Bachelor Thesis

Slitherlink Reloaded

David Westreicher (0716064)
david.westreicher@student.uibk.ac.at

16 November 2011

Supervisor: Dr. René Thiemann

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum	Unterschrift

Abstract

This bachelor thesis is based on the work of Lorenz Thuile's thesis "Slitherlink" [10]. While Lorenz's solutions of the puzzles are generated through patterns and a trial-and-error solver, the aim of this thesis is to encode the rules of Slitherlink into a boolean satisfiability problem (SAT) and evaluate the pros and cons of this approach. Furthermore the GUI of the existing application was improved and the solver was extended to handle degenerated puzzles.

Contents

1. Introduction

Slitherlink is one of the most popular Nikoli logic puzzles and was invented in June 1989. It is also known as ”Fences”, ”Takegaki”, ”Loop the Loop”, ”Loopy”, ”Ouroboros”, ”Suriza” and ”Dotty Dilemma” [5]. In the year 2000 Slitherlink was proved to be NP-complete [12], which means that if NP $\neq$ P holds it is not possible to always solve a puzzle in polynomial time and that any problem in NP can be reduced to an instance of Slitherlink.
In the previous work [10], Slitherlink instances were solved with a combination of finding patterns and a trial-and-error solver. To find these special patterns a huge method (over 7000 lines of code) was constructed. This method is not suited for a formal proof of correctness, because of its length.
The aim of this thesis is to try a different approach to solve the logic puzzle, namely ”boolean SAT solving”.

1.1. SAT Solving

SAT stands for the satisfiability problem of propositional logic. A valid solution of a SAT instance is an assignment of boolean variables which make a boolean formula evaluate to true. To find such assignments, a SAT solver, like SAT4J [2] or MiniSAT [1] is used. To solve a given problem using SAT one has to:

- encode the problem as a boolean formula
- convert the formula into CNF (if the SAT solver accepts only formulas in CNF)
- use a SAT solver to find satisfying assignments of the variables
- and finally interpret these assignments with a decoder to get a solution of the problem.

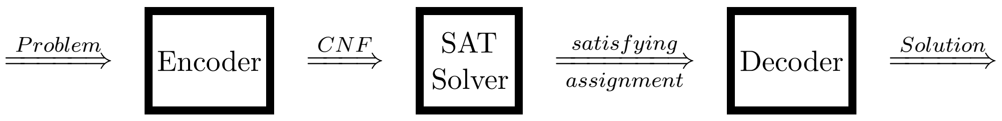


Figure 1.1.: The process of using SAT for problem solving

SAT solving has gained much popularity in recent years, because of faster computers and faster algorithms. Even large instances with millions of clauses and hundred thousands of variables are nowadays solvable in seconds. Some applications of SAT solving include:

- Planning: Given an initial state find actions with preconditions and effects which lead to a given goal state. E.g.: SATPLAN [11], ...
- Dependency Management: Equinox p2 is the OSGi implementation of Eclipse. It is a framework which manages for instance the plugin interface. Equinox, thereby uses SAT-techniques to resolve dependency conflicts of the plugins [6].

1.2. Content

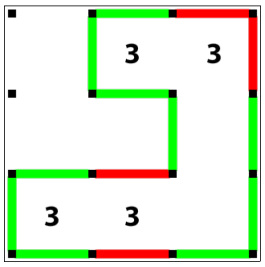
This thesis begins with a short introduction to the game of Slitherlink in Section 2. To solve Slitherlink puzzles with SAT, we have to generate a propositional formula which encodes the rules of the game. In Section 3 we present one encoding of the Rule 1 and two encodings of Rule 2. Rule 3 is observed by two different approaches discussed in Section 4. One further task of the thesis was to integrate the SAT-solving method in the existing application, which is described in Section 5. Finally we compare the runtimes of the existing solver with the ones presented in this work in Section 6.

2. Slitherlink

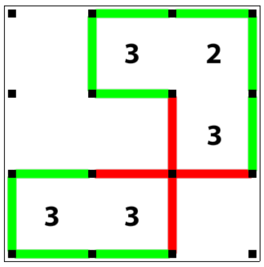
Slitherlink is played on a grid with $m \times n$ cells. Each cell can contain a number between 0 and 3 and is surrounded by 4 lines that can be either set or not set. The goal of the puzzle is to draw a consistent loop (connected lines) between the cells which also complies to the following rules:

1. The value of the cell must be the same as the number of surrounding lines.
2. The loop can't leave the game field and can't cross itself.
3. There has to be exactly one loop.

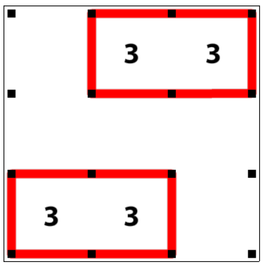
Example 2.1. Figure 2.1 shows some puzzles which violate the rules mentioned above.



(a) Rule 1 is violated



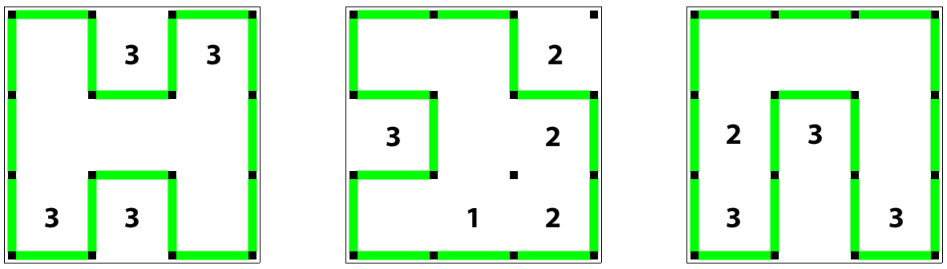
(b) Rule 2 is violated



(c) Rule 3 is violated

Figure 2.1.: Wrong solutions of Slitherlink puzzles.

Example 2.2. Some examples of correctly solved 3×3 puzzles.



3. SAT Encoding for Slitherlink

Now that we know how to play Slitherlink, we will encode Rules 1&2. But first of all we give a short introduction to the basics of propositional logic. We begin by defining what a boolean formula is and what the property satisfiable means.

3.1. Boolean formulae and SAT

In propositional logic a boolean formula is a concatenation of symbols in a special syntax. The symbols are either:

- Constants: $\top$ ("true") and $\bot$ ("false")
- Propositional variables: $x, y, ...$ which represent a truth value.
- Unary and binary operators:
  - Negation: $\neg\Phi$, pronounced "not Φ"
  - Conjunction: $\Phi \wedge \Psi$, pronounced "Φ and Ψ"
  - Disjunction: $\Phi \vee \Psi$, pronounced "Φ or Ψ"
  - Implication: $\Phi \rightarrow \Psi$, pronounced "Φ implies Ψ"
- Parentheses: "(" and ")" which regulate the precedence of the operators.

The following BNF-syntax describes the language of boolean formulae:

$$\Phi ::= \top \mid \bot \mid x \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \rightarrow \Phi)$$

A common procedure in propositional logic is the **evaluation** of a formula. For instance the evaluation of the formula "$(\sqrt{36} = 6)$ and ('The sun is bright')" is true, because each of the arguments of the conjunction evaluates to true.
In the following definitions we introduce the evaluation formally.

Definition 3.1. Let Φ be a boolean formula. The function $P : \Phi \rightarrow \text{Vars}$ returns the set of all variables in Φ.
For instance $P(x \vee (x \wedge \neg y)) = \{x, y\}$

Definition 3.2. An assignment (or model) of a formula Φ is a function $f : P(\Phi) \rightarrow \{\top, \bot\}$, which assigns a truth value to each variable.

Definition 3.3. Let ϕ be a boolean formula and f an assignment of ϕ. Then the evaluation $eval : \Phi \rightarrow \{\top, \bot\}$ of the formula is defined recursively by:

- $eval(p) := f(p)$
- $eval(\neg\chi) := \top$ iff $eval(\chi) = \bot$ and vice versa

- $eval(\chi \wedge \psi) := \top$ iff $eval(\chi)$ and $eval(\psi)$ is $\top$
- $eval(\chi \vee \psi) := \top$ iff $eval(\chi)$ or $eval(\psi)$ is $\top$ (inclusive or)
- $eval(\chi \rightarrow \psi) := \bot$ iff $eval(\chi) = \top$ and $eval(\psi) = \bot$

Definition 3.4. We call a propositional formula Φ satisfiable if there exists an assignment f, which lets Φ evaluate to $\top$.

SAT is the decision problem to check if a given formula is satisfiable or not. Most SAT solvers also return a satisfiable assignment of the formula, if there is any.

Example 3.5. Let $\phi = ((x \vee y \vee z) \wedge (x \vee \neg y) \wedge (\neg x \wedge w))$. Then ϕ is satisfiable by the following assignment:

$$f : x \rightarrow \bot, y \rightarrow \bot, z \rightarrow \top, w \rightarrow \top$$

3.2. Formal Definition of Slitherlink

To encode an instance of Slitherlink into a SAT-problem, we first need to define boolean variables, which represent the integral part of the puzzle.

Definition 3.6. An $m \times n$ instance of Slitherlink consists of the following boolean variables:

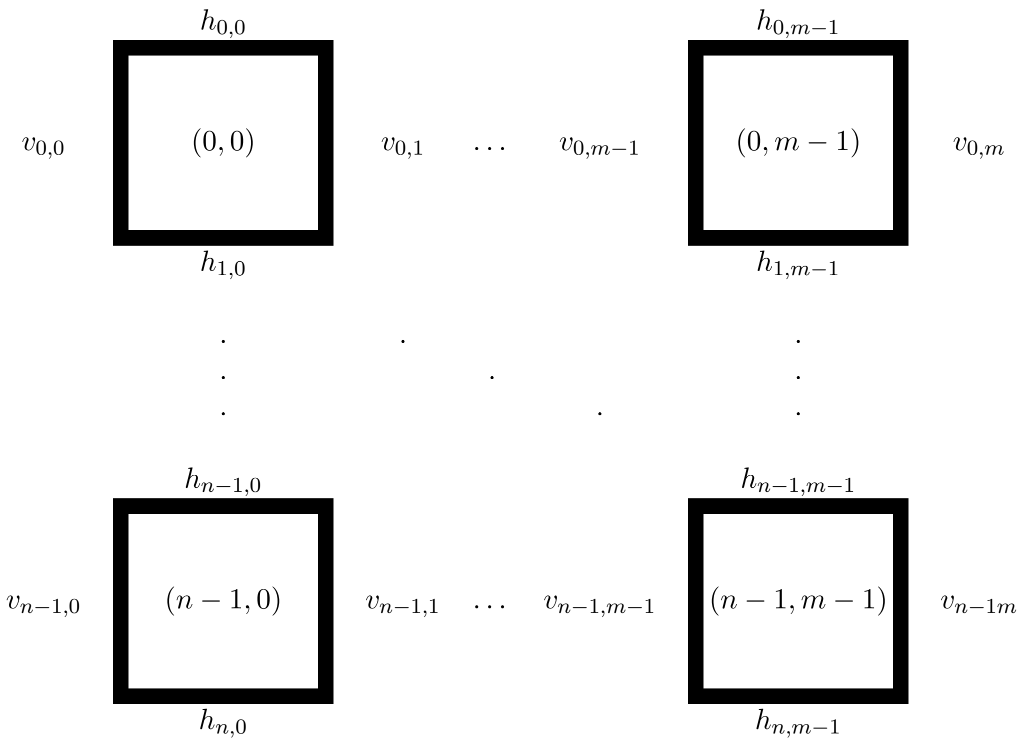


Figure 3.1.: The boolean variables which make up a Slitherlink puzzle

- Vertical lines: $v_{i,l} \mid (i,l) \in \{0,...,n-1\} \times \{0,...,m\}$.

$$v_{i,l} \equiv \text{vertical line } (i,j) \text{ is set} \tag{3.1}$$

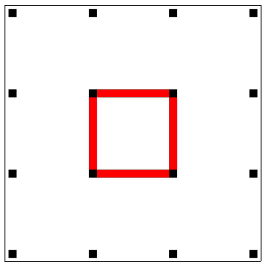
- Horizontal lines: $h_{k,j} \mid (k,j) \in \{0,...,n\} \times \{0,...,m-1\}$.

$$h_{k,j} \equiv \text{horizontal line } (k,j) \text{ is set} \tag{3.2}$$

Definition 3.7. For every cell at $(i,j) \in \{0,...,n-1\} \times \{0,...,m-1\}$ of a puzzle we define $l_{i,j}$ as the set of the surrounding lines:

$$l_{i,j} = \{h_{i,j}, v_{i,j}, h_{i+1,j}, v_{i,j+1}\} \tag{3.3}$$

Example 3.8. All lines of the set $l_{1,1}$ marked in red.



Definition 3.9. For every line $h_{k,j}$ the two sets $preH_{k,j}/succH_{k,j}$ contain the lines to the left/to the right of $h_{k,j}$:

$$preH_{k,j} = \{v_{k,j} \mid k < n\} \cup \{v_{k-1,j} \mid k > 0\} \cup \{h_{k,j-1} \mid j > 0\} \tag{3.4}$$

$$succH_{k,j} = \{v_{k,j+1} \mid j < m\} \cup \{v_{k-1,j+1} \mid k > 0\} \cup \{h_{k,j+1} \mid j < m-1\} \tag{3.5}$$

with $(k,j) \in \{0,...,n\} \times \{0,...,m-1\}$

Definition 3.10. For every line $v_{i,l}$ the two sets $preV_{i,l}/succV_{i,l}$ contain the lines to the top/to the bottom of $v_{i,l}$:

$$preV_{i,l} = \{h_{i,l-1} \mid l > 0\} \cup \{h_{i,l} \mid l < m\} \cup \{v_{i-1,l} \mid i > 0\} \tag{3.6}$$

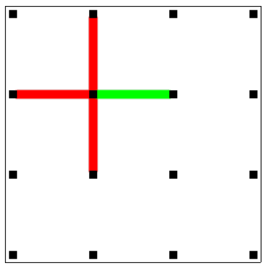
$$succV_{i,l} = \{h_{i+1,l-1} \mid l > 0\} \cup \{h_{i+1,l} \mid l < m\} \cup \{v_{i+1,l} \mid i < n-1\} \tag{3.7}$$

with $(i,l) \in \{0,...,n-1\} \times \{0,...,m\}$

Definition 3.11. For every corner at (k,l) we define $c_{k,l}$ as the set of all lines which touch the corner.

$$\begin{aligned} c_{k,l} = \{v_{k,j} \mid k < n\} \cup \{v_{k-1,j} \mid k > 0\} \cup \\ \{h_{k,j-1} \mid j > 0\} \cup \{h_{k,j}\} \end{aligned} \tag{3.8}$$

Example 3.12. Figure 3.2 and 3.3 show some examples of the *pre*/*succ* and c sets.


(a) $preH_{1,1}$ and its elements

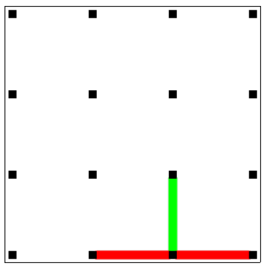
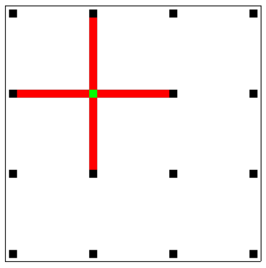

(b) $succV_{2,2}$ and its elements

Figure 3.2.: Examples of *pre* and *succ* sets


(a) $c_{1,1}$ and its elements

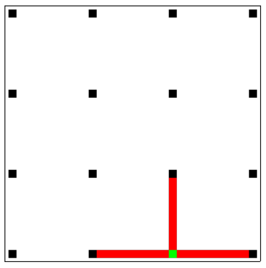

(b) $c_{3,2}$ and its elements

Figure 3.3.: Examples of c sets

3.3. Boolean Helper Functions

To express certain properties of the puzzle we define some helper functions which keep our encodings of manageable length. Let $A = a_0, \ldots, a_{n-1}$ be a set of boolean variables.

Lemma 3.13. *$None(A)$ returns a formula which is satisfiable iff all a_i are $\bot$*

$$None(A) = \bigwedge_{a \in A} \neg a \tag{3.9}$$

Lemma 3.14. *$AtLeastOne(A)$ returns a formula which is satisfiable iff at least one a_i is $\top$*

$$AtleastOne(A) = \bigvee a \in A \tag{3.10}$$

Lemma 3.15. *$AtMostOne(A)$ returns a formula which is satisfiable iff at most one a_i is $\top$*

$$AtMostOne(A) = \bigwedge_{\substack{i \in \{0,\ldots,n-2\} \\ j \in \{i+1,\ldots,n-1\}}} \neg a_i \vee \neg a_j \tag{3.11}$$

Lemma 3.16. *$ExactlyOne(A)$ returns a formula which is satisfiable iff exactly one a_i is $\top$ and all other are $\bot$*

$$ExactlyOne(A) = AtLeastOne(A) \wedge AtMostOne(A) \tag{3.12}$$

Lemma 3.17. *$ExactlyOneNot(A)$ returns a formula which is satisfiable iff exactly one a_i is $\bot$ and all other are $\top$*

$$\begin{aligned} &ExactlyOneNot(A) = ExactlyOne(A') \\ &where\ A' = \{\neg a \mid a \in A\} \end{aligned} \tag{3.13}$$

Proof. To prove that $ExactlyOneNot(A) = ExactlyOne(A')$ holds we substitute a_i with $a'_i = \neg a_i$:
$ExactlyOne(A')$ is trivially $\top$ iff exactly one a'_i is $\top$
Therefore the number of variables which turn out to be $\bot$ has to be one. □

Lemma 3.18. *$ExactlyTwo(A)$ returns a formula which is satisfiable iff exactly two variables a_i and a_j, $i \neq j$ are $\top$ and all other are $\bot$*

$$ExactlyTwo(A) = \bigvee_{\substack{i\in\{0,\dots,n-2\}\\ j\in\{i+1,\dots,n-1\}}} (a_i \wedge a_j \bigwedge_{k\in\{0,\dots,n-1\}/\{i,j\}} \neg a_k) \tag{3.14}$$

Proof. We show that the lemma holds in both directions.
Let us assume that we have the assignment where a_i and a_j, $i < j$ are $\top$ and all $a_k \mid k \neq i \wedge k \neq j$ are $\bot$. Then the disjunction $a_i \wedge a_j \bigwedge_{k\in\{0,\dots,n-1\}/\{i,j\}} \neg a_k$ evaluates to $\top$, which makes $ExactlyTwo(A)$ evaluate to $\top$ and therefore satisfiable.
In the other direction, if $ExactlyTwo(A)$ is satisfiable then exactly one of its disjunctions evaluate to $\top$ and this disjunction contains the positive a_i, a_j and the negative a_k. We show that exactly one disjunction evaluates to $\top$ by the following proof of contradiction:
Let us assume that we have two disjunctions which evaluate to $\top$ in the following form

$$a_{i_0} \wedge a_{j_0} \bigwedge_{k\in\{0,\dots,n-1\}/\{i_0,j_0\}} \neg a_k$$

$$a_{i_1} \wedge a_{j_1} \bigwedge_{k\in\{0,\dots,n-1\}/\{i_1,j_1\}} \neg a_k$$

$$\text{where } (i_0, j_0) \neq (i_1, j_1)$$

Then a_{i_0} or a_{j_0} gets negated in the second disjunction's a_k, because $(i_0, j_0) \neq (i_1, j_1)$. This leads to a contradiction, because the first disjunction states that a_{i_0} and a_{j_0} are $\top$. □

Example 3.19. Let $A = \{a, b, c\}$, then
$ExactlyOne(A) = \underbrace{(a \vee b \vee c)}_{AtLeastOne} \wedge \underbrace{((\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c))}_{AtMostOne}$
$ExactlyTwo(A) = (a \wedge b \wedge \neg c) \vee (a \wedge c \wedge \neg b) \vee (b \wedge c \wedge \neg a)$

3.4. Encoding Slitherlink Rules 1 & 2

First we encode the fact that if the cell at (i, j) has the value k, then k of the surrounding lines $(l_{i,j})$ have to be set. For every cell at (i, j) which contains a number we add a formula depending on the value of the cell.

$$\bigwedge_{(i,j) containsNumber} rule_1(val(i,j), i, j) \tag{3.15}$$

$$rule_1(val, i, j) = \begin{cases} None(l_{i,j}) & \text{if } val = 0 \\ ExactlyOne(l_{i,j}) & \text{if } val = 1 \\ ExactlyTwo(l_{i,j}) & \text{if } val = 2 \\ ExactlyOneNot(l_{i,j}) & \text{if } val = 3 \end{cases}$$

The next formula describes the continuity of a line, by simply stating that each line segment has exactly one predecessor and one successor, we call this subformula $PreSucc$:

$$\begin{aligned} PreSucc := \bigwedge_{(k,j)} h_{k,j} \Rightarrow ExactlyOne(preH_{k,j}) \wedge ExactlyOne(succH_{k,j}) \wedge \\ \bigwedge_{(i,l)} v_{i,l} \Rightarrow ExactlyOne(preV_{i,l}) \wedge ExactlyOne(succV_{i,l}) \end{aligned} \tag{3.16}$$

This encoding also garantuees that no lines cross each other, because therefore the number of predecessor or successor has to be > 1.
An alternative encoding states that the number of lines crossing the point $c_{k,l}$ has to be two or zero, we call this subformula $TwoLine$. Like the other encoding mentioned above, it also encodes that no lines cross each other.

$$TwoLine := \bigwedge_{(k,l)} None(c_{k,l}) \vee ExactlyTwo(c_{k,l}) \tag{3.17}$$

To refer to the encodings later we define $Rule_1$ as the formula generated in Equation 3.15. Furthermore $PreSucc_{1,2} := Rule_1 \wedge PreSucc$ and $TwoLine_{1,2} := Rule_1 \wedge TwoLine$.
With these fairly simple encodings we ensure that the formula is satisfiable iff Rules 1 and 2 hold. The problem is that the encodings also allow many models which violate Rule 3. The bigger the puzzle, the more likely it is that there are two or more loops in the model. Therefore we introduce some methods, which guarantee that rule 3 is observed.

4. Tackling the 1 Loop Problem

To complete the SAT encoding with Rule 3 we present two different approaches:

- Reachability Encoding: the formulas $PreSucc_{1,2}$ and $TwoLine_{1,2}$ are extended with a boolean encoding of Rule 3.
- Mixed Procedures: we use the formula $PreSucc_{1,2}$ or $TwoLine_{1,2}$ to generate models and use an algorithm which decides if these models violate Rule 3.

4.1. Reachability Encoding

This technique was already used in the encoding of Nurikabe [9]. First we need a new set of variables $f_{i,j}$ which tells us whether the cell at (i, j) is inside $(f_{i,j} = \top)$ or outside $(f_{i,j} = \bot)$ of a loop. We propose some kind of filling algorithm:

- For every cell at the border: if the line at the border is set, than the cell is inside the loop. Otherwise, the cell is outside of the loop.
- If there is **no seperating line** between two adjacent cells they must be at the **same** side of the loop.
- If there is a **seperating line** between two adjacent cells they must be at the **opposite** side of the loop.

Formally:

$$\begin{aligned}
&\bigwedge_j (h_{0,j} \Leftrightarrow \neg f_{0,j}) \wedge (h_{n,j} \Leftrightarrow \neg f_{n-1,j}) \\
&\bigwedge_i (v_{i,0} \Leftrightarrow \neg f_{i,0}) \wedge (v_{i,0} \Leftrightarrow \neg f_{i,0}) \\
&\bigwedge_{\substack{1\le i\le n-1 \\ 0\le j\le m-1}} h_{i,j} \Leftrightarrow \neg (f_{i-1,j} \Leftrightarrow f_{i,j}) \\
&\bigwedge_{\substack{0\le i\le n-1 \\ 1\le j\le m-1}} v_{i,j} \Leftrightarrow \neg (f_{i,j-1} \Leftrightarrow f_{i,j})
\end{aligned} \tag{4.1}$$

Now we can garantuee the compliance of Rule 3 by stating the following requirements:

1. **Every cell within the loop should be connected to all other inner cells.**
   If there is more than one loop in an assignment (cf. Figure 4.1(a)) the requirement would not be fulfilled, because the cells of the first loop would not have a connection to the other loops without leaving its boundaries.

2. **Every cell outside the loop should be connected to a cell which is at the border.**
   This is required due to the fact that there exist loops with "holes" (cf. Figure 4.1(b)). But if all outer cells are connected to the border of the puzzle no such holes can exist.

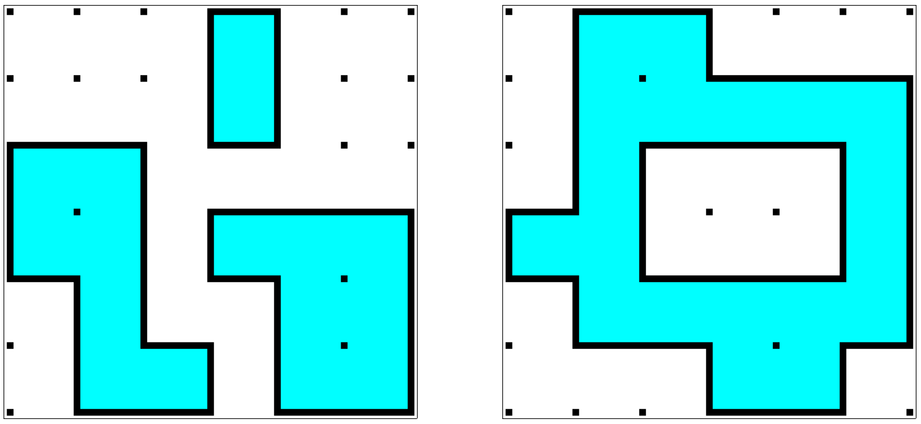

(a) Inner cells not connected (b) Not all outer cells connected to outer border cells

Figure 4.1.: Puzzles violating the requirements.

Two cells are connected if there exists a path of cells, which all share the same $f_{i,j}$ value. In [9] a new set of variables $R^n_{(i,j),(k,l)}$ is defined, meaning that there exists a path from (i,j) to (k,l) with at most n steps. Now we can encode the properties stated above:

1. The following algorithm encodes the first requirement mentioned above:
   a) Find the cell h with the highest value v of the puzzle and ignore the cell if it is in a corner with value 2.
   b) If no such cell could be found return the inefficient encoding:
   $$\bigwedge_{i,j}\bigwedge_{k,l}(f_{i,j} \wedge f_{k,l}) \Rightarrow R^{n_{max}}_{(i,j),(k,l)} \tag{4.2}$$
   c) Otherwise return:
   $$\bigwedge_{(n_i,n_j)\in NB\cap h}\bigwedge_{k,l}(f_{n_i,n_j} \wedge f_{k,l}) \Rightarrow R^{n_{max}}_{(n_i,n_j),(k,l)} \tag{4.3}$$
   where NB is the set of the neighbours of h

   The optimization made in Equation 4.3 reduces the number of clauses generated from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. This is achieved by stating that every inner cell should be connected to some fixed inner cells. These fixed inner cells are found by scanning the initial puzzle before generating the encoding. The number of fixed cells depends on v, the highest value of all cells.

2. The second requirement is encoded with:

$$\bigwedge_{i,j}(\neg f_{i,j} \Rightarrow \bigvee_{(k,l)\in BorderCells} R^n_{(i,j),(k,l)}) \tag{4.4}$$

We define $Reach_3$ as the conjunction of the encodings mentioned above.

4.2. Mixed Procedures

With the basic encoding of Section 3.4 we have a method to quickly generate possible assignments. These assignments can then be checked by an external algorithm if they fulfill the 1 loop constraint. We present two different algorithms to check the property:

4.2.1. Fill Algorithm

The **"Fill Algorithm"** works by partitioning the puzzle into an inner and into an outer side of the loop, similar to the encoding in Section 4.1. In order for this algorithm to work we need a list which saves the cells that were already visited.

1. Mark the cells of the border which are on the outer side of the loop.
2. Scan the puzzle from top left to bottom right line by line and find the first vertical line which is set. This is the beginning of the inner loop and therefore the cell to the right of this line is marked as initial inner cell.
3. Recursively mark adjacent cells if there is no line between them.
4. After the recursion has ended, there exist two possibilities
   - if there exist cells which were not marked then the assignment is not correct
   - otherwise all cells were marked and the assignment is correct

A detailed description in pseudocode can be found on Page 13.

To show that this algorithm is valid, one can show that the requirements of Section 4.1 are fulfilled:

1. **Every cell within the loop should be connected to all other inner cells.**
   By marking the initial inner cell of the puzzle in step 2, all of the inner cells of the first loop are marked, when the algorithm ends. If there are more than one loops, its cells are ignored and therefore not marked when the algorithm ends, resulting in a wrong assignment.
2. **Every cell outside the loop should be connected to a cell which is at the border and outside of the loop too.**
   By marking the outer cells of the border in step 1, all of the reachable outer cells are marked too, when the algorithm ends. This fulfills the requirement and prevents holes in loops.

Algorithm 1 Fill Algorithm

```
Input: a $n \times m$ Slitherlink puzzle
Output: a boolean which is true iff the puzzle is correctly solved
1: $S := \emptyset$ {the set of cells which were visited}
2: $Q := \emptyset$ {the set of cells which should be explored next}
3: $firstFound :=$ false {true iff the beginning of the loop found}
   {Visit the border cells and the first cell of the inner loop}
4: for $x := 0$ to $N - 1$ do
5:    for $y := 0$ to $M - 1$ do
6:       if firstFound $= false$ and $v_{x,y}$ then
7:          $S := S \cup \{s_{x,y}\}$
            $firstFound := true$
8:       end if
9:    end for
10: end for
11: for every outer cell at the border $bc$ do
12:    $S := S \cup bc$
13: end for
    {Recursively visit adjacent cells}
14: $Q := S$
15: while $Q \neq \emptyset$ do
16:    $T := \emptyset$ {temporary set to save the next queue}
17:    for $s \in Q$ do
18:       for every adjacent cell $a \notin S$ of $s$ do
19:          if no line between $a$ and $s$ then
20:             $S := S \cup \{a\}$
                $T := T \cup \{a\}$
21:          end if
22:       end for
23:    end for
24:    $Q := T$
25: end while
26: if $\#S = n * m$ then
27:    return true
28: else
29:    return false
30: end if
```

4.2.2. Linecount Algorithm

The **"Linecount Algorithm"** was presented in [3]. It works by finding a start line and following the loop through the pre/successor of each line, counting the number of lines in the loop. If this number matches the total number of lines, the assignment has to be correct. The pseudocode of this algorithm is presented on the next page.

Algorithm 2 Linecount Algorithm

Input: a $n \times m$ Slitherlink puzzle
Output: a boolean which is true iff the puzzle is correctly solved

```
1: $S := \{h_{i,j} \mid i <= n, j < m, h_{i,j} = \top\} \cup \{v_{i,j} \mid i < n, j <= m, v_{i,j} = \top\}$ {Set of all set lines}
2: $start := null$ {The start line of the loop}
   {Find the start line}
3: for $x := 0$ to $N - 1$ do
4:   for $y := 0$ to $M - 1$ do
5:     if $start = null$ and $v_{x,y}$ then
6:       $start = v_{x,y}$
7:     end if
8:   end for
9: end for
10: removeRecursive(S,start)
11: return $\#S = 0$
```

Algorithm 3 removeRecursive(S,l)

Input: S the set of remaining lines, l the line to be removed

```
1: if $l \in S$ then
2:   $S := S \backslash \{l\}$
3:   for every predecessor/successor $l'$ of $l$ do
4:     if $l' = \top$ then
5:       $removeRecursive(S, l')$
6:     end if
7:   end for
8: end if
```

Example 4.1. Figure 4.2 shows the "Fill Algorithm" on a 3×3 puzzle.

Example 4.2. Figure 4.3 shows the "Linecount Algorithm" on a 3×3 puzzle.

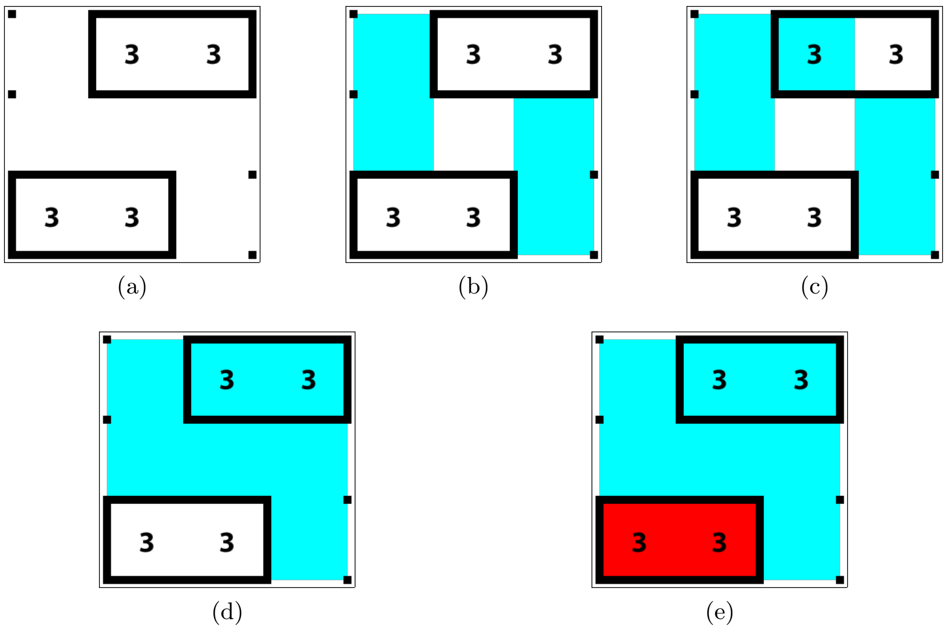


Figure 4.2.: Fill algorithm: (a) the initial assignment, (b) the border cells are marked, (c) the first cell of the inner loop is marked, (d) after one recursion step, (e) return false, because not all cells were marked

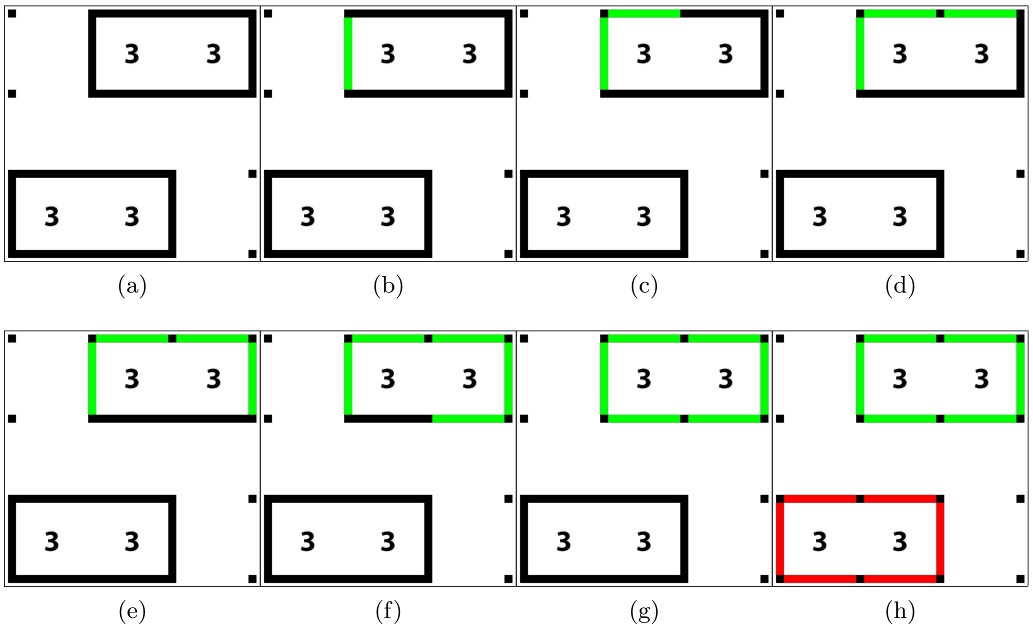


Figure 4.3.: Linecount algorithm: (a) the initial assignment, (b) the start line was found and marked, (c)- (g) the connected loop is marked recursively, (h) return false, because not all lines were marked

4.2.3. Optimizations

The Fill and Linecount algorithm work well for smaller puzzles but finding solutions for puzzles with more than 200 cells does not work. The problem is that the encodings produce many thousands of quite similar "false" assignments. To reduce the solution space we propose a method of "Iterative SAT solving": If we get a false assignment from the solver we deduce as much information as possible, feed the solver with this new information and ask for a new assignment, without resetting the state of the SAT solver.

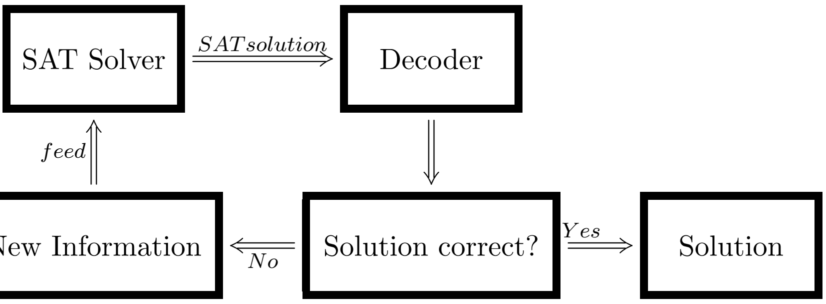


Figure 4.4.: The Iterative SAT solving process

In the case of Slitherlink we modify the algorithms mentioned above to recursively return all loops but the first of the solver's assignment. These loops are added to the solver in negated form, so that the solver ignores these false loops. With this approach it was even possible to solve 30×30 puzzles in a matter of seconds.

Example 4.3. Figure 4.5 shows the Iterative SAT solving of a 6×6 puzzle.

Another optimization was to use the **prework solver** of [10]. The prework solver uses a pattern matching algorithm to iteratively find certain Slitherlink patterns [1]. We use this prework solver to generate a conjuction of the set and unset lines and feed the conjunction to the SAT solver.

[1]Some patterns can be found at http://en.wikipedia.org/wiki/Slitherlink#Solution_methods

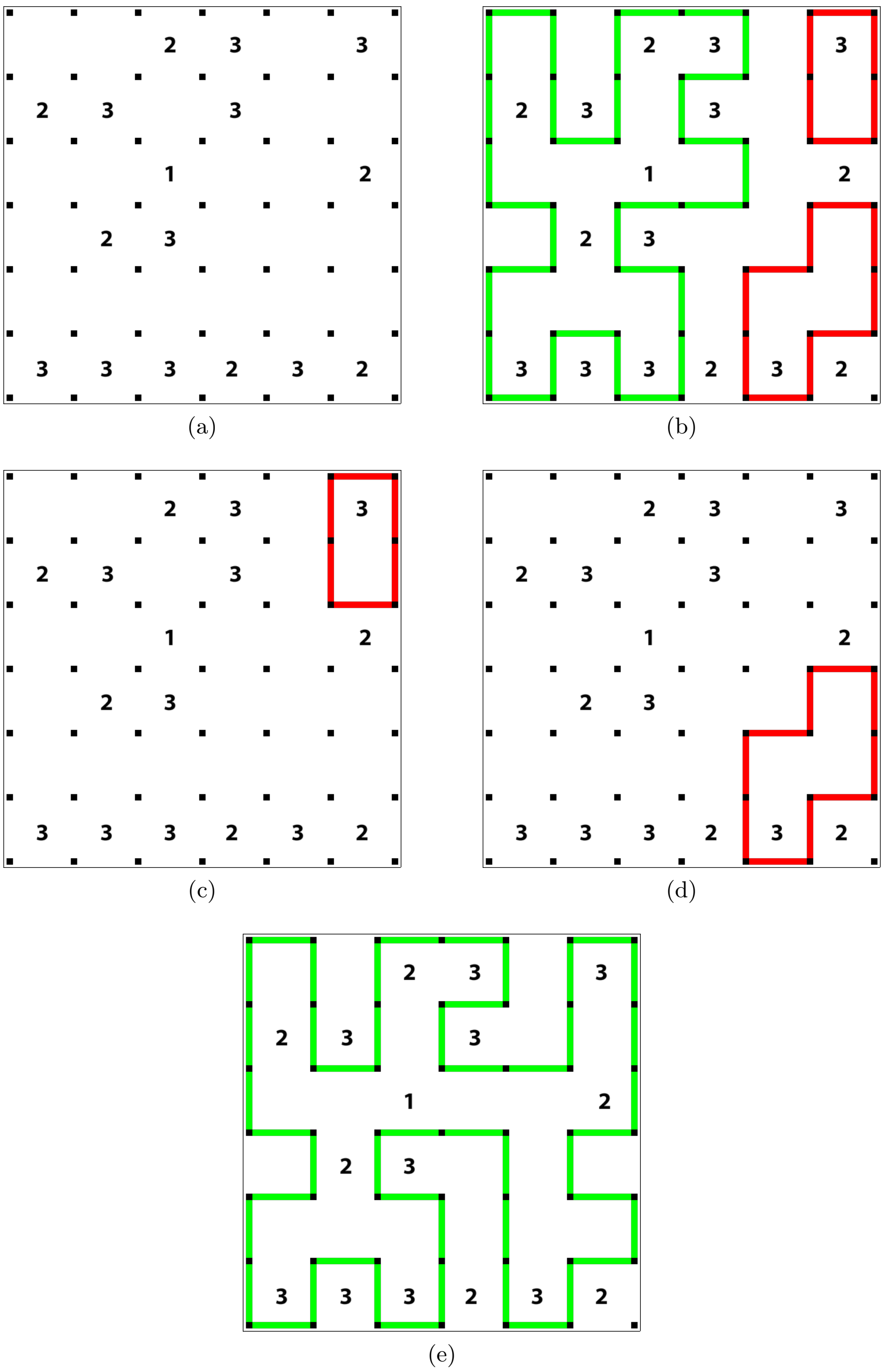


Figure 4.5.: 4.5a the initial puzzle, (b) the SAT solver returned a "wrong" assignment. (c) and (d) The variables of the lines of the wrong loop are grouped in a negated conjunction: $\neg \bigwedge_{l \in \{loop\}} l$ and added to the SAT solver.(e) After some iterations the correct solution is returned by the solver.

5. Implementation

5.1. Class Hierarchy

5.1.1. Propositional Logic Implementation

In order to encode the Slitherlink puzzle in Java, an object oriented definition of propositional logic was defined:

- **Formula**. The base interface for a formula in propositional logic.
  - **Formula convertToCNF()**: returns an equisatisfiable formula in CNF.
  - **boolean isLiteral()**: returns true iff the formula is a literal (normal or negated variable).
  - **int toNum()**: if the formula is a literal, returns the number used for the SAT solver.
- **Operator implements Formula**. The base class for a binary operator. Manages two formula arguments f1 and f2. The subclasses of Operator include: And, Or, Not, Implication and Biimplication.
- **Variable implements Formula**. The base class for a variable. Tracks an ID with every instance and saves all instances in a static list. Can be either set or not set (by SAT solver).
- **Set(Formula... forms)**. A class which groups various formulas in a set.

Example 5.1. The following code

```
Formula f = new Implication(
                new R(),
                new And(new R(),new R()));
System.out.println(f);
System.out.println(f.convertToCNF());
```

leads to the output

```
(R1 => (R2 ^ R3))
((-R1 v R2) ^ (-R1 v R3))
```

Furthermore all of the boolean helper functions of Section 3.3 were implemented using the definitions mentioned above.

5.1.2. Encoding Implementation

EncodedGame(int m, int n) is the abstract base class for an encoding of a $m \times n$ puzzle. It contains the following fields and methods:

- **static boolean USE_PREWORK_SOLVER**: use the prework solver as explained in Section 4.2.3
- **static enum EXTERNAL_ALGO**: can be one of the following enums
  - REACHABILITY_ENCODING: use the reachability encoding of Section 4.1
  - LINE_ALGO: use the line algorithm of Section 4.2
  - FILL_ALGO: use the fill algorithm of Section 4.2
- **static enum CNF_METHOD**: can be one of the following enums
  - STANDARD: use the standard CNF generation of Section 5.2.1
  - TSEITIN: use the tseitin transformation of Section 5.2.2
- **solve(Gamefield gf)**: generates the formula, converts it into CNF and then starts the SAT solving process. If a solution is found the game field is set to the first solution.
- **abstract void getFormula()**: all subclasses implement this method. Each subclass thereby adds arbitrary formulae to a blocking queue.

Due to the large size of the formulas, for even small puzzles, a small performance improvement was made by improving the formula generation:
The problem with first generating the formula and then feeding it to the solver, was that the formula was therefore stored two times in the memory (the first time in the representation explained in Section 5.1.1 and the second time in the SAT solver itself). The memory usage could be cut in half by using a blocking queue which ensures that at most 10 subformulae stay in memory before they are fed to the solver. Furthermore concurrent programming was used to gain performance with multicore systems:

1. In the solve method a new thread is started which only calls the getFormula method and therefore feeds the blocking queue.
2. The main thread meanwhile "consumes" the formulae of the blocking queue by:
   - converting the formula into CNF
   - and passing them to the SAT solver

Example 5.2. A small excerpt of the encoding implementation of the Equation 3.15.

```
//hField and vField are the arrays for the line variables
Set lines = new Set(hField[i][j], hField[i + 1][j],
                    vField[i][j], vField[i][j + 1]);
```

```
//currentGf is the gamefield of the existing application
int val = currentGf.getValue(j, i);
if (val == 0) {
        //cnf is the blocking queue
        cnf.put(new ExactNone(lines));
}...
```

5.2. Generating a CNF

There are at least two different techniques implemented to generate the CNF formulas: The standard approach [7, page 58-65] and the Tseitin Transformation [8, page 12,13].

5.2.1. Standard approach

This approach works by succesively eliminitaing non-CNF operators and using the distribution laws of $\wedge$ and $\vee$ to obtain a CNF formula. This can lead to a exponential blowup of the size of the formula.

5.2.2. Tseitin

The advantage of the Tseitin Transformation, is that it increases the size of the formula only linearly. This is achieved by introducing new variables to a subformula.
The optimized version needs the formula to be in NNF (negations are pushed to the variables) and therefore only needs a implication instead of a biimplication per auxiliary variable:

$tseitin(l) = l$	$t'(l) = \top, x_l = l$ (auxiliary variable $= l$)
$tseitin(\phi) = x_\phi \wedge t'(\phi)$	$t'(p \wedge q) = (x_{p\wedge q} \rightarrow x_p \wedge x_q) \wedge t'(p) \wedge t'(q)$
	$t'(p \vee q) = (x_{p\vee q} \rightarrow x_p \vee x_q) \wedge t'(p) \wedge t'(q)$

Thereby x_ϕ denotes the newly introduced variable of the subformula ϕ and l denotes a literal.

5.3. Trivial Cases

Trivial cases of the Slitherlink puzzle are those puzzles which are solvable by a single loop around two adjacent 3-valued cells. The existing solver of [10] had a problem with trivial cases because a pattern of two adjacent 3's was resolved to a configuration, which is not generally true (cf. Figure 5.1(a)).
To fix this problem the function **"trivialCases"**, described in the Appendix A, was implemented. It gets a $m \times n$ puzzle as an input and returns *null* if there was no trivial case found and the solved puzzle otherwise. The function just looks for the first adjacent cells with 3 values and builds the loop around them. If the puzzle is now a valid solution it is returned.

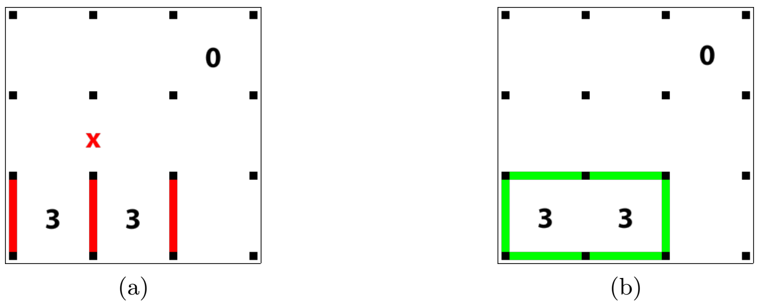


Figure 5.1.: (a) the solution of the prework function of [10], which is not solvable any more, (b) the correct solution of a trivial puzzle

5.4. GUI Improvement

One further task of the thesis was to improve the GUI:
The application now includes an extended solver chooser. You can choose if you want to use the "Standard Solver" (of [10]) or the ones presented in this thesis. Furthermore there were some graphical improvements for the buttons and for the actual game field:

- The look and feel of the swing-application was changed to the more pleasing "Substance" [4].
- The game field is now drawn directly with the Graphics component of Java.awt instead of using JPanel's and JTextField's.
- You can now zoom in/out using your mouse wheel.
- There is now a timer, which counts the seconds you need to solve a puzzle.
- You can now redo steps you have undone.
- The width of the line is now resizable with the slider beneath the "Undo/Redo"-buttons.

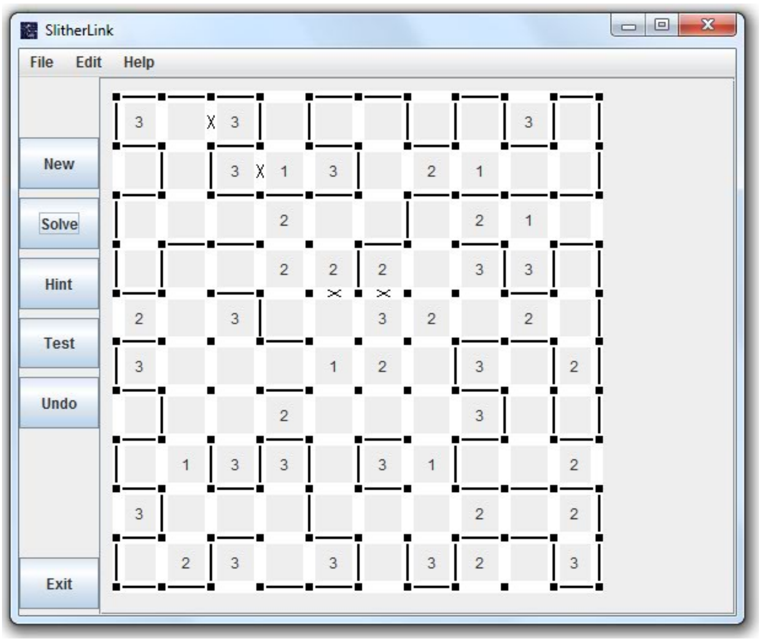


Figure 5.2.: The old GUI

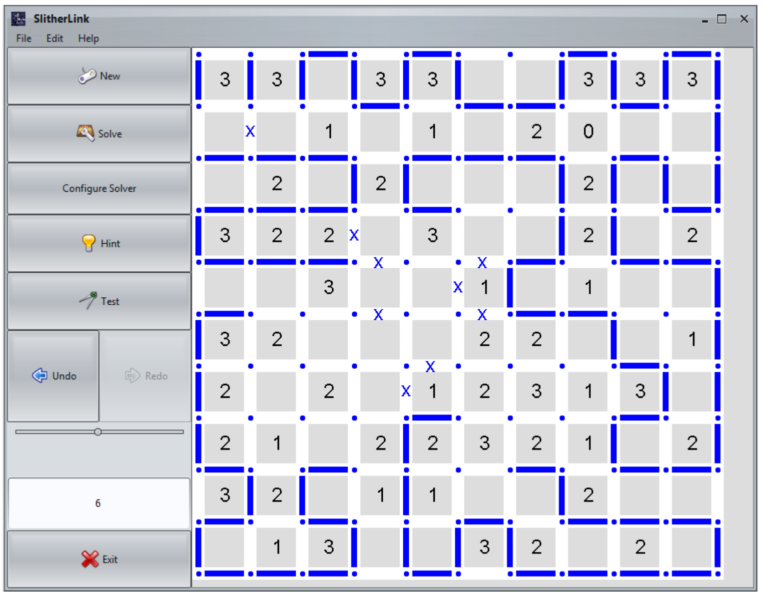


Figure 5.3.: The new GUI.

6. Evaluation

Another aim of the project was to evaluate the exisiting methods with the ones presented in this paper. The following test environment was used for all of the performance tests:

CPU	AMD Athlon 64 X2 Dual Core 2.20GHz
RAM	2.0 GB
Operating System	Windows 7 Ultimate
Java Version	Java SE Runtime (build 1.6.0_17b04)
SAT4J Version	SAT4J 2.2.2

6.1. Performance

We compare the runtimes to solve a puzzle using different configurations and the solver of [10] ("Standard"). The configuration is thereby defined by 4 characters:

1. $[T/P]$ the $TwoLine_{1,2}/PreSucc_{1,2}$ encoding is used.
2. $[R/F]$ the $Reach_3$ encoding/Fill algorithm is used.
3. $[T/S]$ the encoding uses the tseitin/standard cnf conversion.
4. $[T/F]$ the prework solver is turned on/off.

It turned out that, on average, the filling algorithm was faster than the linecount algorithm, so it was used in the mixed procedures encodings.
Figure 6.1 shows the average time t in ms $(y - Axis)$ it took to solve a puzzle with n number of cells$(x - Axis)$. We observe:

- The standard solver of [10] is the fastest solver.
- The fastest configurations are :
  - $PreSucc_{1,2}$ with the prework solver and Tseitin or without the prework solver and the standard CNF conversion.
  - $TwoLine_{1,2}$ with the prework solver and the Tseitin CNF generation.
- The reachability encoding is not applicable to puzzles with more than 50 cells.

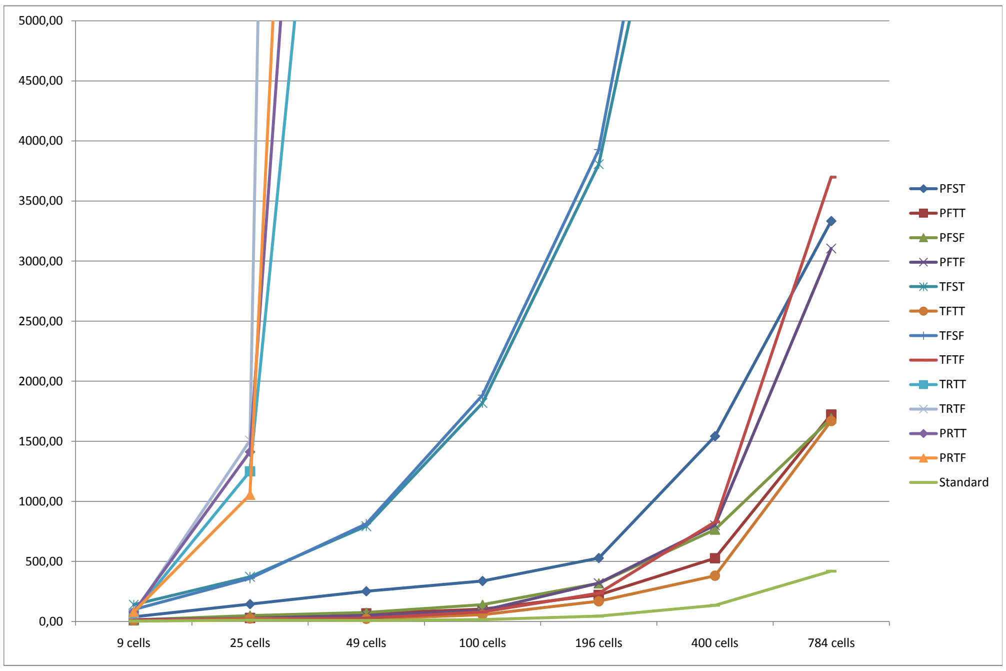


Figure 6.1.: Solving time of the configurations and the Standard Solver in ms

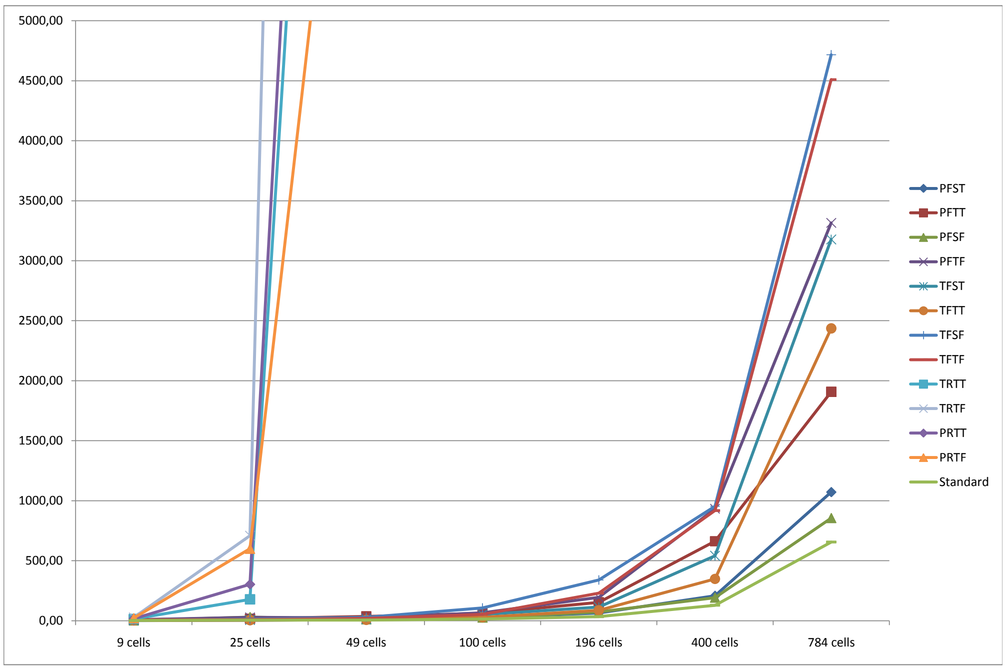


Figure 6.2.: SAT solving time of the configurations and the Standard Solver in ms

Figure 6.2 shows a more optimistic time measurement, by only taking in account the time the SAT solver needs to find all solutions. These times could for instance be achieved, by caching the formulas in files on the hard drive. It shows that the best encodings can even compete with the existing solver in terms of speed.

As pointed out by one of the reviewers of the thesis, the runtime of the SAT solver exceed the runtime of the normal solver, the bigger and more difficult the puzzles get. In Figure 6.3 we see that out of the 7 puzzles [1] only 2 could be solved by the existing solver (the solving process was interrupted after 5 minutes).

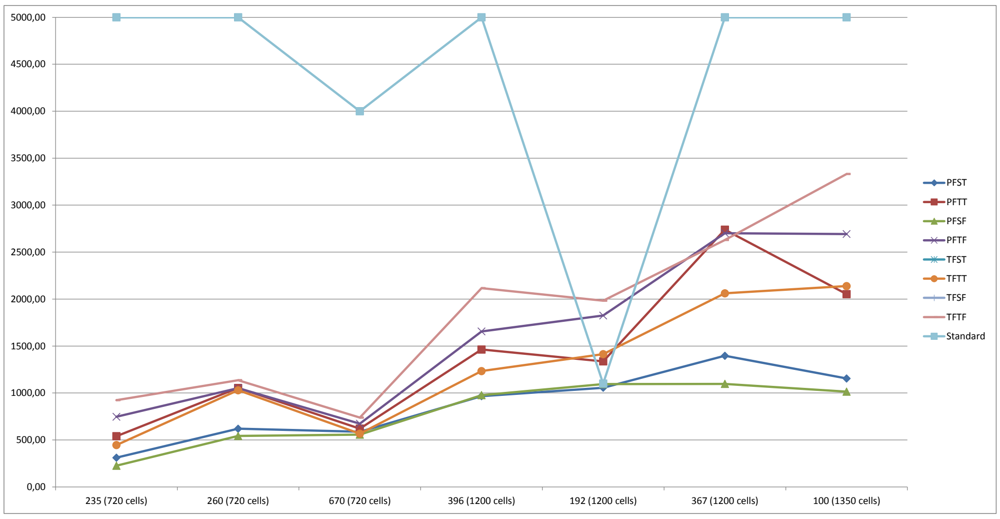


Figure 6.3.: Solving time of the configurations and the Standard Solver in ms

[1]The biggest puzzles with the highest difficulty from `http://www.janko.at/Raetsel/Slitherlink/index.htm` were used. The puzzles that were used for the evaluation correspond to the number on the $x - Axis$

6.2. Encoding Sizes

Next we compare the encodings by the number of variables they use and the number of clauses they generate. Figure 6.4 shows the number of variables generated for each configuration and Figure 6.5 shows the number of clauses generated for each configuration.
We observe:

- that even small puzzles encoded with $Reach_3$ generate huge formulas, no matter if it is transformed with Tseitin or the standard approach.
- that the Tseitin transformations generally generate more variables, while reducing the number of clauses.

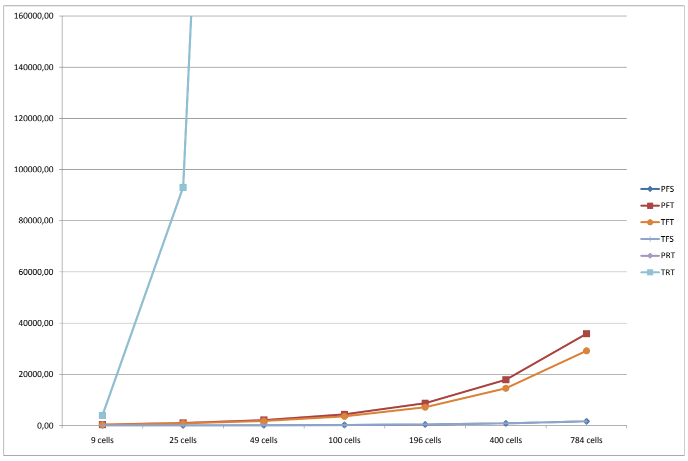


Figure 6.4.: Number of variables generated with configuration

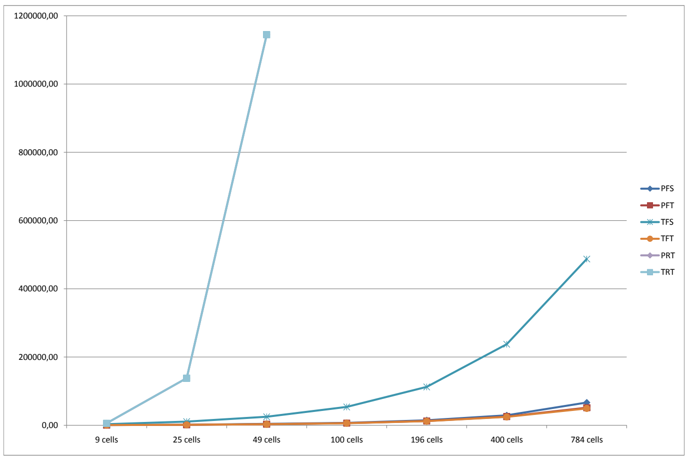


Figure 6.5.: Number of clauses generated with configuration

7. Conclusion

Slitherlink puzzles are fun and challenging[1]. The generation of a solver, which solves puzzles faster than humans, was even more fun.
In this work such a solver was presented, integrated into the existing application and evaluated. The solver uses SAT-encodings of the rules of Slitherlink and an ”Iterative SAT”-technique to speed up the solving process.
As it turned out, the solving process presented in this work was not as fast as the existing solver for small to medium-sized puzzles. But it was shown that the old approach has problems with big puzzles, not generated by the program itself. Most of these puzzles could only be solved by the techniques introduced in this work. I am confident that with more powerful SAT solvers and more optimized encodings, all instances could be solved as fast as with the trial-and-error solver.
Further improvements that could be made, include:

- Linear encodings of the $ExactlyOne(A)/ExactlyTwo(A)$ properties mentioned in Section 3.3
- Cache the formulas on the hard drive, instead of generating them in runtime
- Better encodings of Rule 3
- Better CNF converters
- Faster SAT solver

[1]Some sites with slitherlink puzzles:
`http://de.puzzle-loop.com`
`http://www.krazydad.com/slitherlink/`
`http://www.janko.at/Raetsel/Slitherlink/index.htm`
For Android Users: `market://search?q=pname:jp.ne.sakura.knatt.slitherlink`

Bibliography

[1] MiniSat. http://minisat.se.

[2] SAT4J. http://www.sat4j.org/.

[3] Solving Slitherlink puzzles with a CSP solver. http://bach.istc.kobe-u.ac.jp/sugar/puzzles/slitherlink.html.

[4] Substance - Java Look & Feel. https://substance.dev.java.net.

[5] Wikipedia - Slitherlink. http://en.wikipedia.org/wiki/Slitherlink.

[6] A. P. Daniel Le Berre. On SAT Technologies for dependency management and beyond. http://www.mancoosi.org/papers/leberre-sat-beyond.pdf.

[7] M. Huth and M. Ryan. *Logic in Computer Science 2nd ed.* Cambridge University Press, 2004.

[8] D. Krning and O. Strichman. *Decision Procedures.* Springer Verlag, 2008.

[9] C. Terzer. Nurikabe as SAT Problem. Bachelor's thesis, Computational Logic Group, University of Innsbruck, 2007.

[10] L. Thuile. Slitherlink. Bachelor's thesis, Computational Logic Group, University of Innsbruck, 2008.

[11] D. S. Weld. Recent Advances in AI Planning, 1998. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[12] T. YATO. Complexity and completeness of finding another solution and its application to puzzles, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

A. Trivial Cases

Algorithm 4 trivialCases

```
Input: a $m \times n$ Slitherlink puzzle
Output: a trivial solution or null if none could be found
for $x := 0$ to $N - 1$ do
  for $y := 0$ to $M - 1$ do
    if value of cell $(x, y) = 3$ then
      if $x < N - 1$ and value of cell $(x + 1, y) = 3$ then
        draw loop around cells $(x, y)$ and $(x + 1, y)$
        if solution valid then
          return puzzle
        else
          return null
        end if
      end if
      if $y < M - 1$ and value of cell $(x, y + 1) = 3$ then
        draw loop around cells $(x, y)$ and $(x, y + 1)$
        if solution valid then
          return puzzle
        else
          return null
        end if
      end if
    end if
  end for
end for
return null
```