generally, we want CNF formula in such a form that each satisfying interpretation represents a solution to Hidato. Doing so involves these two steps:

1. We choose a set of variables. This is done so that each interpretation constitutes a potential solution;
2. We then construct a set of clauses over these variables, which restricts satisfying interpretations to those representing a solution.

This will effectively transform Hidato into SAT. As previously stated, SAT is an NP-complete problem [4]. One implication of NP completeness is that all other problems in NP can be transformed into SAT efficiently in a polynomial time. As Hidato is also an NP complete problem [2], we can conclude that during the transformation of Hidato into SAT we did not ignore any aspect of real-life Hidato problem that would allow us to solve it faster directly using a deterministic polynomial algorithm. For this reason, the transformation is justified.

For the rest of the thesis we will use terms *encoding*, *technique of encoding* and *CNF formula* representing Hidato interchangeably.

2. Unary encodings

This chapter examines the task of transforming a Hidato puzzle into a CNF formula using unary encodings. We also prove that a satisfying interpretation to this formula represents a valid solution to the original puzzle.

2.1 Unary encoding

2.1.1 Variables

Let $\mathcal{H}$ be a Hidato puzzle with $C(\mathcal{H}) = \{c_1, c_2, \ldots, c_n\}$. Let UE denote Unary encoding, specifically the final CNF formula. We use variable $v_{c_i,x}$, where $c_i \in C(\mathcal{H})$ and $x \in \{1, 2, \ldots, n\}$, to represent the fact that the i-th cell contains number x. Consequently, literal $\neg v_{c_i,x}$ indicates the opposite. By this definition we need exactly n variables, $\{v_{c_i,1}, v_{c_i,2}, \ldots, v_{c_i,n}\}$, for each cell c_i. Therefore, the set of variables for Unary encoding is defined as

$$var(UE) = \{v_{c_1,1}, v_{c_i,2}, \ldots, v_{c_1,n}, v_{c_2,1}, v_{c_2,2} \ldots, v_{c_2,n}, \ldots, v_{c_n,n}\}.$$

2.1.2 Clauses

After the construction of UE and its processing by SAT solver, from a satisfying interpretation we construct a solution. Every solution to a Hidato puzzle respects conditions 1, 2, 3, 4 and 5 from section 1.3.3. In order for a solution to do so, UE consists of five sets of clauses denoted $C1$, $C2$, $C3$, $C4$ and $C5$. Set $C1$ forces the solution to obey condition 1, set $C2$ makes it respect condition 2 and so on.

Definition: Unary encoding *is formula* $UE = C1 \cup C2 \cup C3 \cup C4 \cup C5$ *with* $C1$, $C2$, $C3$, $C4$ *and* $C5$ *defined as follows:*

$$\begin{aligned}
C1 =& \{C1_{c_i} \mid c_i \in C(\mathcal{H})\}, \\
& \textit{where } C1_{c_i} = \{v_{c_i,1}, v_{c_i,2} \ldots, v_{c_i,n}\}; \\
C2 =& \{C2_{c_i,x_1,x_2} \mid c_i \in C(\mathcal{H}), x_1, x_2 \in \{1, 2, \ldots, n\}, x_1 \neq x_2\}, \\
& \textit{where } C2_{c_i,x_1,x_2} = \{\neg v_{c_i,x_1}, \neg v_{c_i,x_2}\}; \\
C3 =& \{C3_{c_i,c_j,x} \mid c_i, c_j \in C(\mathcal{H}), c_i \neq c_j, x \in \{1, 2, \ldots, n\}\}, \\
& \textit{where } C3_{c_i,c_j,x} = \{\neg v_{c_i,x}, \neg v_{c_j,x}\}; \\
C4 =& \{C4_{c_i,x} \mid c_i \in C(\mathcal{H}), x \in \{1, 2, \ldots, n-1\}\}, \\
& \textit{where } C4_{c_i,x} = \{\neg v_{c_i,x}\} \cup \{v_{c_j,x+1} \mid c_j \in N(c_i)\}; \\
C5 =& \{C5_{c_i,x} \mid (c_i, x) \in Const(\mathcal{H})\}, \\
& \textit{where } C5_{c_i,x} = \{v_{c_i,x}\}.
\end{aligned}$$

2.1.3 Validity

After using UE as the input for SAT solver, we can reconstruct a solution from solver's output $I(UE)$. This process is described in the following definition.

Definition: *Let $I(UE)$ be a satisfying interpretation of UE obtained from output of SAT solver. Let $s \subseteq C(\mathcal{H}) \times \{1, 2, \ldots, n\}$ be defined as follows: $(c_i, x) \in s$ if and only if $v_{c_i,x} \in I(UE)$.*

To prove that a relation s is a solution to $\mathcal{H}$, we must show that it is a bijection $s\colon C(\mathcal{H}) \to \{1, 2, \ldots, n\}$. Moreover, for every $c_i \in C(\mathcal{H})$ this bijection must satisfy either $s(c_i) = n$ or $s(c_j) = s(c_i) + 1$, where $c_j \in N(c)$. Additionally, s must possess a property of $Const(\mathcal{H}) \subseteq s$.

The proof will be split into five parts, four lemmas and a theorem, following the example set by five conditions and five sets of clauses.

Lemma 2: $s(c_i)$ is defined for every cell $c_i \in C(\mathcal{H})$.

Proof: Assume there is a cell $c_i \in C(\mathcal{H})$ with undefined $s(c_i)$. From the definition of interpretation $I(UE)$ and the construction of s, it follows that for every $x \in \{1, 2, \ldots, n\}$, $\neg v_{c_i,x} \in I(UE)$. At the same time $C1_{c_i} = \{v_{c_i,1}, v_{c_i,2}, \ldots, v_{c_i,n}\}$ coupled with the fact that $C1_{c_i} \cap I(UE) = \emptyset$ (lemma 1) contradicts the assumptions. □

One of the corollaries of lemma 2 above is that there are at least n variables in $I(UE)$. Also note that $dom(s) = C(\mathcal{H})$.

Lemma 3: s is a function.

Proof: Let $c_i \in C(\mathcal{H})$ and $x_1, x_2 \in \{1, 2, \ldots, n\}$. We will prove the fact that $(c_i, x_1), (c_i, x_2) \in s$ implies $x_1 = x_2$ by contradiction.

Let $v_{c_i,x_1} \in I(UE)$, $v_{c_i,x_2} \in I(UE)$ and $x_1 \neq x_2$. From lemma 1 we know that $I(UE) \cap C2_{c_i,x_1,x_2} \neq \emptyset$. Since $C2_{c_i,x_1,x_2} = \{\neg v_{c_i,x_1}, \neg v_{c_i,x_2}\}$, either $\neg v_{c_i,x_1}$ or $\neg v_{c_i,x_2}$ is in $I(UE)$. However, both of these cases leads to $I(UE)$ contradicting the definition of an interpretation. □

This lemma coupled with lemma 2 also proves that there are exactly n variables in $I(UE)$.

Lemma 4: s is injective.

Proof: Let $c_i, c_j \in C(\mathcal{H})$. We have to show that $s(c_i) = s(c_j)$ implies $c_i = c_j$ (or equivalently $i = j$).

Assume not. Let there exist such $x \in \{1, 2, \ldots, n\}$ that $x = s(c_i) = s(c_j)$ for $i \neq j$. Then $v_{c_i,x}$ and $v_{c_j,x}$ are both in $I(UE)$. But since $C3_{c_i,c_j,x} = \{\neg v_{c_i,x}, \neg v_{c_j,x}\}$, lemma 1 leads us again to contradiction. □

Since s is an injective function $s\colon C(\mathcal{H}) \to \{1, 2, \ldots, n\}$, from the fact that $|dom(s)| = |C(\mathcal{H})| = n = |\{1, 2, \ldots, n\}| = |rng(s)|$ we can infer that s is also surjective and therefore a bijection.

Lemma 5: $Const(\mathcal{H}) \subseteq s$

Proof: This follows immediately from the fact that for all $(c_i, x) \in Const(\mathcal{H})$, $C5_{c_i,x} = \{v_{c_i,x}\} \cap I(UE) \neq \emptyset$. □

Now we have everything we need for the final theorem summarizing all these partial conclusions and forming some of its own.

Theorem 1: s is a solution.

Proof: What remains to be proven is that $s(c_i) = n$ or $s(c_j) = s(c_i) + 1$, where $c_j \in N(c_i)$, holds for every $c_i \in C(\mathcal{H})$.

Let $c_i \in C(\mathcal{H})$ be an arbitrary cell. If $s(c_i) = n$, we are done. If $s(c_i) = x \neq n$, then $x+1 \in \{1, 2, \ldots, n\}$. Let $s(c_j) \neq x+1$ for every $c_j \in N(c_i)$. This means that for all $c_j \in N(c_i)$, $\neg v_{c_j,x+1} \in I(UE)$. As $C4_{i,x} = \{\neg v_{c_i,x}\} \cup \{v_{c_j,x+1} \mid c_j \in N(c_i)\}$, the fact that $s(c_i) = x$ implies $v_{c_i,x} \in I(UE)$ and $\neg v_{c_i,x} \notin I(UE)$. From lemma 1 it follows that $v_{c_j,x+1} \in I(UE)$ for at least one $c_j \in N(c_i)$. For this c_j the fact that both $v_{c_j,x+1}$ and $\neg v_{c_j,x+1}$ are in $I(UE)$ yields a contradiction, therefore proving the validity of s as a solution. □

2.1.4 Number of variables and clauses

The selection of variables in 2.1.1 is not dependent upon the neighbourhood of a cell. This means that $var(UE)$ is the same for both Hidato and Numbrix puzzles with the same number of cells. The same argument can be used in the case of the number of clauses $|UE|$. Only the size of individual $C4_{c_i,x}$ are affected, their count remains unchanged. Therefore we can omit Numbrix with regard to the number of variables and clauses.

The actual number of clauses is derived from the following observations which in turn are a direct corollary of the definition of Unary encoding in section 2.1.2:

$$\begin{aligned} |C1_{UE}| &= n \\ |C2_{UE}| &= \frac{n^2(n-1)}{2} \\ |C3_{UE}| &= \frac{n^2(n-1)}{2} \\ |C4_{UE}| &= n(n-1) \\ |C5_{UE}| &= |Const(\mathcal{H}) \end{aligned}$$

The actual number of variables is obtained from 2.1.1. The results of these conclusions are expressed in the following identities:

$$\begin{aligned} |var(UE)| &= n^2 \\ |UE| &= n^3 + |Const(\mathcal{H})| \end{aligned}$$

2.2 Reduced encoding

After careful consideration, one can conclude that some clauses in Unary encoding are redundant and can be inferred from other clauses. This lead us to try and reduce the total number of clauses by their elimination.

2.2.1 Variables and clauses

Definition: *Let UE be Unary encoding.* Reduced (Unary) encoding *is formula* $RE = UE \setminus (C1 \cup C3)$

Since the only difference is the absence of $C1$ and $C3$, the set of variables remains the same which means

$$var(RE) = \{v_{c_1,1}, v_{c_i,2} \ldots, v_{c_1,n}, v_{c_2,1}, v_{c_2,2}, \ldots, v_{c_2,n}, \ldots, v_{c_n,n}\}.$$

2.2.2 Validity

To prove the validity of this encoding, instead of constructing relation s we prove that Reduced encoding and Unary encoding are equivalent. In order to do so we must first construct V, a set of specific variables, using only clauses in $C2$, $C4$ and $C5$ from RU.

Notice that $C5_{c_{k_1},1} \in C5 \subseteq RE$ implies $v_{c_{k_1},1} \in I(RE)$ for some cell c_{k_1}. Since $v_{c_{k_1},1} \in I(RE)$ and $C4_{c_{k_1},1} \in RE$, it follows that $v_{c_{k_2},2} \in I(RE)$ for some $c_{k_2} \in N(c_{k_1})$ (lemma 1). We can repeat this process n times and let $V = \{v_{c_{k_1},1}, v_{c_{k_2},2}, \ldots, v_{c_{k_n},n}\} \subseteq I(RE)$.

Note that all the cells represented (appearing as an index of some variable) in V are distinct, because if $c_{k_i} = c_{k_j}$ held for $i \neq j$, the fact that $C2_{c_{k_i},i,j} = C2_{c_{k_j},i,j} = \{\neg v_{c_{k_j},i}, \neg v_{c_{k_j},j}\} \in RE$ would, by lemma 1, result in a contradiction. This means that during the construction of V in each one of n steps a cell distinct from all the others already in V was added. As V contains n distinct cells, every cell is represented in V. Moreover, since both $I(RE)$ and V contain exactly n variables and $V \subseteq I(RE)$, we can disprove the existence of variable $v_{c_i,x} \in (I(RE) \setminus V)$. Consequently, $C1_{c_i} \cap V \neq \emptyset$ for all cells $c_i \in C(\mathcal{H})$.

As advertised before, we utilize the set V and aforementioned observations in proving the following.

Theorem 2: Unary encoding and Reduced encoding are equivalent.

Proof: Let the interpretation $I(UE)$ satisfy UE. Since $RE \subseteq UE$, surely $I(UE)$ satisfies RE as well.

Now let $I(RE)$ be an interpretation satisfying RE but not UE. Let V be a set constructed using RE and $I(RE)$ as shown above. By lemma 1 there must exist at least one clause $X \in UE$ with $X \cap I(RE) = \emptyset$.

Let $X = C3_{c_i,c_j,x}$. Then both $v_{c_i,x}$ and $v_{c_j,x}$ are in $I(RE)$. But if $v_{c_i,x} \in V$, then $v_{c_j,x} \in I(RE) \setminus V$ and vice versa, since $c_i \neq c_j$.

Now let $X = C1_{c_i}$. Then also $C1_{c_i} \cap V = \emptyset$, as $V \subseteq I(RE)$. This c_i is not represented in V. Either way we have reached a contradiction. □

By theorem 2 we can construct a solution s to $\mathcal{H}$ from $I(RE)$ the same way as we would using $I(UE)$. This also renders conditions 1 and 3 obsolete, as a relation respecting only conditions 2, 4 and 5 is still a solution to $\mathcal{H}$.

2.2.3 Number of variables and clauses

Since $var(RE) = var(UE)$ and $RE = UE \setminus (C1 \cup C3)$, the following identities can be easily derived from 2.1.4:

$$
\begin{aligned}
|var(RE)| &= n^2; \\
|RE| &= n(n-1)\binom{n+2}{2} + |Const(\mathcal{H})|.
\end{aligned}
$$

3. Binary and Tseitin encodings

In this chapter we focus on binary encodings and its satisfying interpretations. We also try to improve upon this encoding by using a method called Tseitin transformation.

3.1 Binary encoding

In the previous chapter the fact that cell c_i contained number x was represented by variable $v_{c_i,x}$. On one hand this method of encoding allowed us relatively straightforward construction of clauses for UE. Also a satisfying interpretation represented a solution in a fairly transparent way. On the other hand this simplicity resulted in quadratic size of $|var(UE)|$ and cubic size of $|UE|$ with respect to the number of cells. Not even the removal of redundant clauses can mitigate this problem.

In this chapter we will focus on further reducing the number of variables and the number of clauses as a means to achieve improved efficiency. This will be done by choosing a different representation of the fact that i-th cell contains number x, one that requires less variables per cell and also eliminates the need for clauses enforcing condition 2.

Throughout this chapter we use the letter l to denote a literal. Specifically, literal $l_{c_i,x}$ is used to refer to either $v_{c_i,x}$ or $\neg v_{c_i,x}$. If $l_{c_i,x} = v_{c_i,x}$, then $\neg l_{c_i,x} = \neg v_{c_i,x}$, and vice versa.

3.1.1 Variables

Let $\mathcal{H}$ denote a Hidato puzzle with $C(\mathcal{H}) = \{c_1, c_2, \dots, c_n\}$ and let BE denote Binary encoding. Let v be an arbitrary variable. A literal l in an interpretation can be either in a form $l = v$, or $l = \neg v$, which means that l can embody two values. This leads us to consider l a binary number with $l = v$ interpreted as 1 and $l = \neg v$ as 0. Now if we let b be the smallest number of bits needed to express every x in $\{1, 2, \dots, n\}$ as a binary number, we can easily represent value x in cell c_i as a conjunction of literals $l_{c_i,1}, l_{c_i,2}, \dots, l_{c_i,b}$. The fact that c_i does not contain value x is in turn represented by a disjunction of literals $\neg l_{c_i,1}, \neg l_{c_i,2}, \dots, \neg l_{c_i,b}$. These sets of literals will be the backbone of formula BE, their role analogous to that of literals $v_{c_i,x}$ and $\neg v_{c_i,x}$ in the previous chapter. This is formalized in the following definition.

Definition: *Let $b = \lceil \log_2 n \rceil$, $c_i \in C(\mathcal{H})$ and $x \in \{1, 2, \dots, n\}$. Let us consider $x-1$ written in binary as a sequence $\{a_j\}_1^b$ of 0s and 1s. Let a_1 be the least and a_b the most significant digit. Let $v_{c_i,1}, \dots, v_{c_i,b}$ be variables. We define the set $L_{c_i,x}$ as*

$$L_{c_i,x} = \{v_{c_i,j} \mid a_j = 1\} \cup \{\neg v_{c_i,j} \mid a_j = 0\}$$

and interpret it as a conjunction. The set $\neg L_{c_i,x}$ is defined as

$$\neg L_{c_i,x} = \{v_{c_i,j} \mid a_j = 0\} \cup \{\neg v_{c_i,j} \mid a_j = 1\},$$

which we interpret as a disjunction.

Example: In figure 1 (a) $n = 24$ implies $b = 5$, therefore each $L_{c_i,x}$ contains 5 literals. If we want to construct $L_{c6,8}$, we must first realize that $8 - 1 = 7$ is 11100 in binary (the most significant digit is on the right) and accordingly $L_{c6,8} = \{v_{c6,1}, v_{c6,2}, v_{c6,3}, \neg v_{c6,4}, \neg v_{c6,5}\}$.
Similarly $\neg L_{c6,8}$ is simply $\{\neg v_{c6,1}, \neg v_{c6,2}, \neg v_{c6,3}, v_{c6,4}, v_{c6,5}\}$.

In the light of this definition we let $var(BE)$ be in the form of

$$var(BE) = \{v_{c_1,1}, v_{c_1,2}, \dots, v_{c_1,b}, v_{c_2,1}, v_{c_2,2}, \dots, v_{c_2,b}, \dots, v_{c_n,b}\}.$$

3.1.2 Clauses

This chapter continues in the spirit of the previous one, where the redundancy of conditions 1 and 3 became apparent, as they are respected as long as conditions 2, 4 and 5 are. Because each number has a unique binary representation, the very nature of Binary encoding makes certain that the condition 2 is never violated, thus eliminating the need for clauses forcing s to satisfy it. Therefore, BE will consist only a of set of clauses $C4$ for condition 4 and a set of clauses $C5$ for condition 5.

We will use $T_{c_i,x}$ to denote the set of all transversals T on the collection $\{L_{c_j,x+1} \mid c_j \in N(c_i)\}$.

Definition: *Let $BE = C4 \cup C5$, where*

$$C4 = \bigcup_{\substack{c_i \in C(\mathcal{H}) \\ x \in \{1,2,\dots,n-1\}}} C4_{c_i,x} \text{ and } C4_{c_i,x} = \{\neg L_{c_i,x} \cup T \mid T \in T_{c_i,x}\};$$

$$C5 = \bigcup_{(c_i,x) \in Const(\mathcal{H})} C5_{c_i,x} \text{ and } C5_{c_i,x} = \{\{l_{c_i,1}\}, \{l_{c_i,2}\}, \dots, \{l_{c_i,b}\}\}.$$

3.1.3 Validity

The complexity of this encoding prevents us from defining a relation s and then elegantly proving its validity as a solution, as was done in the previous chapter. Instead, we must choose a more indirect approach. First we will introduce four lemmas which will help us define a set of literals L. Second, we will prove that $L = I(BE)$. This fact together with the process of construction of L will shed some light on the properties of $I(BE)$, utilized in the definition of s and its validation as a solution to $\mathcal{H}$.

Lemma 6: If $L_{c_i,x_1} \subseteq I(BE)$, then $L_{c_i,x_2} \not\subseteq I(BE)$ holds for all $x_2 \neq x_1$.

Proof: This follows immediately from the definition of an interpretation of a formula and the fact that each $x \in \{1, 2, \dots, n\}$ has a unique binary representation. □

Lemma 7: $(c_i, x) \in Const(\mathcal{H})$ implies $L_{c_i,x} \subseteq I(BE)$.

Proof: Let $(c_i, x) \in Const$. Then $C5_{c_i,x} \subseteq C5 \subseteq BE$. Without any loss of generality let $C5_{c_i,x} = \{\{v_{c_i,1}\}, \{v_{c_i,2}\}, \dots, \{v_{c_i,b}\}\}$. From lemma 1, $v_{c_i,j} \in I(BE)$ for every $\{v_{c_i,j}\} \in C5_{c_i,x}$. Consequently, $L_{c_i,x} = \{v_{c_i,1}, v_{c_i,2}, \dots, v_{c_i,b}\} \subseteq I(BE)$.□

Lemma 8: Let $c_i \in C(\mathcal{H})$ and let $L_{c_i,x} \subseteq I(BE)$, where $x \in \{1, 2, \dots, n-1\}$. Then also $L_{c_j,x+1} \subseteq I(BE)$ for at least one of the cells in $N(c_i)$.

Proof: Let $N(c_i) = \{c_{j_1}, c_{j_2}, \dots, c_{j_m}\}$. For the purposes of a proof by contradiction, let us assume that although $L_{c_i,x} \subseteq I(BE)$, $L_{c_{j_k},x+1} \nsubseteq I(BE)$ for all $c_{j_k} \in N(c_i)$.

Let $X = \bigcup_{k=1}^{m} l_k$, where l_k is the literal in $L_{c_{j_k},x+1}$ that is not in $I(BE)$. Since X is a transversal on the collection $\{L_{c_j,x+1} \mid c_j \in N(c_i)\}$, it is also true that $\neg L_{c_i,x} \cup X \in C4_{c_i,x}$. According to lemma 1, $X \cap I(BE) = \emptyset$ means we have reached the desired contradiction. □

Now we have everything we need to construct a set of literals L by repeating the following process, similar to the previous chapter.

According to lemma 7, the fact that $(c_{k_1}, 1) \in Const(\mathcal{H})$ for some $c_{k_1} \in C(\mathcal{H})$ implies $L_{c_{k_1},1} \subseteq I(BE)$. From lemma 8, we also know that $L_{c_{k_2},2} \subseteq I(BE)$ for some $c_{k_2} \in N(c_{k_1})$. After n repetitions, we define the set L as

$$L = \bigcup_{i=1}^{i=n} L_{c_{k_i},i}.$$

Notice that as was the case in the previous chapter, every cell is represented in L. The opposite would result in a cell being represented twice. This follows from Dirichlet (or pigeon-hole) principle, because L would be constructed in n steps using only $(n-1)$ cells. However, this contradicts lemma 6 as $L \subseteq I(BE)$. Another corollary is that for every $c_i \in C(\mathcal{H})$ there exists some $x \in \{1, 2, \dots, n\}$ that $L_{c_i,x} \subseteq I(BE)$.

The purpose of L is to realize that $I(BE)$ has a very specific form. But first we must establish a relationship between them.

Lemma 9: $L = I(BE)$.

Proof: From the construction of L, surely $L \subseteq I(BE)$. The remaining inclusion $I(BE) \subseteq L$ will be proved by contradiction.

Let there exist a literal l satisfying $l \in I(BE)$ and $l \notin L$. Then there also must exist $L_{c_i,x_1} \subseteq I(BE)$ satisfying $l \in L_{c_i,x_1}$ and $L_{c_i,x_1} \nsubseteq L$. If on one hand $L_{c_i,x_2} \subseteq L$ for some $x_2 \neq x_1$, we have a conflict with lemma 6. On the other hand the fact $L_{c_i,x_2} \nsubseteq L$ for all $x_2 \neq x_1$ violates the property of every cell being represented in L.

In both cases we have reached a contradiction, thus $I(BE) \subseteq L$ and consequently $I(BE) = L$. □

As a corollary of the previous lemma, we can consider $I(BE)$ to be a union of disjoint sets, specifically $I(BE) = L_{c_{k_1},1} \cup L_{c_{k_2},2} \cup \dots \cup L_{c_{k_n},n}$. This will enable us to define a relation s and immediately prove that s is a solution to $\mathcal{H}$.

Definition: *Let* $(c_i, x) \in s$ *whenever* $L_{c_i,x} \subseteq I(BE)$.

Theorem 3: s is a solution.

Proof: First we will show that s is a bijection $s\colon C(\mathcal{H}) \to \{1, 2, \dots, n\}$. Since $|C(\mathcal{H})| = n$, doing so encompass proving only three things: s is a function (from lemma 6), domain of s is $C(\mathcal{H})$ (every cell is represented in L) and range of s is $\{1, 2, \dots, n\}$ (from the definition of s). The bijectiveness follows immediately.

Second, as was the case in the previous chapter, we must verify that $s(c_i) = n$ or $s(c_j) = s(c_i) + 1$, where $c_j \in N(c)$, holds for every $c_i \in C(\mathcal{H})$, as does the fact $Const(\mathcal{H}) \subseteq s$. The former is a direct consequence of lemma 8, the latter follows from lemma 7. $\square$

3.1.4 Number of variables and clauses

Variables selected in 3.1.1 do not depend on the neighbourhood of a cell. However, the number of clauses in BE do. This fact has two effects. First, $|BE|$ resulting from encoding a Hidato puzzle will differ from the one we acquire when encoding Numbrix. Second, the following identities are not exact, as the shape of the grid of a puzzle cannot be derived from the number of cells. For this reason we assume the worst case scenario, which is a theoretical situation where every cell has the maximum number of neighbours. For Hidato it is 8 and for Numbrix 4.

In the following we use the fact that $|T_{c_i,x}| = (\lceil \log_2 n \rceil)^8$ and $|T_{c_i,x}| = (\lceil \log_2 n \rceil)^4$ in Hidato and Numbrix respectively:

$$
\begin{aligned}
|var(BE)| &= n\lceil \log_2 n \rceil; \\
|BE_{\mathcal{H}}| &= n(n-1)(\lceil \log_2 n \rceil)^8 + \lceil \log_2 n \rceil |Const(\mathcal{H})|; \\
|BE_{\mathcal{N}}| &= n(n-1)(\lceil \log_2 n \rceil)^4 + \lceil \log_2 n \rceil |Const(\mathcal{H})|.
\end{aligned}
$$

$|BE_{\mathcal{H}}|$ describes the size of Binary encoding of Hidato, $|BE_{\mathcal{N}}|$ expresses the number of clauses in Numbrix.

3.2 Tseitin encoding

The reason behind the introduction of Binary encoding is an attempt to reduce the number of variables and clauses in the final CNF formula. However, at this point it seems that we have accomplished the opposite. The size of $|C5|$ in unary encoding was constant, now it is a factor growing logarithmically. The size of $|C4|$ was quadratic and now, in the worst case scenario, $|C4| \sim n^2 \log_2^8 n$. Neither the elimination of the cubic-sized set $C2$ nor the reduction of the number of variables from n^2 to $n(log_2 n)$ is not enough to mitigate this. Therefore, this technique of encoding is very inefficient with regard to the number of clauses, even to the point of our inability to use it as an input for SAT solver for puzzles with relevant numbers of cells. A conclusion like this would defeat the purpose of Binary encoding, were it not for the fact that we can transform an arbitrary formula to an equisatisfiable CNF formula in polynomial time generating only a linear number of clauses and using only linear number of additional variables. This method is known as the *Tseitin transformation* [10].

3.2.1 Tseitin transformation

Consider an arbitrary non-CNF formula, consisting of a finite number of subformulas, which we want to convert to its CNF representation. The foundation of the Tseitin transformation is the introduction of new variables, one for every non-atomic subformula of the original formula along with clauses to capture the relationship between these new variables and the subformulas. We use $T(G)$ to denote CNF formula resulting from the Tseitin transformation of a formula G.

Note that we do not always have to introduce a new variable whenever we encounter a subformula for the transformation to work.

Example: Let $F = (v_1 \wedge v_2) \vee v_3$. We introduce a new variable v_4 to represent subformula $(v_1 \wedge v_2)$. The formula capturing this relationship is then of the form $(v_4 \vee \neg v_1 \vee \neg v_2) \wedge (\neg v_4 \vee v_1) \wedge (\neg v_4 \vee v_2)$. We add it to F and substitute v_4 for $(v_1 \wedge v_2)$. The result is $T(F) = (v_4 \vee \neg v_1 \vee \neg v_2) \wedge (\neg v_4 \vee v_1) \wedge (\neg v_4 \vee v_2) \wedge (v_4 \vee v_3)$, which is $\{\{v_4, \neg v_1, \neg v_2\}, \{\neg v_4, v_1\}, \{\neg v_4, v_2\}, \{v_4, v_3\}\}$ in set representation.

3.2.2 Variables

Tseitin encoding TE uses the same definitions of $L_{c_i,x}$ and $\neg L_{c_i,x}$ as Binary encoding. However, $var(TE)$ has to account for the additional variables used in Tseitin transformation. Therefore

$$var(TE) = \{v_{c_1,1}, v_{c_1,2}, \ldots, v_{c_1,b}, v_{c_2,1}, v_{c_2,2}, \ldots, v_{c_2,b}, \ldots, v_{c_n,b}\} \cup \bigcup_{\substack{c_i \in C(\mathcal{H}) \\ x \in \{2,3,\ldots,n\}}} v_{L_{c_i,x}}.$$

3.2.3 Clauses

In our case we use Tseitin transformation to reduce the size of $|C4|$ from binary encoding by replacing each $C4_{c_i,x}$ with an equisatisfiable formula $T(C4_{c_i,x})$. To do this, we must first realize that the conjunction $L_{c_i,x} = \{l_{c_i,1}, l_{c_i,2}, \ldots, l_{c_i,b}\}$ after the Tseitin transformation is

$$T(L_{c_i,x}) = \{\{v_{L_{c_i,x}}, \neg l_{c_i,1}, \ldots, \neg l_{c_i,b}\}, \{\neg v_{L_{c_i,x}}, l_{c_i,1}\}, \ldots, \{\neg v_{L_{c_i,x}}, l_{c_i,b}\}\}.$$

Finally, we have everything we need for the definition of Tseitin encoding, an improved Binary encoding, in the form of TE.

Definition: *Tseitin encoding is a formula $TE = C4 \cup C5$ with $C4$ and $C5$ defined as follows:*

$$C4 = \bigcup_{\substack{c_i \in C(\mathcal{H}) \\ x \in \{1,2,\ldots,n-1\}}} T(C4_{c_i,x}),$$

where $T(C4_{c_i,x}) = \{\neg L_{c_i,x} \cup \{v_{L_{c_j,x+1}} \mid c_j \in N(c_i)\}\} \cup \{T(L_{c_j,x+1} \mid c_j \in N(c_i)\}$;

$$C5 = \bigcup_{(c_i,x) \in Const(\mathcal{H})} C5_{c_i,x},$$

where $C5_{c_i,x} = \{\{l_{c_i,1}\}, \{l_{c_i,2}\}, \ldots, \{l_{c_i,b}\}\}$.

3.2.4 Validity

Theorem 4: Binary and Tseitin encoding are equisatisfiable.

Proof: Let $BE = C4 \cup C5$ be Binary and $TE = C4' \cup C5'$ Tseitin encoding. Let us consider arbitrary $C4_{c_i,x} \in C4$. $C4_{c_i,x}$ is equivalent to the disjunction of $\neg L_{c_i,x}, L_{c_{j_1},x+1}, L_{c_{j_2},x+1}, \ldots, L_{c_{j_m},x+1}$, where $N(c_i) = \{c_{j1}, c_{j2}, \ldots, c_{j_m}\}$, which, according to [10], is in turn equisatisfiable with $T(C4_{c_i,x}) \in C4'$. Consequently, $C4$ and $C4'$ are equisatisfiable. This together with the fact that $C5 = C5'$ proves the theorem.

Since BE and TE are equisatisfiable and $var(BE) \subseteq var(TE)$, a satisfying interpretation $I(TE)$ satisfies also BE and therefore encodes a valid solution s.

3.2.5 Number of variables and clauses

After examining corresponding definitions we arrived at these conclusions:

$$|C4| = n(n-1)\left(\lceil \log_2 n \rceil + 2\right);$$
$$|C5| = \lceil \log_2 n \rceil |Const(\mathcal{H}_n)|.$$

The idea behind finding the value of $|C4_{TE}|$ is to reorganize the clauses into two groups and count them separately. First, there are clauses in $T(L_{c_i,x})$, where c_i can be any of the cells in $C(\mathcal{H})$ and x any of the values in $\{2, \ldots, n\}$. Since each $|T(L_{c_i,x})| = \lceil \log_2 n \rceil + 1$, we have a total of $n(n-1)(\lceil \log_2 n \rceil + 1)$ clauses in the first group. The second group is comprised of clauses which combine $\neg L_{c_i,x}$ with $v_{L_{c_j,x+1}}$, where c_j can be any of $N(c_i)$. While the size of $N(c_i)$ affects the size of clauses in the second group, it does not do so for their count. This also means that $|TE|$ will be the same for $\mathcal{H}_n$ and $\mathcal{N}_n$. As c_i can be any of the cells in $C(\mathcal{H})$ and x any of the values in $\{2, \ldots, n\}$, we are left with $n(n-1)$ clauses in the second group. Thus we arrive at the aforementioned identity.

The fact that the total number of clauses in Tseitin encoding remains the same after switching from Hidato to Numbrix is also true for the number of variables. This means that we can omit Numbrix with regard to both and list only figures expressing the values of $|var(TE)|$ and $|TE|$ for Hidato.

Since $TE = C4 \cup C5$, using observations above we can easily deduce the following identities:

$$|var(TE)| = n(n - 1 + \lceil \log_2 n \rceil);$$
$$|TE| = n(n-1)\left(\lceil \log_2 n \rceil + 2\right) + \lceil \log_2 n \rceil |Const(\mathcal{H}_n)|.$$

The value of $|var(TE)|$ is derived directly from the selection in 3.2.2.

4. Implementation

This chapter describes one of the possible implementations of **`Codec`** using an object oriented programming terminology. **`Codec`** is a program designed with two main functions. The first is encoding a Hidato or Numbrix puzzle into a formula in CNF. This puts into practise all of the techniques of encoding presented in previous chapters. The second function is decoding output of SAT solver (a satisfying interpretation) into human-readable format.

4.1 Overview

Our **`Codec`** is implemented as a simple console application. It consists of two main classes. The first one is class **`Puzzle`**, which processes input and also provides puzzle representation. The second and main class of **`Codec`** is class **`Encoding`**, responsible for creating clauses. The technique of encoding used to produce the final CNF formula or to decode a satisfying interpretation is determined by the particularities of the implementation of class **`Encoding`**.

4.2 Input and output

`Codec` is controlled using simple console commands. A command is a list of string arguments separated by spaces formatted as one of the following:

1. `encode unary|reduced|tseitin puzzle outputFile hidato|numbrix`;
2. `decode unary|reduced|tseitin puzzle inputFile outputFile`;
3. `exit|close|end|quit|q`.

First command tells **`Codec`** to encode file `puzzle` using specified type of encoding into file `outputFile` and that `puzzle` should be encoded as either Hidato or Numbrix. The second command makes **`Codec`** decode a satisfying interpretation in file `inputFile` into `outputFile` using chosen technique of encoding. The last command terminates the application.

Files `inputFile` and `outputFile` in encode and decode commands are in DIMACS format, file `puzzle` is a text representation of Hidato or Numbrix. Both these formats are described in sections 4.2.1 and 4.2.2.

4.2.1 DIMACS format

DIMACS format is a simple text format widely accepted as the standard for formulas in CNF. In a DIMACS file each line beginning with a lower case letter `"c"` followed be a space is a comment. The file usually contains a single block of comments at the beginning, but it can be also interleaved with them. Comment lines are followed by a single problem line. This is in the form of `"p cnf` ***`nbvars nbclauses`***`"`, where ***`nbvars`*** and ***`nbclauses`*** are integers standing for the total number of variables and clauses respectively. Next there are ***`nbclauses`*** lines, each one representing a *DIMACS clause*. This is defined using

a space separated sequence of *DIMACS literals* terminated by "0". DIMACS representation of a literal is a non-zero integer from the interval $[-\mathit{nbvars}, \mathit{nbvars}]$. Positive integer designates a *DIMACS variable* (a positive literal), and negative integer a negation of one.

```
c DIMACS representation of
formula
c (v₁ ∨ ¬v₂ ∨ v₃) ∧ (¬v₃ ∨ v₄) ∧
¬v₄
c looks like this:
p cnf 4 3
1 -2 3 0
-3 4 0
-4 0
```

Figure 3: An example of a formula is DIMACS format

Converting a CNF formula (a set of sets of literals) into DIMACS format can be done in a single iteration over all the literals. For this reason we continue to use the set representation, which provides us with a more convenient and flexible notation.

4.2.2 Puzzle format

Formally there are only two types of cells in a puzzle. Cells with a pre-filled number and cells with no number. However, we consider an additional third type called invalid cells. These cells are added to the grid of irregularly shaped puzzles in order to make it square or rectangular.

The format of a puzzle is a following. Each line represents one row in the grid of the puzzle. Individual cells in the row, separated by spaces, are denoted using either an integer, "." or "x", depending on the type of the cell. For pre-filled cells it is the number they contain, for empty cells the character "." and for invalid cells the letter "x". See figure 4.

```
.  .  .  4  x  x
.  8  .  .  x  x
.  .  .  .  24 x
1  11 21 .  19 x
x  x  .  14 .  17
x  x  x  x  .  .
```

(a)

```
7  6  5  4  x  x
9  8  3  23 x  x
10 2  22 20 24 x
1  11 21 13 19 x
x  x  12 14 18 17
x  x  x  x  15 16
```

(b)

Figure 4: An example of an unsolved (a) and solved (b) Hidato puzzle in text format

4.3 Class `Puzzle`

A cell in a puzzle is represented by an ordered pair of integers. The first one, *cell identifier* or *cell id*, is a value ranging from 0 to n and defines a numbering on cells with 0 reserved for invalid cells. Cells are numbered from left to right and from top to bottom as illustrated in section 1.3.1. The first cell is therefore a cell with id equal to 1 and so on. The second number in the pair, *cell value*, corresponds to the pre-filled number in the cell and is set to 0 if the cell has no number or is invalid. Cell representation is summarized in the following list:

- $(0, 0)$ for invalid cells;
- $(i, 0)$, where $i \in \{1, \ldots, n\}$ for empty cells;
- (i, x), where $i \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, n\}$ for pre-filled cells.

The puzzle is represented by a two dimensional array of these pairs embedded in class **`Puzzle`**. Apart from this array, **`Puzzle`** provides us also with access to the neighbouring cells via method `ListNeighbours`. Particularities of the implementation of `ListNeighbours` is dependent on whether we want to encode a Hidato or a Numbrix puzzle as the definitions for the neighbourhood of a cells differ. See section 1.3.2.

4.4 Class `Encoding`

As each implementation of class **`Encoding`** corresponds to one technique of encoding, all the implementations in our program share some common characteristics.

First, an implementation must be able to provide a conversion mechanism, the means to convert a (cell id,cell value) pair into a set of DIMACS literals and vice versa. It is put into practice through method `Convert` and `IConvert`.

Second, every implementation must contain five methods for producing clauses forcing a solution to respect five conditions from section 1.3.3. They utilize method `Convert` to do their bidding. These methods are called `Exists`, `IsUnique`, `AreNotEqual`, `Precedes` and `IsEqual`. Methods `Exists` and `IsUnique` take a cell id, method `AreNotEqual` takes two cell identifiers, method `Precedes` takes a cell id and a set of cell identifiers and `IsEqual` takes both cell id and cell value as an argument.

The cooperation of these five methods is controlled by `Encode`, the main method of class **`Encoding`**. Arguments for `Encode` are an instance of class **`Puzzle`** and `outputFile`.

Last, **`Encoding`** contains method `Decode` responsible for decoding a satisfying interpretation (a set of DIMACS literals) back into a solved puzzle in text format. To accomplish this, `Decode` employs method `IConvert` described above.

Algorithm in method `Decode` is the same for all encodings. It consists of two steps. First is a cycle. In each iteration we take the first x literals from argument `inputFile` (a satisfying interpretation provided by SAT solver) and pass them to method `IConvert`. Variable x is either n or b (defined in 3.1.1), depending on whether `inputFile` is a result of Unary (n), Reduced (n) or Tseitin encoding (b). The output of `IConvert` is stored in a set called *values*. In second step, this

set is disassembled and , together with **`Puzzle`** as a template, used to typeset `outputFile`. See algorithm 1.

Algorithm 1 `Decode`(**`Puzzle`**,`inputFile`)

---

for $i = 1$ to n **do**
  ints $\leftarrow$ first x literals from `inputFile`
  remove the first x literals from `inputFile`
  add the output of `IConvert`(*ints*) to *values*
end for
use **`Puzzle`** as a template to typeset *values* into output file

---

4.4.1 Unary encoding

In this technique of encoding we assign $(i-1)n+x$ to be the DIMACS variable representing the expression $\nu(c_i) = x$. This is summarized in method `Convert` and can be characterized by formula

$$l = (id-1)n + x,$$

where the cell id *id* and the cell value x are the arguments and l is the resulting DIMACS variable. This means that all the values of all the cells can be expressed by DIMACS variables from the set $\{1, 2 \ldots, n^2\}$.

The algorithm 2 in method `IConvert` is a simple loop over all DIMACS literals in the argument *ints*, selecting the positive one. Since it can be any number from 1 to n^2, we apply operation $(\mod n)$ to acquire the value it represents. Appropriate incrementation and decrementation prevents $(\bmod\ n)$ from producing 0 when a cell contains number n.

Algorithm 2 `IConvert`(*ints*)

---

for all i in *ints* **do**
  if $i > 0$ **then**
    return $(i - 1 \bmod n) + 1$
  end if
end for

The implementation of method `Exists` is very simple. The formula is constructed by concatenating the output of `Convert`(*id*,*x*) in a cycle from $x = 1$ to $x = n$, where *id* is an argument of `Exists`.

Method `IsUnique` cycles through all unordered pairs of numbers $\{x_1, x_2\}$ from the set $\{1, 2 \ldots, n\}$ and builds a set of clauses using building blocks in the form of $\{-$`Convert`$(id, x_1), -$`Convert`$(id, y_2)\}$, see algorithm 3.

Algorithm 3 `IsUnique`(id)

$CNF \leftarrow \emptyset$
for $x_1 = 1$ **to** $x_1 = n - 1$ **do**
  for $x_2 = x1$ **to** $x_2 = n$ **do**
    add clause $\{-\texttt{Convert}(id, x_1), -\texttt{Convert}(id, x_2)\}$ to CNF
  end for
end for
return CNF

Method `AreNotEqual` (algorithm 4) works in a similar manner, except the cycle is from $x = 1$ to n and $\{-\texttt{Convert}(id_1, x), -\texttt{Convert}(id_2, x)\}$ are the building blocks. Id id_1 and id_2 are two different cell identifiers provided as arguments.

Algorithm 4 `AreNotEqual`(id_1,id_2)

$CNF \leftarrow \emptyset$
for $x = 1$ **to** $x = n$ **do**
  add clause $\{-\texttt{Convert}(id_1, x), -\texttt{Convert}(id_2, x)\}$ to CNF
end for
return CNF

The inner workings of method `Precedes` can be best understood from algorithm 5. The second argument, ids, is a set of identifiers of cells neighbouring with a cell with id provided as the first argument id. The algorithm uses two cycles, an outer and nested one, to produce clause in the form

$$\{-\texttt{Convert}(id, x), \texttt{Convert}(i_1, x + 1), \texttt{Convert}(i_2, x + 1), \ldots, \texttt{Convert}(i_m, x + 1)\},$$

where $x = 1, \ldots, n - 1$ and $ids = \{i_1, i_2, \ldots, i_m\}$.

Algorithm 5 `Precedes`(id, ids)

$CNF \leftarrow \emptyset$
for $x = 1$ **to** $x = n - 1$ **do**
  $clause \leftarrow \emptyset$
  add $-\texttt{Convert}(id, x)$ to $clause$
  for all i in ids **do**
    add $\texttt{Convert}(i, x + 1)$ to $clause$
  end for
  add $clause$ to CNF
end for
return CNF

Method `IsEqual` simply returns $\{\texttt{Convert}(id, x)\}$ for provided cell id id and cell value x.

As the main method of class **`Encoding`**, `Encode` coordinates the cooperation of methods to encode Hidato and Numbrix puzzles into CNF formula. Method contains five cycles. Each cycle iterates over the cells in an instance of class **`Puzzle`** and passes them as arguments to one of the methods in **`Encoding`** (`Exists`,

`IsUnique`, `AreNotEqual`, `Precedes`, `IsEqual`). The returned clauses `Encode` writes into the `outputFile`. This process is outlined in algorithm 6

Algorithm 6 `Encode`(**`Puzzle`**,`outputFile`)

```
CNF ← ∅
for all cells c in Puzzle do
  add the output of Exists(id of c) to CNF
end for
for all cells c in Puzzle do
  add the output of IsUnique(id of c) to CNF
end for
for all pairs of cells (c1, c2) in Puzzle do
  add the output of AreNotEqual(id of c1, id of c2) to CNF
end for
for all cells c in Puzzle do
  ids ← Puzzle.ListNeighbours(id of c)
  add the output of Precedes(id of c, ids) to CNF
end for
for all cells c in Puzzle do
  if c has a non-zero value x then
    add the output of IsEqual(c, x) to CNF
  end if
end for
write CNF to outputFile
```

4.4.2 Reduced encoding

This implementation is identical to the previous one, with the exception of method `Encode`. Differences can be seen in algorithm 7

Algorithm 7 `Encode`(**`Puzzle`**,`outputFile`)

```
CNF ← ∅
for all cells c in Puzzle do
  add the output of IsUnique(id of c) to CNF
end for
for all cells c in Puzzle do
  ids ← Puzzle.ListNeighbours(id of c)
  add the output of Precedes(id of c, ids) to CNF
end for
for all cells c in Puzzle do
  if c has a non-zero value x then
    add the output of IsEqual(c, x) to CNF
  end if
end for
write CNF to outputFile
```

4.4.3 Tseitin encoding

In Tseitin encoding $\nu(c_i) = x$ is expressed as a set of literals. We assign DIMACS variables $(i-1)b+1, (i-1)b+2. \ldots, ib$ to represent the value of the i-th cell. Therefore, for all the values of all the cells we use DIMACS variables from the interval $1, \ldots, nb$. Values $nb+1$ and forth are reserved for additional variables acquired during the Tseitin transformation.

First we describe the implementation of the conversion mechanism. Let id be a cell id and x a cell value. We consider x a binary number. Method `Convert` takes id and x as arguments and outputs a set of integers $ints$, using the following algorithm. First, we decrement x by one. This is done because values of cells are from the set $\{1, 2 \ldots, n\}$ and since b literals can represent only numbers from $\{0, 1, \ldots, 2^b - 1\}$, the situation $n = 2^b$ would either require an additional DIMACS literal or a more complex conversion mechanism. Next we look at the last bit of x. If it is 1, we add the corresponding positive integer to $ints$. Otherwise we add its negative counterpart. After this, we remove the last bit of x and repeat the whole process. We finish after $ints$ contain b integers. See algorithm 13.

Algorithm 8 `Convert`(id,x)

$x \leftarrow x - 1$
$ints \leftarrow \emptyset$
for $i = 1$ **to** b **do**
  if the last bit of x is 1 **then**
    add $(id - 1) * b + i$ to $ints$
  else
    add $-((id - 1) * b + i)$ to $ints$
  end if
  remove the last bit of x
end for
return $ints$

From this it is clear that in the expression $\nu(c_i) = x$ the value of $(i-1)b+1$ represents the first (least significant) bit of x, the value of $(i-1)b+2$ the second and so on.

Algorithm 9 in method `IConverse` is a variation on a binary to decimal conversion algorithm using Horner's method. We consider the argument $ints$ to be an ordered set of integers. The order is dictated by ascending absolute values of contained numbers. The final incrementation compensates the decrementation performed in `Connvert`.

Algorithm 9 `IConverse`($ints$)

```
result ← 0
power ← 1
for all i in ints do
  if i > 0 then
    result ← result + power
  end if
  power ← power · 2
end for
return result + 1
```

To produce the clauses, this implementation of **`Encoding`** uses an additional data structure: a hash table. The reason for this is that during the Tseitin transformation, additional variables are assigned to subformulas and the hash table stores this relation. To manipulate the hash table **`Encoding`** utilizes two supplementary methods: `Hash` and `Contains`. `Hash` receives a clause and returns a unique DIMACS variable assigned to it in the hash table. Method `Contains`, returning True or False, checks whether hash table contains an entry for the clause received as an argument.

The bulk of the work is done in methods `Precedes` and `Tseitin`. `Tseitin` is responsible for the the Tseitin transformation of clauses. First, using `Contains`, it tests whether the Tseitin transformation of a clause provided as argument have already happened. In that case it returns an empty set to avoid a duplicity of clauses in the output file. In case the check fails it produces a set of clauses in the form of

$$\{\{h, -i_1, -i_2, \ldots, -i_m\}, \{-h, i_1\}, \{-h, i_2\}, \ldots, \{-h, i_m\}\},$$

where $ints = \{i_1, i_2, \ldots, i_m\}$ and $h = \texttt{Hash}(ints)$. The details are described in algorithm 10.

Algorithm 10 `Tseitin`($ints$)

```
CNF ← ∅
if Contains(ints) then
  return CNF
else
  clause ← ∅
  h ← Hash(ints)
  add h to clause
  for all i in ints do
    add −i to clause
  end for
  add clause to CNF
  for all i in ints do
    add {−h, i} to CNF
  end for
  return CNF
end if
```

`Precedes` receives two arguments: a cell id (id) and a set of cell ids of the neighbouring cells (ids). In each iteration of the main cycle it produces a set of clauses. With the exception of the last one they are the result of the method `Tseitin` called for every $i \in ids$ in a nested cycle. The last clause is the one tying together all the additional DIMACS variables added during the Tseitin transformation with DIMACS literals representing the values of the cell. It is constructed by concatenating the negation of DIMACS literals in the output of `Convert`(id, x) with the output of `Hash`(`Convert`$(i, x + 1)$) for all $i \in ids$. This process is illustrated in algorithm 11.

Algorithm 11 `Precedes`(id, ids)

$CNF \leftarrow \emptyset$
for $x = 1$ **to** n **do**
  $clause \leftarrow \emptyset$
  for all i in `Convert`(id,x) **do**
    add $-i$ to $clause$
  end for
  for all i in ids **do**
    add the output of `Hash`(`Convert`(i,$x + 1$)) to $clause$
    add the output of `Tseitin`(`Convert`(i,$x + 1$)) to CNF
  end for
  add $clause$ to CNF
end for
return CNF

The result of a call to `IsEqual` is the set of clauses $\{\{i_1\}, \{i_2\}, \ldots, \{i_b\}\}$, where `Convert`$(id, x) = \{i_1, i_2, \ldots, i_b\}$ and id with x are arguments. See algorithm 12.

Algorithm 12 `IsEqual`

$CNF \leftarrow \emptyset$
for all i in `Convert`(id,x) **do**
  add i to CNF
end for
return CNF

The rest of the methods, namely `Exists`, `IsUnique` and `AreNotEqual`, are not needed and therefore won't be implemented.

Method `Encode` is again very similar to its counterparts in other encodings. The difference is only an absence of another cycle. See algorithm 13.

Algorithm 13 `Encode`(**`Puzzle`**,`outputFile`)

```
for all cells c in Puzzle do
  ids ← Puzzle.ListNeighbours(id of c)
  add the output of Precedes(id of c, ids) to CNF
end for
for all cells c in Puzzle do
  if c has a non-zero value x then
    add the output of IsEqual(c, x) to CNF
  end if
end for
write CNF to outputFile
```

5. Experiments

In this chapter we aim to present and interpret the results of conducted experiments, which test the efficiency of various encoding techniques described in previous chapters. Individual encodings are compared using several criteria and multiple SAT solvers.

5.1 Comparison criteria

Criteria used to measure the efficiency of individual encodings are summarized in the following list:

- total number of variables;
- total number of clauses;
- size of the encoding, measured in MB;
- the time it takes SAT solver to find a satisfying interpretation, measured in seconds;
- the amount of memory SAT solver uses, measured in MB.

Criteria are interpreted with respect to the size of the problem. In our case it is the number of cells of a puzzle and the number of solutions of a puzzle.

5.2 Methodology

In order to examine the efficiency of encoding techniques we need a sequence of Hidato and Numbrix puzzles comparably difficult and expanding in size. As it turns out, finding a reasonably large puzzle with non-trivial solution is an increasingly hard task. The more pre-filled cells the puzzle has, the more trivial the solution is. On the other hand a lower amount of pre-filled cells means a greater probability of multiple solutions, which conflicts with the definition of Hidato and Numbrix.

To address this issue we conduct two sets of experiments. The first set uses simple puzzles with unique solutions, puzzles for the second set are allowed to have multiple solutions. Both sets of experiments are carried out using several different SAT solvers regularly participating in SAT competitions, namely Lingeling [3], MiniSAT [5] and ProbSAT [1]. As the first three criteria are affected only by puzzle and encoding technique, we will examine them independently and use SAT solvers only to test the fourth and fifth criterion. The first set of experiments investigates their dependency on the number of cells of a puzzle, the second on the number of solutions of a puzzle with a fixed size. We also limit the time SAT solvers run by one hour. This is done in terminal by setting the corresponding parameter of the solver. For details see solver's help page.

The solvers were run on AMD Athlon II Dual-Core M300 2.00 Ghz CPU with 4.00 GB of RAM. The operating system was Linus Mint.

Puzzles used in experiments can be found in folder `puzzles` on the attached CD. Encodings of these puzzles are in `dimacs` and the results of experiments are summarised in folder `experiments`. Source code with installation instruction for each SAT solver is also in corresponding folders.

The results of Binary encoding regarding criteria three, four and five are not included, as the sheer number of clauses prevents SAT solvers from successfully resolving even the smallest of test puzzle. The number of variables and clauses can be found in folder `experiments`.

5.3 Puzzles with unique solutions

For the first set of experiments we utilize square puzzles with a spiral solution starting from the top left corner an ending in the centre. To achieve uniqueness we fill every other cell with a number. This enable us to easily generate arbitrarily large puzzles while preserving comparable difficulty. An example of such a puzzle can be seen in figure 5. Note that this type of puzzle is in fact a Numbrix puzzle, but we will use it as a Hidato puzzle as well.

Experiments were carried out on puzzles of magnitudes $n = 100$ up to $n = 2500$, where n is the number of cells. Puzzles were encoded using an application similar to one described in chapter 4, which can be found on the attached CD in folder `Codec`. The resulting files in DIMACS format were then given as input to SAT solver.

1		3		5		7		9		11
	41		43		45		47		49	
39		73		75		77		79		13
	71		97		99		101		51	
37		95		113		115		81		15
	69		111		121		103		53	
35		93		119		117		83		17
	67		109		107		105		55	
33		91		89		87		85		19
	65		63		61		59		57	
31		29		27		25		23		21

Figure 5: An example of a test puzzle with 121 cells

5.3.1 Number of variables

The values of $|var(UE)|$, $|var(RE)|$ and $|var(TE)|$ were already listed in sections 2.1.4, 2.2.3 and 3.2.5. However, not all experimental results are presented here. They were selected from the low end of the spectrum to demonstrate the differences between individual encodings. All results can be found in folder `experiments` on the attached CD.

For graphical comparison see figure 6.

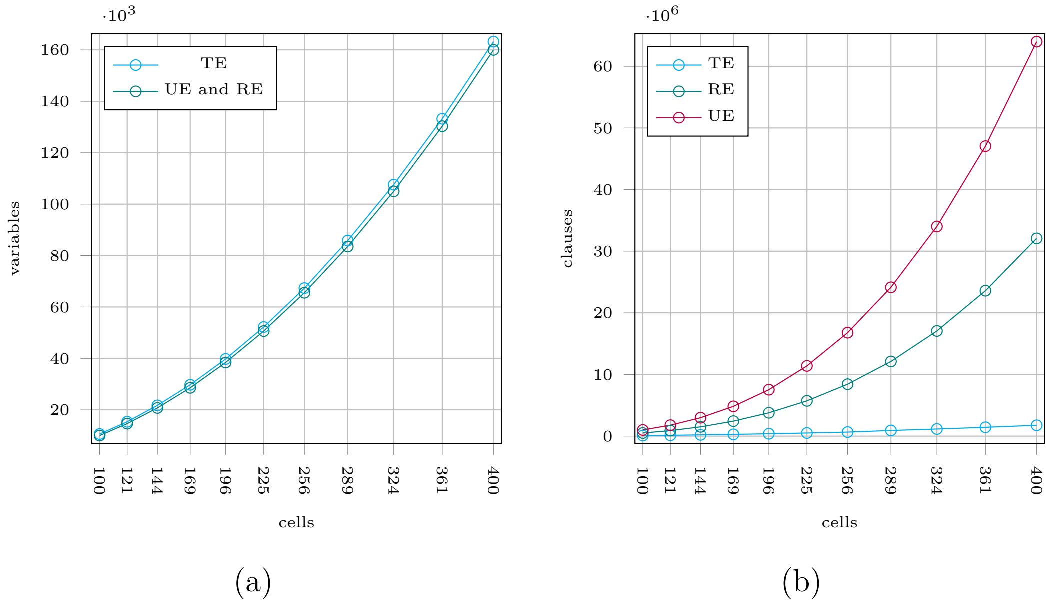


Figure 6: Number of variables (a) and clauses (b) for different encodings

The results indicate a negligible difference between encodings regarding the number of variables. Even though Tseitin encoding uses only $\mathcal{O}(nlog_2n)$ variables to represent all possible values in every cell, the additional variables added during Tseitin transformation more that compensate for this deficiency.

5.3.2 Number of clauses

This criterion is again investigated in sections 2.1.4, 2.2.3 and 3.2.5. The results displayed in figure 6 are again selected from the low end of the spectrum so as to demonstrate the differences between individual encodings. The rest is located in folder `experiments` on the attached CD.

In this criterion Tseitin encoding dominates, followed by Reduced encoding. Unary encoding finishing last. This criterion partially manifests itself also in the following property, the size of formula.

5.3.3 Size of encoding

This criterion measures the amount of space the text file containing the encoding as the output of our encoder takes on disk. Values in graph 7 are in megabytes (10^6 bytes) and are derived from values listed in Windows Explorer or Unix `ls -l` command.

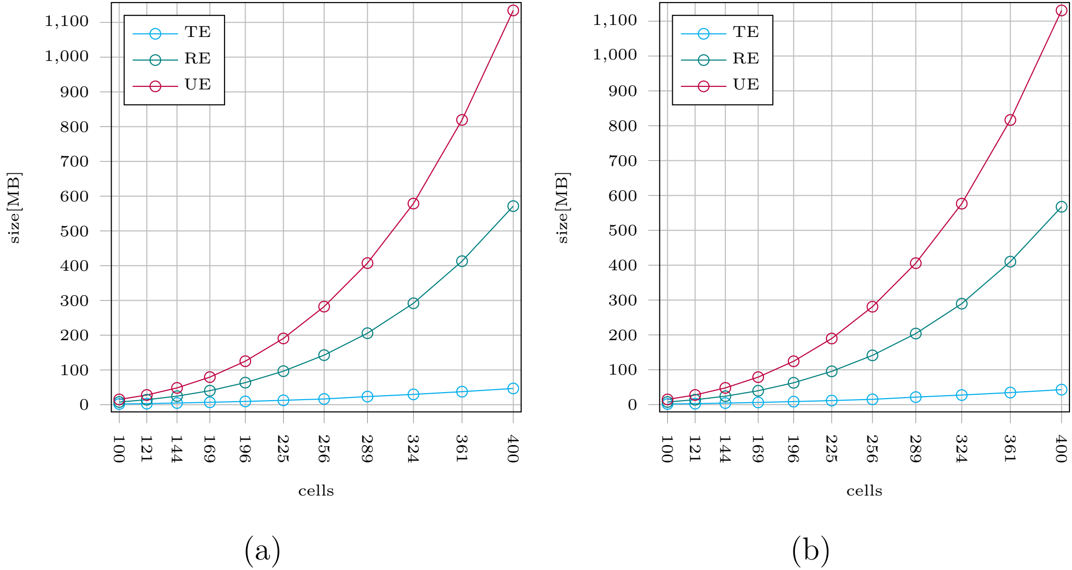


Figure 7: Size of the resulting formula, after applying encodings on Hidato (a) and Numbrix (b) puzzles with unique solution

Notice that the difference between Hidato and Numbrix puzzles are almost non-existent. As we previously stated, the neighbourhood of a cell affects only the size of individual clauses, not their count. The maximum number of neighbouring cells in Hidato and Numbrix is 8 and 4 respectively. Therefore, the affected clause contain at most 4 additional literals.

5.3.4 Time and spatial complexity

Values in figures regarding these two criteria are provided by SAT solvers as part of their output. Absence of a value means that solver failed to resolve the formula by either running out of memory or time. The amount of memory is limited by hardware, time limit is set to one hour.

Lingeling

Lingeling is a SAT solver based on Conflict-Driven Clause Learning (CDCL) algorithm [3], which is in turn based on Davis-Putnam-Logemann-Loveland (DPLL) algorithm.

Given a CNF formula F, DPLL performs a backtrack search where at each step a variable $v \in var(F)$ is selected and either v or $\neg v$ is added to an interpretation I. This addition is called branching. After each branching step DPLL evaluates its logical consequences. If a clause $C \in F$ such that $\{\neg l \mid l \in C\} \subseteq I$ (a conflict) is found, backtracking is executed.

DPLL enhances backtracking by employing *unit propagation* and *pure literal elimination*. In order to explain these concepts, we need to understand what unit clauses and pure literals are.

Unit clause contains only a single literal, i.e. it is of the form $\{l\}$. Unit propagation removes from F all clauses C, such that $l \in C$ and C is not a unit clause. It also removes all occurrences of $\neg l$ from all the clauses. This will leave us with a formula equivalent to F.

Pure literal is a literal l such that $\neg l \notin C$ for all clauses $C \in F$. Clauses containing l can always be made satisfiable after we add l to I. Therefore, they no longer constrain the search and can be eliminated from F while preserving equivalence in a process called pure literal elimination.

CDCL is an improved DPLL algorithm that learns new clauses from conflicts during backtrack search [9].

The result are displayed in figures 8 and 9.

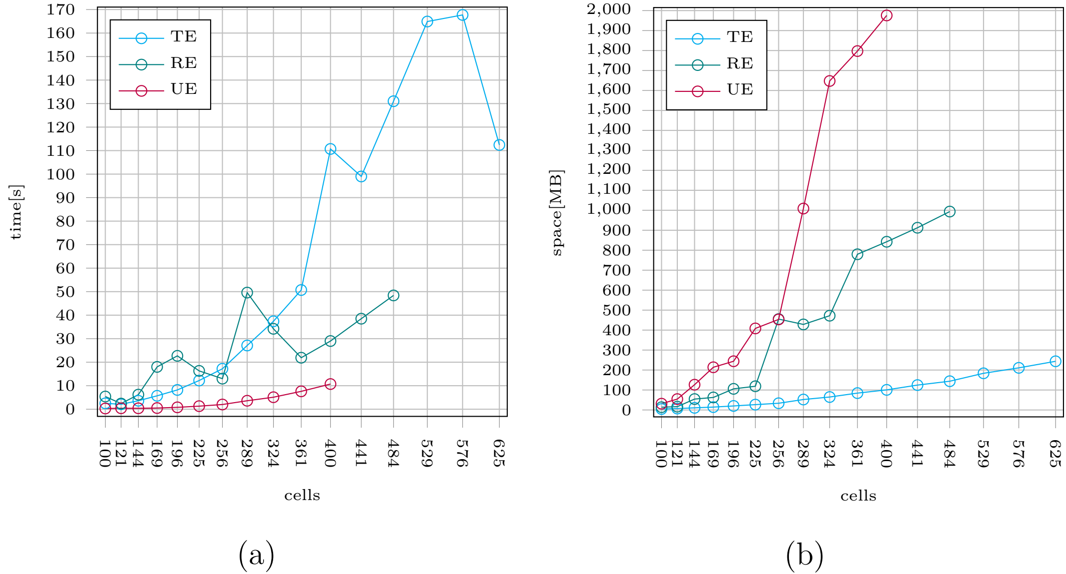


Figure 8: Computation time (a) and spatial complexity (b) of Lingeling for different encodings of Hidato puzzles with unique solution

Concerning time complexity, Unary encoding outperforms both Reduced and Tseitin encoding on Hidato puzzles. On the other hand, Tseitin encoding dominated spatial complexity criterion. It is possible that the more spatially complex an encoding, the less complex it is regarding time. The spatial efficiency of Tseitin and the size of encoding is probably the reason why Lingeling is able to resolve Tseitin encodings of much larger puzzles.

The same can be said about performance of encodings of Numbrix puzzles, whose spatial complexity is almost identical. However, differences in time complexity seem to be only minor, with Reduced encoding being slightly advantageous. Also note that spatial complexity in case of both Hidato and Numbrix correlates with the previous criterion, the size of encoding.

MiniSAT

Minisat is a another CDCL bases SAT solver [5]. We include it to demonstrate that even SAT solvers based on the same algorithm can yield completely different results. The results can be seen in figures 10 and 11.

Here the time complexity correlates with spatial complexity, even though the efficiency of both is generally lower than that of Lingeling. Also the results for Hidato and Numbrix are similar, with Numbrix being faintly less time consuming to solve and space requiring to allocate. Tseitin encoding got the best results, followed by Reduced encoding and Unary encoding. As was the case with Lingeling,

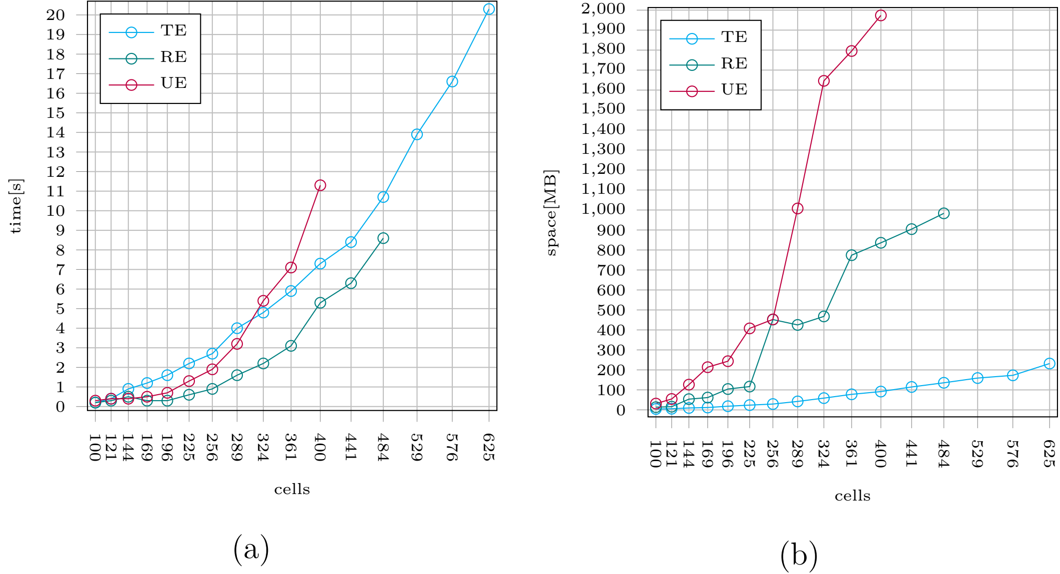


Figure 9: Computation time (a) and spatial complexity (b) of Lingeling for different encodings of Numbrix puzzles with unique solution

spatial complexity of encodings of Hidato and Numbrix puzzles corresponds to their size.

Probsat

ProbSAT is a probability distribution based stochastic local search (SLS) SAT solver [1]. Given a CNF formula F, SLS solvers operate on interpretations $I(F)$ and try to satisfy F by flipping literals, i.e changing $l \in I(F)$ to $\neg l \in I(F)$ or vice versa. Literals to be flipped are selected according to a given heuristic. ProbSAT's heuristic is characterized by the usage of probability distribution function instead of decision procedure.

In figure 12 we present only the time analysis of ProbSAT performance, as ProbSAT does not provide informations about its memory management.

Overall, Reduced encoding is the most successful both in Hidato and Numbrix puzzles.

Performance of Tseitin encoding on large puzzles

As it turns out, Lingeling and MiniSAT are able to find solutions even to puzzles with 1681 cells when they are encoded using tseitin encoding. This is almost four times the size when we compare it to unary encoding and reduced unary encoding. In figures 13 and 14 are the results of such experiments.

5.4 Puzzles with multiple solutions

The experiments presented here use Hidato and Numbrix puzzles with a fixed number of cells. In the first case it is an arbitrary 5×5 Hidato and in the second a 8×8 Numbrix. The difference in size is caused by a higher number of possible solutions of Hidato and Numbrix puzzles with the same grid.

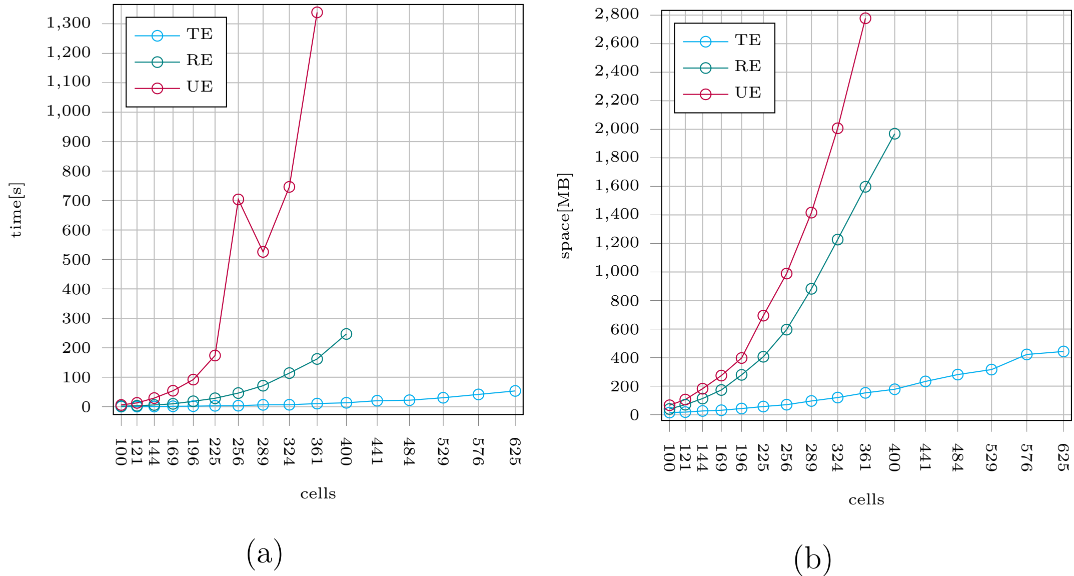


Figure 10: Computation time (a) and spatial complexity (b) of MiniSAT for different encodings of Hidato puzzles with unique solution

Experiments are conducted using the following methodology. We start with both puzzles solved. Then we gradually empty one cell at a time and in each iteration encode the puzzle. Numbers not included in the removal process are the highest and the lowest one. Encoding is given to a SAT solver. After solver finds a satisfying interpretation, it is negated and added as a clause to encoding. This clause ensures that should the solver find another satisfying interpretation, it will not be the same one. This process is repeated until the solver deems the encoding unsatisfiable.

The results reside in folder `experiments` on the attached CD.

5.4.1 Number of variables and clauses, size of formula and spatial complexity

Since the puzzle is fixed, the number of variables is constant and corresponds to the values derived above in section 2.1.4, 2.2.3 and 3.2.5.

The number of clauses is consistently increased by one and can be easily derived by the addition of the number of satisfying interpretations (i.e the number of iterations) to identities mentioned in section 5.3.2.

The size of formula can change significantly, because the number of clauses added to an encoding is equal to the number of solutions to puzzle. However, the rate at which the number of solution increases is fast enough to render the increase in formula size negligible. This is because the amount of time involved in finding all solutions is so great that on the scale of practically conductible experiment, the size of formula is nearly constant.

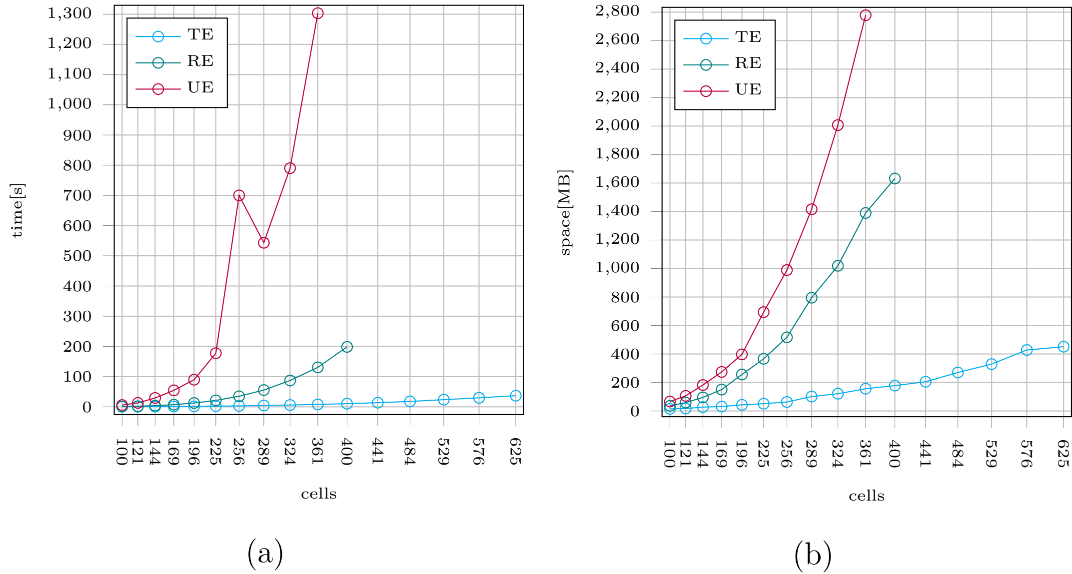


Figure 11: Computation time (a) and spatial complexity (b) of MiniSAT for different encodings of Numbrix puzzles with unique solution

5.4.2 Time complexity

Lingeling

The results of Lingeling are listed in figure 15.

We see that while Tseitin and Reduced encoding perform similarly, Unary encoding is significantly better.

MiniSAT

Figure 16 illustrate the performance of MiniSAT.

Here Unary encoding is again the most efficient.

ProbSAT

The efficiency of different encodings with regard to puzzles with multiple solutions and ProbSAT can be seen in figure 17.

In ProbSAT the Unary encoding is on a par with the Reduced encoding. Tseitin encoding is the worst by a very large margin, especially in Numbrix.

5.5 Discusion

5.5.1 Number of variables, number of clauses, size of encoding and spatial complexity

The results of experiments indicate that the first two criteria, the number of variables and the number of clauses, manifest themselves in the third criterion, the size of encoding, which in turn correlates with the spatial complexity of encodings. This is true for both Hidato and Numbrix if we use Lingeling or

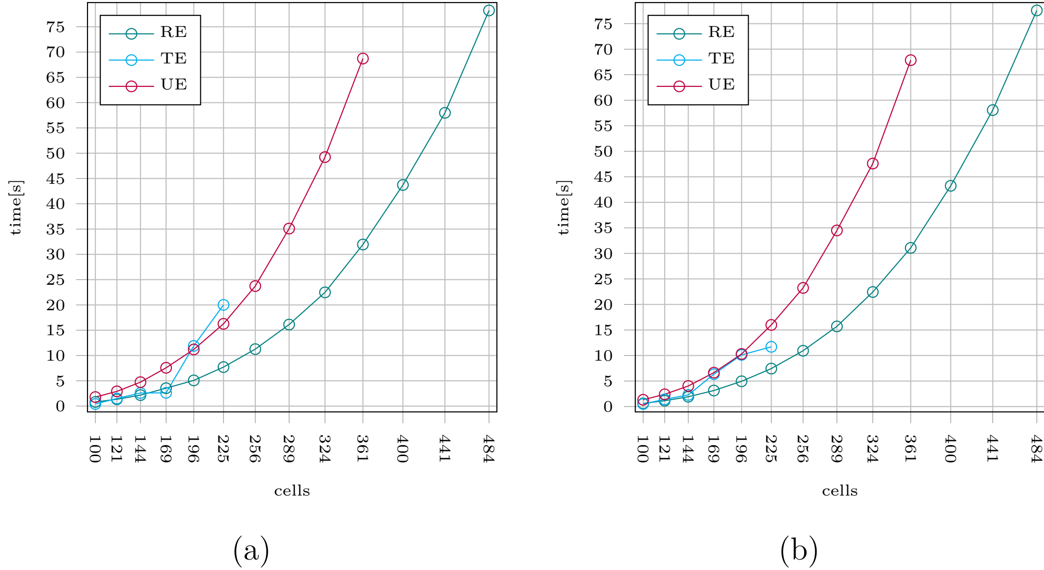


Figure 12: Computation time of ProbSAT for different encodings of Hidato (a) and Numbrix (b) puzzles with unique solution

MiniSAT. The unavailability of this information in case of ProbSAT prevents us from coming to a similar conclusion.

Regarding this criterion, Tseitin encoding is the most efficient in all cases, including small and large Hidato and Numbrix puzzles, Lingeling and MiniSAT. On the other side of the spectrum is Unary encoding, being the most spatially complex.

5.5.2 Time complexity

According to Lingeling, the most time efficient encoding of a unique Hidato puzzle is Unary encoding, while Reduced encoding outperformed the other two when it comes to unique Numbrix. MiniSAT is dominated by Tseitin encoding of both Hidato and Numbrix unique puzzles. ProbSAT is best suited for Reduced encoding of unique puzzles. The comparison of the fastest encodings is illustrated in figure 18.

Regarding puzzles with multiple solution, it is apparent that not all properties of encodings carried over from unique puzzles. The acquired data indicate that Unary encoding produces the best results in terms of time efficiency regardless of solver type or whether we encode Hidato or Numbrix.

5.6 Summary

The fastest way to solve a small Hidato puzzle with unique solution is using Unary encoding and Lingeling, for unique Numbrix it is Reduced encoding with the same SAT solver. Hoverer, the performance of MiniSAT indicate that Tseitin encoding can be used to efficiently encode small Hidato and Numbrix puzzles with unique solutions as well. For larger puzzles with unique solutions the only choice is Tseitin encoding, as SAT solvers fail to provide solutions when using Unary or Reduced encoding. Large Numbrix puzzles with unique solutions are

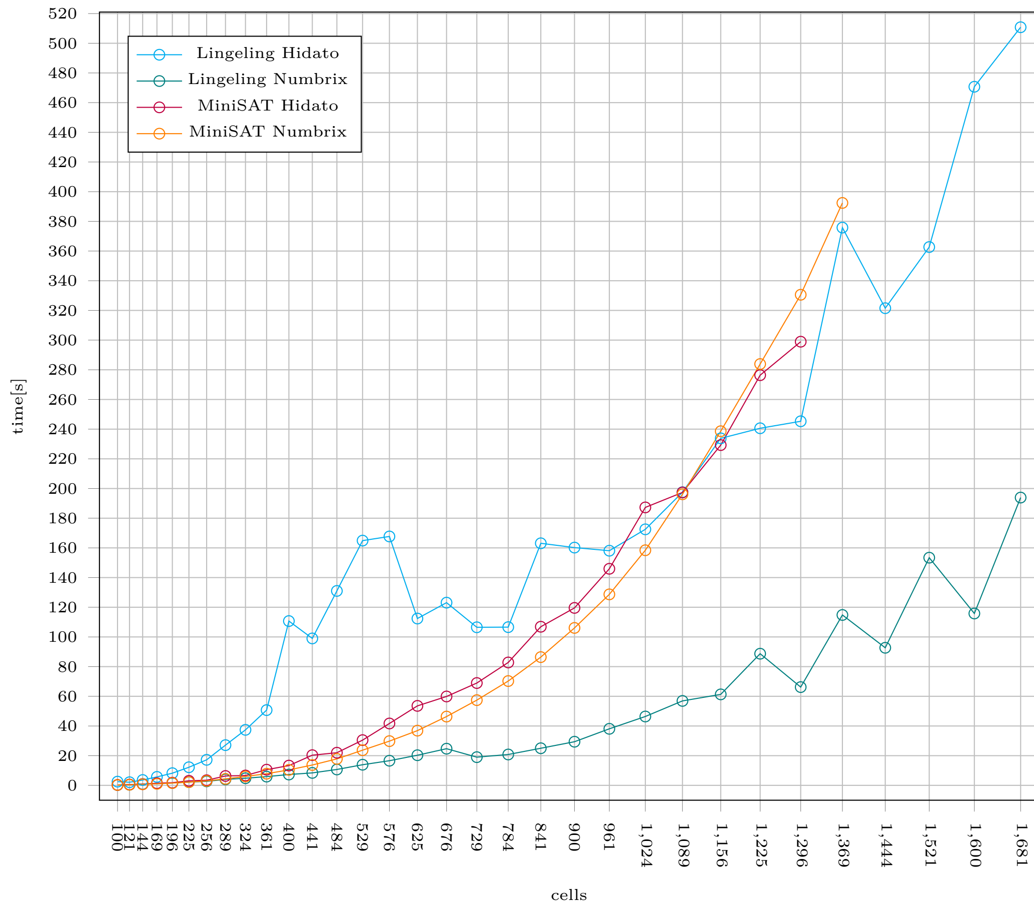


Figure 13: Computation time of Lingeling and MiniSAT for tseitin encoding of Hidato and Numbrix puzzles with unique solutions

best suited for Lingeling, while Lingeling and MiniSAT yield similar results for unique Hidato. ProbSAT is the least efficient when it comes to time complexity of smaller puzzles with unique solutions and it also failed to solve larger ones.

When it comes to puzzles with multiple solutions, the encoding of choice is unanimously the Unary encoding.

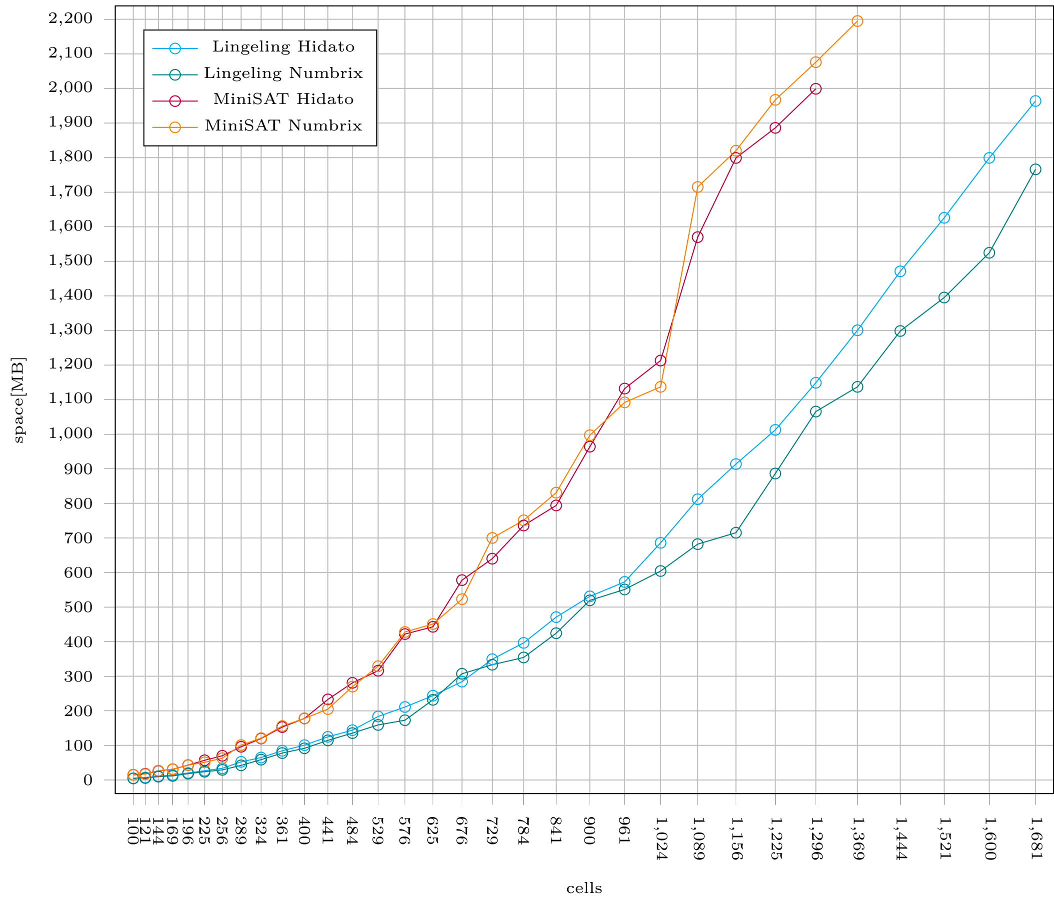


Figure 14: Spatial complexity of Lingeling and Minisat for tseitin encoding of Hidato and Numbrix puzzles with unique solution

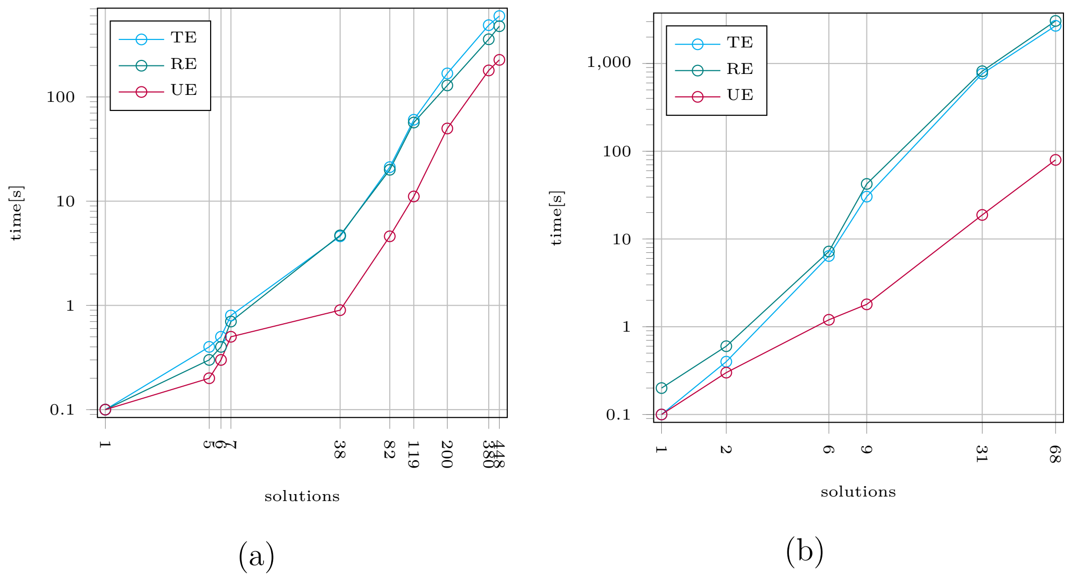
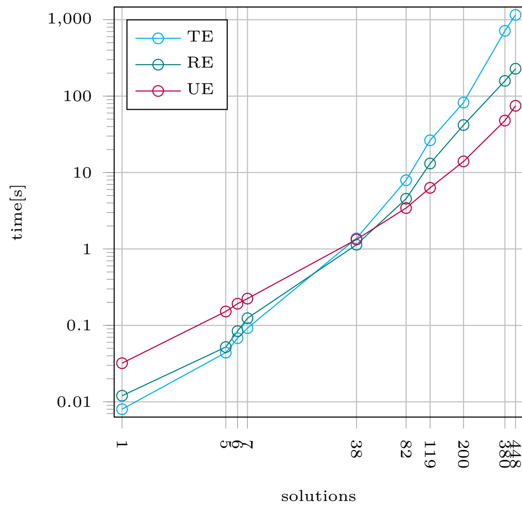
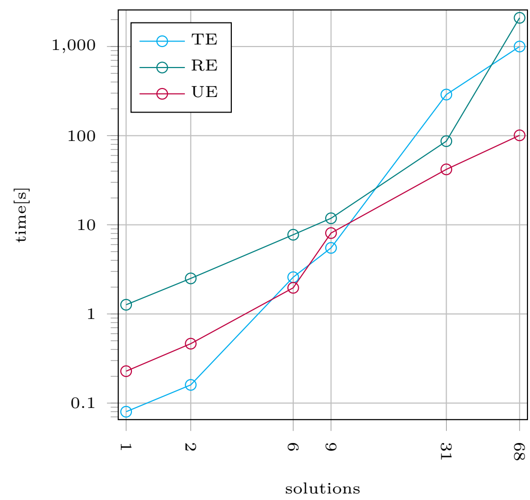


Figure 15: Computation time of Lingeling for different encodings of Hidato (a) and Numbrix (b) puzzles with multiple solutions

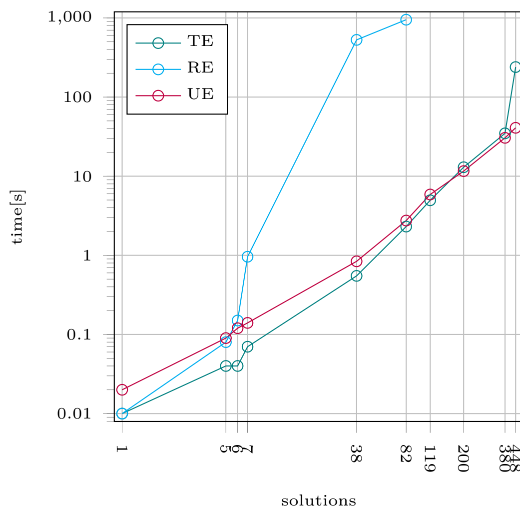


(a)

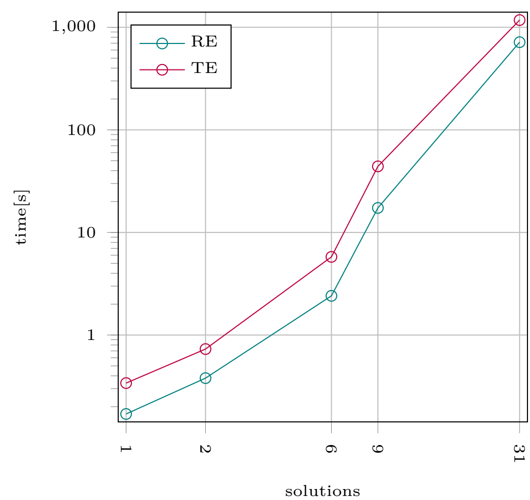


(b)

Figure 16: Computation time of MiniSAT for different encodings of Hidato (a) and Numbrix (b) puzzles with multiple solutions



(a)



(b)

Figure 17: Computation time of ProbSAT for different encodings of Hidato (a) and Numbrix (b) puzzles with multiple solutions

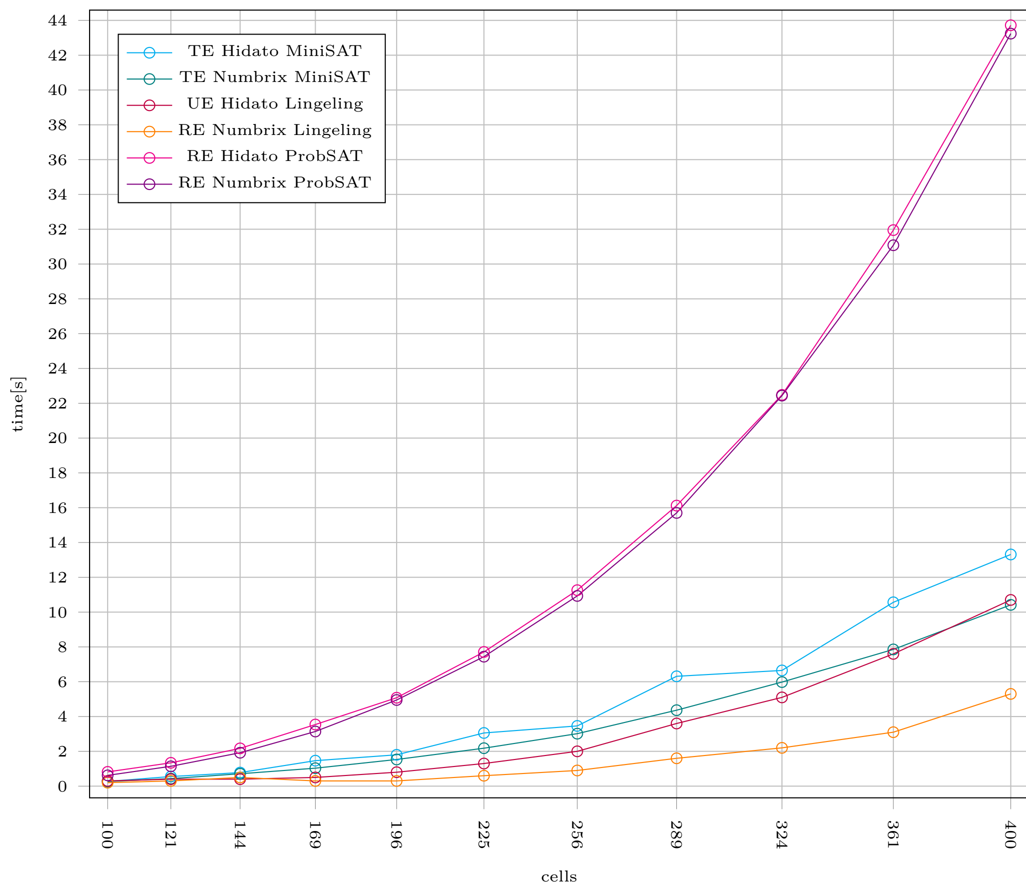


Figure 18: Computation time of the most time efficient encodings of Hidato and Numbrix puzzles

Conclusion

In this thesis we explored the possibility of encoding Hidato and Numbrix puzzles into CNF formulas of various forms and the impact of a particular technique of encoding on the performance of various SAT solvers. The chosen approach was one of minimizing the number of variables and clauses used in encodings in order to maximize the solver's efficiency.

Encodings were classified into two groups, unary encodings and binary encodings, according to how the fact that a certain cell of a puzzle contain a certain number is represented. In the former a variable represents only one value of a cell, in the latter a set of variables characterises cell's value as a binary number. In the first group two encodings were constructed: Unary encoding and Reduced encoding. The same is true for the second group with Binary encoding and Tseitin encoding as its representatives. Validity of all these encodings was examined and confirmed, as we proved that a solution to an encoded puzzle can be constructed from satisfying interpretations of their respective formulas.

We described one of possible implementations of an application designed to encode Hidato and Numbrix puzzles using mentioned encodings, as well as decode satisfying interpretation obtained from the output of SAT solver. The implementation is based on object oriented language.

Encodings were then compared using several criteria, namely the number of variables, clauses, size of formula in MB and time and spacial complexity of a solver's computation, on both Hidato and Numbrix puzzles with unique and multiple solutions. For the last two criteria we used Lingeling, MiniSAT and ProbSAT. These are modern SAT solvers frequently participating in various benchmarks and competitions.

The results indicate that there is no single choice when it comes to the most efficient encoding, as its performance depends heavily on various circumstances, including the size and form of a puzzle and the type of employed SAT solver.

Future work

There are many potential encodings of Hidato and Numbrix puzzles into CNF formulas, some of which are not included in the scope of this thesis. One such encoding is dual encoding. It is based on the combination of our unary and binary encoding, in which we add clauses tying variables from both sets representing corresponding facts. When it is later needed to enforce some condition or restriction, either set of variables can be chosen and a more convenient clauses formed. The foundation of another type of encoding not explored in this thesis is the usage of variables representing intervals. In this technique a variable does not represent the fact that a certain cell contain a certain number. Instead, it represents that the contained number is larger or equal than some other number.

Another way to get the most out of SAT solver in terms of efficiency is acquiring thorough knowledge of its algorithm and devising an encoding exploiting the details of its inner-workings. However, such an encoding will need to evolve with its SAT solver in order to stay competitive.

Also, most solvers have a number of parameters that can be change. These

parameters dictate the behaviour of various algorithms and heuristics underneath and can have a significant impact on solver run time. Despite the obvious advantages, finding better values for parameters can prove to be a very difficult task and the time involved may very well be better spent on encoding.

Bibliography

[1] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2012.

[2] Marzio De Biasi. Hidato is np-complete, 2013. [Online; accessed 23-July-2014 at `http://www.nearly42.org/cstheory/hidato-is-np-complete/`].

[3] Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013*, page 51, 2013.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[5] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[6] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[7] Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. Finding efficient circuits using sat-solvers. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 32–44. Springer, 2009.

[8] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[9] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[10] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[11] Wikipedia. Hidato and numbrix, 2014. [Online; accessed 23-July-2014 at `http://en.wikipedia.org/wiki/Hidato`].