Sudoku, Akari, Arukone: Logical Encodings and Complexity

master thesis in computer science

by

Caroline Terzer

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements
for the degree of Master of Science

supervisor: Univ.-Prof. Dr. Aart Middeldorp,
Institute of Computer Science

Innsbruck, 4 May 2009

Master Thesis

Sudoku, Akari, Arukone: Logical Encodings and Complexity

Caroline Terzer (0016292)
Caroline.Terzer@student.uibk.ac.at

4 May 2009

Supervisor: Univ.-Prof. Dr. Aart Middeldorp

Abstract

In this thesis three pencil-and-paper puzzles are explained and analysed: Sudoku, Akari and Arukone. For each puzzle we present encodings in propositional logic, in pseudo-boolean logic and in SMT. Experimental results for the different encodings and for different solvers are reported. Further we focus on the complexity of these puzzles. The existing NP-completeness proofs for Sudoku and Akari are reviewed and for Arukone, which was not proved to be NP-complete before, a proof is presented.

Acknowledgments

Foremost I would like to thank my supervisor Univ.-Prof. Dr. Aart Middeldorp for his support throughout the past years. He did not only spark my interest in logic with his excellent lectures but also motivated me to immerge deeper into this fascinating world by providing me with interesting new topics for seminars, bachelor projects and for my master thesis. Special thanks go to my whole family for putting me through university and, even more, for being there for me during the ups and downs of my studies. Further I want to express my gratitude to my partner Peter who always was up to occupy himself with any new subject just to assist me with all my questions and problems of a theoretical nature. Final thanks go to the developers of $\mathsf{T\!T\!T_2}$ for providing me with the libraries of $\mathsf{T\!T\!T_2}$.

Contents

1 Introduction

Ever since the broader emergence of Sudoku, paper-and-pencil puzzles enjoy great popularity. Together with this growth in popularity, people started to design individual solvers for almost every puzzle. The idea of such solvers is normally to implement the solving strategies which are applied by humans. A rather different approach to find solutions to puzzles is to encode puzzles in logic and then to use a generic satisfiability solver to compute a solution. The big advantage of this method is that only one solver has to be implemented and increasing the performance of this single solver results in a performance increase for all puzzles. As a price this solver is not able to implement any solving strategies which are specific for a single puzzle but has to be capable of handling very heterogeneous problems. In fact this shortcoming can be partly compensated by a wise construction of the encodings.

In this thesis this approach is applied to three different Japanese puzzles. For each puzzle we consider three different kinds of logic: propositional logic, pseudo-boolean (PB) logic and SMT. Propositional logic provides a very restricted language and therefore the resulting formulas are often large and hard to read. Nevertheless the field of SAT solving has experienced a lot of research during the past decades and therefore modern SAT solvers achieve outstanding results. Pseudo-boolean logic is particularly useful if it is necessary to characterise arithmetic connections. Often it allows a more concise representation of problems than propositional logic. But as Chapter 4 shows there exist classes of problems where the language of PB is not suitable at all, e.g. if there are no arithmetic relations between variables or constraints appear mainly in the shape of implications. Currently there are two approaches to testing a set of PB constraints for satisfiability. The first is to transform the PB constraints to propositional logic and then to use a SAT solver. This method is implemented in `Minisat+` [4]. The second is to try to apply the solving strategies which are well-working for SAT solvers directly to PB constraints. The strategies implemented in the PB solver `Pueblo` [21] are designed according to this principle. The last language which is considered is SMT (satisfiability modulo theories). More precisely SMT is not one specific language but a framework where different logical theories can be used to encode problems. As there is more freedom to chose a theory which goes with the properties of the original problem, SMT often provides very convenient means to construct small formulas which are also human-readable. As a drawback, our experiments showed that for all encoded puzzles SMT performs dramatically worse than propositional logic and at best is equally fast than PB logic.

In Chapter 2 we deal with the problem of converting propositional formulas to CNF. Chapter 3 is entirely dedicated to Sudoku. We introduce and comment different encodings, present experimental results and finally illustrate the NP-

completeness proof by Seta and Yato [27]. Chapter 4 treats the puzzle Akari and similar to Chapter 3 we present encodings, experimental results and an NP-completeness proof by McPhail [17]. Chapter 5 handles the puzzle Arukone. To encode Sudoku and Akari it suffices completely to define a series of local constraints. However encoding Arukone is more demanding as it requires to find a way to represent connectivity of single cells in a grid. In this chapter we present two different approaches for this problem. Further we present an NP-completeness proof of Arukone. Finally Chapter 6 deals with the description of a program which transforms instances of Sudoku, Akari and Arukone into formulas according to the presented encodings. Subsequent to the generation of a formula the program is able to run a solver on the generated formula and to output a solution to the input puzzle or to declare it unsolvable.

2 Preliminaries

Most SAT solvers expect the input to be in conjunctive normal form (CNF). There exists a standard technique to transform propositional formulas into CNF which relies on the repeated application of various propositional laws. A definition of this approach is for example given in [10]. However this method is problematic as it may cause an exponential blow-up of the initial formula. An alternative method to derive a CNF is the application of Tseitin's transformation [25]. The size of the resulting CNF depends linearly on the size of the original formula but in turn requires the introduction of additional variables. Moreover a formula and the CNF generated by Tseitin's transformation are not semantically equivalent but are equisatisfiable.

Definition 2.1. Two formulas ϕ and ψ are *equisatisfiable* if ϕ is satisfiable if and only if ψ is satisfiable.

For a propositional formula ϕ Tseitin's transformation defines an equisatisfiable formula $T(\phi)$ as follows:

Definition 2.2.

$$T(\phi) = p_\phi \wedge \bigwedge_{\substack{\psi \in NASub(\phi) \\ \psi = \psi_1 * \psi_2}} (p_\psi \leftrightarrow p_{\psi_1} * p_{\psi_2}) \wedge \bigwedge_{\substack{\psi \in NASub(\phi) \\ \psi = \neg\psi_1}} (p_\psi \leftrightarrow \neg p_{\psi_1})$$

where $NASub(\phi)$ denotes the set of all non-atomic subformulas of ϕ and $*$ stands for a binary connective. If ψ_1 is an atom then p_{ψ_1} stands for ψ_1 otherwise it is a fresh variable. The same holds for p_{ψ_2} and for p_ϕ.

The actual CNF for a formula ϕ is derived from $T(\phi)$ by applying the standard transformation to each equivalence. For each of these subformulas the standard transformation yields at most four clauses [28].

In 1986 Greenbaum and Plaisted [19] presented a structure-preserving improvement of Tseitin's transformation. If a formula is in negated normal form (NNF), i.e., negations appear only next to atoms, it suffices to use an implication instead of an equivalence in the definition of $T(\phi)$. This approach yields less clauses but it may not increase performance if the formula is not in NNF already.

In the same year Blair *et al.* [9] proposed another refinement of Tseitin's transformation which allows to reduce the number of new variables. This is achieved by introducing just one variable for disjunctions and conjunctions independent of the number of $\vee$ or $\wedge$ connectives. In fact for a disjunction even no new variable may be introduced but the disjunction itself is reused. Moreover they apply the idea of Greenbaum and Plaisted to further reduce the resulting CNF.

Most of the encodings which are presented in this thesis naturally yield a formula in CNF or can be easily transformed to CNF by the application of the standard transformation. The only formulas where a non-standard approach is necessary appear in Chapter 5. Here the transformation according to Blair *et al.* is applied.

3 Sudoku

Sudoku is one of the most famous paper-and-pencil puzzles. Invented by Howard Garns and first published in 1979 under the name Number Place it achieved tremendous popularity after the Japanese company Nikoli [5] started publishing it in 1984 [20]. Normally Sudoku puzzles which appear in newspapers and books are in the form of a 9×9 grid and have a unique solution. Depending on the number of initial entries and the deduction rules which have to be applied in order to solve a puzzle it is assigned a certain difficulty level. Often these published Sudoku puzzles can be solved by applying only reasoning, i.e., guessing and backtracking is not necessary. Due to the popularity of Sudoku a lot of different solution strategies have been developed and documented and there exists already a large number of specialised Sudoku solvers implementing these strategies.

In this chapter we introduce a formal definition of Sudoku generalised to $n^2 \times n^2$ grids on the basis of which we present encodings of Sudoku in three different logical languages: propositional logic, pseudo-boolean constraints and SMT. Further the different translations are compared and their weaknesses and strengths are pointed out. The chapter is concluded with an NP-completeness proof for Sudoku.

	2	4	
	4		2
			3

Figure 3.1: 4×4 Sudoku grid with unique solution.

3.1 Problem Definition

Sudoku is played on a grid consisting of $n^2 \times n^2$ cells, an example grid is given in Figure 3.1. The goal is to fill the cells of the grid with numbers such that the following conditions are fulfilled:

- in each cell there is exactly one number,
- in each row each number from 1 to n^2 appears exactly once,
- in each column each number from 1 to n^2 appears exactly once,

- the whole $n^2 \times n^2$ grid can be divided into n^2 subgrids of size $n \times n$, in each of these subgrids each number from 1 to n^2 appears exactly once.

Some of the cells may contain predefined entries in order to make it easier to find a solution or to define a unique solution. So as to make it possible to subdivide a Sudoku grid in equally sized, quadratic subgrids as required by the last condition the number of rows and columns has not only to be equal but to be the power of a natural number.

3	2	4	1
1	4	3	2
4	1	2	3
2	3	1	4

Figure 3.2: Solution to the Sudoku grid of Figure 3.1.

In this and the following chapters we use the notation (i, j) where i denotes the index of the row and j the index of the column to refer to a single cell of a grid. The indices of rows and columns start in the left upper corner of the grid.

Example 3.1. Consider the Sudoku grid as given in Figure 3.1. Using the predefined entries the number which belongs to cell $(1, 4)$ can be identified in the following way: by the rule that in each row each number appears exactly once we can exclude 2 and 4 from the set of possible entries, by the corresponding rule for columns we know that 3 is also not possible in $(1, 4)$ and hence the only valid choice for cell $(1, 4)$ is 1. Now it follows immediately that cell $(4, 4)$ contains a 4 as this is the only number which is still missing in column 4. Similarly we know that the only number which fits in cell $(1, 1)$ is 3 as it is the last missing number in the first row. Using similar deduction steps the grid can be completed. For this example grid the final solution which is shown in Figure 3.2 can be computed without guessing.

Now that we have introduced the puzzle in an informal way and are familiar with the rules by doing an example we are ready to give a formal definition of Sudoku.

Definition 3.2. For $n \in \mathbb{N}$ with $n \geq 2$ an $n^2 \times n^2$ *Sudoku grid* is a function $S : N \times N \to N \cup \{w\}$ with $N = \{x \mid 1 \leq x \leq n^2\}$. n is called *dimension* of the Sudoku grid and w refers to a blank entry. A *solution* to a Sudoku grid S is a function $S' : N \times N \to N$ such that

1. if $S(i, j) \neq w$ then $S'(i, j) = S(i, j)$,
2. $\forall m \in N,\ \forall i \in N,\ \exists j \in N\colon S'(i, j) = m$,
3. $\forall m \in N,\ \forall j \in N,\ \exists i \in N\colon S'(i, j) = m$,
4. $\forall m \in N,\ \forall\ 0 \leq i, j \leq n - 1,\ \exists\ 1 \leq k, l \leq n\colon S'(i * n + k, j * n + l) = m$.

3.2 Propositional Logic

In this section two different translations from Sudoku to a formula in propositional logic are presented. For any translation the first step is to fix a set of variables and to determine their meaning. For each cell (i, j) in an $n^2 \times n^2$ Sudoku grid and each number m from 1 to n^2 a variable of the shape $C_{i,j,m}$ is introduced. If the value of the variable $C_{i,j,m}$ is true then the number m is in cell (i, j) otherwise it isn't. In total this yields $n^2 * n^2 * n^2$ variables which are necessary to encode the Sudoku grid.

The second step is to translate each rule of the puzzle. The following propositional formulas suffice to express the conditions that have to hold for a solution to an $n^2 \times n^2$ Sudoku grid S:

- each cell contains at most one number:

  $\bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigwedge_{v=1}^{n^2-1} \bigwedge_{w=v+1}^{n^2} (\neg C_{i,j,v} \vee \neg C_{i,j,w})$,

- each number appears at least once in each row:

  $\bigwedge_{i=1}^{n^2} \bigwedge_{v=1}^{n^2} \bigvee_{j=1}^{n^2} C_{i,j,v}$,

- each number appears at least once in each column:

  $\bigwedge_{j=1}^{n^2} \bigwedge_{v=1}^{n^2} \bigvee_{i=1}^{n^2} C_{i,j,v}$,

- each number appears at least once in each $n \times n$ subgrid:

  $\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n-1} \bigwedge_{v=1}^{n^2} \bigvee_{k=1}^{n} \bigvee_{l=1}^{n} C_{i*n+k,j*n+l,v}$,

- $\forall i \in N$, $\forall j \in N$ with $S(i, j) \neq w$ add $C_{i,j,S(i,j)}$ to the set of clauses.

This encoding is minimal in the sense that no clause can be omitted. If any clause was left out than a solution to this formula probably wouldn't correspond to a valid solution of S. For each cell we need $n^2 * \frac{1}{2}(n^2 - 1)$ binary clauses to guarantee that it contains at most one number and as there are n^4 different cells we get $n^4 * n^2 * \frac{1}{2}(n^2 - 1)$ binary clauses for the first statement. For each of the remaining properties we need n^4 clauses of arity n^2 which makes $3 * n^4$ clauses. The number of single literal clauses depends of course on the number of initial entries in S but can not be more than n^2.

Another minimal encoding can be constructed by using the same set of variables but the requirements of having exactly one number per cell and all numbers from 1 to n^2 per row, column and subgrid is expressed by using an "at least" clause for each cell and "at most" clauses for rows, columns and subgrids – exactly the other way around as in the first minimal encoding:

- each number appears at least once in each cell:

  $\bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigvee_{v=1}^{n^2} C_{i,j,v}$,

- each number appears at most once in each row:

  $\bigwedge_{i=1}^{n^2} \bigwedge_{v=1}^{n^2} \bigwedge_{j=1}^{n^2-1} \bigwedge_{k=j+1}^{n^2} (\neg C_{i,j,v} \vee \neg C_{i,k,v})$,

- each number appears at most once in each column:
  $\bigwedge_{j=1}^{n^2} \bigwedge_{v=1}^{n^2} \bigwedge_{i=1}^{n^2-1} \bigwedge_{k=i+1}^{n^2} (\neg C_{i,j,v} \vee \neg C_{k,j,v})$,
- each number appears at most once in each $n \times n$ subgrid:
  $\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n-1} \bigwedge_{v=1}^{n^2} \bigwedge_{k=1}^{n} \bigwedge_{l=1}^{n-1} \bigwedge_{m=l+1}^{n} (\neg C_{i*n+k,j*n+l,v} \vee \neg C_{i*n+k,j*n+m,v})$
  $\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n-1} \bigwedge_{v=1}^{n^2} \bigwedge_{k=1}^{n-1} \bigwedge_{l=1}^{n} \bigwedge_{m=k+1}^{n} \bigwedge_{o=1}^{n} (\neg C_{i*n+k,j*n+l,v} \vee \neg C_{i*n+m,j*n+o,v})$,
- $\forall i \in N$, $\forall j \in N$ with $S(i,j) \neq w$ add $C_{i,j,S(i,j)}$ to the set of clauses.

This encoding produces more clauses than the first one, but as most of them are binary this may be beneficial due to the fact that they could allow more unit propagation than the first minimal encoding. For the condition that in every cell there has to be at least one number n^4 clauses of arity n^2 are needed. The other three properties are encoded using binary clauses. We have $n^4 * n^2 * \frac{1}{2}(n^2-1)$ clauses for each property which in total yields $3 * n^6 * \frac{1}{2}(n^2-1)$ binary clauses. Finally an unknown number of unary clauses is added to the formula.

If we compare these two encodings in terms of computation time it turns out that a SAT solver performs dramatically worse on the first encoding than on the second. On the first sight this might seem strange as one would expect the smaller formula, as produced by the first encoding, to be solved faster. The weakness of this formula is that the capability of deriving new information from existing information is rather low. A SAT solver does this inference of new information by applying unit propagation which is commonly implemented rather efficient. Together with the fact that no guessing is involved in unit propagation it allows the fast derivation of compulsory information. The following example points out the shortcomings of both encodings regarding unit propagation.

Example 3.3. Consider the 4×4 Sudoku grid from Figure 3.1. The first implication we observed in Example 3.1 is that the only valid choice for cell $(1,4)$ is 1. At best we expect that an encoding is able to make such easy implications as well. In our encodings the placement of number 1 in cell $(1,4)$ is represented by the fact that the variable $C_{1,4,1}$ is true. In both encodings $C_{1,4,1}$ is not a unary clause hence we want to derive it using unit propagation.

Concerning the first encoding there are three clauses were $C_{1,4,1}$ appears positively and therefore could be assigned to true by unit propagation:

- $C_{1,1,1} \vee C_{1,2,1} \vee C_{1,3,1} \vee C_{1,4,1}$ (1 appears at least once in row 1),
- $C_{1,4,1} \vee C_{2,4,1} \vee C_{3,4,1} \vee C_{4,4,1}$ (1 appears at least once in column 4),
- $C_{1,3,1} \vee C_{1,4,1} \vee C_{2,3,1} \vee C_{2,4,1}$ (1 appears at least once in the subgrid starting at cell $(1,3)$).

If we consider the first clause we need to have that $C_{1,1,1}$, $C_{1,2,1}$ and $C_{1,3,1}$ are false in order to derive that $C_{1,4,1}$ has to be true. $\neg C_{1,2,1}$ and $\neg C_{1,3,1}$ are found by unit propagation on the two binary clauses

- $\neg C_{1,2,1} \vee \neg C_{1,2,2}$ ($C_{1,2,2}$ is true as 2 is in $(1,2)$),

- $\neg C_{1,3,1} \vee \neg C_{1,3,4}$ ($C_{1,3,4}$ is true as 4 is in $(1,3)$).

Anyway it is not possible to derive that $C_{1,1,1}$ is also false. Likewise the other two 4-ary clauses containing $C_{1,4,1}$ are not appropriate to derive this easy information which can be found by a human immediately.

Using the second encoding unit propagation is able to show that $C_{1,4,1}$ has to be true. There is only one clause which allows to imply $C_{1,4,1}$, namely $C_{1,4,1} \vee C_{1,4,2} \vee C_{1,4,3} \vee C_{1,4,4}$. That all variables except the desired one are false is implied by the following binary clauses:

- $\neg C_{1,4,2} \vee \neg C_{1,2,2}$ (2 at most once in row 1),
- $\neg C_{1,4,3} \vee \neg C_{3,4,3}$ (3 at most once in column 4),
- $\neg C_{1,4,4} \vee \neg C_{1,3,4}$ (4 at most once in column 1).

To illustrate a drawback of the second encoding we consider the Sudoku grid given in Figure 3.3. The immediate implication in this grid is that in cell $(3,2)$ there has to be a 4. If we want to derive this implication by unit propagation there is only one clause which could possibly be useful to get $C_{3,2,4}$ which is $C_{3,2,1} \vee C_{3,2,2} \vee C_{3,2,3} \vee C_{3,2,4}$. But as there is no 2 and no 3 in column 2 or row 3 we can not make any implication using this clause. A clause that demands that there is at least one 4 in column 2 as generated by the first encoding would help.

4	1		
		4	
			4

Figure 3.3: 4×4 Sudoku grid.

The latter example shows that it could be advantageous to put both encodings together to one extended Sudoku encoding with clauses which are redundant but allow more implications. This approach is also recommended in [16] and supported by the experimental results given in Section 3.5.

As a final remark note that a SAT solver typically takes its input in conjunctive normal form. Fortunately the presented encodings already create formulas in CNF and therefore no further transformation effort is necessary.

3.3 Pseudo-Boolean Logic

For the PB encoding we may use the same number of variables with the same meaning as in the propositional encoding. Hence we have again n^6 variables of the form $C_{i,j,m}$ for an $n^2 \times n^2$ Sudoku grid. In comparison to propositional logic, PB constraints allow a very compact representation of the properties of a correctly filled out Sudoku puzzle:

- each number appears exactly once in each cell:
  $\forall i \in N,\ \forall j \in N\colon\ \sum_{m=1}^{n^2} C_{i,j,m} = 1,$
- each number appears exactly once in each row:
  $\forall i \in N,\ \forall m \in N\colon\ \sum_{j=1}^{n^2} C_{i,j,m} = 1,$
- each number appears exactly once in each column:
  $\forall j \in N,\ \forall m \in N\colon\ \sum_{i=1}^{n^2} C_{i,j,m} = 1,$
- each number appears exactly once in each $n \times n$ subgrid:
  $\forall\ 0 \le i,j \le n-1,\ \forall m \in N\colon\ \sum_{k=1}^{n} \sum_{l=1}^{n} C_{i*n+k,j*n+l,m} = 1,$
- $\forall j \in N,\ \forall m \in N$ with $S(i,j) \neq w\colon\ C_{i,j,S(i,j)} = 1.$

The encoding of a Sudoku puzzle of dimension n needs $4 * n^4$ PB constraints to represent the rules and a changing number of constraints to store the predefined entries.

It has to be mentioned that a strict definition of PB constraints does not comprise equalities as generated by the latter encoding but allows only inequalities. Nevertheless equalities are often considered PB constraints for reasons of convenience. In our Sudoku encoding the additional use of equalities instead of using only inequalities allows the expression of the "exactly one number" property in a very concise way, i.e, it is not necessary to split all rules into an "at most" and an "at least" part as it has to be done for the propositional encoding. Moreover this encoding approach is reasonable as both PB solvers which were used in the experiments accept equalities as input.

3.4 SMT

To define an SMT encoding we first have to decide upon an appropriate theory. For Sudoku it seemed suitable to use the theory of linear arithmetic or a subset of linear arithmetic. The SMT solver `Yices` [6] which was used in our experiments decides the theory modulo which satisfiability is computed on its own if the input is specified in the `Yices` input format. If the input formula is in SMT-LIB format [3] a theory has to be explicitly specified. The main difference in comparison to PB constraints is that in the theory of linear arithmetic variables may range over natural or rational numbers whereas in PB variables are boolean. Moreover we are not only allowed to use linear equalities and inequalities but also to combine them with boolean connectives like $\wedge$ and $\vee$. It is easy to see that PB constraints are a subset of all valid expressions which can be formed with the theory of linear arithmetic. The immediate consequence is that the PB encoding presented in the former section is also an SMT encoding of Sudoku. But as linear arithmetic allows integer variables an SMT encoding with less variables can be constructed. The set of variables for an $n^2 \times n^2$ Sudoku grid consists of n^4 variables of the form $C_{i,j}$. In the encoding the value of a variable $C_{i,j}$ is restricted to a value in-between 1 and n^2. Given a solution

of an SMT formula which is the encoding of a Sudoku puzzle the value of $C_{i,j}$ indicates the number which is located in cell (i, j) in the completed puzzle.

A minimal encoding for an $n^2 \times n^2$ Sudoku grid is constructed in the following way:

- each number appears at most once in each row:

  $\forall i \in N,\ \forall\ 1 \leq j \leq n^2 - 1,\ \forall\ j + 1 \leq k \leq n^2$: $C_{i,j} \neq C_{i,k}$,

- each number appears at most once in each column:

  $\forall j \in N,\ \forall\ 1 \leq i \leq n^2 - 1,\ \forall\ i + 1 \leq k \leq n^2$: $C_{i,j} \neq C_{k,j}$,

- each number appears at most once in each $n \times n$ subgrid:

  $\forall\ 0 \leq i, j \leq n - 1,\ \forall\ 1 \leq k \leq n,\ \forall\ 1 \leq l < m \leq n$: $C_{i*n+k,j*n+l} \neq C_{i*n+k,j*n+m}$,

  $\forall\ 0 \leq i, j \leq n-1,\ \forall\ 1 \leq k < m \leq n,\ \forall\ 1 \leq l, o \leq n$: $C_{i*n+k,j*n+l} \neq C_{i*n+m,j*n+o}$,

- $\forall i \in N,\ \forall j \in N$ with $S(i, j) \neq w$: $C_{i,j} = S(i, j)$.

It remains to guarantee that the value of each variable must not be larger than n^2 and not less than 1. In the language of the SMT solver `Yices` this can be achieved by defining a subtype of the natural numbers and assigning this subtype to each variable. For example for a 4×4 Sudoku grid this looks as follows:

```
(define-type num (subtype (n::int) (and (>= n 1) (<= n 4))))
(define C11::num)
(define C12::num)

        ⋮

(define C44::num)
```

Interestingly the experiments showed that `Yices` performs better if the range of the variables is not only restricted by the definition of the corresponding type but by an explicit declaration in the form of constraints:

- $\forall i \in N,\ \forall j \in N$: $\bigvee_{m=1}^{n^2} C_{i,j} = m$.

Similar to the minimal propositional encodings this SMT encoding yields rather bad results. If it is used to transform a Sudoku grid of dimension 6 with a usual number of predefined entries `Yices` is not able to find a solution in reasonable time. Again the results are improved if redundant constraints are added in order to have additional information. Using the very same idea as in the propositional encoding we get the redundant constraints to express the "exactly one" requirement by adding the missing "at least" and "at most" constraints to the encoding. The property that there is exactly one number in each cell is guaranteed by the fact that for each cell there is only one variable and that a variable has exactly one value:

- each number appears at least once in each row:

  $\forall m \in N,\ \forall i \in N\colon \bigvee_{j=1}^{n^2} C_{i,j} = m,$

- each number appears at least once in each column:

  $\forall m \in N,\ \forall j \in N\colon \bigvee_{i=1}^{n^2} C_{i,j} = m,$

- each number appears at least once in each $n \times n$ subgrid:

  $\forall m \in N,\ \forall\ 0 \leq i, j \leq n-1\colon \bigvee_{k=1}^{n} \bigvee_{l=1}^{n} C_{i*n+k,j*n+l} = m.$

The minimal encoding yields $3 * n^4 * \frac{1}{2}(n^2 - 1)$ constraints. If the definition of the range is given as constraints then additional n^4 constraints are necessary. Finally the extension adds another $3 * n^4$ redundant constraints.

3.5 Experimental Results

The Sudoku grids corresponding to the test results shown in Tables 3.1, 3.2 and 3.3 were randomly generated and have all a unique solution. For each gridsize – from 9×9 up to 36×36 – 10 puzzles were tested and the results are give in terms of the arithmetic mean. In Table 3.1 the computation results

	Grid size	Minimal I	Minimal II	Minimal I+II
Decisions		44829.10	27.80	2.90
Conflicts	9×9	23850.70	12.50	0.70
CPU time		0.56 s	0.01 s	0.01 s
Decisions		~ 3 m	565.00	54.90
Conflicts	16×16	~ 4 m	271.60	23.40
CPU time		1826.02 s	0.08 s	0.07 s
Decisions			77072.60	337.20
Conflicts	25×25	>1 h	41845.90	97.90
CPU time			22.69 s	0.34 s
Decisions				405.40
Conflicts	36×36	>1 h	>1 h	121.50
CPU time				1.37 s

Table 3.1: Performance of `Minisat2` on Sudoku grids with unique solution using different SAT encodings.

for the 3 different propositional encodings are given. To test for satisfiability the SAT solver `Minisat2` [4] was used. As pointed out in Section 3.2 the minimal encodings on their own are less powerful in comparison to a formula consisting of both encodings together. If we consider the results for the Sudoku grids of dimension 3 in Table 3.1 then there is no big difference between the single encodings in terms of computation time. But concerning the number of conflicts and decisions the differences are already outstanding. A small number of conflicts is preferable as the cause of each conflict was a wrong guess and as consequence backtracking is necessary which makes the whole process slow as

it corresponds to branching in the decision tree. Especially the first minimal encoding which is also the shortest concerning the number of clauses produces a large number of conflicts and flops even at rather small grids of dimension 4 where the other two encodings yield satisfactory results. If we consider the larger Sudoku grids of dimension 6 then the correlation between number of conflicts and CPU time becomes even more clear. Using only the minimal encodings after one hour `Minisat2` still had not found any result whereas a solution to the combination of both encodings could be found within seconds. Of course not only the number of conflicts but also the number of variables which appear in a formula have a big impact on the computation time.

	Grid size	`Minisat+`	`Pueblo`
Decisions	9×9	3.50	–
Conflicts		0.90	–
CPU time		0.01 s	0.02 s
Decisions	16×16	15.70	–
Conflicts		5.60	–
CPU time		0.05 s	0.15 s
Decisions	25×25	399.30	–
Conflicts		143.20	–
CPU time		0.23 s	0.76 s
Decisions	36×36	484.30	–
Conflicts		111.90	–
CPU time		0.54 s	3.34 s

Table 3.2: Performance of `Minisat+` and `Pueblo` on Sudoku grids with unique solution using the PB encoding.

Table 3.2 shows the computation results for the two PB solvers `Minisat+` and `Pueblo`. Unfortunately it was not possible to get the number of conflicts and decisions `Pueblo` needs in the solving process. Even on the larger benchmarks both solvers are rather fast although `Minisat+` performs a bit better throughout all sizes of input files. The working principle of `Minisat+` is to translate the PB constraints into CNF and then to take a version of the SAT solver `Minisat` as back end to find a solution to the translated set of PB constraints. All the more surprising that despite of the translation `Minisat+` is faster than `Minisat2`. The reason could be that `Minisat+` already optimises the resulting propositional formula during the transformation process. Nevertheless for the tested instances the differences between the performance of the two PB solvers and the SAT solver are rather small.

Table 3.3 illustrates the experimental results for the SMT solver `Yices` using the SMT encoding on one and the PB encoding on the other hand. The experiments show that it is not suitable to use the PB encoding on the SMT solver, it performs equally bad than the first propositional encoding. In comparison to the results of the PB and SAT solvers also the SMT encoding together with `Yices` performs worse than any other combination of solver and encoding.

	Grid size	PB encoding	SMT encoding
CPU time	9×9	0.45 s	0.16 s
CPU time	16×16	112.92 s	0.84 s
CPU time	25×25	> 1 h	15.82 s
CPU time	36×36	> 1 h	45.69 s

Table 3.3: Performance of `Yices` on Sudoku grids with unique solution using the SMT and the PB encoding.

Finally the performance of each solver on an empty Sudoku grid was tested. The idea is to evaluate how the solvers perform if there is no initial information to start from and to which unit propagation may be applied. For the experiments the combination of both minimal Sudoku encodings was taken as propositional encoding. The results which are given in Table 3.4 show that `Minisat2` clearly outperforms the other solvers. Only `Minisat2` is powerful enough to solve puzzles of dimension 6 where a larger amount of guessing is necessary within seconds. `Pueblo` already needs more than one minute and `Minisat+` and `Yices` couldn't find a solution for at least one hour. Surprisingly `Minisat+` which uses `Minisat` as back end – and which needs only a few seconds on this problem – is very slow, also in comparison to `Pueblo`.

	Grid size	`Minisat2`	`Minisat+`	`Pueblo`	`Yices`
CPU time	9×9	0.02 s	0.04 s	0.02 s	10.51 s
CPU time	16×16	0.08 s	0.30 s	0.23 s	1659.50 s
CPU time	25×25	0.59 s	1.57 s	2.67 s	> 1 h
CPU time	36×36	5.66 s	> 1 h	104.09 s	> 1 h

Table 3.4: Performance of the single solvers on empty Sudoku grids.

3.6 NP-Completeness

The task of finding a solution to an $n^2 \times n^2$ Sudoku grid is NP-complete. This was first proven in 2003 by Seta and Yato [27]. On the contrary the smaller class of Sudoku puzzles which have a unique solution and which can be solved only by reasoning is decidable in polynomial time [16]. In this section we give a short review of the NP-completeness proof. The main idea is to give a polynomial-time reduction from the Latin square problem which was proven to be NP-complete by Colbourn *et al.* in 1984 [11]. For a general introduction to the theory of NP-completeness see for example [18].

Definition 3.4. For $m \in \mathbb{N}$ with $m \geq 2$ a *partial Latin square* of dimension m is a function $L : M \times M \to M \cup \{w\}$ where $M = \{x \mid 1 \leq x \leq m\}$. A *complete Latin square* for a partial Latin square L is a function $L' : M \times M \to M$ such that

1. if $L(i,j) \neq w$ then $L'(i,j) = L(i,j)$,
2. $\forall n \in M,\ \forall i \in M,\ \exists j \in M\colon\ L'(i,j) = n$,
3. $\forall n \in M,\ \forall j \in M,\ \exists i \in M\colon\ L'(i,j) = n$.

Observe that the definition of a complete Latin square is almost the same as the definition of the solution of a Sudoku grid. The main difference is that in a complete Latin square there are no requirements for subgrids. As a consequence it is not necessary to demand that the square root of the dimension m of a Latin square is a natural number as it has to be in a Sudoku grid.

Theorem 3.5. *The problem of finding a valid solution for an $n^2 \times n^2$ Sudoku grid is NP-complete.*

Proof. **Membership** To test if a completed grid S' meets all requirements in order to be a valid solution of the Sudoku grid S the following procedure can be used:

1. Check if the number of rows and columns in S and S' is equal.
2. For every cell (i,j) of S' verify that $1 \leq S'(i,j) \leq n^2$ and that $S(i,j) = S'(i,j)$ if $S(i,j) \in \mathbb{N}$.
3. For every row, column and $n \times n$ subgrid in S' compare every two numbers to guarantee that they are distinct.

The first and the second step require each at most one iteration through the grid. In the third step, for each row, each column and each subgrid $n^2 * \frac{1}{2}(n^2-1)$ comparisons are necessary. Hence a solution can be verified in polynomial time.

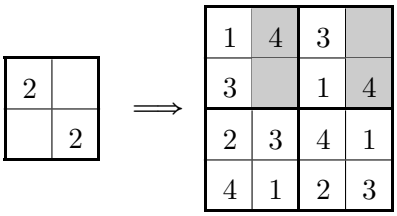


Figure 3.4: A partial Latin square and the Sudoku grid it is reduced to.

Hardness Given a partial Latin square L of dimension m it can be mapped to a Sudoku grid S_L of dimension m as follows:

$$S_L(i,j) = \begin{cases} L(i, \frac{j}{m}) * m & \text{if } (i,j) \in B, L(i, \frac{j}{m}) \neq w \\ w & \text{if } (i,j) \in B, L(i, \frac{j}{m}) = w \\ (((i-1) \bmod m) * m + \lfloor \frac{i-1}{m} \rfloor + j) \bmod m^2 & \text{otherwise} \end{cases}$$

where $B = \{(i,j) \mid 1 \leq i \leq m \text{ and } j \bmod m = 0\}$.

The set B contains respectively all cells of the last column of all $m \times m$ subgrids in the first row of the Sudoku grid. Figure 3.4 illustrates the relation between the Latin square and the constructed Sudoku grid. This reduction is clearly computable in polynomial time and as the Latin square problem is NP-complete, it remains only to show that there exists a complete Latin square for L if and only if there exists a solution for S_L.

First we proof that all cells which are in the first m rows of S_L and which are not in B must not contain a number which is divisible by m without rest. Further we show that all cells which are in the same column as cells in B but are not element of B do not contain which are numbers divisible by m.

Lemma 3.6. *Given a partial Latin square L then for each $(i,j) \in N \times N$ with $N = \{1,\dots,m^2\}$ it holds that*

1. *if $1 \leq i \leq m$ and $j \bmod m \neq 0$ then $S_L(i,j) \bmod m \neq 0$,*
2. *if $m < i \leq m^2$ and $j \bmod m = 0$ then $S_L(i,j) \bmod m \neq 0$.*

Proof.
1. By definition $S_L(i,j) = (((i-1) \bmod m) * m + \lfloor \frac{i-1}{m} \rfloor + j) \bmod m^2$. By the fact that $1 \leq i \leq m$ we have that $\lfloor \frac{i-1}{m} \rfloor = 0$ and therefore $S_L(i,j) = (((i-1) \bmod m) * m + j) \bmod m^2$. If we abbreviate $((i-1) \bmod m) * m$ with $v * m$ and write $m * m$ for m^2 then $S_L(i,j) = (v - w * m) * m + j$ for some $w \in \{0,1\}$. As $((v - w * m) * m) \bmod m = 0$ and $j \bmod m \neq 0$ we have that $S_L(i,j) \bmod m \neq 0$.
2. By definition $S_L(i,j) = (((i-1) \bmod m) * m + \lfloor \frac{i-1}{m} \rfloor + j) \bmod m^2$. As $j \bmod m = 0$ we can write j in the form of $u * m$ and by abbreviating $(i-1) \bmod m$ with v we get $S_L(i,j) = ((v+u) * m + \lfloor \frac{i-1}{m} \rfloor) \bmod m^2$. This can be further rewritten to $S_L(i,j) = (v + u - w * m) * m + \lfloor \frac{i-1}{m} \rfloor$ for some w. As $m < i \leq m^2$ we have that $1 \leq \lfloor \frac{i-1}{m} \rfloor < m$ and therefore $\lfloor \frac{i-1}{m} \rfloor \bmod m \neq 0$. Now it is immediate that $S_L(i,j) \bmod m \neq 0$.

□

From this lemma it follows immediately that in a solution S'_L of S_L all cells in B contain numbers which are divided by m without rest.

($\Leftarrow$) Let S_L be the Sudoku grid corresponding to a partial Latin square L with dimension m and with solution S'_L. We construct a completion L' by taking the first m elements of each column j of S'_L with $j \bmod m = 0$ divided by m as column $\frac{j}{m}$ of L'. These elements of S'_L are all in B and therefore are divided by m without rest. As already mentioned this fact follows from Lemma 3.6. To show that L' is really a completion of L we assume that in one row of L' a number appears twice. Then it follows immediately that S'_L is not a valid Sudoku solution to S_L as the rows of L' are part of the rows of S' which must not contain any symbol twice. The same argument can be used to deduce that all columns in L' only contain distinct symbols. Finally we have to verify that $L'(i,j) = L(i,j)$ if $L(i,j) \neq w$. This holds by definition.

($\Rightarrow$) Let L be a partial Latin square with completion L' and S_L the Sudoku grid constructed from L. A solution S'_L to S_L is given by the following function

$$S'_L(i,j) = \begin{cases} S_L(i,j) & \text{if } S_L(i,j) \neq w \\ L'(i, \frac{j}{m}) * m & \text{otherwise} \end{cases}$$

Next we have to prove that S'_L is a valid Sudoku solution for S_L.

- Every number in S_L is at the same position as in S'_L by definition.
- Clearly in every cell of S'_L there is a number in-between 1 and m^2.
- By contradiction we show that in every row each number appears at most once. Therefore we assume that there exist a row i and two columns j, j' with $j \neq j'$ such that $S'_L(i,j) = S'_L(i,j')$.
  1. $(i,j),(i,j') \notin B$: from $S_L(i,j) = S_L(i,j')$ it follows by definition that

     $$(((i-1) \bmod m) * m + \lfloor \tfrac{i-1}{m} \rfloor + j) \bmod m^2 =$$
     $$(((i-1) \bmod m) * m + \lfloor \tfrac{i-1}{m} \rfloor + j') \bmod m^2.$$

     This equation can be simplified to $j \bmod m^2 = j' \bmod m^2$. But as $j \neq j'$ and $1 \leq j, j' \leq m^2$ this yields a contradiction.
  2. $(i,j),(i,j') \in B$: then $L'(i, \frac{j}{m}) * m = L'(i, \frac{j'}{m}) * m$ and hence $L'(i, \frac{j}{m}) = L'(i, \frac{j'}{m})$ which is contradicting the fact that L' is a complete Latin square.
  3. $(i,j) \in B, (i,j') \notin B$: $S'_L(i,j) = L'(i, \frac{j}{m}) * m$ and hence $S'_L(i,j) \bmod m = 0$. By Lemma 3.6 we have that $S'_L(i,j') \bmod m \neq 0$. If follows that $S_L(i,j) \neq S'_L(i,j')$ which yields a contradiction.
  4. $(i,j') \in B, (i,j) \notin B$: same proof as in c).
- To prove that in every column and every $m \times m$ subgrid every number is unique a similar argumentation can be used.

□

4 Akari

In comparison to Sudoku, Akari, also known as Light Up, is a relatively new puzzle – it was developed in 2001 by the Japanese publisher Nikoli. Nevertheless there are several online tutorials and games and Nikoli has already published two books with Akari puzzles. Deciding if a given Akari puzzle has a solution is NP-complete. This was proven in 2005 by Brandon McPhail [17]. At the end of this chapter the main steps of the proof are illustrated.

The first part of this chapter is dedicated to different encodings of Akari, a propositional and a pseudo-boolean encoding are discussed. For Akari no separate SMT encoding is introduced but in the experiments the PB encoding is taken as input for the SMT solver `Yices`. A detailed explanation of the reasons can be found in Section 4.4. Further we give experimental results on the presented encodings and different solvers.

4.1 Problem Definition

Like most other logic puzzles by Nikoli, Akari is played on a rectangular grid. Grid cells are either black or white and in addition some of the black cells may contain a number between 1 and 4. Figure 4.1 shows an example of an initial Akari grid. During the game light bulbs are placed in white cells. If a light bulb

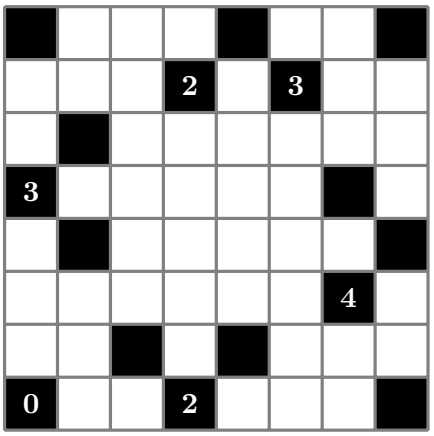


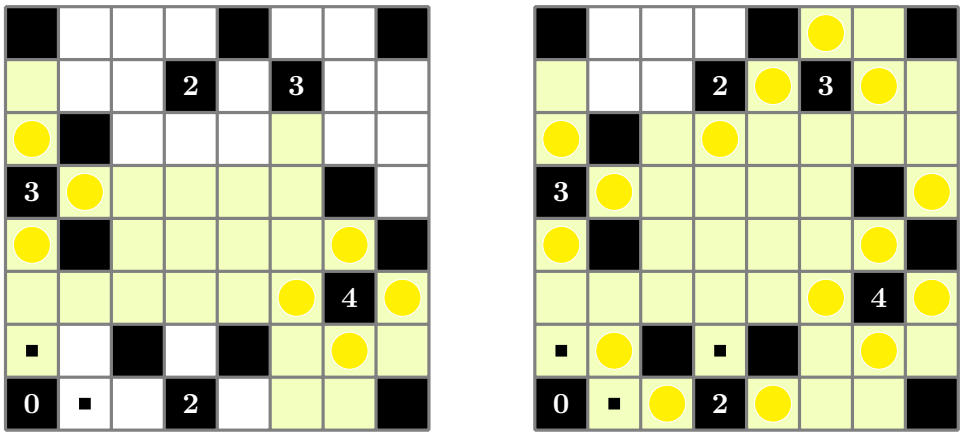
Figure 4.1: Akari grid of size 8×8.

is situated in a cell it illuminates all white cells in the same row and the same column until the beam reaches the border or a black cell. An Akari puzzle is solved correctly if all light bulbs are positioned in such a way that the following conditions are fulfilled:

- no two light bulbs shine on each other,
- every cell is illuminated,
- for every black cell containing a number n there are exactly n light bulbs in the four adjacent cells.

Example 4.1. In this example we solve the Akari puzzle given in Figure 4.1 to point out the basic solving strategies:

- In the first step we mark every cell which is adjacent to a black cell labelled with 0. A mark in a cell indicates that there must not be a light in this cell. In our example there is only one cell labelled with 0 namely cell $(8, 1)$ and hence we mark the cells $(7, 1)$ and $(8, 2)$ which are direct neighbours of $(8, 1)$.
- Next we search every cell labelled with a number n and check if it has exactly n adjacent white cells. If so, there is only one possibility to position the according light bulbs, i.e., to place a bulb in every adjacent white cell. In our example cell $(6, 7)$ is labelled with 4 and therefore it is immediately possible to place four lights around cell $(6, 7)$. The second cell to which this strategy is applicable is $(4, 1)$. This cell is labelled with 3 and as it is next to the left border it only has 3 adjacent cells where lights can be put. After the application of the first two steps we get the partly solved Akari puzzle shown in Figure 4.2(a).



(a) (b)

Figure 4.2: Partial solutions of the Akari grid given in Figure 4.1.

- In a next step we search if the light bulbs we have already placed allow to deduce new information. And indeed cell $(2, 6)$ which is labelled with a 3 allows us to place all of the three light bulbs. Imagine we put a light bulb in cell $(3, 6)$ which is already illuminated then one light would be hit by the beam of the other and hence there must not be a light in this cell.

- Further for the three cells $(7,2)$, $(8,3)$ and $(8,5)$ there is only one possibility to illuminate these cells namely by putting a light bulb directly inside theses cells. As $(8,3)$ and $(8,5)$ are both adjacent to cell $(8,4)$ which is labelled with 2, we know that there must not be more light bulbs adjacent to $(8,4)$ and hence we mark cell $(7,4)$. Now it follows that cell $(7,4)$ can only be illuminated if there is a light bulb in cell $(3,4)$. After applying these steps we see that there has to be also a light in cell $(4,8)$ as there is no other chance to illuminate this cell. The resulting Akari grid is shown in Figure 4.2(b).
- Finally there are only five cells in the upper left corner left which are not lit up by now. The two cells $(1,4)$ and $(2,3)$ surely must not contain a light bulb as they are adjacent to a cell labelled with 2 which has already two lights in its neighbourhood. Hence there are only three cells left where we may put a light bulb. If there was a light in $(1,2)$ then cell $(2,3)$ couldn't be illuminated any more. Hence we mark $(1,2)$ too and now the only possibility which is left to light up all these five cells is to place a light bulb in each of the two last unmarked cells $(1,3)$ and $(2,2)$. The definite solution is shown in Figure 4.3.

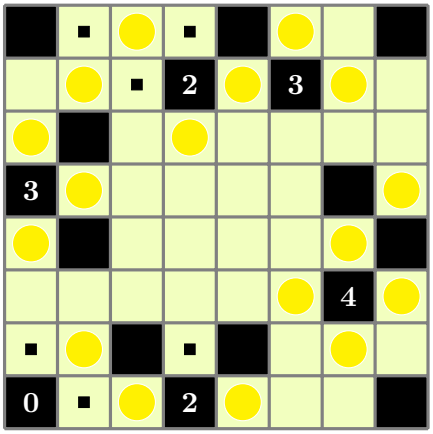


Figure 4.3: Final solution of the Akari grid shown in Figure 4.1.

As usual we also need to introduce a more formal definition of Akari in order to have the means to formulate the different encodings in a concise way. Again a playing field is defined as a function:

Definition 4.2. For $m, n \in \mathbb{N}^+$ an $m \times n$ *Akari grid* is a function

$$A : \{1, \ldots, m\} \times \{1, \ldots, n\} \to \{w, b, 0, \ldots 4\}$$

where m stands for the number of rows, n for the number of columns, w denotes white cells and b is used for black cells without a number.

In the following sections the abbreviations $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$ are used to ease readability. In order to express all requirements for a correctly solved Akari puzzle the following preliminary definitions are needed:

Definition 4.3. Given an $m \times n$ Akari grid A two cells $(i,j), (k,l) \in M \times N$ are *adjacent* iff

$$|i-k| + |j-l| = 1.$$

Two cells (i,j) and (k,l) are *diagonally adjacent* iff

$$|i-k| = |j-l| = 1.$$

For each cell $(i,j) \in M \times N$ the *4-neighbourhood* $N_4(i,j)$ contains all cells (k,l) with $A(k,l) = w$ that are adjacent to (i,j).

Definition 4.4. Given an $m \times n$ Akari grid A we describe the rows and columns of A as functions R and C:

- $R : M \to \mathcal{P}(M \times N),\ R(i) = \{(i,j) \mid j \in N\}$
- $C : N \to \mathcal{P}(M \times N),\ C(j) = \{(i,j) \mid i \in M\}$

A row or column may be split by black cells into "subrows" and "subcolumns". For each cell (i,j) the subrow $SR(i,j)$ and the subcolumn $SC(i,j)$ it belongs to are defined as follows:

- $SR : M \times N \to \mathcal{P}(M \times N)$,

$$SR(i,j) = \begin{cases} SR'(i,j) & \text{if } A(i,j) = w \\ \bot & \text{otherwise} \end{cases}$$

  with $(i,j) \in SR'(i,j)$ and $\forall (k,l) \in R(i)$ we have $(k,l) \in SR'(i,j)$ iff
  1. $A(k,l) = w$
  2. $\forall (u,v) \in R(i)$ with $j < v < l$ or $l < v < j$: $A(u,v) = w$

- $SC : M \times N \to \mathcal{P}(M \times N)$,

$$SC(i,j) = \begin{cases} SC'(i,j) & \text{if } A(i,j) = w \\ \bot & \text{otherwise} \end{cases}$$

  with $(i,j) \in SC'(i,j)$ and $\forall (k,l) \in C(j)$ we have $(k,l) \in SC'(i,j)$ iff
  1. $A(k,l) = w$
  2. $\forall (u,v) \in C(j)$ with $i < u < k$ or $k < u < i$: $A(u,v) = w$

All subrows can be merged to one set SR containing all distinct subrows, likewise a set SC with all subcolumns is constructed:

- $SR = \{SR(i,j) \mid (i,j) \in M \times N\}$
- $SC = \{SC(i,j) \mid (i,j) \in M \times N\}$

Definition 4.5. A *solution* to an $m \times n$ Akari grid A is a function

$$A' : \{1, \ldots, m\} \times \{1, \ldots, n\} \to \{w, b, l, 0, \ldots 4\}$$

where l stands for light and the following conditions $\forall (i,j) \in M \times N$ are fulfilled:

1. if $A(i,j) = w$ then $A'(i,j) = w$ or $A'(i,j) = l$, otherwise $A'(i,j) = A(i,j)$,
2. if $A'(i,j) = l$ then $\forall (i',j') \in SR(i,j) \cup SC(i,j)$ with $(i,j) \neq (i',j')$ it holds that $A'(i',j') \neq l$,
3. if $A(i,j) = w$ then $\exists (i',j') \in SR(i,j) \cup SC(i,j)$ such that $A'(i',j') = l$,
4. if $A(i,j) \in \{0, \ldots, 4\}$ then there are exactly $A(i,j)$ cells $(i',j') \in N_4(i,j)$ with $A'(i',j') = l$.

4.2 Propositional Logic

The set of variables which is necessary for the propositional encoding presented in this section is rather small. In order to find a solution for a given Akari puzzle we have to decide for every cell if a light bulb is positioned inside that specific cell or not. To model this fact a single variable $L(i,j)$ for a cell (i,j) suffices. The meaning of such a variable $L(i,j)$ is that a light bulb is in cell (i,j) if and only if $L(i,j)$ is true. As lights may only be placed in white cells variables are only introduced for those cells. All other cells represent static information and therefore no variables are needed. The total number of variables clearly does not exceed $m * n$, for a typical $m \times n$ Akari grid it is about 80% of $m * n$. As the number of non-white cells is not fixed and depends on the difficulty of the actual puzzle it is not possible to state the exact number of variables in general.

Sticking to Definition 4.5 the construction of a propositional encoding isn't too difficult. The first property stated in the definition is guaranteed by the choice of variables. The second and the third condition are encoded in the following way:

- The fact that no two lights bulbs shine on each other can be split into two subparts. First, in each subrow there is at most one light bulb and second, in each subcolumn there is at most one light bulb:

  $\bigwedge_{sr \in SR} \bigwedge_{(i,j) \in sr} \bigwedge_{(i',j') \in sr, (i,j) \neq (i',j')} (\neg L(i,j) \vee \neg L(i',j'))$

  $\bigwedge_{sc \in SC} \bigwedge_{(i,j) \in sc} \bigwedge_{(i',j') \in sc, (i,j) \neq (i',j')} (\neg L(i,j) \vee \neg L(i',j'))$

- To guarantee that a white cell is lit up there has to be at least one light bulb in the corresponding subrow or subcolumn:

  $\bigwedge_{(i,j) \in M \times N, A(i,j) = w} \bigvee_{(i',j') \in SR(i,j) \cup SC(i,j)} L(i',j')$

The encoding of the last property of a correctly solved Akari grid takes a bit more thinking. Recall, that this characteristic requires the counting of lights in the neighbourhood of a cell. Mainly there are two approaches to realise addition in propositional logic. On the one hand we may use a network of adders to

sum up the values of propositional variables. On the other hand a formula can be generated by enumerating all possible combinations of variables that correspond to the correct outcome of the addition. It happens frequently that this approach is unfeasible due to the rapid growth of the resulting formula. Using adder networks in turn creates more compact formulas but requires the use of additional variables to keep track of intermediate sums and carries. Furthermore the conversion to CNF of a formula corresponding to a full adder by the application of the standard transformation again increases the final formula. As explained in Chapter 2 the resulting CNF can be kept small by applying a kind of Tseitin's transformation which is linear in the length of the initial formula but introduces additional variables.

Considering Akari puzzles counting is applied only in a very limited way. The result of the addition must not be anything else than 1, 2 or 3 – to decide about the adjacent cells of cells labelled with 0 and 4 no addition is required at all. The corresponding lights may be placed immediately. Moreover not more than four variables are added together, often it's even less because of black cells or vicinity to the border. Due to these reasons an encoding which establishes formulas by enumerating possibilities instead of using adder networks is applicable. It yields compact formulas, needs no additional variables and is closer to preserving the reasoning steps of a human by allowing more unit propagation.

The following encodings assume that for the currently treated cell (i, j) all four neighbours exist and are white. This is the most complex case to translate. Simpler combinations are handled later on. The two cases for a cell labelled with 0 or 4 are straight forward:

- if $A(i, j) = 0$ then $\bigwedge_{(i',j')\in N_4(i,j)} \neg L(i', j')$,
- if $A(i, j) = 4$ then $\bigwedge_{(i',j')\in N_4(i,j)} L(i', j')$.

The other cases have to be explained in more detail:

- Let $A(i, j) = 1$. This case amounts to the fact that at least one variable in $N_4(i, j)$ has to be true as well as at most one variable has to be true. Similar to the usage of "at least" and "at most" clauses in the propositional Sudoku encoding presented in Section 3.2 we get the formulas:

  $\bigvee_{(k,l)\in N_4(i,j)} L(k, l)$

  $\bigwedge_{(k,l)\in N_4(i,j)} \bigwedge_{(k',l')\in N_4(i,j),(k,l)\neq(k',l')} (\neg L(k, l) \vee \neg L(k', l'))$

- Let $A(i, j) = 2$. The fact that two of the four neighbours have to be true can't be expressed as directly as before. The following formula states that for every three variables in $N_4(i, j)$ it is not possible that all three are true or all three are false at the same time:

  $\bigwedge_{(k,l)\in N_4(i,j)} \bigwedge_{(k',l')\in N_4(i,j)} \bigwedge_{(k'',l'')\in N_4(i,j)}$ with $(k, l) \neq (k', l') \neq (k'', l'')$

  $(L(k, l) \vee L(k', l') \vee L(k'', l'')) \wedge (\neg L(k, l) \vee \neg L(k', l') \vee \neg L(k'', l''))$

- Given $A(i, j) = 3$. This case is very similar to the handling of cells labelled with 1. We require that at least one and at most one of the four adjacent

cells is false:

$\bigvee_{(k,l)\in N_4(i,j)} \neg L(k,l)$

$\bigwedge_{(k,l)\in N_4(i,j)} \bigwedge_{(k',l')\in N_4(i,j),(k,l)\neq(k',l')} (L(k,l) \vee L(k',l'))$

Next we consider the cases where $N_4(i,j)$ of a labelled cell (i,j) has less than four elements. If $|N_4(i,j)| = A(i,j)$ and $A(i,j) > 0$ then all elements of $N_4(i,j)$ can be set to true immediately. If $|N_4(i,j)| < A(i,j)$ then there exists no solution to the given puzzle. This leaves only following cases to consider: cells labelled with 1 and cells labelled with 2 having less than four adjacent cells. If $A(i,j) = 1$ then the same formula can be used irrespective of the number of elements in $N_4(i,j)$. Finally for a cell labelled with 2 having three neighbours we use the formulas which were introduced for cells labelled with 3 having four neighbours.

It is rather hard to state the number of clauses generated by this encoding in a precise way. First it depends on the number of subcolumns and subrows which are not always the same and it depends on the numbers the cells are labelled with. Hence we present an upper bound instead of giving an exact formula. Remember also, that the number of clauses is not that important. If they are short and/or allow unit propagation it is even better to have more clauses. To find an upper bound for the encoding of the first two properties we consider an Akari grid consisting only of white cells. The encoding of the fact that no two lights see each other produces $\frac{1}{2}(m+1) * m * n + \frac{1}{2}(n+1) * n * m$ clauses on such an empty grid. To express that every cell is illuminated in the empty grid takes $m * n$ clauses. The clauses used to express addition are more difficult to estimate. The encoding of addition for a cell labelled with a number needs at most 8 clauses. If we had a grid where every cell is labelled with a number then we get an upper bound of $8 * m * n$. Of course such a puzzle is not solvable, but it surely yields an upper bound. Putting everything together the number of clauses generated for an $m \times n$ Akari grid is bounded by $O(m^2 * n + n^2 * m)$.

4.3 Pseudo-Boolean Logic

For a pseudo-boolean encoding the same set of variables can be used as for the propositional encoding, i.e., for every white cell (i,j) a variable $L(i,j)$ is introduced and the assignment of $L(i,j)$ in a solution indicates if there is a light in (i,j) or not.

The pseudo-boolean encoding itself is very similar to the propositional one, nevertheless pseudo-boolean logic provides better means to express the different requirements of a correct solution for an Akari puzzle. For example the encoding of an "at most" condition requires a larger number of propositional clauses whereas in pseudo-boolean logic we need exactly one $\leq$-constraint. Another big advantage is that addition is supported naturally in pseudo-boolean logic. Hence it is not necessary to think about adder networks or similar concepts but an addition of variables with a desired result is expressed simply by stating a linear equation. A further advantage is that the pseudo-boolean encoding is closer to a human readable representation than the propositional encoding.

The actual encoding for an $m \times n$ Akari grid A is as follows:

- no two light bulbs shine on each other:

  $\forall sr \in SR$: $\sum_{(i,j)\in sr} L(i,j) \leq 1$,

  $\forall sc \in SC$: $\sum_{(i,j)\in sc} L(i,j) \leq 1$,

- each cell is illuminated:

  $\forall (i,j) \in M \times N$ with $A(i,j) = w$: $\sum_{(i',j')\in SR(i,j)\cup SC(i,j)} L(i',j') \geq 1$,

- ever cell labelled with a number n has exactly n adjacent cells containing a light bulb:

  $\forall (i,j) \in M \times N$ with $A(i,j) \in \mathbb{N}$: $\sum_{(i',j')\in N_4(i,j)} L(i',j') = A(i,j)$.

In [22] Helmut Simonis presents an extension to this encoding which proves to be useful in constraint programming. In the following the basic idea is explained and illustrated with an example. The usefulness of this extension concerning pseudo-boolean logic is discussed in Section 4.5 where the experimental results are presented.

The aim of the extension is to improve the decisions for cells which are diagonally adjacent to cells labelled with a number. Consider Figure 4.4. It is clear that all four cells diagonally adjacent to the cell labelled with 3 have to be marked – no matter in which way the three light bulbs are placed, the four diagonally adjacent cells are always in one row or column with a light. The pseudo-boolean encoding as presented previously in this section does not allow this deduction step. In order to develop an encoding of this property we formu-

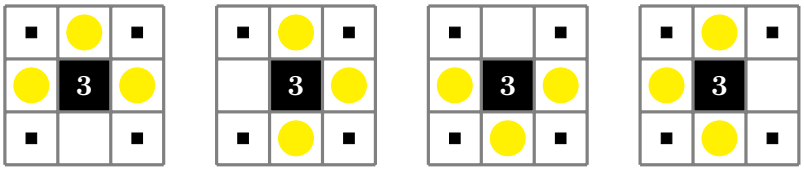


Figure 4.4: Example Akari grids for the extension according to Simonis.

late it in a more general way such that it does not only hold for cells labelled with 3 but also for cells labelled with 1 and 2. As the labels 0 and 4 allow direct positioning of all lights in these cases this extension is not useful. Given a cell (i,j) with integer label $1 \leq A(i,j) \leq 3$ and $|N_4(i,j)| > A(i,j)$ a general constraint for a cell (k,l) which is diagonally adjacent to cell (i,j) is derived using the following equations:

1. $\sum_{(i',j')\in N_4(i,j)} L(i',j') = A(i,j)$,
2. $\forall (k',l') \in N_4(i,j) \cap N_4(k,l)$: $L(k,l) + L(k',l') \leq 1$.

Let $c = |N_4(i,j) \cap N_4(k,l)|$, then some arithmetical conversions on the previous constraints yield the inequality

$$\sum_{(k'',l'')\in N_4(i,j)\setminus N_4(k,l)} L(k'',l'') - c * L(k,l) \geq A(i,j) - c.$$

If c is 0 then such an additional constraint gives no extra information and can be omitted.

Example 4.6. In Figure 4.4 the three initial equations respectively inequations for cell $(1,1)$ which is diagonally adjacent to cell $(2,2)$ are

$$\begin{aligned} L(2,1) + L(1,2) + L(3,2) + L(2,3) &= 3 \\ L(2,1) + L(1,1) &\leq 1 \\ L(1,2) + L(1,1) &\leq 1 \end{aligned}$$

and the final constraint is

$$L(3,2) + L(2,3) - 2 * L(1,1) \geq 1.$$

Now assume that there is a light bulb in $(1,1)$, thus that $L(1,1)$ is true. Then this inequality is never fulfilled even if $L(3,2)$ and $L(2,3)$ are both true. Therefore this constraint allows to deduce that there must not be a light in $(1,1)$. Note that if the cell in the middle of the grid wasn't labelled with 3 but with 2 than the resulting constraint would be $L(3,2) + L(2,3) - 2 * L(1,1) \geq 0$ and reasoning as in the former case wouldn't be possible any more. In that case it would be necessary to first set at least one of the two variables $L(3,2)$ or $L(2,3)$ to false in order to derive that there is no light in $(1,1)$.

In pseudo-boolean logic it is possible to express the above mentioned property in a very concise way. If we want to encode the very same fact in propositional logic some difficulties arise and it is not immediately clear how the encoding is done best. One possibility is to generate the PB constraints which correspond to the extension and then to translate them into SAT, e.g. using translation techniques like BDD's or adders as done in `Minisat+`. Another approach is to directly create a propositional formula which can be done by a kind of enumeration of all possibilities or by the use of propositional counting. As all of these methods need some effort and as the propositional encoding given in Section 4.2 performs very good in the experiments we omit the presentation of a propositional translation of this extension.

4.4 SMT

As already mentioned one main advantage of SMT is that by the use of different theories variables may range not only over boolean values but also over integers or even over reals. Unfortunately there is no use for integer and real variables in Akari. For each cell we have to make only a yes/no decision and therefore boolean variables suffice completely. As a consequence there is no new encoding presented in this section. For the experiments in Section 4.5 the performance of `Yices` on the minimal and on the extended PB encoding is evaluated.

4.5 Experimental Result

The Akari puzzles which were tested in the experiments and for which results are presented in this section were randomly generated by the tool `lightup` which is part of "Simon Tatham's Portable Puzzle Collection" [23]. All puzzles belong to the hardest difficulty level which can be specified in this tool and have a unique solution. For each puzzle size 10 different puzzles were tested. The given results are always an average of these 10 instances.

Grid size	Decisions	Conflicts	CPU time
10×10	40.80	3.50	0.002 s
24×14	188.40	4.80	0.003 s
36×36	2139.20	22.60	0.005 s
50×100	23180.10	54.20	0.032 s
50×200	84310.70	85.60	0.085 s
50×400	353959.20	173.50	0.219 s

Table 4.1: Performance of `Minisat2` on Akari grids.

The largest puzzles which where checked are of size 50×400, a grid size which would take a human hours or days to find a solution. It is also very large in comparison to the largest Sudoku grids for which results were presented in Section 3.5. Still for these Akari puzzles a solution can be found within seconds by all solvers. A propositional or PB encoding of a Sudoku grid of size 36×36 uses 36^3 which is 46656 variables whereas for an Akari puzzle of the same size only 36^2 variables are necessary. To encode a 50×400 grid we need 20000 variables which roughly corresponds to a 25×25 Sudoku grid concerning the number of variables. This explains mostly why the performance results for Akari grids are dramatically better than for equally sized Sudoku grids.

	`Minisat+`			`Pueblo`
Grid size	Decisions	Conflicts	CPU time	CPU time
10×10	28.90	5.00	0.002 s	0.005 s
24×14	43.90	6.60	0.007 s	0.018 s
36×36	270.00	29.20	0.024 s	0.049 s
50×100	1170.50	65.10	0.074 s	0.219 s
50×200	2383.00	119.70	0.094 s	0.432 s
50×400	8274.10	230.50	0.154 s	1.048 s

Table 4.2: Performance of `Minisat+` and `Pueblo` on Akari grids using the PB encoding without extension.

Table 4.1 illustrates the performance of the SAT solver `Minisat2` on Akari grids encoded with the propositional encoding which is described in Section 4.2. For all instances a solution is found in less than one second.

	Minisat+			Pueblo
Grid size	Decisions	Conflicts	CPU time	CPU time
10×10	24.80	1.30	0.002 s	0.001 s
24×14	39.90	2.80	0.003 s	0.014 s
36×36	72.00	4.70	0.010 s	0.042 s
50×100	321.40	17.20	0.048 s	0.215 s
50×200	458.60	42.80	0.084 s	0.419 s
50×400	347.80	55.90	0.117 s	1.038 s

Table 4.3: Performance of `Minisat+` and `Pueblo` on Akari grids using the PB encoding with extension.

Experimental results for the PB encoding and its extension as presented in Section 4.3 are given in Tables 4.2 and 4.3. The results show that for both solvers the additional constraints of the extension improve the CPU time which is required to find a solution only marginally. Nevertheless it is able to reduce the number of conflicts reasonable and therefore may improve runtime on even larger benchmarks for which no tests were enforced. Comparing the performance of the different solvers it is easy to see that `Minisat+` is faster than `Pueblo` on all tested instances, also `Minisat2` together with the propositional encoding performs better than `Pueblo`. On all Akari grids except the largest

Grid size	Minimal PB encoding CPU time	Extended PB encoding CPU time
10×10	0.052 s	0.054 s
24×14	0.127 s	0.141 s
36×36	0.321 s	0.353 s
50×100	0.724 s	0.828 s
50×200	1.319 s	1.345 s
50×400	2.451 s	2.734 s

Table 4.4: Performance of `Yices` on Akari grids encoded with the minimal and the extended PB encoding.

ones of size 50×400 `Minisat2` is faster or equally fast as `Minisat+` irrespective of the chosen PB encoding and despite the fact that the SAT encoding uses no extension. Only for the largest class of Akari puzzles but there on both PB encodings `Minisat+` performs better than `Minisat2`.

Table 4.4 shows the performance results for `Yices`. As already the case for Sudoku puzzles, the SMT solver `Yices` is slower than all other solvers and encodings. Interestingly the use of the extended PB encoding which improved the runtime of `Minisat+` and `Pueblo` doesn't allow any advancement for `Yices` but makes it even slower.

4.6 NP-Completeness

At the beginning of this chapter we mentioned already that Akari is NP-complete, the proof was established by Brandon McPhail in 2005. He gives a polynomial-time computable reduction from the NP-complete problem Circuit-SAT to Akari. NP-completeness of Circuit-SAT again is proven by a reduction from SAT – the propositional satisfiability problem – which was the first problem for which an NP-completeness proof was found. In the 1970s Stephen Cook [12] and Leonid Levin [14] independently proved the NP-completeness of SAT.

Definition 4.7. A *boolean circuit* is a directed acyclic graph $C = (V, E)$ which satisfies the following conditions:

- $V = \{1, \ldots, n\}$ and each node $i \in V$ is called a *gate* of C.
- An edge (i, j) is called *wire* and for each $(i, j) \in E$ we have $i < j$.
- Let $I = \{T, F\} \cup \{x_1, \ldots, x_m\}$ then to each node i a sort $s(i) \in I \cup \{\vee, \wedge, \neg, \mathit{split}\}$ is assigned.
- The indegree of a node i depends on its sort:
  - if $s(i) \in I$ then the indegree of i is 0 and i is called *input*,
  - if $s(i) \in \{\neg, \mathit{split}\}$ then the indegree of i is 1,
  - if $s(i) \in \{\vee, \wedge\}$ then the indegree of i is 2.
- Gate n is not of sort *split*, has outdegree 0 and is called *output*. Every gate with associated sort *split* has outdegree 2 and every other gate $i < n$ has outdegree 1.
- For each $x \in \{x_1, \ldots, x_m\}$ there is exactly one input i with $s(i) = x$.

A boolean circuit C with m variables defines a distinct boolean function $f_C : \{0,1\}^m \mapsto \{0,1\}$. The value of the output depends on the value of the input variables, i.e., the output n has value $f_C(a_1, \ldots, a_m)$ if the input variables $x_1, \ldots x_m$ are of value $a_1, \ldots, a_m$. f_C is computed inductively starting at the inputs of the circuit and then using the usual semantics of the boolean connectives corresponding to the sort of each gate. Gates with sort *split* can be disregarded in the computation of f_C as a gate of this sort does not change the value of the input wire but simply forwards it to two output wires. A boolean circuit could of course be defined without the use of *split*. Here it is used to have a fixed outdegree for each gate, a fact which simplifies the reduction from circuits to Akari. A further simplification of boolean circuits is to disallow the usage of the connective $\wedge$. This is admissible as $x \wedge y \equiv \neg(\neg x \vee \neg y)$ according to De Morgan's law.

Definition 4.8. Given a boolean circuit C with boolean function f_C the circuit satisfiability problem *Circuit-SAT* is the problem of deciding if there exists a tuple $(a_1, \ldots, a_m) \in \{0,1\}^m$ such that $f_C(a_1, \ldots, a_m) = 1$. If so, C is said to be *satisfiable*.

The reduction from SAT to Circuit-SAT which is necessary to show NP-completeness of Circuit-SAT is rather easy as both problems decide upon satisfiability of different representations of boolean functions and therefore is not given here.

In general it is not possible to draw a circuit in such a way that no two wires cross each other. If a graph can be drawn on a plane without crossing edges it is called *planar*. Concerning a reduction there are two strategies in order to deal with an eventual non-planarity of a circuit. The first approach is to restrict the set of circuits which are reduced to the puzzle to planar circuits. For that purpose the problem *planar-3SAT* which is also NP-complete as shown in [15] can be used. A formula belonging to this set of problems has a planar occurrence graph, that is a graph for a formula in CNF which has a node for every variable and for every clause and edges (x_i, c_j) if x_i or $\neg x_i$ appears in clause c_j. Converting a planar occurrence graph into a planar circuit which is equivalent to the original formula is straightforward. The drawback of this method is that the resulting circuit usually has more than one output node and induction on the structure of the input formula to prove the equivalence of formula and reduction is not directly applicable. But the restriction to planar circuits is not essential. Another method of dealing with non-planar circuits is to model a crossing of two wires with three *XOR-* (or $\oplus$-) gates and three *split*-gates as shown in Figure 4.5 (*split*-gates are drawn as black spots). In this second approach we don't have to deal with multiple outputs but in turn an additional gate has to be taken care of in the reduction. Concerning the reduction to Akari we get the mapping of an $\oplus$-gate to the puzzle for free by a reduction of $\vee$-gates and hence there is no need to restrict ourselves to planar circuits. To simplify our reduction we additionally demand for any circuit that two edges (i_1, j) and (i_2, j) with $s(i_1), s(i_2) \in I$ must not form a crossing. It is easy to see that this is not an actual restriction.

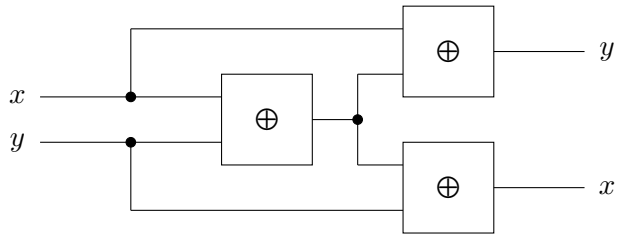


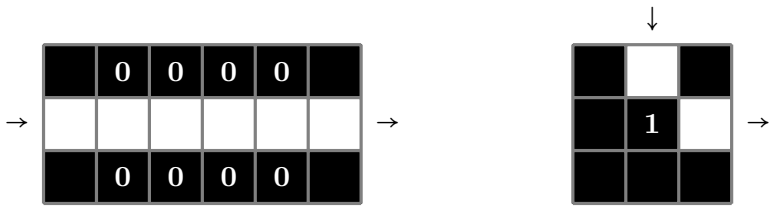
Figure 4.5: Modelling a wire crossing with a sequence of *split*- and $\oplus$-gates.

Theorem 4.9. *The problem of finding a solution for an $m \times n$ Akari grid is NP-complete.*

Proof. **Membership** It is not too hard to see that the verification of a solution to an Akari grid can be done in polynomial time. To check if the number of lights in the neighbourhood of a labelled cell is correct the solution has to be searched once. While iterating over the cells of the grid we can check for each white cell if it is illuminated by searching the subrow and

subcolumn the white cell belongs to. This needs at most $m + n$ steps per cell. At the same time we verify that there are no two lights in a subrow or subcolumn with no additional time overhead. It remains to show that the solution is really a solution of the initial Akari grid. This can also be done by iterating once through all grid cells of both grids. All together this yields a time bound of $O(m^2 * n + n^2 * m)$.

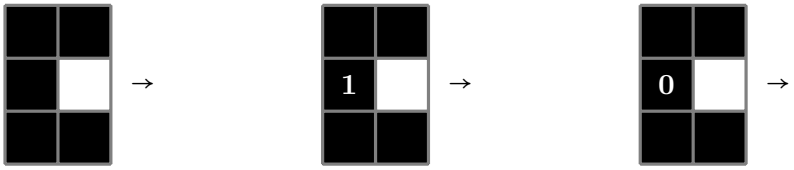
Hardness The reduction from Circuit-SAT to Akari is defined on the basis of so-called tiles – minimal building blocks used to construct an Akari puzzle from a circuit. The idea of the reduction is to rebuild a circuit kind of graphically. Every wire and every gate is replaced by a special tile and all these tiles together then form an Akari grid. Similar to gates in a circuit tiles have input and output sides. Throughout the figures in this section input and output sides of tiles are indicated by arrows.



(a) straight-wire-tile (b) corner-wire-tile

Figure 4.6: Tiles corresponding to reductions from wires.

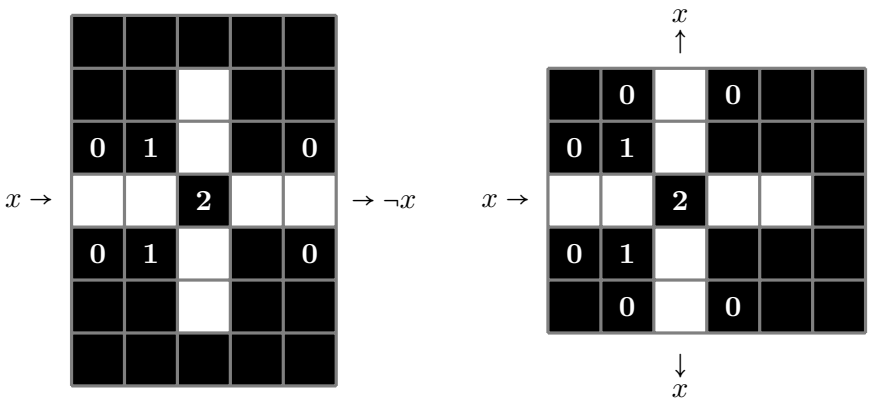
First of all a reduction for the wires is needed, therefore wire-tiles as shown in Figure 4.6 are introduced. A straight-wire-tile may not contain any 0's but can be stretched by inserting as many 0's as desired. Both wire-tiles have the property that a light either is in the first or the last cell of the tile but nowhere else. This property is utilised to keep the connection to boolean circuits by fixing a meaning in terms of boolean values to a tile. If a light bulb is placed at the very beginning – the input side – of the wire-tile this corresponds to value 1. Otherwise the light is in the last cell – on the output side – and the value of the tile is 0. Thus a wire-tile forwards a certain boolean value like the wire in the circuit does.



(a) variable-tile (b) T-input-tile (c) F-input-tile

Figure 4.7: All possible tiles corresponding to inputs.

Next we define the representation of the input nodes in the reduction. Therefore we distinguish three different kinds of inputs: T, F and variables. For each of these a tile is defined as shown in Figure 4.7. An input-tile is either placed at the beginning of a wire or directly yields an input value for a tile corresponding to a gate. The variable-tile has no cells which are labelled with numbers and hence the value of e.g. a wire which starts with a variable-tile is not fixed. This is due to the fact that a variable either has value 0 or 1 and therefore the position of the light bulb must not be predefined. The T-input- and F-input-tile which correspond to input values 1 and 0 are used to fix the position of the light bulb to the beginning respectively the end of a wire starting with these tiles.



(a) ¬-tile (b) *split*-tile

Figure 4.8: One tile for ¬- and *split*-gates.

Figure 4.8 shows two nearly identical tiles. The ¬-tile which is given in picture (a) is the reduction of a ¬-gate. Note that the 0's are not actually part of the ¬-tile but are the end respectively beginning of the input and output wires. To illustrate the idea of the ¬-tile imagine that the value on the input wire is 1 and hence there is no light bulb in cell $(4, 2)$. As consequence there has to be a light in cell $(3, 3)$ and one in $(5, 3)$. Thus the 2 in the middle has already two adjacent cells with lights and does not allow any more lights in its neighbourhood. Hence there must not be a light bulb in cell $(4, 4)$ which is the first cell of the output wire. By the previously explained connection between wires and truth values this gives value 0 on the output wire. On the other hand, if the incoming value is 0 then there is a light in $(4, 2)$ and as a consequence of the two 1's there are no lights in $(3, 3)$ and $(5, 3)$. Then by the 2 a light has to be placed in cell $(4, 4)$ and therefore the value on the output wire is 1.

With a few simple modifications the ¬-tile is changed to a *split*-tile (picture (b) in Figure 4.8) which is used as representation of *split*-gates. The *split*-tile copies the position of the light on the input wire to the output wire and therefore does not change the incoming value but forwards it.

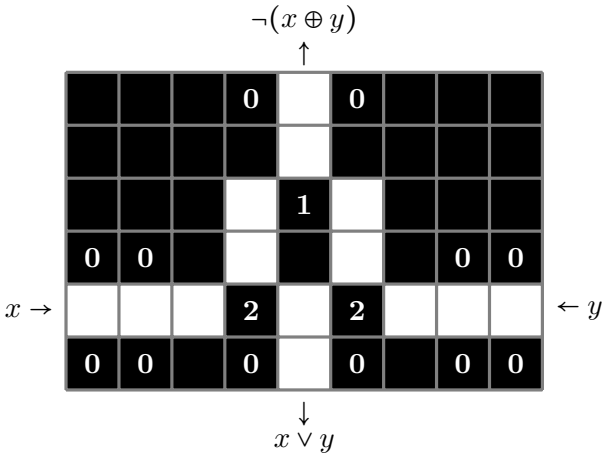


Figure 4.9: One tile for ∨- and *XNOR*-gates.

The next tile which is introduced and which is pictured in Figure 4.9 corresponds to the two boolean functions ∨ and *XNOR* which is the complement of ⊕ and also happens to be equivalent with ↔. Depending on the gate which is represented, one of the two output wires is locked with a black cell. Also for the ∨- and the *XNOR*-tile the idea is to simulate the correlation between input and output values of the corresponding boolean functions. As for the ¬-tile one can easily verify that the tiles work as desired by trying to find solutions for different positions of lights on the input wires. The usefulness of the ∨-tile is immediately clear but one might wonder why an *XNOR*-tile is of any use. Remember that we have to provide a reduction of wire crossings via ⊕. To do so an ⊕-tile is modelled by an *XNOR*-tile followed by a ¬-tile.

Figure 4.10 illustrates the tile which is used to indicate the end of the circuit. The T-output-tile is placed at the end of that wire which comes out of the tile which represents the output of the circuit. It demands that in this wire a light is put in the first cell. This corresponds to requiring an output value of 1.

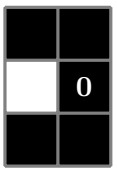


Figure 4.10: The T-output-tile.

Having defined a tile for each element of a boolean circuit it is easy to reduce the whole circuit to an Akari puzzle in polynomial time using recursive computations. To reduce a gate i first the circuits which yield

the inputs for i are computed recursively. If this is done these two smaller Akari grids and the tile for gate i can be placed appropriately and are then connected with wire-tiles. This placement of tiles is the only part of the reduction where a bit more computation is involved.

The last step of the NP-completeness proof is to show that for a boolean circuit C and its reduction A_C it holds that C is satisfiable if and only if there exists a solution to the Akari grid A_C. The truth of this statement follows immediately from the following two lemmas. The proof for Lemma 4.10 is given in detail whereas Lemma 4.11 is only declared but not proven. The proof for this Lemma is very similar to the one of Lemma 4.10 and therefore it is not stated explicitly.

Lemma 4.10. *Given a boolean circuit C with boolean function f_C. If there is a tuple $(a_1, \ldots, a_m) \in \{0,1\}^m$ with $f_C(a_1, \ldots, a_m) = 1$ then A_C has a solution.*

Proof. In the proof we consider only circuits which are already drawn in a plane with the use of *XNOR*-gates. Then we show the statement by induction on the number of gates i with $s(i) \in G$ in a circuit C where $G = \{\neg, \vee, \mathit{split}, \mathit{XNOR}\}$.

(a) A_C for a variable (b) A_C for T

Figure 4.11: Reductions corresponding to circuits consisting of one gate.

For the base case we have to consider all possible circuits which have no gate i with $s(i) \in G$. Clearly these circuits consist of a single input node and therefore we have to distinguish three different cases. The first possible circuit consists of a single variable node. Hence it is satisfiable as $f_C(1) = 1$. Then we have to verify that A_C which is given Figure 4.11(a) has a solution. Therefore just put a light in the left white cell. The second case is a single T-node. A solution to the reduction A_C as shown in picture (b) of Figure 4.11 is again found by placing a light in the left white cell. The third case is a single F-node. As for this node there is no assignment under which f_C evaluates to 1 the statement holds trivially.

For the step case we consider a boolean circuit C with $n+1$ gates which are not input nodes, i.e, they don't have a sort which is element of G. Assume that there is a tuple $(a_1, \ldots, a_m) \in \{0,1\}^m$ such that $f_C(a_1, \ldots, a_m) = 1$. Next we search for the smallest gate j with $s(j) \in G$ such that for all $(i, j) \in E$ $s(i) \in I$. As the number of non-input gates is at least one such

a gate j exists. If this wasn't the case then somewhere in C there would be a cycle and therefore C wouldn't be a circuit. Under the assumption that the variables $x_1, \ldots, x_m$ which occur in C are of value $a_1, \ldots, a_m$ the gate j has a completely determined boolean value v. Then we construct a circuit C' with n non-input gates by removing gate j and all nodes i with $(i, j) \in E$ as well as all corresponding edges.

- If $s(j) \neq \mathit{split}$ and $j < n$ then there is one gate k with $(j, k) \in E$ in circuit C. But as j is missing in C' and hence k misses one incoming edge we have to find a replacement. If $v = 1$ then we add a new input node i' with $s(i') = T$ to C' and add the edge (i', k). If $v = 0$ then the new node i' and the edge (i', k) are added as well but $s(i') = F$.
- If $s(j) \neq \mathit{split}$ and $j = n$ then j has no outgoing edges. The new circuit C' consists of a single input with sort T if $v = 1$ and of a single input with sort F if $v = 0$.
- If $s(j) = \mathit{split}$ there are two nodes k_1, k_2 with $(j, k_1), (j, k_2) \in E$ in circuit C and therefore two new inputs and two new edges have to be added. Both new inputs have the same sort which again depends on v.

The boolean function $f_{C'}$ of this new circuit C' clearly is 1 for the values $a_1, \ldots, a_m$. Hence we can apply the induction hypothesis to get that the Akari grid $A_{C'}$ has a solution which again can be used to generate a solution for A_C. Therefore we have to consider all possible combinations of gates which could have been removed. Here we explain only the cases for $s(j) = \neg$. For all other cases where $s(j) \in \{\vee, \mathit{split}, \mathit{XNOR}\}$ a solution to A_C can be found in a similar way.

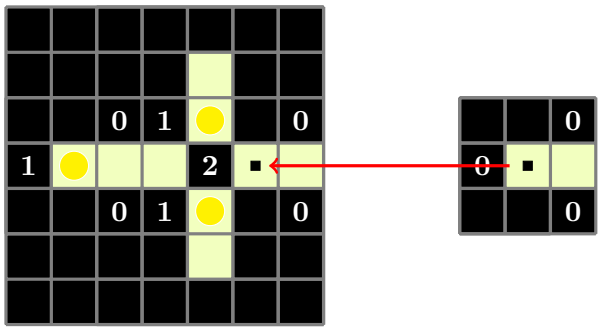


Figure 4.12: Solution to A_C if $\neg T$ is added to C'.

Assume $s(j) = \neg$ and there is an edge from i to j then the actual solution of A_C depends on the sort of i.

- If $s(i) = T$ then independent of a variable assignment the value of v is always 0. Therefore the gate i' which was added to complete circuit C' was of sort F. In the reduction $A_{C'}$ this gate i' is represented by an F-input-tile. If we take the Akari grid $A_{C'}$, chop of the F-input-tile for gate i' and add a $\neg$-tile and a T-input-tile we get an Akari grid

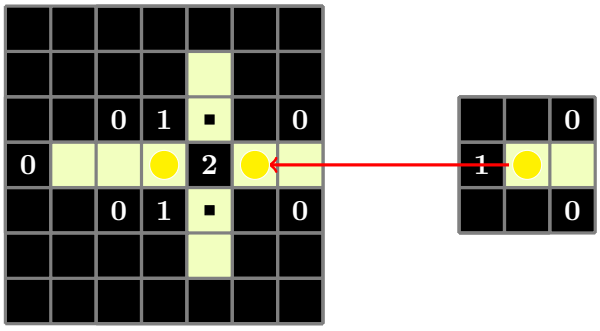


Figure 4.13: Solution to A_C if $\neg F$ is added to C'.

which is the reduction of C disregarding the concrete wire-tiles. A solution to this grid A_C can be constructed by adopting the solution of $A_{C'}$ and completing the new tiles as shown in Figure 4.12.

- If $s(i) = F$ then the value of v is 1. Therefore the gate which was added to get C' had sort T. In a solution of $A_{C'}$ there is always a light in the first white cell of the T-input-tile which represents gate i'. To get a grid which is the reduction of C we remove the T-input-tile and add a $\neg$-tile and an F-input-tile. The actual solution is then derived by completing the solution of $A_{C'}$ as illustrated in Figure 4.13.
- If $s(i) \in \{x_1, \ldots, x_m\}$ then v could have value 0 or value 1. Therefore we have to guarantee that A_C has a solution in either of both cases. This time we get A_C by again removing the input-tile which corresponds to the new gate i' and by adding a $\neg$-tile and a variable-tile. That a solution for A_C exists independent of the value of v is shown in Figure 4.14.

□

Lemma 4.11. *Given a boolean circuit C with boolean function f_C. If A_C has a solution then there is a tuple $(a_1, \ldots, a_m) \in \{0,1\}^m$ such that $f_C(a_1, \ldots, a_m) = 1$.*

The proof for this lemma has the same structure as the one for Lemma 4.10, i.e., the lemma is shown by induction on the number of gates which are not inputs. The base case is almost the same. In the step case again a circuit C' is considered where we remove a gate which has only edges coming from variable nodes.

□

To conclude this section about NP-completeness an example reduction for a small circuit is given.

Example 4.12. The circuit for which an Akari grid is created is given in Figure 4.15. It consists of six gates, three of them inputs namely x, y and z, a

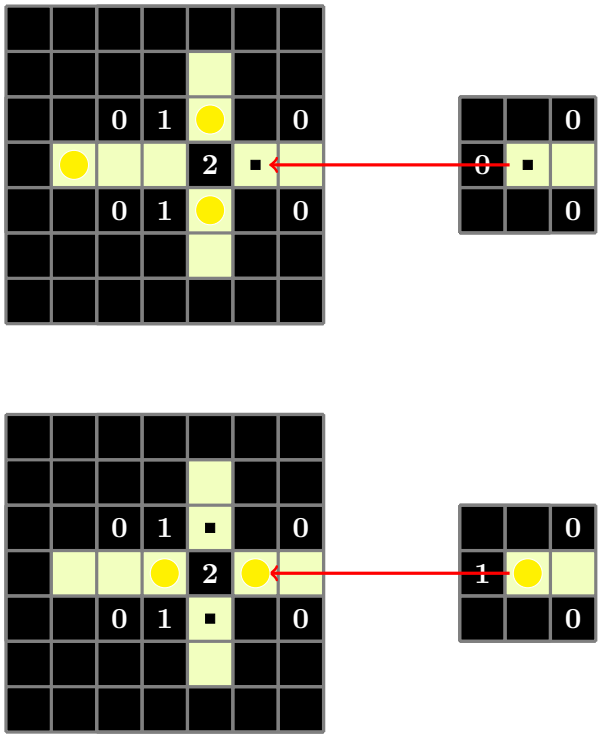


Figure 4.14: Solution to A_C if $\neg x$ is added to C'.

¬-gate and two ∨-gates. The reduction of this circuit is shown in Figure 4.16. In this picture the edges of the single tiles are indicated by blue boxes. For each of the inputs one variable-tile is placed on the Akari grid. Further there are two ∨-tiles and one ¬-tile corresponding to the two ∨-gates and the ¬-gate. The two lower variable-tiles correspond to inputs x and y and therefore are connected with wire-tiles to the first ∨-tile. The uppermost variable-tile is the reduction of z. It is attached to the ¬-tile with a straight-wire-tile. The ¬-tile and the first ∨-tile then yield the two inputs for the second ∨-tile. The last tile on the right side of the grid is a T-output-tile. It closes the output wire of the second ∨-tile and stands for an output value of 1.

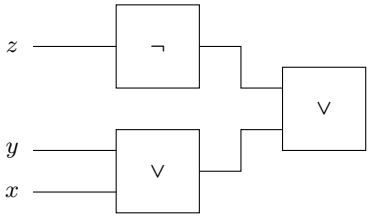


Figure 4.15: Boolean circuit corresponding to $\neg z \vee (x \vee y)$.

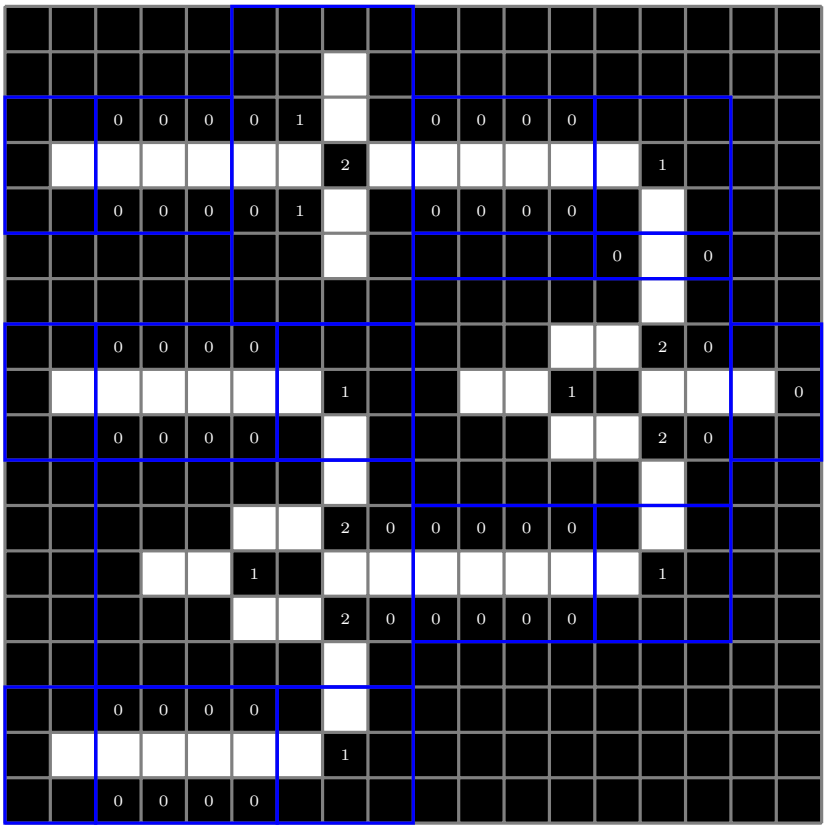


Figure 4.16: Reduction of the boolean circuit shown in Figure 4.15.

5 Arukone

Arukone, also called Number Link, is another logic puzzle originating from the Japanese publisher Nikoli. It seems to be one of the less common puzzles as only very little information on this puzzle is available.

At the beginning of this chapter the rules of Arukone are explained and we give a formal definition of the game. Then the different encodings are introduced and discussed. The key problem concerning an encoding is to express that two cells in a grid are connected with a continuous line. The representation of this property requests to think about a whole concept which expresses connectivity. This task is not as simple as the encodings of the features of Akari and Sudoku which used only constraints that followed immediately from the definition of the puzzles. In the following we introduce two approaches to define connectivity which are known from graph theory and explain how they can be encoded into logic. The section about experimental results focuses on comparing the performance of these two methods for each logical language. In the last section we present an NP-completeness proof for Arukone. This appears to be a new result.

				1
4		**2**	**3**	
				1
3	**2**			**4**

Figure 5.1: Arukone grid of size 5×5.

5.1 Problem Definition

Arukone is played on a rectangular grid of white cells in which some of the cells contain numbers. In each Arukone grid there are at least two numbers and each number appears exactly twice on the grid. An example grid is given in Figure 5.1. To construct a solution we have to draw lines through all white cells such that the following conditions hold:

- each matching pair of numbers is connected with a continuous line,
- in each white cell without number there is exactly one line,

- lines must not cross, branch-off or travel through cells with numbers.

Solving an Arukone puzzle demands more sophisticated reasoning than solving an Akari or Sudoku grid. This is due to the fact that often the only means to come to the decision if a certain line can be placed in a cell is by considering the influence of placing that line on the possibility to connect the other numbers.

Example 5.1. The puzzle shown in Figure 5.1 is a very easy one but still the outline of its solving process illustrates the main techniques which are commonly used to derive a solution.

- A possible starting point is the 3 in cell $(5, 1)$. There is only one possibility to draw a line starting at this 3, that is through cells $(4, 1)$ and $(4, 2)$ whereas the shape of the line in $(4, 2)$ is not clear yet. As cell $(4, 2)$ is already reserved for number 3, the line starting at the 2 in cell $(5, 2)$ can only take a path via cell $(5, 3)$. The concrete shape of the line in $(5, 3)$ is then determined by the fact that cell $(5, 4)$ has to contain a line starting at the 4 in $(5, 5)$. This is the case because cell $(5, 4)$ is the only neighbour cell of $(5, 5)$ which does not have a number. The grid with all these newly located lines is given in Figure 5.2(a).

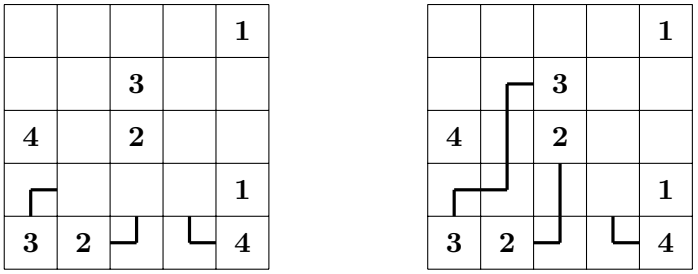


Figure 5.2: Partial solutions of the Arukone grid given in Figure 5.1.

- Now it is easy to see that there is only one kind of line admissible in cell $(4, 3)$ – the one which goes straight through a cell from the bottom to the top. A turn to the left is not possible as cell $(4, 2)$ is reserved for a line connecting the 3's and a turn to the right is not feasible as cell $(4, 4)$ is the only possible continuation of the line in cell $(5, 4)$.

- Next the line connecting the two 3's can be finished the way it is shown in Figure 5.2(b). The cells $(4, 2)$ and $(3, 2)$ are clearly the only possibility to continue the line coming from cell $(4, 1)$. Any longer path is not possible as such a path has to go through the cells $(1, 2)$ and $(1, 3)$ and as a consequence the 4 in cell $(3, 1)$ would be cut off from the rest of the grid and therefore could not be connected to the other 4 any more.

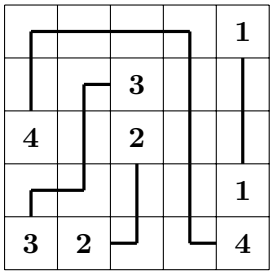


Figure 5.3: Solution of the Arukone grid given in Figure 5.1.

- After these steps the solution of the grid given in Figure 5.3 is very easy to find. It is the only possible connection for the last two remaining pairs of numbers.

The formal definition of an Arukone grid as a function A and a correct solution to A are defined in the following way:

Definition 5.2. For $m, n \in \mathbb{N}$ with $m, n \geq 2$ an $m \times n$ *Arukone grid* is a function

$$A : M \times N \to \{w\} \cup \mathbb{N}$$

with $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$ where w stands for white and it holds that

1. $\exists (i,j) \in M \times N\colon\ A(i,j) \in \mathbb{N}$,
2. $A(i,j) \in \mathbb{N} \implies \exists i', j'$ with $(i,j) \neq (i',j')$ and $A(i,j) = A(i',j')$ such that $(\forall (k,l) \in M \times N\colon\ (i,j) \neq (k,l) \neq (i',j') \implies A(k,l) \neq A(i,j))$.

Definition 5.3. A *solution* to an $m \times n$ Arukone grid A is a function

$$A' : M \times N \to \mathit{lines} \cup \mathbb{N}$$

where $\mathit{lines} = \{-, |, \ulcorner, \llcorner, \lrcorner, \urcorner\}$ and where the following properties hold:

1. if $A(i,j) = w$ then $A'(i,j) \in \mathit{lines}$, otherwise $A'(i,j) = A(i,j)$,
2. $\forall (i,j), (k,l) \in M \times N$ with $(i,j) \neq (k,l)$ and $A(i,j), A(k,l) \in \mathbb{N}$:
   - if $A(i,j) = A(k,l)$ then there is a line from (i,j) to (k,l),
   - if $A(i,j) \neq A(k,l)$ then there is no line from (i,j) to (k,l),
3. if $A(i,j) \in \mathbb{N}$ then (i,j) is directly connected to exactly one cell $(k,l) \in N_4(i,j)$,
4. if $A(i,j) = w$ then there is a line to at least one (i,j) with $A(i,j) \in \mathbb{N}$.

The meaning of the term "there is a line from c_1 to c_2" is not defined formally as it would require a formal definition of connectivity but intuitively it is clear that it signifies that there is a continuous line from c_1 to c_2 which passes at least one other cell. The above mentioned notion of direct connectivity is defined as follows:

Definition 5.4. Two cells $(i,j),(k,l) \in M \times N$ are *directly connected* if and only if (i,j) and (k,l) are adjacent and one of the following two properties holds:

- $l = j + 1$ and $A'(i,j) \in \{-, \ulcorner, \llcorner\}$ and $A'(k,l) \in \{-, \lrcorner, \urcorner\}$,
- $k = i + 1$ and $A'(i,j) \in \{|, \ulcorner, \urcorner\}$ and $A'(k,l) \in \{|, \llcorner, \lrcorner\}$.

5.2 Propositional Logic

The first step is to define a set of variables for the propositional encoding. As there are six different ways to draw a line through a cell, for each white cell (i,j) we introduce six variables $C_{i,j,L}$ with $L \in \mathit{lines}$. If a variable $C_{i,j,L}$ is true then a line of shape L is in cell (i,j). To refer to cells which are labelled with numbers we have different possibilities. One is to use a single variable $C_{i,j}$ to denote the numbered cell (i,j), another is to define the six variables for the six lines as well as for white cells and set all of them to true. In the propositional encoding which is presented in this section the second approach is taken as it allows an easier description and implementation of the encoding. In total this yields $6 * m * n$ variables for an $m \times n$ Arukone grid.

Given an $m \times n$ Arukone grid we first consider all numbered cells and set all possible lines to true:

$$\forall 1 \le i \le m,\ \forall 1 \le j \le n \text{ with } A(i,j) \in \mathbb{N}\colon\ \bigwedge_{L \in lines} C_{i,j,L}$$

For all white cells we have to guarantee that there is exactly one line per cell:

$$\forall 1 \le i \le m,\ \forall 1 \le j \le n \text{ with } A(i,j) = w\colon\ \mathit{exact}(i,j,\mathit{lines})$$

where

$$\mathit{exact}(i,j,S) = \bigvee_{L \in S} C_{i,j,L} \wedge \bigwedge_{L \in S} \bigwedge_{L' \in S \setminus \{L\}} (\neg C_{i,j,L} \vee \neg C_{i,j,L'})$$

This minimal encoding of the possible lines in white cells can be replaced by another minimal set of formulas which takes into consideration that the actual number of lines per cell is not always six but may be restricted by the border of the grid. If we consider for example cell $(1,1)$ in the upper left corner, here the only line which is admissible is $\ulcorner$ and therefore the variable $C_{1,1,\ulcorner}$ can be set immediately to true whereas all other variables for $(1,1)$ are set to false. If this encoding approach is used some of the clauses are smaller and the values of some variables are fixed already at the beginning of the computation. For that reason we extend the definition of $\mathit{exact}(i,j,S)$ to $\mathit{exact}(i,j,S) \wedge \bigwedge_{L \in lines \setminus S} \neg C_{i,j,L}$. The possible lines per cell are then encoded by

$$\forall 1 \le i \le m,\ \forall 1 \le j \le n \text{ with } A(i,j) = w\colon\ \mathit{exact}(i,j,\mathit{lin}(i,j))$$

where $\mathit{lin}(i,j)$ is the set of all lines which are possible in cell (i,j):

$$lin(i,j) = \begin{cases} \{\ulcorner\} & \text{if } i = 1,\ j = 1 \\ \{-, \ulcorner, \urcorner\} & \text{if } i = 1,\ 1 < j < n \\ \{\urcorner\} & \text{if } i = 1,\ j = n \\ \{|, \ulcorner, \llcorner\} & \text{if } 1 < i < m,\ j = 1 \\ \{|, \lrcorner, \urcorner\} & \text{if } 1 < i < m,\ j = n \\ \{\llcorner\} & \text{if } i = m,\ j = 1 \\ \{-, \llcorner, \lrcorner\} & \text{if } i = m,\ 1 < j < n \\ \{\lrcorner\} & \text{if } i = m,\ j = n \\ lines & \text{otherwise} \end{cases} \tag{5.1}$$

To encode any of the other features of Arukone we first need to define variables which allow us to express that two cells are connected. In this section we introduce two different connectivity concepts which are not only used for the propositional encoding but also for PB and SMT. The ideas of these two concepts are also discussed in [8]. In Section 5.5 the performance of both approaches is compared. The first representation of connectivity is an encoding of the Floyd-Warshall algorithm. This algorithm is named after its inventors Robert Floyd [13] and Stephen Warshall [26] and computes the shortest path between each two nodes of a graph. In the version we present here we use the basic idea of the algorithm but instead of the shortest path we simply compute if there is a path between two nodes.

The algorithm as presented by Warshall starts with the adjacency matrix R_0 of a graph (E, V) which is a boolean matrix. Let us fix $e = |E|$ then the algorithm needs e iterations to compute a boolean matrix R_e in which an entry at position $[u, v]$ is 1 exactly if u and v are connected in the graph. The concrete computation looks as follows:

$$R_0[u,v] = \begin{cases} 1 & \text{if } u = v \text{ or } (u,v) \in V \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

$$\forall 1 \le k \le e\colon\ R_k[u,v] = R_{k-1}[u,v] \vee (R_{k-1}[u,k] \wedge R_{k-1}[k,v])$$

The idea is that two nodes u and v are connected at level k if there is a path from u to the intermediate node k and another path from k to v at level $k-1$ or, of course, if u and v are already connected at a lower level.

To apply the Floyd-Warshall algorithm to an Arukone grid we view a grid of size $m \times n$ as a graph with $m * n$ vertices where each cell (i_u, j_u) of the grid corresponds to a vertex u with $u = (i_u - 1) * n + j_u$. In the following we write (i_u, j_u) for u and vice versa whenever it simplifies the representation of formulas. Between two vertices there is an edge if they are directly connected in the solution of the Arukone grid. Next the previous definition has to be translated from matrices to propositional formulas. Therefore for each matrix entry $R_k[u, v]$ a variable $R_{u,v,k}$ is introduced which yields $e^2 * (e + 1) = e^3 + e^2$ new variables with $e = |E| = m * n$. In fact the number of necessary variables per matrix R_k can be reduced from e^2 to $\frac{1}{2} * (e+1) * e$ by defining only one variable for $R_k[u, v]$ and $R_k[v, u]$. This is admissible as the graph corresponding to

an Arukone grid is undirected and therefore the adjacency matrix R_0 and as a consequence all other matrices R_k are symmetric and hence two entries $R_k[u, v]$ and $R_k[v, u]$ are always equal. In this manner the total amount of variables for the Floyd-Warshall encoding is reduced from $e^3 + e^2$ to $\frac{1}{2} * (e^3 + 2 * e^2 + e)$.

As the actual lines on the grid and therefore the edges of the corresponding graph are not assessed initially but are computed during the solving process the definition of the initial matrix R_0 has to be changed in order to be suitable for a changing placement of lines:

$$\forall 1 \leq u \leq v \leq e\colon\ R_{u,v,0} \leftrightarrow \begin{cases} T & \text{if } u = v \\ \psi(\{-, \ulcorner, \llcorner\}, \{-, \lrcorner, \urcorner\}, u, v) & \text{if } i_u = i_v,\ j_v = j_u + 1 \\ \psi(\{|, \ulcorner, \urcorner\}, \{|, \llcorner, \lrcorner\}, u, v) & \text{if } j_u = j_v,\ i_v = i_u + 1 \\ F & \text{otherwise} \end{cases}$$

where (5.3)

$$\psi(S, S', u, v) = \begin{cases} F & \text{if } A(i_u, j_u),\ A(i_v, j_v) \in \mathbb{N} \\ \bigvee_{L \in S}\ \bigvee_{L' \in S'}\ C_{i_u,j_u,L} \wedge C_{i_v,j_v,L'} & \text{otherwise} \end{cases}$$

If u and v are equal or if they are not adjacent the value of $R_{u,v,0}$ can be determined immediately. In turn, if both cells are adjacent the variable $R_{u,v,0}$ is false if u and v both contain a number. If this is not the case, $R_{u,v,0}$ is true if and only if the two lines which are currently placed in u and v form a continuous line. These properties for adjacent cells are expressed in ψ. For matrices with index $k > 0$ the definition of the Floyd-Warshall algorithm can be applied one-to-one to Arukone grids:

$$\forall 1 \leq k \leq e,\ \forall 1 \leq u \leq v \leq e:\ R_{u,v,k} \leftrightarrow R_{u,v,k-1} \vee (R_{u,k,k-1} \wedge R_{k,v,k-1})$$

The last step in this encoding is the transformation of the single formulas to CNF. For all equivalences concerning variables with $k > 0$ this requires only a few steps in the standard transformation. Also the $\leftarrow$-side of equivalences concerning variables of level 0 is transformed instantly to CNF by replacing the implication by its definition via $\vee$. The only kind of formula which is left to transform is the $\rightarrow$-side of level 0 equivalences. Unfortunately the right-hand side of these formulas is in DNF and the CNF generated by the application of the standard transformation is very large. Normally in such cases Tseitin's transformation is used to create an equisatisfiable formula in CNF. But in this case we might as well apply another method to get an equivalent formula in CNF namely by reading the CNF out of the truth table. This is done only once as structure and size of the formulas in question are always identical. This way we can omit the use of additional variables. Moreover a few experiments showed that for these formulas CNF generation via truth tables performs better than CNF generation with Tseitin's transformation.

The second approach concerning connectivity is the logarithmic encoding. A detailed description of this encoding and of its translation to CNF is given in [24]. The advantage of the logarithmic encoding is that – without considering

the transformation to CNF – it requires less variables than the one based on Floyd-Warshall. The idea is that if there is a path of length n between two nodes u and v then there exist an intermediate node k on this path such that the path from u to k and the path from k to v are at most of length $\frac{n+1}{2}$. If the two endpoints of a subpath are not equal then this path is again searched for an intermediate node which serves to split the subpath into two almost equally long smaller paths. This process is continued until the whole path is divided into subpaths of length 1. Due to the consequent bisecting of paths the number of iterations which are necessary to delimit the connectivity of all nodes is limited to $\lceil \log_2(e-1) \rceil$.

If we apply this idea to an $m \times n$ Arukone grid again variables $R_{u,v,k}$ are introduced but their meaning slightly changes. $R_{u,v,k}$ is true if there is a path from u to v with length at most 2^k. The equations which correspond to the definition of the adjacency matrix R_0 are the same as in the encoding of the Floyd-Warshall algorithm. This is intuitively clear as the starting point of the computation – the adjacency matrix – does not change if another algorithm is chosen. We set the number of necessary iterations max to $\lceil \log_2(e-1) \rceil$ for $e = m * n$ and then define all variables of higher levels as follows:

$$\forall 1 \leq k \leq max,\ \forall 1 \leq u \leq v \leq e:\ R_{u,v,k} \leftrightarrow \bigvee_{c=1}^{e} R_{u,c,k} \wedge R_{v,c,k}$$

The number of variables which is required by this encoding is $\frac{1}{2} * (e+1) * e * \lceil \log_2(e-1) \rceil$. In comparison to the first encoding of connectivity were $\frac{1}{2} * (e+1) * e * (e+1)$ variables are used this is an improvement. But in turn we need a lot of additional variables as it is necessary to apply Tseitin's transformation in order to get equisatisfiable formulas in CNF. The $\leftarrow$-side of the equivalence is immediately transformed to CNF and hence we only have to care about the other direction were Tseitin's transformation together with the improvements which are explained in Chapter 2 is applied. The formulas in question are all of the following shape:

$$R_{u,v,k} \rightarrow \bigvee_{c=1}^{e} R_{u,c,k} \wedge R_{v,c,k} \ \equiv\ \neg R_{u,v,k} \vee \bigvee_{c=1}^{e} R_{u,c,k} \wedge R_{v,c,k}$$

For each of the conjunctions $R_{u,c,k} \wedge R_{v,c,k}$ in this large disjunction a new variable $x_{u,v,c,k}$ is introduced. Then a formula which is satisfiable if and only if the original disjunction is satisfiable is established:

$$(\neg R_{u,v,k} \vee \bigvee_{c=1}^{e} x_{u,v,c,k}) \ \wedge\ \bigwedge_{c=1}^{e} (x_{u,v,c,k} \rightarrow R_{u,c,k} \wedge R_{v,c,k})$$

To convert this formula to CNF only the small implications have to be transformed which is straightforward. Nevertheless we have to keep in mind that per disjunction e new variables are needed and as for each level $k > 0$ $\frac{1}{2} * (e+1) * e$ such disjunctions appear we have $\frac{1}{2} * (e+1) * e^2 * \lceil \log_2(e-1) \rceil$ additional variables.

Having defined these two methods to encode connectivity in Arukone grids we are now ready to give an encoding of the remaining characteristics of a correctly solved $m \times n$ Arukone grid A as it is defined in Definition 5.3. If the Floyd-Warshall encoding is used to express connectivity then $max = m * n$, for the logarithmic encoding $max = \lceil \log_2(m * n - 1) \rceil$:

- two equal numbers are connected:

  $\forall u, v \in M \times N$ with $u \neq v$, $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$ and $A(i_u, j_u) = A(i_v, j_v)$: $R_{u,v,max}$,

- two different numbers must not be connected:

  $\forall u, v \in M \times N$ with $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$, $A(i_u, j_u) \neq A(i_v, j_v)$: $\neg R_{u,v,max}$,

- a cell with a number is directly connected to exactly one cell:

  $\forall t \in M \times N$ with $A(i_t, j_t) \in \mathbb{N}$:

  $\bigvee_{u \in N_4(i_t,j_t)} R_{t,u,0} \wedge \bigwedge_{u \in N_4(i_t,j_t)} \bigwedge_{v \in N_4(i_t,j_t), u \neq v} (\neg R_{t,u,0} \vee \neg R_{t,v,0})$,

- each white cell is connected to a number:

  $\forall u \in M \times N$ with $A(i_u, j_u) = w$: $\bigvee_{v \in M \times N, A(i_v,j_v) \in \mathbb{N}} R_{u,v,max}$.

Note that by requiring that two nodes are not reachable at the highest level (second item) they also must not be reachable at any lower level. This is a consequence in both connectivity concepts. The disjunction which is described in the fourth item – each white cell reaches at least one number – could be reduced by adding just one variable per pair of equal numbers to the disjunction instead of both.

The formulas which are presented up to now cover all properties that have to be encoded in order to get a correct solution and constitute a minimal encoding. Nevertheless there is an additional feature of Arukone for which an encoding is presented as it significantly improves the performance. Consider a white cell of an Arukone grid where a certain line is placed. This line connects the white cell to exactly two of its neighbours at level 0 whereas the other two neighbours are not connected to this cell at level 0 (if the white cell is not at the border). To encode this property we use the following formulas:

- $C_{i,j,-} \rightarrow R_{(i,j-1),(i,j),0} \wedge R_{(i,j),(i,j+1),0} \wedge \neg R_{(i-1,j),(i,j),0} \wedge \neg R_{(i,j),(i+1,j),0}$,
- $C_{i,j,|} \rightarrow R_{(i-1,j),(i,j),0} \wedge R_{(i,j),(i+1,j),0} \wedge \neg R_{(i,j-1),(i,j),0} \wedge \neg R_{(i,j),(i,j+1),0}$,
- $C_{i,j,\ulcorner} \rightarrow R_{(i,j),(i,j+1),0} \wedge R_{(i,j),(i+1,j),0} \wedge \neg R_{(i-1,j),(i,j),0} \wedge \neg R_{(i,j-1),(i,j),0}$,
- $C_{i,j,\llcorner} \rightarrow R_{(i-1,j),(i,j),0} \wedge R_{(i,j),(i,j+1),0} \wedge \neg R_{(i,j-1),(i,j),0} \wedge \neg R_{(i,j),(i+1,j),0}$,
- $C_{i,j,\lrcorner} \rightarrow R_{(i-1,j),(i,j),0} \wedge R_{(i,j-1),(i,j),0} \wedge \neg R_{(i,j),(i,j+1),0} \wedge \neg R_{(i,j),(i+1,j),0}$,
- $C_{i,j,\urcorner} \rightarrow R_{(i,j-1),(i,j),0} \wedge R_{(i,j),(i+1,j),0} \wedge \neg R_{(i-1,j),(i,j),0} \wedge \neg R_{(i,j),(i,j+1),0}$.

Note that such an implication for a cell (i, j) and a line L is only added to the set of formulas if it is in principle admissible to put a line L in cell (i, j). Further a negated variable $\neg R_{u,v,0}$ is only added to the conjunction on the right-hand side of the implication if u and v are on the grid.

5.3 Pseudo-Boolean Logic

For Sudoku and Akari pseudo-boolean logic provides mostly very convenient means to encode the single features of the puzzles. Interestingly it turns out that pseudo-boolean logic isn't that suitable to express more complex properties like connectivity. A major drawback is that there exists no construction which is analogue to the propositional implication. This is the main reason why some of the pseudo-boolean constraints which are presented in this section are not intuitively understandable and not very easy to read.

The pseudo-boolean encoding of Arukone uses the same set of variables as the propositional one. Hence for each cell (i, j) in an $m \times n$ Arukone grid we have six variables $C_{i,j,L}$ – one for each of kind of line. Again we require that in each cell without number there is exactly one line and in each cell with number all lines are true. As in the propositional encoding we take care that lines which can never be placed in a cell due to the border are excluded from the set of possible lines:

$\forall 1 \leq i \leq m,\ \forall 1 \leq j \leq n$:

- if $A(i,j) \in \mathbb{N}$ then $\forall L \in \mathit{lines}\colon\ C_{i,j,L} = 1$,
- if $A(i,j) = w$ then $\sum_{L \in lin(i,j)} C_{i,j,L} = 1$ and $\sum_{L \in lines \setminus lin(i,j)} C_{i,j,L} = 0$

where the definition of $lin(i, j)$ corresponds to equation 5.1.

To express connectivity we again use the two approaches which where already presented in Section 5.2 and define their representation as pseudo-boolean constraints. Remember that independent of the chosen connectivity concept – Floyd-Warshall or logarithmic – the encoding of the adjacency matrix, i.e., the definition of the connectivity variables of level 0, is the same. To find a pseudo-boolean representation of the adjacency matrix we take its propositional encoding according to equation 5.3 and transform it. The easy part is to translate $R_{u,v,k} \leftrightarrow T$ to $R_{u,v,k} = 1$ and $R_{u,v,k} \leftrightarrow F$ to $R_{u,v,k} = 0$. The more difficult part is to find an appropriate pseudo-boolean constraint to encode that two cells are reachable at level 0 if they are directly connected. Therefore we recall the corresponding propositional formula for adjacent cells which is

$$R_{u,v,0} \leftrightarrow \bigvee_{L \in S}\ \bigvee_{L' \in S'}\ C_{i_u,j_u,L} \wedge C_{i_v,j_v,L'}$$

In the $\rightarrow$-direction we have that if $R_{u,v,0}$ is true then one line from S is in u and one line from S' is in v. In pseudo-boolean logic these implications are expressed by the following two constraints:

- $-R_{u,v,0} + \sum_{L \in S} C_{i_u,j_u,L} \geq 0$,
- $-R_{u,v,0} + \sum_{L \in S'} C_{i_v,j_v,L} \geq 0$.

The other direction of the equivalence states that $R_{u,v,0}$ is true whenever there are two variables $C_{i_u,j_u,L}$ and $C_{i_v,j_v,L'}$ with $L \in S$ and $L' \in S'$ which are true:

$$\forall L \in S,\ \forall L' \in S' :\ 2 * R_{u,v,0} - C_{i_u,j_u,L} - C_{i_v,j_v,L'} \geq -1$$

To express connectivity on a higher level the two different approaches have to be distinguished. First we focus on the encoding corresponding to the Floyd-Warshall algorithm as defined in equation 5.2. The two pseudo-boolean constraints which encode the definition of an entry $R_k[u, v]$ of a matrix R_k with $k > 0$ are

- $2 * R_{u,v,k-1} + R_{u,k,k-1} + R_{k,v,k-1} - 3 * R_{u,v,k} \geq -1$,
- $2 * R_{u,v,k-1} + R_{u,k,k-1} + R_{k,v,k-1} - 3 * R_{u,v,k} \leq 1$.

To demonstrate that these constraints really determine the value of $R_{u,v,k}$ correctly assume that a variable $R_{u,v,k}$ is true. Then by the first constraint the variable $R_{u,v,k-1}$ which states that u and v are reachable already at level $k-1$ has to be true or – if $R_{u,v,k-1}$ is false – then both $R_{u,k,k-1}$ and $R_{k,v,k-1}$ have to be true and therefore u and v are connected via node k. On the other hand, if $R_{u,v,k}$ is false then it is not possible that $R_{u,v,k-1}$ is true because this would violate the second constraint. The second constraint also forbids that $R_{u,k,k-1}$ and $R_{k,v,k-1}$ are true at the same time if $R_{u,v,k}$ is false.

The second approach to express connectivity of different cells in the Arukone grid is the logarithmic encoding. To find a suitable PB encoding for reachability of higher levels we start with the corresponding expressions in propositional logic which are always of the following form:

$$R_{u,v,k} \leftrightarrow \bigvee_{c=1}^{e} R_{u,c,k} \wedge R_{v,c,k}$$

We first consider the easier direction of the equivalence which is the one from right to left. By an application of De Morgan's law we get

$$R_{u,v,k} \leftarrow \bigvee_{c=1}^{e} R_{u,c,k} \wedge R_{v,c,k} \;\equiv\; \bigwedge_{c=1}^{e} (R_{u,v,k} \vee \neg R_{u,c,k} \vee \neg R_{v,c,k})$$

Now it is easy to define PB constraints which are equivalent to the clauses of this formula in CNF:

$$\forall 1 \leq c \leq e: \; -2 * R_{u,v,k} + R_{u,c,k} + R_{v,c,k} \leq 0$$

The PB constraints corresponding to the $\rightarrow$-direction of the equation require the use of additional variables like it is the case for propositional logic. As starting point of the construction we take the propositional formula after Tseitin's transformation:

$$(\neg R_{u,v,k} \vee \bigvee_{c=1}^{e} x_{u,v,c,k}) \;\wedge\; \bigwedge_{c=1}^{e} (x_{u,v,c,k} \rightarrow R_{u,c,k} \wedge R_{v,c,k})$$

The first part of the formula, $\neg R_{u,v,k} \vee \bigvee_{c=1}^{e} x_{u,v,c,k}$, is translated one-to-one by demanding that at least one of the variables in this clause is true:

$$-R_{u,v,k} + \sum_{c=1}^{e} x_{u,v,c,k} \geq 0$$

Each implication $x_{u,v,c,k} \rightarrow R_{u,c,k} \wedge R_{v,c,k}$ is expressed in PB by the following constraint:

$$-2 * x_{u,v,c,k} + R_{u,c,k} + R_{v,c,k} \geq 0$$

To improve performance we add redundant constraints of the form

$$-2 * x_{u,v,c,k} + R_{u,c,k} + R_{v,c,k} \leq 1$$

which state that if both reachability variables are true then also the new variable $x_{u,v,c,k}$ has to be true.

Clearly in an $m \times n$ Arukone grid all of these constraints have to be introduced for all levels $1 \leq k \leq \lceil \log_2(m * n - 1) \rceil$ and all nodes $1 \leq u \leq v \leq m * n$. For the logarithmic as well as for the Floyd-Warshall encoding the number of required variables is the same as for the connectivity encodings in propositional logic.

It remains to present the encoding of the remaining properties which constitute a minimal representation of an Arukone grid. Depending on the chosen concept to express connectivity $max = m * n$ in case of the Floyd-Warshall encoding and $max = \lceil \log_2(m * n - 1) \rceil$ in case of the logarithmic encoding:

- two equal numbers are connected:

  $\forall u, v \in M \times N$ with $u \neq v$, $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$ and $A(i_u, j_u) = A(i_v, j_v)$: $R_{u,v,max} = 1$,

- two different numbers must not be connected:

  $\forall u, v \in M \times N$ with $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$ and $A(i_u, j_u) \neq A(i_v, j_v)$: $R_{u,v,max} = 0$,

- a cell with a number is directly connected to exactly one cell:

  $\forall u \in M \times N$ with $A(i_u, j_u) \in \mathbb{N}$: $\sum_{v \in N_4(i_u, j_u)} R_{u,v,0} = 1$,

- each white cell is connected to a number:

  $\forall u \in M \times N$ with $A(i_u, j_u) = w$: $\sum_{v : A(i_v, j_v) \in \mathbb{N}} R_{u,v,max} \geq 1$.

As in the propositional encoding we add a set of redundant constraints which deals with the implications of the placement of a certain line L in a cell (i, j). Assume that L is placed (and allowed to place) in cell (i, j). As a consequence there are two cells u_1 and u_2 which are directly connected to cell (i, j). For these two cells and this line L this implication is reflected by the following PB constraint:

$$-2 * C_{i,j,L} + R_{(i,j),u_1,0} + R_{(i,j),u_2,0} \geq 0$$

Moreover if L is in cell (i, j) there is a set of cells S containing all cells adjacent to (i, j) which must not be directly connected to (i, j) as a consequence of L. If (i, j) is in a corner of the grid this set is empty, if it is somewhere at the border but not in a corner then S contains one element and in all other cases it has two elements. From this fact a constraint of the following form can be derived

$$|S| * C_{i,j,L} + \sum_{u \in S} R_{(i,j),u,0} \leq |S|$$

which demands that all variables $R_{(i,j),u,0}$ are false if L is in (i, j). This constraint has another, rather convenient implication. If one of the reachability variables $R_{(i,j),u,0}$ is true and hence (i, j) and u are directly connected then it is not possible any more to have line L in (i, j).

These previous two constraints are added for every cell and for every line which can be put in that cell. In addition the second constraint is only constructible if S is not the empty set. We omit a complete listing of all constraints as the structure is the same as given in Section 5.2 and can be easily adjusted to pseudo-boolean logic.

5.4 SMT

Depending on the chosen theory SMT may provide a very rich language which in the case of Arukone allows to specify a solution to an Arukone grid in a very compact way. In Section 3.4 it was already mentioned that the SMT solver `Yices`, the solver on which all SMT encodings were tested in our experiments, decides the respective theory on its own. Therefore we do not care so much about the theory but simply use the language features `Yices` provides to its users.

For an $m \times n$ Arukone grid two sets of variables are defined. One consists of integer variables ranging from 1 to 6 where for each cell (i, j) one variable $C_{i,j}$ is introduced. The value of a variable $C_{i,j}$ indicates the line which is placed in this cell (i, j). The other set consists of boolean variables $R_{u,v,k}$ which are used to express connectivity as introduced in the previous sections. The size of this set depends on the connectivity concept which is chosen in the actual encoding. To define integer variables with a limited range in `Yices` a subset of the natural numbers has to be determined by defining a subtype $L = \{1, \ldots, 6\}$. Then the type of each variable $C_{i,j}$ is defined to be L. In addition we use redundant constraints of the form $C_{i,j} = 1 \vee \ldots \vee C_{i,j} = 6$ which state all possible values of a variable explicitly to improve performance. The type of all variables of kind $R_{u,v,k}$ is set to $\mathbb{B}$. In comparison to the previous Arukone encodings it is clearly not possible that for a cell containing a number all possible lines are set to true as a variable has exactly one value. Therefore the value of the variable for such a cell is always set to 1:

$\forall 1 \leq i \leq m,\ \forall 1 \leq j \leq n$:

- if $A(i, j) = w$ then $\bigvee_{l=1}^{6} C_{i,j} = l$,
- if $A(i, j) \in \mathbb{N}$ then $C_{i,j} = 1$.

Before we state the two encodings of connectivity in SMT we need to define a few auxiliary functions:

- $up : L \to \mathbb{B}$ where $up(x) = (x = 2) \vee (x = 4) \vee (x = 5)$,
- $down : L \to \mathbb{B}$ where $down(x) = (x = 2) \vee (x = 3) \vee (x = 6)$,
- $left : L \to \mathbb{B}$ where $left(x) = (x = 1) \vee (x = 5) \vee (x = 6)$,
- $right : L \to \mathbb{B}$ where $right(x) = (x = 1) \vee (x = 3) \vee (x = 4)$,
- $horizontal : L \times L \to \mathbb{B}$ where $horizontal(x, y) = right(x) \wedge left(y)$,

- $vertical : L \times L \to \mathbb{B}$ where $vertical(x, y) = down(x) \wedge up(y)$.

To make these functions more understandable we keep the following mapping from numbers in L to actual lines in mind:

—	|	⌜	⌞	⌟	⌝
1	2	3	4	5	6

Then it is clear that $up(x)$ is true if x refers to a type of line which connects a cell to its upper neighbour. The meaning of *down*, *left* and *right* is similar. The functions *horizontal* and *vertical* are merely introduced as abbreviations. $horizontal(x, y)$ is true whenever x is a line connecting a cell to its right neighbour and y is a line connecting a cell to its left neighbour. In the encoding this function comes in if we want to check if two horizontally adjacent cells are directly connected. The function itself does not guarantee that x and y are adjacent and that x is the left neighbour of y therefore *horizontal* has to be applied carefully. The same holds for *vertical* which is used to determine if two cells which are vertically adjacent are directly connected.

In a next step these functions are used to define the adjacency matrix for the Arukone grid:

$$\forall 1 \le u \le v \le e : \; R_{u,v,0} = \begin{cases} T & \text{if } u = v \\ \psi_1(u,v) & \text{if } i_u = i_v \text{ and } j_v = j_u + 1 \\ \psi_2(u,v) & \text{if } j_u = j_v \text{ and } i_v = i_u + 1 \\ F & \text{otherwise} \end{cases}$$

where

$$\psi_1(u,v) = \begin{cases} F & \text{if } A(i_u, j_u),\ A(i_v, j_v) \in \mathbb{N} \\ left(C_{i_v,j_v}) & \text{if } A(i_u, j_u) \in \mathbb{N} \text{ and } A(i_v, j_v) = w \\ right(C_{i_u,j_u}) & \text{if } A(i_v, j_v) \in \mathbb{N} \text{ and } A(i_u, j_u) = w \\ horizontal(C_{i_u,j_u}, C_{i_v,j_v}) & \text{otherwise} \end{cases}$$

$$\psi_2(u,v) = \begin{cases} F & \text{if } A(i_u, j_u),\ A(i_v, j_v) \in \mathbb{N} \\ top(C_{i_v,j_v}) & \text{if } A(i_u, j_u) \in \mathbb{N} \text{ and } A(i_v, j_v) = w \\ down(C_{i_u,j_u}) & \text{if } A(i_v, j_v) \in \mathbb{N} \text{ and } A(i_u, j_u) = w \\ vertical(C_{i_u,j_u}, C_{i_v,j_v}) & \text{otherwise} \end{cases}$$

The definition of $R_{u,v,0}$ for two adjacent cells is more complicated than for propositional and pseudo-boolean logic as in these encodings all lines of a cell with a number are set to true but for SMT we have that there is only one line in a numbered cell. Assume u is the upper neighbour of v and contains a number whereas v is white. By the SMT encoding the line in u is always of shape —. Normally this line does not allow u to be connected to its lower neighbour. Hence we ignore in the definition of ψ_2 the line in cell u completely and demand only that v contains a type of line which allows a cell to be connected to its upper neighbour.

The first connectivity concept we examine is the one on the basis of the Floyd-Warshall algorithm. The translation of the definition for all matrices with $k > 0$ is immediate due to the rich language of SMT. For each entry $R_k[u, v]$ in a matrix R_k with $1 \leq k \leq m * n$ a variable $R_{u,v,k}$ is introduced and the following constraint is added:

$$R_{u,v,k} = R_{u,v,k-1} \vee (R_{u,k,k-1} \wedge R_{k,v,k-1})$$

Also for the second approach, the logarithmic encoding, SMT allows us to directly take over the formula as defined originally. Hence for an $m \times n$ Arukone grid and for all $1 \leq k \leq \lceil \log_2(n-1) \rceil$ the following formula is added:

$$R_{u,v,k} = \bigvee_{c=1}^{m*n} R_{u,c,k} \wedge R_{v,c,k}$$

The SMT encoding of logarithmic connectivity requires far less variables as it is not necessary to squeeze the constraints into the corset of CNF or pseudo-boolean logic.

The encoding of the remaining properties of Arukone is very similar to the encodings given in propositional and pseudo-boolean logic. Again, in case of the Floyd-Warshall encoding $max = m * n$ and in case of the logarithmic encoding $max = \lceil \log_2(m * n - 1) \rceil$:

- two equal numbers are connected:

  $\forall u, v \in M \times N$ with $u \neq v$, $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$ and $A(i_u, j_u) = A(i_v, j_v)$: $R_{u,v,max} = T$,

- two different numbers must not be connected:

  $\forall u, v \in M \times N$ with $A(i_u, j_u), A(i_v, j_v) \in \mathbb{N}$ and $A(i_u, j_u) \neq A(i_v, j_v)$: $R_{u,v,max} = F$,

- a cell with a number is directly connected to exactly one cell:

  $\forall t \in M \times N$ with $A(i_t, j_t) \in \mathbb{N}$:

  $\bigvee_{u \in N_4(i_t, j_t)} R_{t,u,0} \wedge \bigwedge_{u \in N_4(i_t, j_t)} \bigwedge_{v \in N_4(i_t, j_t), u \neq v} (\neg R_{t,u,0} \vee \neg R_{t,v,0})$,

- each white cell is connected to a number:

  $\forall u \in M \times N$ with $A(i_u, j_u) = w$: $\bigvee_{v \in M \times N, A(i_v, j_v) \in \mathbb{N}} R_{u,v,max}$.

The very last property – the fact that lines imply certain cells to be directly connected and others to be not – is expressed similar to the propositional encoding given in Section 5.2.

5.5 Experimental Results

To test the performance of the different encodings, experiments on 50 Arukone puzzles were run. According to their sizes the puzzles are grouped into 5 classes each containing ten instances of the same grid size. All results in this section are given as an arithmetic mean of these ten Arukone puzzles per class. Besides

	Grid size	Floyd-Warshall	Logarithmic
Decisions		7801.70	39303.00
Conflicts	5×5	0.10	0.60
CPU time		0.038 s	0.189 s
Decisions		25694.60	142218.00
Conflicts	6×6	21.60	26.20
CPU time		0.124 s	0.778 s
Decisions		131620.60	791627.00
Conflicts	8×8	74.60	79.60
CPU time		0.926 s	5.773 s
Decisions		268350.90	~ 2 m
Conflicts	9×9	764.50	1092.50
CPU time		5.802 s	91.364 s
Decisions		526774.70	~ 4 m
Conflicts	10×10	11316.20	13834.50
CPU time		78.195 s	1765.661 s

Table 5.1: Performance of `Minisat2` on Arukone grids.

the comparison of the different encodings, the tests aimed also to compare the performance of the two different approaches which are used to express connectivity. All tested Arukone grids are taken from "Janko.at" [1] or self-generated. The time limit was set to one hour, puzzles which needed more than one hour were taken with 3600 seconds into consideration. This affects results where the average CPU time is more than 1500 seconds but less than one hour. If a solver needs more than one hour on all instances of a certain problem class then we write > 1 h. In all tables in this section the letter "m" is used as abbreviation for "million".

Table 5.1 shows the performance results for the SAT solver `Minisat2` on the propositional encoding for both connectivity approaches. Throughout all instances the encoding of connectivity according to the Floyd-Warshall algorithm performs better than the logarithmic encoding. If we compare the amount of variables required by the two different encodings then the logarithmic encoding uses far more variables to express reachability than the encoding of the Floyd-Warshall algorithm. As explained in Section 5.2 this is due to the transformation into CNF which adds a lot of auxiliary variables to the logarithmic encoding. Another reason for the poor performance of the logarithmic encoding could be the fact that it creates very large formulas whereas the formulas generated by the Floyd-Warshall encoding are very compact. In comparison to the results for Sudoku and Akari, the SAT solver needs already a rather long time to solve the relatively small puzzles of size 10×10. This is due to the complex encoding of connectivity which does not allow too much unit propagation and necessitates a large number of variables.

The pseudo-boolean encoding performs even worse than the SAT encoding. Especially the solver `Pueblo` has huge difficulties (Table 5.3) to solve even very

	Grid size	Floyd-Warshall	Logarithmic
Decisions		1.90	60556.30
Conflicts	5×5	0.20	960.20
CPU time		0.816 s	7.873 s
Decisions		259.80	578113.00
Conflicts	6×6	59.40	3413.40
CPU time		2.826 s	46.332 s
Decisions		1086.00	~ 9 m
Conflicts	8×8	284.40	32594.10
CPU time		16.214 s	1862.673 s
Decisions		5031.90	
Conflicts	9×9	1495.80	> 1 h
CPU time		45.550 s	
Decisions		79653.70	
Conflicts	10×10	26583.70	> 1 h
CPU time		498.709 s	

Table 5.2: Performance of `Minisat+` on Arukone grids.

small Arukone grids. Using the logarithmic encoding `Pueblo` is only able to find a solution for puzzles smaller than 8×8 within an hour. This is worse than any human player who is familiar with Arukone. But also the results for `Minisat+` which are given in Table 5.2 are rather bad in comparison to `Minisat2`. The unsatisfying runtime results for the pseudo-boolean encoding reflect the fact that pseudo-boolean logic is not very suitable to encode the properties of Arukone.

	Floyd-Warshall	Logarithmic
Grid size	CPU time	CPU time
5×5	0.568 s	19.179 s
6×6	3.124 s	632.289 s
8×8	181.889 s	> 1 h
9×9	799.710 s	> 1 h
10×10	> 1 h	> 1 h

Table 5.3: Performance of `Pueblo` on Arukone grids.

The results for the SMT solver `Yices` (Table 5.4) are better than for the two pseudo-boolean solvers. This is a bit surprising as for Sudoku and Akari, `Yices` always performed worst. Especially on the logarithmic connectivity encoding the difference to `Minisat+` and `Pueblo` is outstanding. As it is not necessary in SMT to use additional variables for the logarithmic encoding SMT allows a more direct comparisons of the two different approaches to express connectivity. Interestingly, `Yices` is faster on the Floyd-Warshall encoding which uses notably more variables than the logarithmic encoding. The reason might be the fact

	Floyd-Warshall	Logarithmic
Grid size	CPU time	CPU time
5×5	0.597 s	0.916 s
6×6	1.860 s	4.509 s
8×8	11.732 s	100.202 s
9×9	42.745 s	1920.427 s
10×10	477.671 s	>1 h

Table 5.4: Performance of `Yices` on Arukone grids.

that the formulas generated by the Floyd-Warshall algorithm are considerable smaller.

5.6 NP-Completeness

In this section we illustrate how Arukone is proven to be NP-complete by defining a reduction from the NP-complete problem planar 3SAT which was shortly introduced in Section 4.6.

Definition 5.5. A propositional formula ϕ is in *3CNF* if it is in CNF and if all clauses in ϕ have exactly three literals. *3SAT* is the problem of deciding if a given formula which is in 3CNF is satisfiable. *Planar 3SAT* is a restricted form of 3SAT where only planar instances of 3SAT, i.e, 3CNF formulas which have a planar occurrence graph $G(\phi)$, are considered. For a formula ϕ in CNF let X be the set of all variables in ϕ and let C be the set of all clauses. The *occurrence graph* $G(\phi)$ is defined as follows:

$$G(\phi) = (X \cup C, E) \text{ with } E = \{(x_i, c_j) \mid x_i \text{ or } \neg x_i \text{ is in } c_j\}$$

In order to reduce a formula ϕ which is in 3CNF and has a planar occurrence graph to an Arukone grid we create an intermediate planar boolean circuit from the planar occurrence graph. Roughly, this circuit is satisfied by every assignment which satisfies ϕ and is not satisfied by an assignment which does not satisfy ϕ. In a next step the circuit is reduced to Arukone by applying the same principle as in the NP-completeness proof of Akari. For each component of the circuit – gates and wires – a corresponding tile which is a small part of an Arukone grid is defined.

In comparison to Akari where a general boolean circuit is reduced to an Akari grid, the usage of planar circuits allows some simplifications. The most important one is that it is not necessary to define a tile for *XOR*-gates as by the planarity of the circuit there are no wire-crossings. Another advantage is that there is no need to define tiles which correspond to inputs of sort T and F as ϕ contains only variables. The disadvantage of this approach is that the circuits which are derived from planar 3CNF formulas usually have not only one but multiple outputs. Therefore Definition 4.7 has to be changed to allow more than just one output node (nodes with outdegree 0).

To construct a circuit from a planar 3CNF formula ϕ the occurrence graph $G(\phi)$ is used. For each node x_i in $G(\phi)$ which corresponds to a variable of ϕ in the circuit an input node with sort x_i is defined. For each node c_j which corresponds to a clause with three literals we introduce two $\vee$-gates. If the outdegree of a node x_i is larger than 1 several *split*-gates are used to fix the outdegree to 1 as it is required by Definition 4.7. Also for the reduction it is essential to have a fixed outdegree for each gate. The edges (x_i, c_j) of $G(\phi)$ serve as wires. If x_i appears negatively in c_j an additional $\neg$-gate is added before the edge starting at input node x_i enters the $\vee$-gate corresponding to c_j. The circuit which is constructed by this transformation is planar.

In all figures in this section which show tiles or complete Arukone grids, characters are used instead of natural numbers to indicate labelled cells. This is done to improve readability and of course has no influence on the solution of a puzzle. Further in each tile certain cells are marked with black spots. These spots indicate inputs and outputs of the tile. In fact two spots together form an input or an output for one signal. Spots on the left-hand side correspond to inputs, black spots on the right-hand side point at cells which forward the signal to the next tile. Moreover in the figures arrows are used to point out the input and the output side of a tile. Further the cells marked with black spots are the only cells through which a line from a previous tile is allowed to enter respectively a line is allowed to leave in order to enter a successor tile. These lines which pass several tiles are used to forward truth values from one tile to the next. A line which enters a tile through the upper black spot of an input corresponds to an input signal of 1, if the line enters at the lower spot of the input this means that the value of the input is 0. The definition of values which are transported by lines leaving the tile is the same: leaving through the upper black spot of the output means 1 and leaving the tile through the lower black spot of the output means 0. To guarantee that lines are not allowed to enter or leave through cells which are different from the ones marked with spots, all other cells have to be blocked by "walls" which are made out of labels. The concrete implementation of walls is illustrated later on in this section as the explanation requires the knowledge of the different tiles.

For all tiles and grids which are presented we have to distinguish different kinds of labels. A label of the form a_i appears twice per tile and hence the line which connects these two labels chooses a path within the tile. These labels have an auxiliary function and are used to direct the incoming and outgoing lines appropriately. Then there are labels of shape $\bar{v}$ or $\bar{v_i}$ – these label cells to which input lines are connected to. Accordingly there is one label v per tile or in case of the *split*-tile which is explained later there are two labels v_1 and v_2 where the outgoing line(s) start(s). In the concrete reduction we have to take care that the label v of one tile and the label $\bar{v}$ of its successive tile (straight-wire-tiles are not considered as they only have auxiliary labels) are the same number or character. This way it is possible to draw lines which cross the border of a tile and therefore carry information from one tile to the next. Finally there are labels x_i which appear only outside of tiles and are used to form walls.

Straight-wire-tile

To transfer a value from one tile to the next so-called wire-tiles are used. As the name already suggests they are the equivalent of wires in a circuit. The shortest straight-wire-tile is shown in Figure 5.4. Imagine a line entering through the upper black spot of the tile as shown in picture (b) which corresponds to an input value of 1. Clearly the two labels a_1 make it impossible that the line leaves the tile through the lower left spot on the right-hand side and hence the input value 1 is preserved by the wire-tile. The same holds if the input line enters through the lower black spot – there is no way this line reaches the upper spot of the output. This case is illustrated in picture (c) of Figure 5.4.

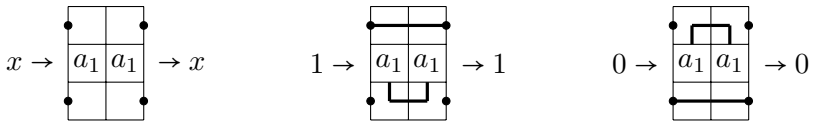


Figure 5.4: The shortest straight-wire-tile.

The length of straight-wire-tiles can be increased by adding an even number of white cells in-between the two cells which are labelled with a_1. A straight-wire-tile which is stretched by two cells is given in picture (a) of Figure 5.5. Picture (b) illustrates that also for stretched wire-tiles it is not possible that the propagated value changes. In this picture we try to switch from input 1 to output 0. But the corresponding line separates the two a_1's from each other and hence there is no solution for this grid. Picture (c) shows a straight-wire-tile which is stretched by one cell – an odd number of cells. Also for this grid there exists no solution as there is no possibility to draw a line through all three cells in the third row. Note that there are several ways to connect the two a_1's but none of them yields a solution. This problem appears for any straight-wire-tile which is extended with an odd number of cells.

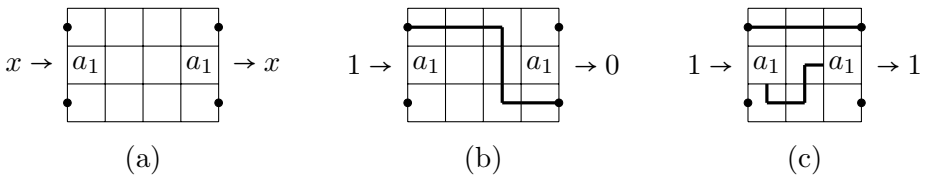
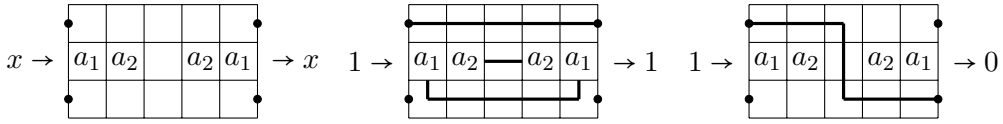


Figure 5.5: Larger straight-wire-tiles.

If a straight-wire-tile with odd length is necessary the tile given in Figure 5.6 can be used. This tile may be enlarged by inserting any number of white cells in the middle of the tile. Picture (b) shows a solution of this tile if the input is 1. The case for input 0 is similar. Picture (c) illustrates that also for straight-wire-tiles of odd length there is no solution which allows to have input and

output with different values. As before, if input and output values are different the auxiliary labels in the tile can not be connected any more.

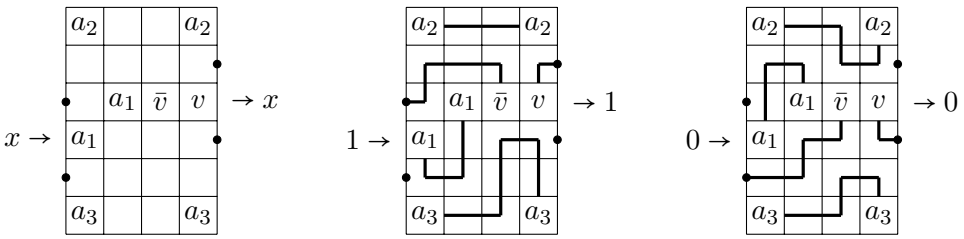


(a) (b) (c)

Figure 5.6: Straight-wire-tile with odd length.

Bend-wire-tile

A tile which allows small corrections of the path of a wire is the bend-wire-tile. It forwards the signal and at the same time defers it by one cell. The bend-wire-tile which is shown in picture (a) of Figure 5.7 moves the signal to the next higher row in the grid. If this tile is mirrored horizontally the resulting tile is a bend which defers the input line to the next lower row. Picture (b) shows the solution of the tile if the input is 1. The entering line has to connect to $\bar{v}$ from above and therefore the only starting point for a line in cell $(2, 4)$ is v. Hence the output line has value 1. Picture (c) shows a solution for the bend-wire-tile if the input is 0. Here the way the input line connects to $\bar{v}$ forces the output line starting at v to be 0. That it is not possible to have an output which is different from the input follows immediately as the tile has two completely separated areas.



(a) (b) (c)

Figure 5.7: The bend-wire-tile allows small corrections.

Corner-wire-tile

The last tile which is necessary to form wires is the corner-wire-tile. The tile shown in Figure 5.8 allows to bend the signal for example from the left border to the top border. Of course it can be flipped to achieve additional corners. But no matter how the tile is flipped it is not possible to get a corner-wire-tile which bends a line from the left side to the lower side of the tile. To get such a

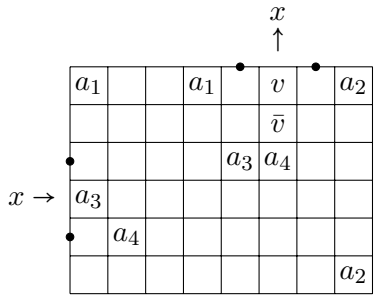


Figure 5.8: The corner-wire-tile.

tile the corner-wire-tile has to be mirrored vertically in the middle. Similar to the other wire-tiles this tile is divided into two areas and the input line is not able to enter one area from the other which guarantees that the input value is not changed but forwarded. This separation is achieved by the cells labelled with a_3 and a_4.

Figure 5.9(a) shows a solution of the corner-wire-tile where the input is 1. This solution is not unique but as the input line entering at the upper black spot of the input must not separate the two a_3's and the two a_4's, this line connects to $\bar{v}$ from the left side and hence the outgoing line has to leave via cell $(1, 5)$. The very same idea is used if the input is 0. This case is illustrated in Figure 5.9(b). Also here the shown solution is not unique.

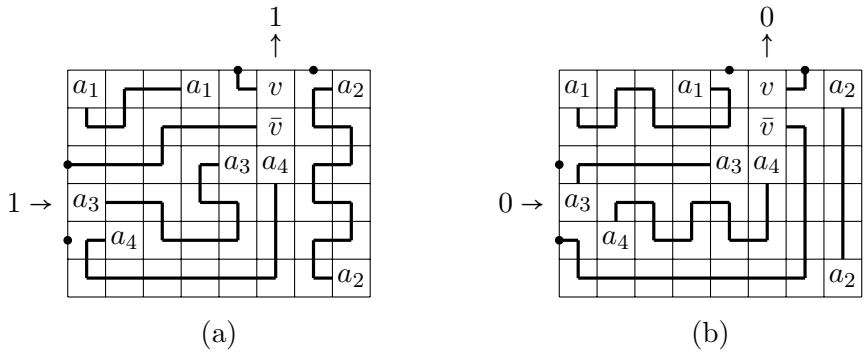


Figure 5.9: Solutions of the corner-wire-tile.

Variable-tile

The tile which is shown in picture (a) of Figure 5.10 is the reduction of an input node which has a variable as sort. As already mentioned there is no need to define tiles for inputs of sort F and T as they do not appear in a 3CNF formula. The variable tile has two output spots which form together one output and as it is a tile which constitutes an input node there are no input spots. The line starting at v is allowed to leave the variable-tile through either of the two black

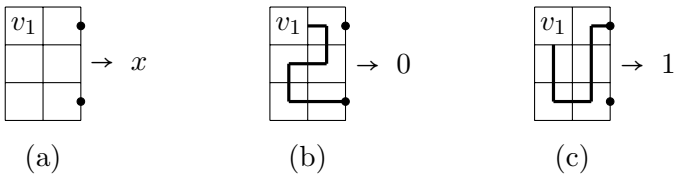


Figure 5.10: The variable-tile.

spots. This corresponds to the fact that the value of a variable can be either 0 or 1 and is not fixed at this point. Pictures (b) and (c) illustrate how all white cells of the tile are passed while the line leaves via the lower or the upper black spot.

T-output-tile

Figure 5.11 shows the T-output-tile which is used to finalise every reduction of an output node of the circuit. Assuming that this tile is connected to a wire and surrounded by walls, the solution given in Figure 5.11(b) is unique. The two auxiliary labels a_1 in the T-output-tile guarantee that the input line which has to be connected to $\bar{v}$ is only allowed to enter the tile through the upper black spot. This corresponds to an input value of 1 and hence the value of each tile which is directly before the T-output-tile has to be 1 in any solution of the grid. In terms of the original formula ϕ this means that there exists a variable assignment for ϕ which satisfies every clause and hence all outputs of the circuit which is constructed from $G(\phi)$ have value 1 under this assignment.

Figure 5.11: The T-output-tile.

¬-tile

The next step in the encoding is to define the reduction of a ¬-gate. It is mapped to the ¬-tile which is shown in picture (a) of Figure 5.12. If the input line which is coming from the previous tile and which has to be connected to $\bar{v}$ enters the ¬-tile via the upper black dot on the left side – this corresponds to input signal 1 – then the output line starting at v is forced to leave the ¬-tile through the lower black spot on the right-hand side as shown in Figure 5.12(b). This is equivalent to having an output signal of 0. If the line ending at $\bar{v}$ enters through the lower black spot the line which is leaving the tile has to take a path via the right, upper black dot and again the signal is flipped. A complete

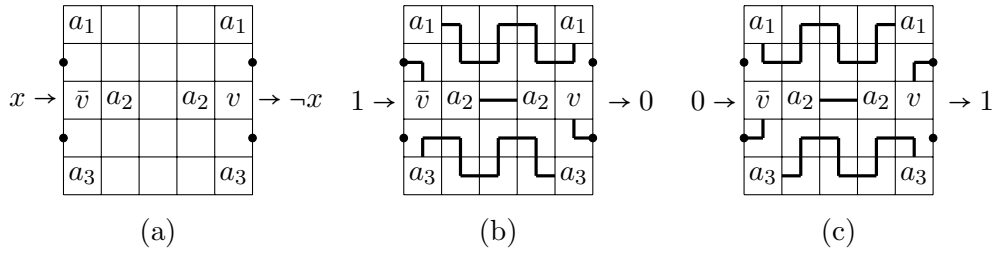


Figure 5.12: The ¬-tile.

solution for input 0 is given in Figure 5.12(c). Clearly these properties are only true if lines are not allowed to leave the tile through any cells except the ones which are marked with black spots. This is achieved by placing a wire tile next to input and output and by blocking all other cells with walls. To guarantee that the ¬-tile really works as desired it is not only necessary to verify that it is possible to find a solution for every input signal where input and output correspond to different values but also to show that there exists no solution where input line and output line of a ¬-tile enter and leave with the same value. That it is not possible to have input and output of the same value can be seen rather quickly by trying to find a solution which meets this constraint.

∨-tile

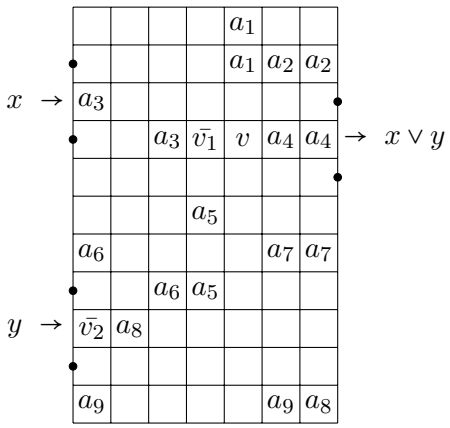


Figure 5.13: The ∨-tile.

The representation of a ∨-gate as Arukone grid requires the more complicated tile given in Figure 5.13. According to the two input wires of a ∨-gate we have four black spots at the left-hand side which allow two lines from previous tiles to enter the ∨-tile. The line entering at the upper input which is labelled with x has to be connected to $\bar{v_1}$, the input line y is connected to $\bar{v_2}$. To simulate the output of a ∨-gate, the output line starting at v is forced to leave the ∨-tile

via the lower black spot at the right-hand side precisely when both input lines are entering via their lower input spots.

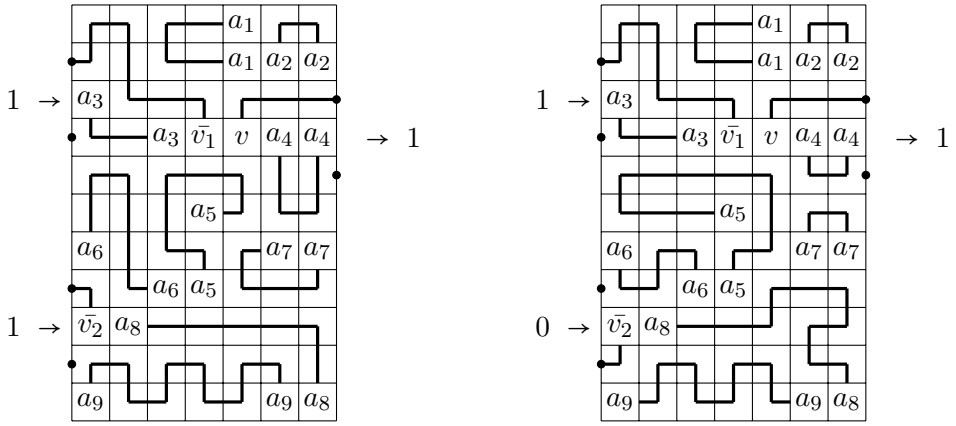


Figure 5.14: Solutions of the ∨-tile where the upper input is 1.

If the upper input x is 1 then the corresponding input line enters via the upper black spot of x. By the two cells labelled with a_3 this line is not able to pass the bottleneck at cell $(4, 2)$ and therefore connects to $\bar{v_1}$ from above. Then the three remaining cells in row three can only be filled out if the output of the tile is 1. This way the output is determined independent of the second input and it remains to show that there exists a solution which completes the lower part of the ∨-tile for both input values of y. Two possible but no unique solutions are given in Figure 5.14.

Figure 5.15 shows how the output value of the ∨-tile is determined if the upper input is 0 and the lower input value is 1. Picture (a) gives a correct solution with output value 1. Picture (b) illustrates why it is not possible to have an output value of 0. The first important observation in this case is that if input x is 0 and the output is 1 then the line connecting the two a_3's must not reach a cell beyond row 3 otherwise the upper part could not be solved. Hence the cells which belong to the upper and the lower part are completely defined. Moreover the line which is connected to $\bar{v_2}$ is uniquely determined and the start and the endpoint of the output line are fixed. If we now try to find a placement for the remaining lines it turns out that there is no solution within these bounds. In Figure 5.15(b) for example it is not possible to have a line connecting the two cells labelled with a_7 such that all white cells are crossed by a line.

Figure 5.16(a) shows a valid solution of the ∨-tile if both input signals are 0. In picture (b) of Figure 5.16 we try to find a solution of the ∨-tile where both inputs are 0 and the output is 1. Again the cells of the upper and the lower part of the tile are delimited by the line which connects the two a_3's. This line must not reach any other cell in row 4 or 5 as then the upper input line could not be connected any more to $\bar{v_1}$. Further the lines in the lower part of the

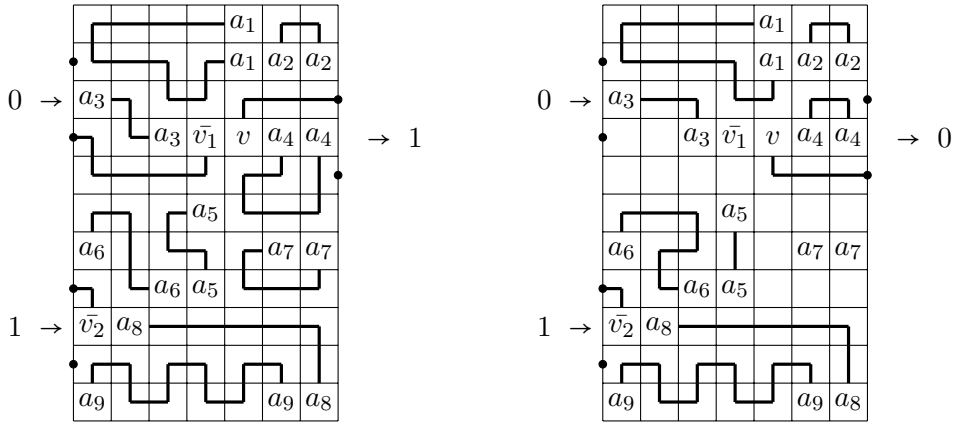


Figure 5.15: ∨-tile where input x is 0 and input y is 1.

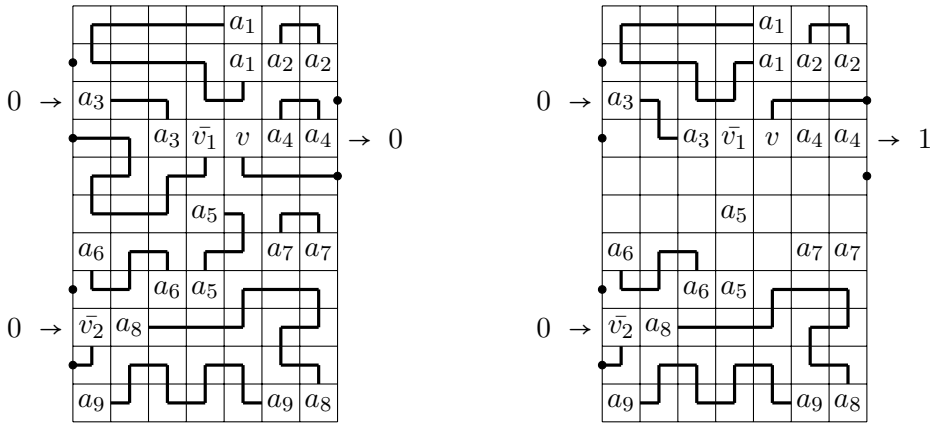


Figure 5.16: A ∨-tile where both inputs are 0.

tile in picture (b) can not be changed in such a way that the white cells in the middle change. But now it is not possible any more to connect the upper input line starting at the lower black spot of the upper input to $\bar{v}_1$ without having at least one white cell which is not touched by a line.

split-tile

The idea of the *split*-tile which is shown in Figure 5.17 and to which a *split*-gate is reduced is simple. If the input line enters via the upper input spot which corresponds to signal 1 then the two cells labelled with a_2 force the input line to be connected to $\bar{v}$ via the upper adjacent cell of $\bar{v}$. This in turn forces the first output starting at v_1 to leave the tile through its upper exit point. Then the

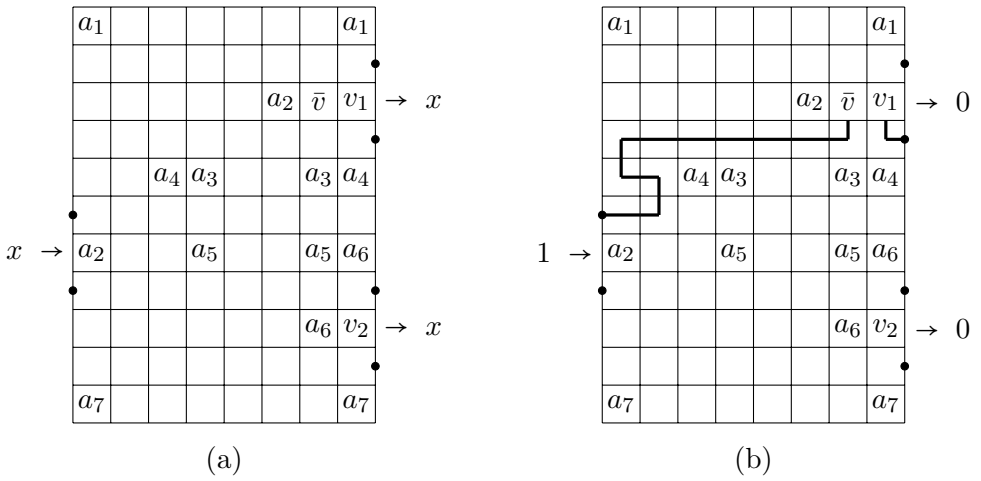


Figure 5.17: The *split*-tile.

signal is propagated to the second output line starting at v_2 by the cells on the right-hand side of the grid labelled with a_4 and a_6. In picture (b) of Figure 5.17 we have an input signal of 1 and at the same time try to have an output signal of 0. Hence the line starting at v_1 leaves through the lower black spot of the first output. Then the input line has to connect to $\bar{v}$ from below, otherwise there could not be a line in cell $(2, 8)$. But now the two a_2's are separated and we can not find a solution any more. Also if we try to have output 1 with an input signal of 0 it follows immediately that the input line connected to $\bar{v}$ forbids to connect the two cells labelled with a_2.

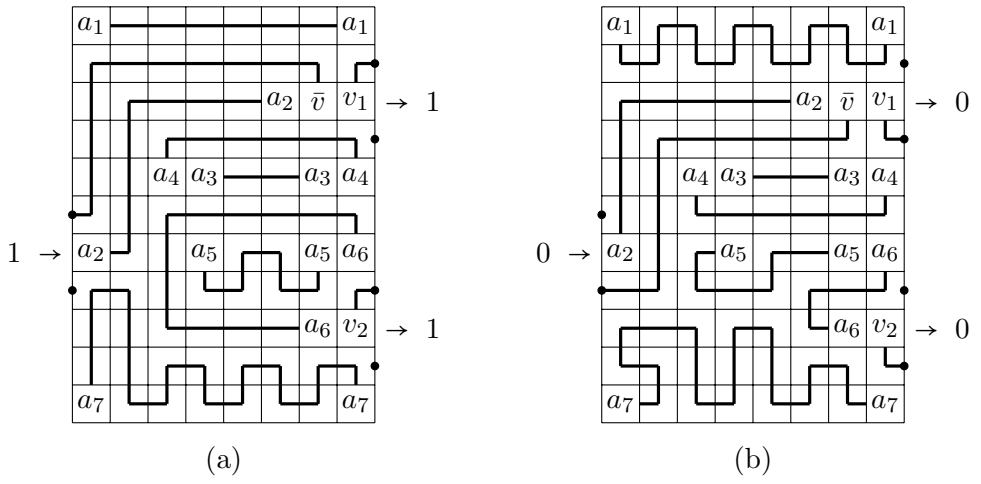


Figure 5.18: Valid solutions for the *split*-tile.

Figure 5.18(a) shows a valid solution of the *split*-tile for an input value of 1. As already explained the entering line has to connect to $\bar{v}$ from above and

therefore the first output has to be 1. Now the line in cell $(4, 8)$ has to come from cell $(5, 8)$ which is labelled with a_4. As a consequence the line in cell $(6, 8)$ has to start in cell $(7, 8)$ which is labelled with a_6 which in turn forces the second output to be 1 as well. The working principle is the same if the input has value 0 as shown in Figure 5.18(b). Also in this case the output value is determined for the upper output by the way the input line has to connect to $\bar{v}$ and then forwarded through the labels on the right-hand side to the second output.

Walls

Figure 5.19 shows the actual construction of walls. Therefore some of the cells which are not part of a tile are filled with labels in order to get a rectangular

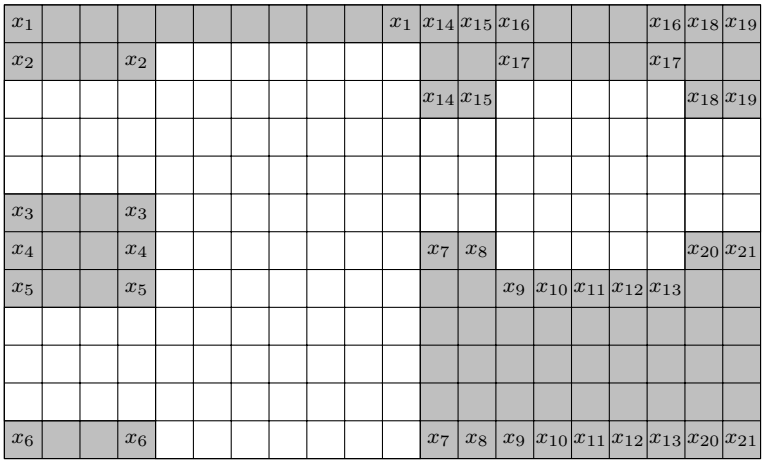


Figure 5.19: Cells which are not part of a tile are covered with walls.

Arukone grid. In this figure all gray cells denote cells which are outside of tiles – these are the cells where walls are placed – and all white cells are part of tiles. Of course the construction of walls has to be done in such a way that the working principle of all tiles is not changed. Especially we have to take care that no formerly impossible combination of input and output lines becomes possible. An important observation is that it is not possible that lines which

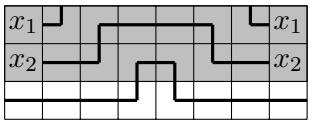
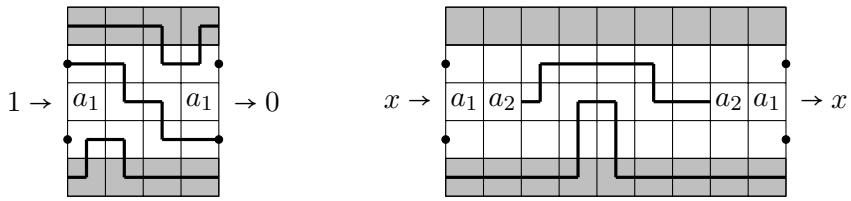


Figure 5.20: Lines from inside a tile must not touch cells of the wall.

start inside the tile are ever able to reach a cell which is outside of that or any other tile e.g. successor wire-tiles. Figure 5.20 illustrates this property of the reduction. It shows a fragment of an Arukone grid where a line starting

inside the white cells of a tile occupies also cells of the wall. As a consequence the line connecting the two x_2's has to take a path around these cells which again makes it impossible to connect the two x_1's. Obviously the cells which are passed by the line from inside the tile are missing. Due to the construction of walls it is always the case that not all labels of the wall can be connected if lines from outside the wall are interfering. Hence we examine only cases where a line starting somewhere within the walls has a path which passes also cells inside of one or more tiles.



(a) (b)

Figure 5.21: Lines of the wall interfering with wire-tiles.

Consider for example the straight-wire-tile in Figure 5.21(a). It is possible that the path of a line which belongs to the wall covers cells of the wire but the line which enters the wire-tile as input is never able to switch from the upper input to the lower output or vice versa due to the fact that then the two cells labelled with a_1 could not be connected any more. Figure 5.21(b) shows that for straight-wire-tiles of odd length there is no solution at all if lines from the wall interfere. Such a tile always has a unique solution. For the other two wire-tiles – bends and corners – it is also clear that lines starting at the wall are not able to change the output value as both tiles are separated in two areas and therefore it is completely determined through which neighbour cell of $\bar{v}$ the input line connects to $\bar{v}$. Then also the path of the outgoing line is exactly defined.

Also for most other tiles it is clear that lines from outside are not able to induce severe changes inside the tile. For the variable-tile we have that the output is independent of any input and that the outgoing line may leave via the 0 or the 1 exit of the tile. Therefore lines starting inside the wall can not change the correlation between input and output signal. For the T-output-tile the case is similar. As input 0 is blocked with a label, a line entering this tile from the previous tile has no choice but to enter via the upper spot of the input.

Figure 5.22 shows a ¬-tile with two straight-wire-tiles – one for the input, one for the output line. As this grid is only a part of a complete Arukone grid the labels of the walls are not complete. In the concrete reduction of a formula the ¬-tile (and every other tile which is not a straight-wire-tile) always appears next to two straight-wire-tiles. In Figure 5.22 we try to find a solution if input and output of the ¬-tile are 1. The lines which are already drawn inside of the tiles are completely determined. Without lines from the outside there is no solution for this combination of input and output values. The picture shows

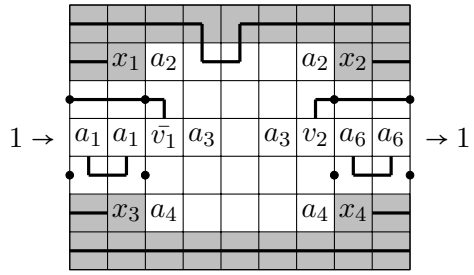


Figure 5.22: Walls do not change the outcome of the ¬-tile.

that if a line from outside enters the ¬-tile from above then the two a_2's are separated. On the other hand, if a line from the wall enters at the lower border of the ¬-tile then it can not have any influence on the lines in the upper half of the ¬-tile as cell $(4, 5)$ can only be passed by a line which belongs to the two a_3's. Also if input and output are both 0 there exist no solution due to the same argumentation as the ¬-tile is symmetric. For these reasons it is not possible that the outcome of the ¬-tile is influenced by a line of the wall.

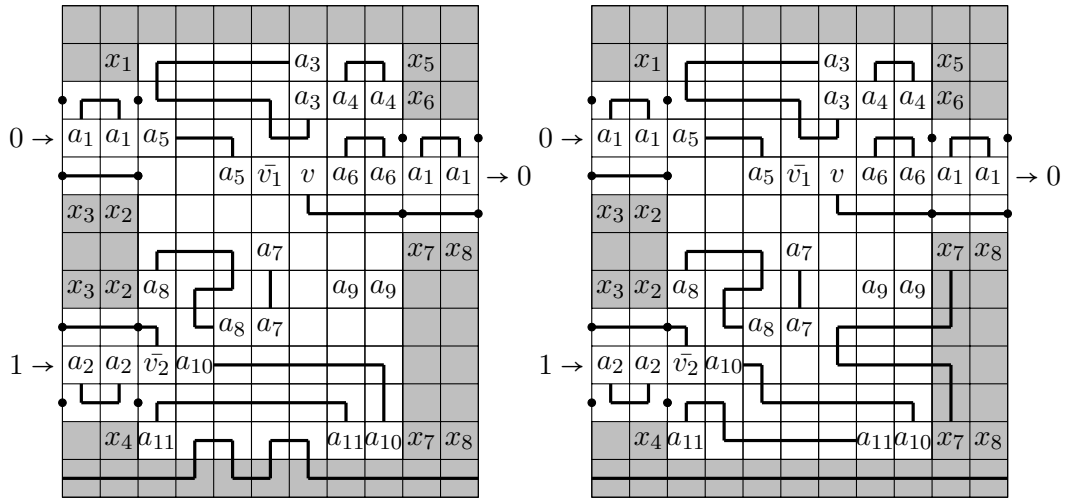


Figure 5.23: Walls and the ∨-tile.

For the ∨-tile it is not immediately obvious that the walls do not disturb the lines inside the tile. If the upper input line enters at the upper black spot then the way this line connects to $\bar{v_1}$ forces the line starting at v to leave via the upper spot and this consequence can not be influenced by any other line in- or outside the tile. If the upper input is 0 then the two different cases for the lower input have to be distinguished. In Figure 5.23 we try to find a solution of the ∨-tile where input x is 0, input y is 1 and the output is 0. This should of course not be possible. A possible entry point of a line from outside the tile

is the lower border of the ∨-tile. If we compare the area which is covered by the line for a_9 in Figure 5.15(a) and the area for a_{11} (the corresponding label in this figure) together with the line starting at the wall in Figure 5.23(a) then it turns out that it is the same and in addition that there is no way to enlarge this area. Hence there are no changes possible which result from lines entering at this lower border of the ∨-tile. In picture (b) of Figure 5.23 a line from a wall enters at the right side of the tile. Also in this case the new line does not help to construct a solution as cell $(10, 6)$ has no line. Also if the line entering at this side of the tile covers only two or four cells instead of six as it does in picture (b) there are always cells which can not be covered. This way we have detected that there is no solution for the subarea containing the two cells labelled with a_9 also with the help of additional lines from the outside and that an output value of 0 still is not possible for inputs 1 and 0.

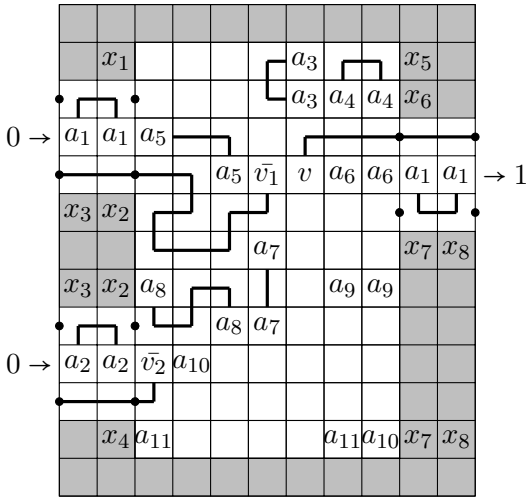
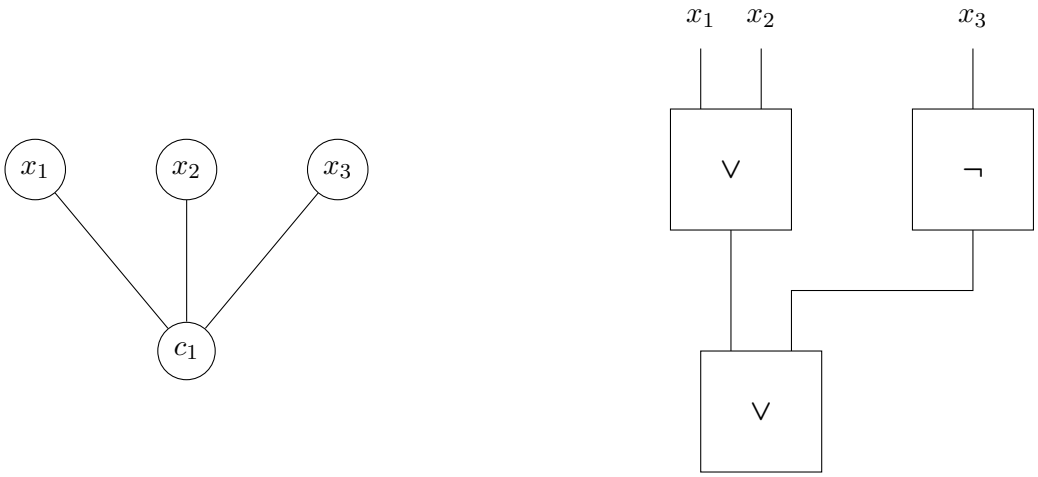


Figure 5.24: Walls and the ∨-tile.

It remains to check if the walls allow to have output 1 if both input signals of the ∨-tile are 0. Figure 5.16 already illustrated that there exists no solution for this combination of input and output values without walls. It is for example not possible to cover all cells in the white area between the cells labelled with a_3, a_5 and a_6. In order to have a solution for this part of the grid as shown in Figure 5.24 the line connecting the two a_5's (a_3's in Figure 5.16) has to change as given in this grid. But then cell $(4, 6)$ has no line and it is also not possible to change the placement of lines with the help of lines from the outside in order have a line in this cell.

The last tile we have to cover is the *split*-tile. As illustrated in Figure 5.17(b) the two labels a_2 divide the tile in two parts and this determines uniquely from which side the input line connects to $\bar{v}$ which in turn determines the output value. This principle can not be disturbed by any cells from the outside as they are not able to reach any cells at the right border which propagate the truth value.



(a) (b)

Figure 5.25: Occurrence graph and circuit for $(x_1 \vee x_2 \vee \neg x_3)$.

Example 5.6. The formula which is reduced to Arukone is kept rather small as the main purpose of this example is to illustrate the different steps of the reduction. Moreover, already for such a small formula the resulting grid is quite large. The formula in 3CNF which is transformed to an Arukone grid is the following:

$$\phi = (x_1 \vee x_2 \vee \neg x_3)$$

The first step is to create the occurrence graph $G(\phi)$ for ϕ. The resulting graph is shown in picture (a) of Figure 5.25 and as easily can be seen this graph is planar. Then a planar circuit is created out of $G(\phi)$. The transformation is straightforward, the only thing we have to take care of is that negations of literals do not appear in $G(\phi)$ but of course have to be considered in the circuit. Picture (b) of Figure 5.25 shows the circuit which is generated by the transformation. For node c_1 in $G(\phi)$ two $\vee$-gates are introduced, for each node x_i we add an input to the circuit and the value of x_3 is negated before it enters the second $\vee$-gate.

Then the transformation of the circuit into an Arukone grid is accomplished. For each component of the circuit a corresponding tile is placed in a similar position on a big Arukone grid. Figure 5.26 shows the large Arukone grid which is created this way. As in Figure 4.16 in Section 4.6 blue boxes are used to indicate the border of the single tiles. This grid is not unique, e.g. the corner-wire-tiles could also be replaced by bend-wire-tiles. Every cell which is not part of a tile is highlighted in gray. In this figure the actual implementation of walls is not given as it is not exactly determined and the multiplicity of labels may confuse the reader. To ensure that all white cells of the wall can be covered with lines we have to guarantee that there is no self-contained gray area which

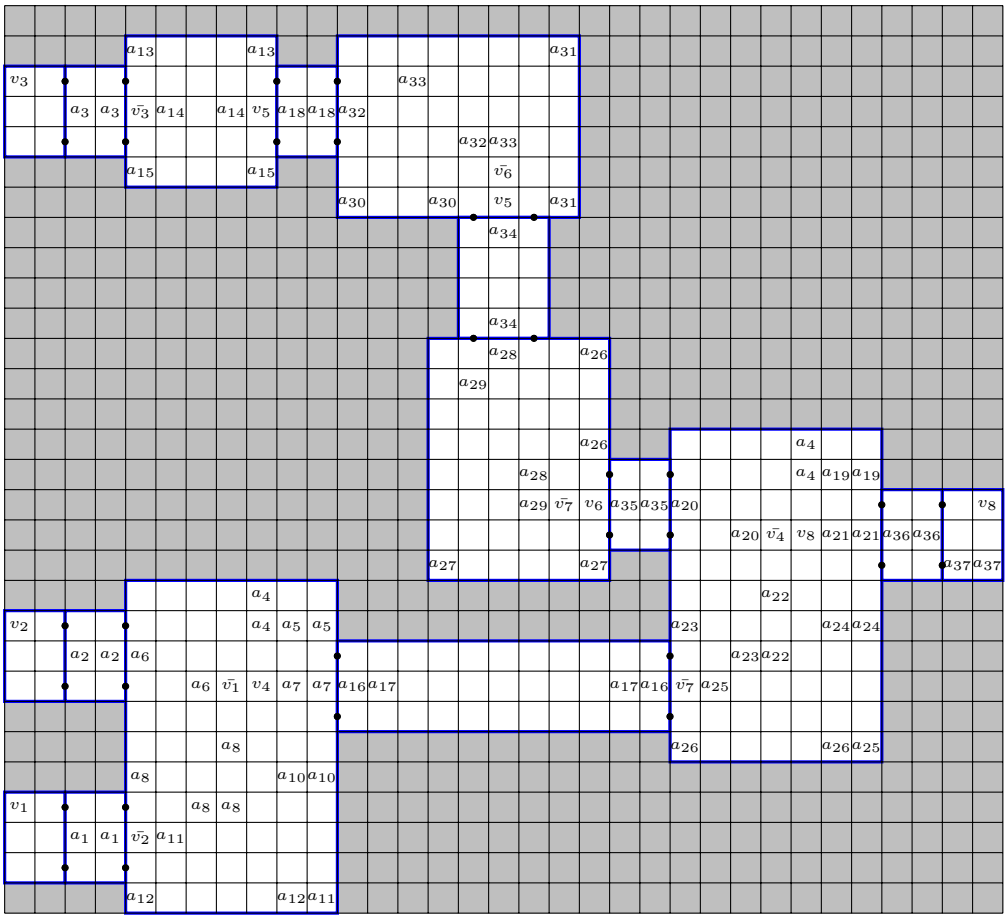


Figure 5.26: Arukone grid which is the reduction of $(x_1 \vee x_2 \vee \neg x_3)$.

has less than four cells. This is achieved easily by using longer wires if necessary or by increasing the cells which are reserved for walls. In this example the first row of the grid is added for this reason. Further there is always a straight-wire-tile between every two tiles which are not straight-wire-tiles to guarantee that input and output line really take a correct path.

Having defined a reduction A from planar boolean circuits to Arukone next we have to show the following statement:

Theorem 5.7. *Let ϕ be a 3CNF formula and $A(\phi)$ the previously defined reduction to an Arukone grid. Then ϕ is satisfiable if and only if $A(\phi)$ has a solution.*

The proof is not given here as it is very tedious. To obtain the NP-completeness of Arukone, the remaining task is to show that Arukone lies in NP by giving a polynomial-time algorithm to verify a solution A' for an Arukone grid A:

1. For each cell (i, j) with $A(i, j) \in \mathbb{N}$ verify that $A(i, j) = A'(i, j)$.
2. For each white cell check if there is exactly one line in that cell.

3. Follow each line starting at a number and verify that the numbers at both ends are matching. If a visited cell is not stored already, memorise it.

4. Count the number of stored cells. If it is equal to the cells in the grid then the solution is correct.

6 Implementation

With this thesis comes a program which incorporates all presented encodings. The program is implement in OCaml. To obtain an executable in byte code use `make`, for native compilation the command `make native` may be used. The user has to given an input file which defines a puzzle and to specify an encoding. Then the program generates a formula and chooses a solver to compute satisfiability of that formula. If there is a satisfying assignment the program displays the solution of the initial puzzles. The solvers which are integrated in the program are the same as used in the experiments: `Minisat2`, `Minisat+`, `Pueblo` and `Yices`. For PB encodings `Minisat+` is taken by default. To use `Pueblo` add the option `-solver pueblo`. The option `-file-only` circumvents the computation of a solution but simply generates a file containing the generated formula. The output file is named after the input file but the ending is changed to .cnf, .opb or .ys according to the kind of formula it determines and is stored in the directory of the input file.

A complete list of all options which are allowed can be obtained with the command `./PuzzleTool -help`. Each option which is used to specify an encoding has to be followed by a file and consists of three parts. The first letter stands for the kind of puzzle which is given in the input file: `S` for Sudoku, `A` for Akari and `N` for Arukone (or Number Link). The second part states the logical language, i.e., `SAT`, `PB` or `SMT`, and the last part is either empty or is a natural number which determines the refinement of the encoding as presented in the thesis. For instance the option `SSAT1` tells the solver that the input file contains a Sudoku and that it has to use the first minimal SAT encoding as given in Chapter 3. A call to the program might look as follows:

```
./PuzzleTool -SPB examples/sudoku/3x3_01.sin -solver pueblo
```

There are two different input formats in which puzzles are specified. An example input file for Sudoku and Arukone for which the same format is used is given in Figure 6.1. The grid is rebuild as a sequence of characters where

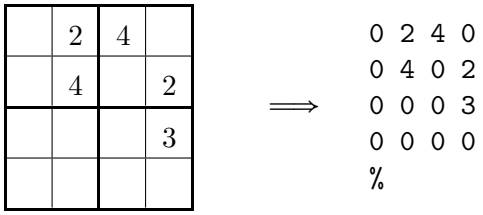
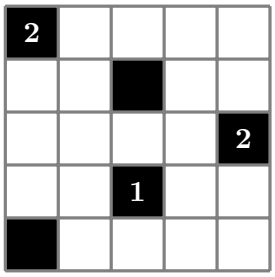


Figure 6.1: Input format for Sudoku and Arukone.

0 stands for a white cell, a numbered cell is represented by the corresponding number, newlines indicate the start of a new row and the end of the puzzle is marked with the symbol %.




```
5x5
1,1,2
2,3,b
3,5,2
4,3,1
5,1,b
```

Figure 6.2: Input format for Akari.

For Akari a less verbose format is used. This is due to the fact that for Akari very large puzzles where tested in the experiments. Figure 6.2 shows how an Akari grid is represented in the input file. The first row gives the number of rows and columns of the puzzle. Then for each non-white cell an entry consisting of the row, the column and a number or, in case of entirely black cells, the symbol b is added.

To communicate with the different solvers the program makes use of files. Depending on the chosen solver encodings are generated in different standardised formats and then written to a file which serves as input for the solver. Especially in case of CNF formulas these input files may require a considerable amount of disk space. `Minisat2` expects a CNF formula in DIMACS CNF format – a detailed description is given in [7]. Both PB solvers expect a version of the OPB format. A short overview can be found on the website of the PB evaluation 2005 [2]. An example input for `Minisat+` looks as follows:

```
* #variable= 5 #constraint= 3
* comments
min:  +1*x1 -1*x3;
-1*x2 +3*x4 -2*x5 >= -2;
-1*x1 +4*x2 -5*x3 >= +3;
-8*x4 +4*x3 >= +6;
```

The format for `Pueblo` is slightly different: constants and variables are not separated by * but by a blank. The SMT solver `Yices` has its own input language which is explained on the `Yices` website [6]. Additionally the authors provide a large number of examples.

Bibliography

[1] Janko.at. www.janko.at.

[2] PB Evaluation 2005. http://www.cril.univ-artois.fr/PB05/solver_req.html.

[3] SMT-LIB - The Satisfiability Modulo Theories Library. http://combination.cs.uiowa.edu/smtlib/.

[4] The MiniSAT Page. www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html.

[5] Web Nikoli. www.nikoli.co.jp/en/.

[6] Yices: An SMT Solver. http://yices.csl.sri.com.

[7] Satisfiability: Suggested Format, 1993. www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

[8] J. C. Beck, K. N. Brown, I. Miguel, and P. Prosser. Enforcing connectivity in a fixed degree graph: A constraint programming case study. 2008.

[9] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 13(5):633–645, 1986.

[10] C.-L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[11] C. J. Colbourn, M. J. Colbourn, and D. R. Stinson. The computational complexity of recognizing critical sets. In *Proceedings of the First Southeast Asian Conference on Graph Theory*, volume 1073, pages 248–253. Lecture Notes in Mathematics, 1984.

[12] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[13] R. W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[14] L. Levin. Universal sorting problems. In *Problems of Information Transmission*, volume 9, pages 265–266, 1973.

[15] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[16] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale*. Springer, 2006.

[17] B. McPhail. Light up is NP-complete. `www.cs.umass.edu/~mcphailb/papers/2005lightup.pdf`, 2005.

[18] C. H. Papadimitriou. *Computational Complexity*. Adison-Wesley, University of California – San Diego, 1995.

[19] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[20] G. Rote. Sudoku, 2006. `http://page.mi.fu-berlin.de/rote/Papers/slides/Sudoku.pdf`.

[21] H. M. Sheini and K. A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:2006, 2006.

[22] H. Simonis. A tale of two puzzles. `http://4c.ucc.ie/~hsimonis/A Tale of Two Puzzles.pdf`, 2007.

[23] S. Tatham. Simon Tatham's portable puzzle collection. `http://www.chiark.greenend.org.uk/~sgtatham/puzzles/`.

[24] C. Terzer. Nurikabe as SAT Problem, 2007. Bachelor Thesis.

[25] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[26] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[27] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IECE Transactions Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1052–1060, 2003.

[28] H. Zankl. BDD and SAT techniques for precedence based orders. Master's thesis, University of Innsbruck, 2006.