Bachelor Thesis

Yajilin

Armin Gufler
Florian Hagenauer
armin.gufler@student.uibk.ac.at
florian.hagenauer@student.uibk.ac.at

1 July 2011

Supervisor: Univ.-Prof. Dr. Aart Middeldorp

Abstract

This bachelor thesis is about the Japanese logical puzzle Yajilin. The thesis covers the development of a powerful solver and generator for Yajilin puzzles and the realization of an intuitive user interface which allows to play the puzzle. The first chapters are about the puzzle in general and the main technologies used throughout the development. These chapters are followed by the two main chapters about the solving and generation process. Within these chapters the realization of the components is explained in detail. The next chapter is about the results of extensive benchmarks regarding the solver and generators. Afterwards a description of the implemented user interface and the underlying architecture is given. A final conclusion outlines which goals have been achieved.

Contents

1 Introduction

1.1 Motivation

During the summer break between semester four and five we started to look for a suitable bachelor thesis. Our aim was to find a topic which is both about programming and about creating new algorithms. The project about the puzzle Yajilin therefore was really appealing for us.
During our search we found out that currently there are three books published by Nikoli with Yajilin example problems. There are also a few blogs offering some puzzles but there is nothing like a database for sample puzzles. Moreover, we did not find any other automatic solvers nor generators for Yajilin puzzles.

1.2 Goals

There are three major goals to fulfil in the project described in this thesis.

1. Provide a graphical user interface similar to the official Nikoli Player[1].
2. Construct and write an automatic solver for arbitrary Yajilin puzzles.
3. Construct and write a generator which generates puzzles of different complexity.

To build the generator we first had to make a working solver. The GUI was developed alongside the other two parts of the thesis.

1.3 Overview

Chapter 2 introduces the rules for the puzzle itself and gives examples of errors. The next chapter describes the main goals of the thesis and what format we chose to represent the puzzles in our program. Chapters 4 and 5 are about the solving and generation of new puzzles. Another chapter shows the results of benchmarks regarding the solver and the generators. The last chapter is about the graphical user interface and what technologies we used in our project.

[1] `http://www.nikoli.com/en/puzzles/yajilin/`

2 Yajilin

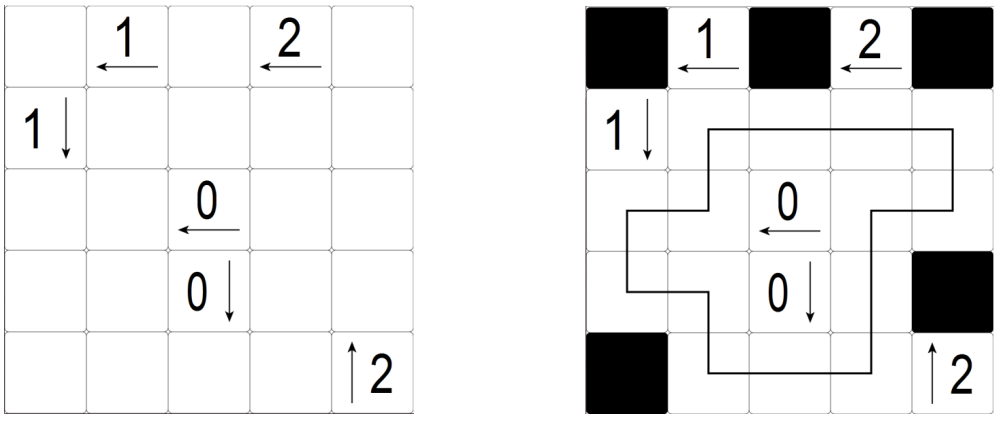
Yajilin is a Japanese logical puzzle published by Nikoli in the year 1999.[1] Yajilin is played in a grid with different kinds of cells. A cell can be either black, an arrow with a number, or a line. Initially there are only arrows and empty cells in the puzzle.

2.1 Rules

There are four rules which apply and have to hold to consider a puzzle solved:

1. All cells have to be filled (either black, arrow or line).
2. A line should be drawn which makes a single loop. The line isn't allowed to cross itself or to branch.
3. Black cells are not allowed to touch vertically or horizontally.
4. An arrow indicates the number of black cells in the given direction.

In Figure 2.1(a) a simple unsolved puzzle can be seen and in Figure 2.1(b) the solution is displayed. It can easily be observed that all rules stated above are fulfilled.



(a) The empty puzzle

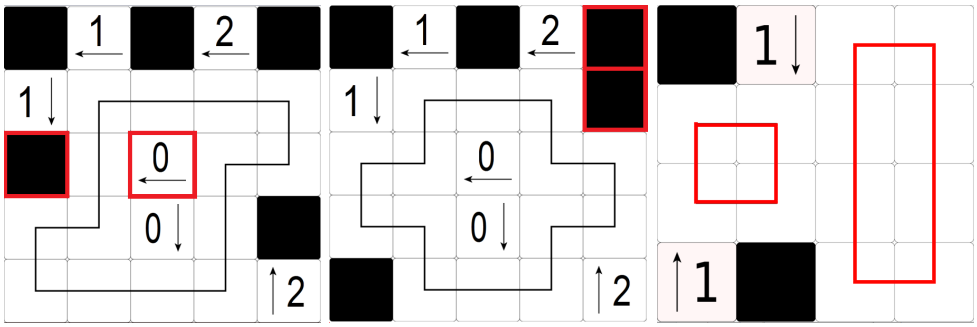
(b) The solved puzzle

Figure 2.1: An empty puzzle and its solution

[1] http://www.nikoli.co.jp/en/puzzles/yajilin/

It is also considered that the puzzles are only allowed to have one solution. This is not mentioned explicitly in the official description but without this last condition it would be possible to create puzzles with many different solutions.

Some incorrect attempts to solve a Yajilin puzzle are shown in Figure 2.2.



(a) Arrow Error (b) Black Error (c) Line Error

Figure 2.2: Different error scenarios

Looking at Figure 2.2(a) it can be observed that the arrow in the middle, with number 0, points to the left which means that in this row no black cells are allowed. But there is one placed at the beginning of the row resulting in a violation of rule 4.

Rule 3 is violated in the puzzle shown in Figure 2.2(b). The number of black cells indicated by the arrow in the lower right corner is correct, but the two black cells are not allowed to be adjacent.

Figure 2.2(c) shows a puzzle which satisfies rules 1, 3 and 4. But rule 2 does not apply because two loops are drawn and only one is allowed. The player's task is now to correctly fill the puzzle in a way to conform to the four rules.

3 Scope of the Thesis

3.1 Problems Addressed in this Thesis

There are three major topics covered in this thesis. The first is building a graphical user interface which allows a user to play the puzzle. The features of this program should be similar to the one provided by Nikoli[1]. That means that the player should be able to play the puzzle and also have a trial & error mode. In this mode he can try out some ideas and make them permanent or discard them. In addition to these features it is also possible to generate and solve puzzles directly from the interface.

The second topic is building a solver for such Yajilin puzzles. Such a solver should be able to perform two tasks. On the one hand it should be capable of finding out if the puzzle is solvable and if so return the correct solution. On the other hand the solver ought to be able to find out if there exists more than one solution. During our research we did not find any other automatic solver. So it was unfortunately not possible to compare our solver to other implementations. The details about the solver are presented in Chapter 4.

The last topic is the automatic generation of new Yajilin puzzles. Generated puzzles should not only be correct and solvable but also appealing to solve and interesting for the player. Puzzles with for example many arrows can be created very easily but are neither very fun or hard to solve. On the other hand, puzzles with only a few arrows are quite hard to solve, because they only have a small number of hints. The generator should be able to create puzzles of different difficulties and sizes. Generation of such puzzles can be a time-consuming task. A detailed description of our generators can be found in Chapter 5.

Furthermore the program contains a simple puzzle creator which allows the user to create new puzzles from scratch. Such created puzzles can be saved directly to XML and can be played afterwards if they fulfil all rules and have only one solution.

3.2 Representation of Yajilin Puzzles

For an easy representation of Yajilin puzzles a simple, XML based format which offers good portability has been chosen. The program also offers methods for

[1] `http://www.nikoli.com/swf/yl.swf?loadUrl=/nfp/yl-0001.nfp&lang=1`

loading puzzles from XML directly into the program and saving puzzles in their XML representation.

The XML representation is divided into two parts. At first is a general description of the puzzle with height, width and the puzzle name. The name is used to distinguish between different puzzles, width and height alone wouldn't be enough because there are many puzzles with the same parameters. Secondly a list of tiles follows. This list should contain all tiles of the puzzle which are not empty. That means, tiles not listed explicitly in this list are considered empty. The advantages of this representation are that it is readable by machines and by humans and that it is very compact due to leaving out the empty cells. Also the existing tools for handling XML files are quite fast and easy to use with all major programming languages. The disadvantage is that a human can not see the whole puzzle because the cells are only in list form and not in any kind of real visual representation. The arrow tile and the line tile have `dir` tags. These tags describe the two directions of the line in case of a line tile. When used in an arrow tile the direction in which the arrow points is described. The tags `xPos` and `yPos` correspond to the coordinates on the vertical and horizontal axis. This may look a bit unfamiliar at first but is similar to the two-dimensional array notation which also addresses cells in this order.

```
<puzzle>
  <width>3</width> <height>3</height>
  <name>Simple No1</name>
    <tiles>
      <tile>
        <xPos>0</xPos>  <yPos>0</yPos>
        <black/>
      </tile>
      <tile>
        <xPos>1</xPos>  <yPos>0</yPos>
        <arrow>
          <dir>DOWN</dir>
          <count>1</count>
        </arrow>
      </tile>
      <tile>
        <xPos>1</xPos> <yPos>1</yPos>
        <line>
          <dir1>UP</dir1>
          <dir2>DOWN</dir2>
        </line>
      </tile>
  </tiles>
</puzzle>
```

Listing 3.1: Simple XML representation of a puzzle

In Listing 3.1 a simple puzzle in XML format can be seen. The corresponding puzzle is shown in Figure 3.1.

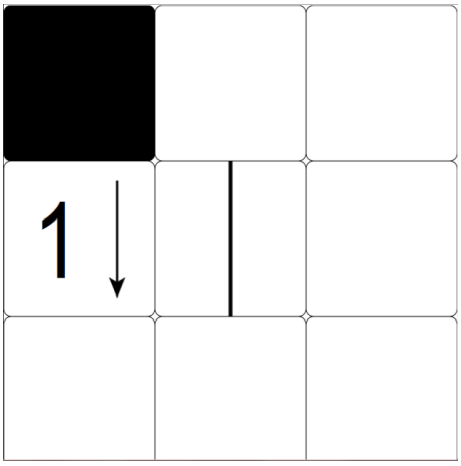


Figure 3.1: The puzzle that corresponds to Listing 3.1

In the implementation object orientation paradigms are used to represent the puzzle in an intuitive way. The main class `Puzzle` has an array of `Tile`. `Tile` is an interface from which the four tile types inherit. We decided to use `EmptyTile` too, instead of dropping it like in the XML format because it makes working with the puzzles much more convenient. With the Java operator `instanceof` the tile type can easily be checked and no special attributes or enumerations are needed. The UML graph of these classes can be seen in Figure 3.2.

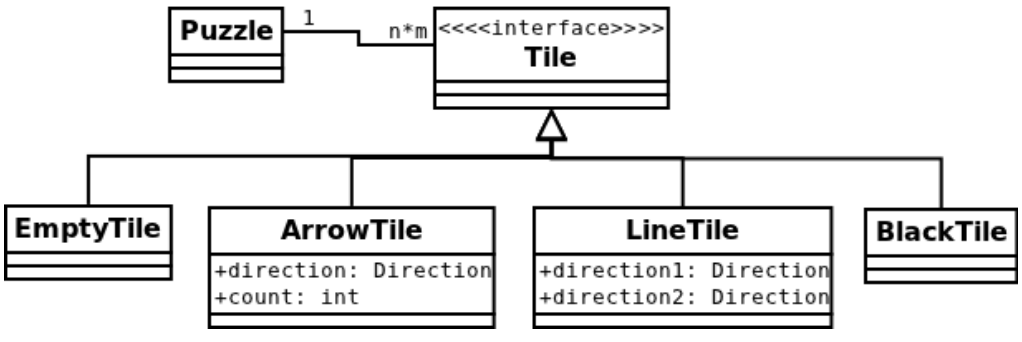


Figure 3.2: UML representation of a puzzle.

4 Automatic Solver

One of the main goals of this thesis was to implement methods to solve every correct puzzle automatically. Within this chapter the process of solving a Yajilin puzzle using a SAT solver is described in detail. Most importantly it will be shown how to construct a formula which encodes the important constraints of a puzzle.

4.1 SAT Solving

SAT solvers are used to solve large satisfiability problems (SAT) of propositional logic. These are problems where it should be determined if a given boolean formula can evaluate to `true` by assigning the variables in a certain way. This means to decide whether a formula is satisfiable or unsatisfiable. Such problems may be very complex and are indeed NP-complete. For a detailed introduction into SAT solving the reader is referred to [2].

In order to get a concrete and powerful SAT solver `SAT4J` is used [3]. The `SAT4J` project offers powerful SAT solving technologies for `Java`.

4.2 Solving Overview

This section is dedicated to describe how to get a solved Yajilin puzzle.

1. First the puzzle has to be represented as a boolean formula in order to solve it using a SAT solver. This process of converting a puzzle to a boolean formula is described starting with Section 4.3.
2. The generated formula has to be in conjunctive normal form (CNF) in order for the SAT solver to work with it. If it does not have this form already it has to be transformed.
3. The formula has to be encoded in a format suitable for the SAT solver.
4. The SAT solver now tries to solve the puzzle given the encoded formula as input. If the instance is satisfiable the solver will return a representation which shows the assignment of each variable of the formula.
5. If the solver finds a satisfying assignment it has to be checked if there exists only one line forming a single loop. If this is the case, the solution

is found, otherwise the solver has to try to find another solution. This check is necessary because the formula does not contain the constraint of a single loop. This will be described further in Section 4.7.2.

Figure 4.1 visualizes this course of actions.

The two main components regarding solving are the `FormulaBuilder` which constructs a formula and the `Solver` component itself. The latter one just invokes the `FormulaBuilder` to get the formula for the puzzle and then uses the concrete SAT solver, trying to find a correct solution.

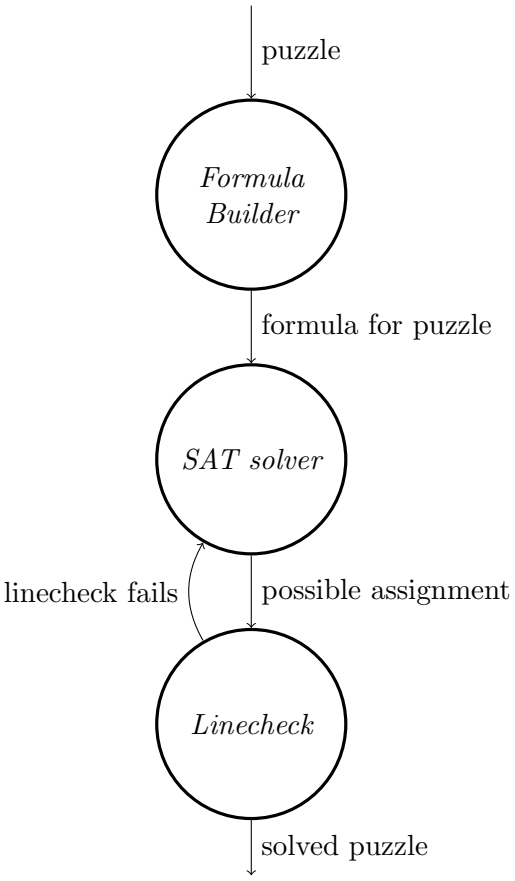


Figure 4.1: Overview of the solving process

4.3 Logical Representation

This section describes how a Yajilin puzzle can be transformed into a boolean formula.
To get such a formula which represents the whole puzzle, all the constraints and rules of the puzzle have to be encoded. Some of these rules are easy to convert

whereas others are more complex. As mentioned earlier in Section 2, there are basically four important rules for Yajilin puzzles. The following subsections depict the basic approach to convert these rules into a boolean formula.

Throughout the following description several variable names are used to denote a certain fact:

- $L_{i,j}^{D}$ denotes that the tile at position (i, j) contains a line with direction D, where $D \in \{d, u, r, l\}$.
- $B_{i,j}$ denotes that the cell at position (i, j) is black.

The values (d, u, r, l) of the direction D stand for *down, up, right, left.*
The height H and the width W of a certain puzzle define the range of the position indices i and j. More precisely this means $i \in \{0, ..., H-1\}$ and $j \in \{0, ..., W-1\}$. Note that the first index describes the vertical position and the second index the horizontal position. The position $(0, 0)$ is in the top left corner.

Having these variables for each cell of an empty puzzle, a boolean formula for the solution can be constructed. There is no need for a variable describing an arrow cell, because the arrow cells of a puzzle are given and cannot be changed.

4.3.1 Constraints for an Empty Cell

For an empty cell there are only a few correct states. First of all the cell is not allowed to stay empty. Then there are two correct possibilities:

1. The cell is black and therefore does not contain any line segments.
2. The cell is not black and contains exactly two line segments (two directions). For example a line left and one down, or a line left and one right.

These constraints can be described with the following formula by enumerating all correct states of a cell at position (i, j):

$$\begin{aligned}
&(B_{i,j} \wedge \neg L_{i,j}^{r} \wedge \neg L_{i,j}^{l} \wedge \neg L_{i,j}^{u} \wedge \neg L_{i,j}^{d}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{u} \wedge L_{i,j}^{d} \wedge \neg L_{i,j}^{r} \wedge \neg L_{i,j}^{l}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{u} \wedge L_{i,j}^{r} \wedge \neg L_{i,j}^{d} \wedge \neg L_{i,j}^{l}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{u} \wedge L_{i,j}^{l} \wedge \neg L_{i,j}^{r} \wedge \neg L_{i,j}^{d}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{d} \wedge L_{i,j}^{l} \wedge \neg L_{i,j}^{r} \wedge \neg L_{i,j}^{u}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{d} \wedge L_{i,j}^{r} \wedge \neg L_{i,j}^{u} \wedge \neg L_{i,j}^{l}) \vee \\
&(\neg B_{i,j} \wedge L_{i,j}^{r} \wedge L_{i,j}^{l} \wedge \neg L_{i,j}^{u} \wedge \neg L_{i,j}^{d})
\end{aligned} \tag{4.1}$$

Furthermore a line in a certain direction does also affect neighbour cells. Hence if there is a line which goes up and the cell above is also an existing empty cell, the upper cell must have a line going down. The following formula encodes this

constraint:

$$\begin{aligned}&\bigwedge_{i=1}^{H-1}\bigwedge_{j=0}^{W-1} L^u_{i,j} \leftrightarrow L^d_{i-1,j} \wedge \\ &\bigwedge_{i=0}^{H-1}\bigwedge_{j=1}^{W-1} L^l_{i,j} \leftrightarrow L^r_{i,j-1}\end{aligned} \tag{4.2}$$

If the neighbour cell is an arrow cell, this constraint can be ignored.

For cells at a border it is possible to forbid a line that would implicate a neighbour cell outside the puzzle. As an example cells at the upper border are not allowed to contain a line going upwards. Using the following formula all such invalid lines at the border can be prohibited:

$$\bigwedge_{j=0}^{W-1} \neg L^u_{0,j} \wedge \bigwedge_{j=0}^{W-1} \neg L^d_{H-1,j} \wedge \bigwedge_{i=0}^{H-1} \neg L^l_{i,0} \wedge \bigwedge_{i=0}^{H-1} \neg L^r_{i,W-1} \tag{4.3}$$

4.3.2 No Consecutive Black Cells

This rule is easy to encode as a logical formula:

$$\begin{aligned}&\bigwedge_{i=1}^{H-1}\bigwedge_{j=0}^{W-1} B_{i,j} \rightarrow \neg B_{i-1,j} \wedge \\ &\bigwedge_{i=0}^{H-2}\bigwedge_{j=0}^{W-1} B_{i,j} \rightarrow \neg B_{i+1,j} \wedge \\ &\bigwedge_{i=0}^{H-1}\bigwedge_{j=1}^{W-1} B_{i,j} \rightarrow \neg B_{i,j-1} \wedge \\ &\bigwedge_{i=0}^{H-1}\bigwedge_{j=0}^{W-2} B_{i,j} \rightarrow \neg B_{i,j+1}\end{aligned} \tag{4.4}$$

Formula 4.4 just states that if the cell at position (i, j) is black the neighbour cells are not allowed to be black. If the neighbour cell is an arrow cell, this constraint can also be ignored.

4.3.3 Rules as a Result of Arrow Cells

Arrow cells of a Yajilin puzzle are special cells because they are fixed and it is not possible to change them. Therefore they may not be changed during the solving process, but there are some constraints that have to be added because of an arrow cell found in a puzzle.
An arrow pointing in a certain direction means that in this column or row a

correct amount of black cells has to be placed, but there may be many combinations of how to place these black cells in the given space. The formula therefore has to encode every possible combination as a solution.

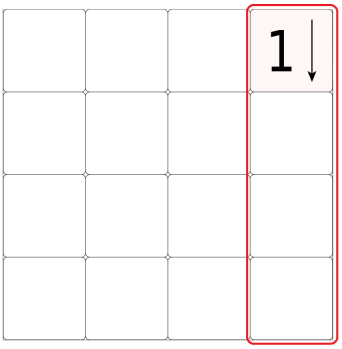


Figure 4.2: Example puzzle

The puzzle shown in Figure 4.2 has an arrow at position $(0, 3)$ with value 1. Therefore the last column has to contain exactly one black cell and it can easily be seen, that there are three correct possibilities to place the black cell.
This results in the following formula:

$$(B_{1,3} \wedge \neg B_{2,3} \wedge \neg B_{3,3}) \vee$$
$$(B_{2,3} \wedge \neg B_{3,3} \wedge \neg B_{1,3}) \vee$$
$$(B_{3,3} \wedge \neg B_{1,3} \wedge \neg B_{2,3})$$

General Computation of the Combinations

To encode the possible placements of black cells a binary format is used. The algorithm works as follows:

1. Count the number of all possible cells for a given arrow. This will be the length n of the binary number B.
2. Encode every possible combination of black cells using a binary number. This binary number is of the shape

   $$B = b_0 b_1 \dots b_{n-1}$$

   Every bit b_i corresponds to a relevant cell where b_0 is the first cell following the arrow in the direction the arrow points. If the bit b_i is 1 then it encodes a black cell otherwise an empty cell. By starting with $B = 0$ (no bit set) and incrementing the number till every bit is set, all combinations can be obtained.
3. Drop every encoding with consecutive ones because adjacent black cells are not allowed.
4. Drop every encoding where the number of ones is not equal to the number of the arrow.
5. The formula for every remaining combination is constructed as follows:
   - Every 0 is encoded as $\neg B_{i,j}$ and every 1 as $B_{i,j}$.
   - All these variables have to be connected with logical `AND`.

6. In a final step all the formulas have to be connected with logical `OR`.

Obviously these steps can be simplified. For example it is possible to do steps three and four during the computation of step two.

The example shown in Figure 4.2 would lead to the following three binary encodings and their corresponding boolean formula:

$$\begin{aligned} 100 &= B_{1,3} \wedge \neg B_{2,3} \wedge \neg B_{3,3} \\ 010 &= \neg B_{1,3} \wedge B_{2,3} \wedge \neg B_{3,3} \\ 001 &= \neg B_{1,3} \wedge \neg B_{2,3} \wedge B_{3,3} \end{aligned}$$

All other combinations of binary numbers are dropped during the algorithm.

4.4 Building the Formula for a Whole Puzzle

By combining all the above formulas for every cell in a Yajilin puzzle, a single large formula can be constructed. This can be done by just connecting all the formulas with a logical `AND`, hence all these constraints must be fulfilled to get a solved puzzle.

But in order to pass the formula to a SAT solver it first has to be in conjunctive normal form. Such a formula consists of conjunctions of disjunctions of literals, where a literal is either an atom a or the negation of an atom, $\neg a$.
A simple example: $(a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$.

In order to get such a formula in CNF for a Yajilin puzzle two approaches may be used. Either the formula is first generated exactly like described above and then transformed to CNF using an algorithm. The other possibility is to create formulas in CNF immediately out of the puzzle. Both approaches have been realized resulting in two different solvers and `FormulaBuilder` components which are described in the following sections.

4.5 Automatic Transformation

This approach of building a formula for a puzzle uses the Tseitin transformation [4] to convert an arbitrary boolean formula into an equisatisfiable formula in conjunctive normal form. This transformation performs well even for large formulas. The SAT4J project offers an implementation of this transformation using logical gates in the `GateTranslator` class.[1] Therefore the formula can be constructed exactly like described in Section 4.3 and will then be transformed automatically.

[1] `http://www.sat4j.org/maven21/org.sat4j.core/apidocs/org/sat4j/tools/GateTranslator.html`

4.6 Immediate CNF Generation

This section is about creating a formula for a Yajilin puzzle that is in conjunctive normal form. To achieve this it is necessary to build CNF formulas which are equivalent to the formulas described earlier in Section 4.3. Even though this construction is based on these formulas, some constraints have to be encoded differently.

The building procedure works on a central formula which at all times is considered to be in CNF. Throughout the process clauses will be added to this formula, where clauses are defined as disjunctions of literals. Listing 4.1 shows in a simplified way how the formula is being built by just iterating over all the cells and adding relevant clauses.

```
for(i=0; i<puzzle.height; i++){
  for(j=0; j<puzzle.width; j++){
    if(puzzle[i,j] == ArrowCell){
      formula.add(clausesRegardingArrow);
    }
    else{
      formula.add(clausesForEmptyCells);
    }
  }
}
```

Listing 4.1: Pseudo code to build a formula

Note that it is enough to distinguish only between cells with an arrow and empty cells, because an empty Yajilin puzzle does not contain black cells or lines. Now it is necessary to add the correct clauses in both cases.

4.6.1 Empty Cells

In simple words these cells have to be either a black cell or a correct line.

First it can be stated that the cells are not allowed to contain line parts and be a black cell the same time. Even if such things seem natural it has to be encoded in the formula in order for the solving process to work correctly. This is achieved with Formula 4.5. It is also necessary to exclude the case where the cell remains empty, which is captured by Formula 4.6:

$$(\neg B_{i,j} \vee \neg L^u_{i,j}) \wedge (\neg B_{i,j} \vee \neg L^d_{i,j}) \wedge (\neg B_{i,j} \vee \neg L^r_{i,j}) \wedge (\neg B_{i,j} \vee \neg L^l_{i,j}) \quad (4.5)$$

$$L^u_{i,j} \vee L^d_{i,j} \vee L^r_{i,j} \vee L^l_{i,j} \vee B_{i,j} \quad (4.6)$$

Having these two formulas (4.5 and 4.6) it is assured that the cell at position (i, j) is not empty and does either contain some line or is turned into a black cell.

Correct line

So far the formula for the empty cell does only make sure that there is either some line or the cell is black. But the combinations of lines have to be restricted to all combinations where there are exactly two different directions. This has already been described in Section 4.3.1 presenting Formula 4.1 as a solution. To encode the same constraint with a formula in CNF it is necessary to encode all the wrong possibilities. And those are the ones where there are not exactly two line segments. This results in the following formula for a cell at position (i, j):

$$\begin{aligned}
&(\neg L^u_{i,j} \vee \neg L^d_{i,j} \vee \neg L^r_{i,j} \vee \neg L^l_{i,j}) \wedge \\
&(\neg L^u_{i,j} \vee \neg L^d_{i,j} \vee \neg L^r_{i,j} \vee L^l_{i,j}) \wedge \\
&(\neg L^u_{i,j} \vee \neg L^d_{i,j} \vee L^r_{i,j} \vee \neg L^l_{i,j}) \wedge \\
&(\neg L^u_{i,j} \vee L^d_{i,j} \vee \neg L^r_{i,j} \vee \neg L^l_{i,j}) \wedge \\
&(\neg L^u_{i,j} \vee L^d_{i,j} \vee L^r_{i,j} \vee L^l_{i,j}) \wedge \\
&(L^u_{i,j} \vee \neg L^d_{i,j} \vee \neg L^r_{i,j} \vee \neg L^l_{i,j}) \wedge \\
&(L^u_{i,j} \vee \neg L^d_{i,j} \vee L^r_{i,j} \vee L^l_{i,j}) \wedge \\
&(L^u_{i,j} \vee L^d_{i,j} \vee \neg L^r_{i,j} \vee L^l_{i,j}) \wedge \\
&(L^u_{i,j} \vee L^d_{i,j} \vee L^r_{i,j} \vee \neg L^l_{i,j})
\end{aligned} \tag{4.7}$$

Forced line due to neighbour line

If a cell has a line in a certain direction this automatically means that the affected neighbour must have the corresponding line. This is captured by the already presented Formula 4.2, which can easily be transformed to CNF. For any neighbour which has to be considered, the following clause has to be inserted:

$$\neg L^{Dir}_{i,j} \vee L^{otherDir}_{neighbour}$$

The corresponding other direction ($otherDir$) has to be computed and is just the opposite direction, i.e. left and right, up and down.

Given a cell at position (i, j) where all the four neighbours have to be considered, the following formula will be constructed:

$$\begin{aligned}
&(\neg L^u_{i,j} \vee L^d_{i-1,j}) \wedge \\
&(\neg L^d_{i,j} \vee L^u_{i+1,j}) \wedge \\
&(\neg L^l_{i,j} \vee L^r_{i,j-1}) \wedge \\
&(\neg L^r_{i,j} \vee L^l_{i,j+1})
\end{aligned} \tag{4.8}$$

No consecutive black cells

To construct CNF clauses stating that two neighbour cells are not allowed to be black at the same time is quite easy. It is enough to insert a clause of the following shape for every neighbour: $\neg B_{i,j} \vee \neg B_{neighbour}$.

There will be at most four clauses generated for a single cell. For the cell at position (i, j) the cells at position $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$ have to be considered. If the neighbour cell exists and if it does not contain an arrow, the clause will be added to the formula.

4.6.2 Arrow Cells

When finding an arrow cell it is necessary to encode that in a certain direction there has to be a specific amount of black cells. In simple words this can be achieved by stating all possible combinations of black cells in the column or row. As the formula should be in CNF it will be necessary to compute all the wrong placements of black cells.

The concrete steps that need to be done:

1. **Look for all relevant cells**, that means cells that are in the row/column in the direction of the arrow
2. **Compute combinations** of black cells where the amount is not correct
3. **Encode the combinations** as a clause and add the clause to the formula

The following example should visualize this procedure. As base the puzzle in Figure 4.3 is considered. Assume the arrow in the lower right corner is being found. It is at position $(4, 4)$, the arrow has a count of 2 and it points up.

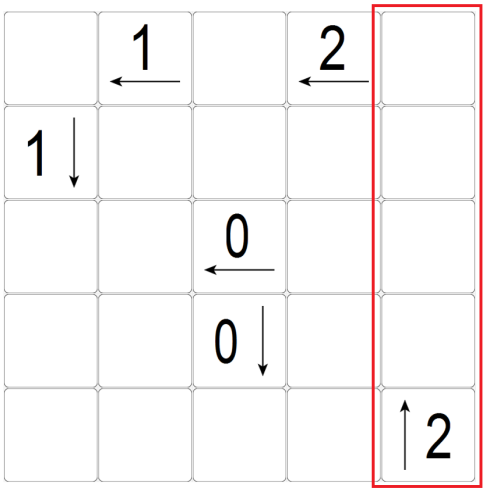


Figure 4.3: Example puzzle

1. The relevant cells are those at positions $(3,4),(2,4),(1,4)$ and $(0,4)$.

2. The wrong combinations now are those with an amount of black cells that is smaller than 2 or greater than 2. Now a very convenient fact comes into play. Since it was already encoded earlier that black cells are not allowed to be adjacent, at this point it is not necessary to consider the combinations where black cells are back-to-back and therefore a lot of combinations can be omitted.
   In this example now the case that every relevant cell is not black and the cases where only one cell is black have to be encoded. In these cases black cells are not consecutive and the amount is smaller than 2. All combinations with more than two black cells can be omitted because there are consecutive black cells. The combinations are computed similarly as described in Section 4.3.3. The difference is just that now encodings with the wrong amount of bits set are used.

3. The combinations are easily written as clauses in conjunctive normal form. For the example the formula will be enhanced by the following part:

$$\begin{aligned}&(B_{3,4} \vee B_{2,4} \vee B_{1,4} \vee B_{0,4}) \wedge \\ &(B_{3,4} \vee B_{2,4} \vee B_{1,4} \vee \neg B_{0,4}) \wedge \\ &(B_{3,4} \vee B_{2,4} \vee \neg B_{1,4} \vee B_{0,4}) \wedge \\ &(B_{3,4} \vee \neg B_{2,4} \vee B_{1,4} \vee B_{0,4}) \wedge \\ &(\neg B_{3,4} \vee B_{2,4} \vee B_{1,4} \vee B_{0,4})\end{aligned} \tag{4.9}$$

Arrows with a count of zero

The case that an arrow with the number zero is encountered can be handled separately to get a shorter formula. The number zero means simply that every cell in this direction is not allowed to be black. Therefore all the combinations computed as described above are wrong combinations and so a lot of clauses will be added. The solution is really simple: It is enough to just add a clause with one literal for every cell that is affected by the arrow stating that this cell is not allowed to be black ($\neg B_{i,j}$).

So if looking again at the puzzle from Figure 4.3 an arrow at position $(2,2)$ pointing to the left is present. The relevant cells affected by this arrow cell are at position $(2,1)$ and $(2,0)$. This means the clauses $\neg B_{2,1}$ and $\neg B_{2,0}$ will be added to the formula and no other steps are required.

4.7 The Solver

The solver is the central component to handle the solving process of a Yajilin puzzle. The main task of the solver is to start an actual SAT solver provided by SAT4J[3]. First the solver has to get the formula for a puzzle by using a

`FormulaBuilder` and then the SAT solver is able to search for solutions.

Throughout the thesis two different solvers have been realized.
The `ImmediateCnf` solver is based on the construction of the formula as described in Section 4.6 whereas the `AutoTransform` solver uses the Tseitin algorithm to transform the formula (see Section 4.5).

4.7.1 Formula Representation for the Solver

When using the component of SAT4J to transform the formula to conjunctive normal form it is not necessary to worry about the representation of the formula. This is done implicitly by SAT4J.

When constructing the CNF formula manually, the clauses have to be transformed into a format which SAT4J can handle. A clause that can be added to the SAT solver can be represented as a sequence of integers. Each integer denotes a specific literal, if it is negative then it denotes the negation of the literal. This corresponds to the notation for clauses as described by the DIMACS format[1].

As an example the clause $(1 \wedge \neg 2 \wedge 3 \wedge \neg 4)$ is represented as a sequence containing the elements 1, -2, 3 and -4.
All clauses produced during the formula construction are passed in such a format directly to the SAT solver.

4.7.2 Functionality

The actual solver tries to solve a puzzle by finding correct variable assignments for the formula.

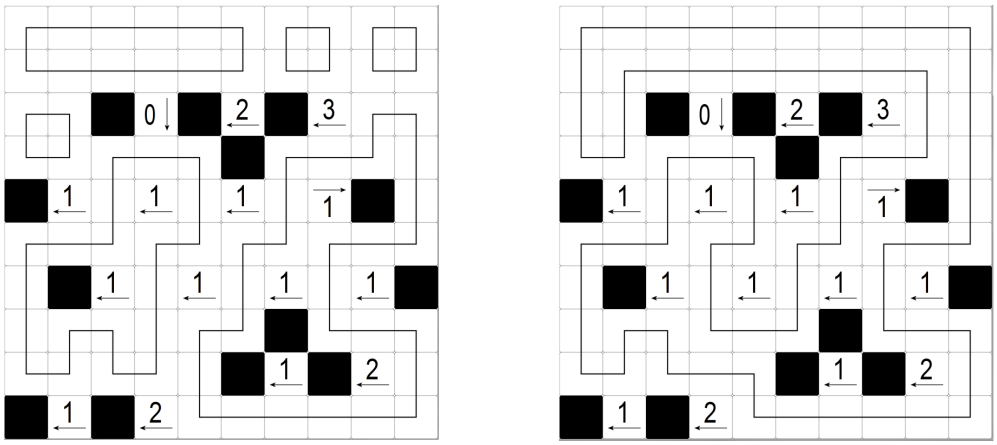
Based on the formula SAT4J constructs a model for the problem. The SAT solver is now able to check if there are solutions for the problem and computes a variable assignment such that the formula is evaluated to `true` (correct model). After getting this model it is necessary to build a solved puzzle out of it.

Building the Puzzle

Every literal in the formula corresponds to a certain state of a specific cell in the puzzle, therefore it is quite easy to construct the puzzle. It is just important to save the mapping of a certain literal to the corresponding number of the literal within the formula representation. This mapping is then used to convert the number back to a literal and then to a state of a cell.
At this point a puzzle has been constructed, where the constraints encoded within the formula are all fulfilled. But there is one major constraint that was not encoded into the boolean formula, which is the demand of a single line creating a loop.

Single Loop Problem

Because the single loop constraint is not encoded in the formula, the solver may find solutions that are not correct regarding to the rules of Yajilin. Such a solution looks like the one shown in Figure 4.4(a). The puzzle has been taken from the Nikoli website (Sample puzzle 1).[2]



(a) Too many loops (b) A single loop

Figure 4.4: A wrong and a correct solution

Now there has to be a routine that checks the line within the puzzle. This routine just walks along the first line part it finds, marking every cell it visits. If the start cell is reached again, every line cell of the puzzle has to be marked. After this process the routine should not be able to find a line cell which was not marked. The whole process can be seen in Figure 4.5.

This means that after building the puzzle out of the solution model this check is performed and if it succeeds the model is really the valid solution and the solving procedure can stop now. The correct solution for the example puzzle is shown in Figure 4.4(b). If the check fails, the SAT solver goes on and tries to find the next model. This application flow has already been depicted in Figure 4.1.

4.8 Performance

In the previous sections it has been described how to solve a Yajilin puzzle using a SAT solver. By building the formula for the solver exactly like explained, it

[2] `http://www.nikoli.com/en/puzzles/yajilin/`

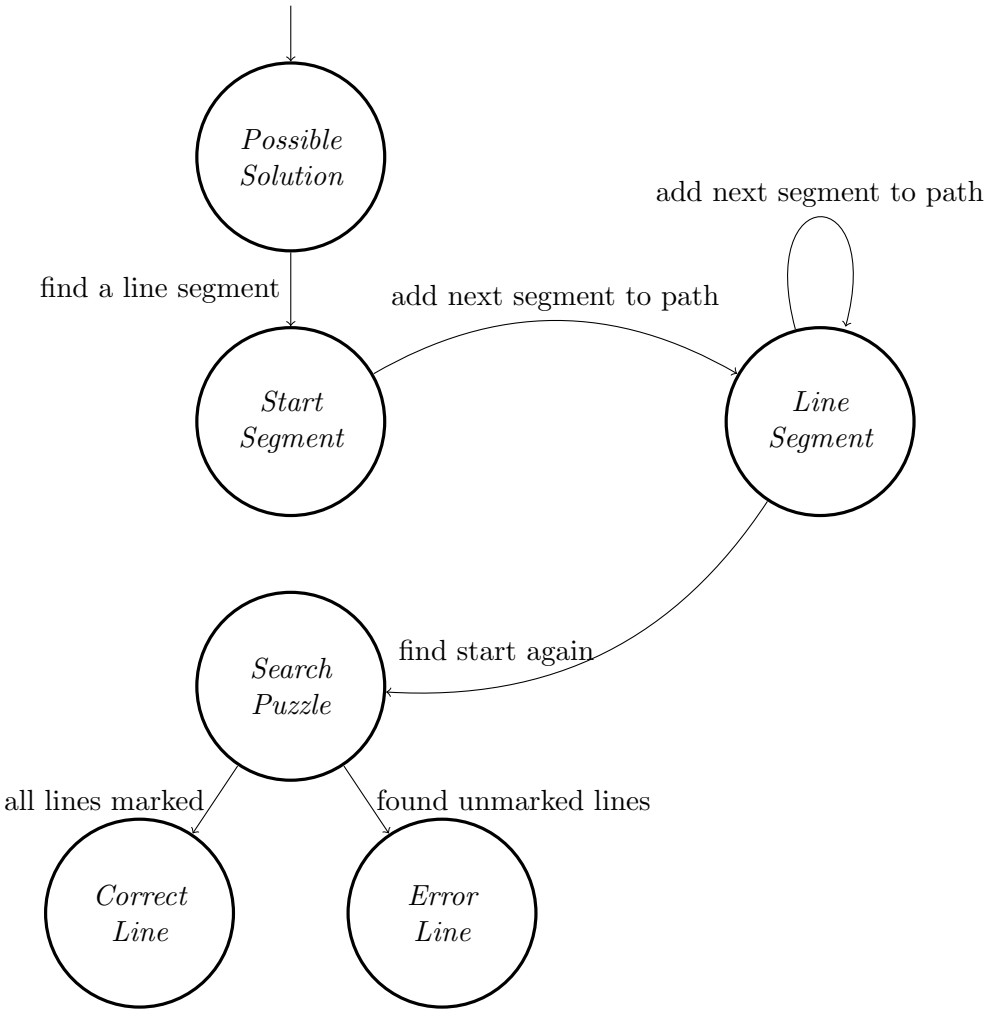


Figure 4.5: Linecheck algorithm

will result in a correct formula. But for bigger puzzles the performance is not good. As it turned out for the SAT solver to find the solution really fast, it is crucial that the formula generated by the builder is good. Good does not only mean short and not a lot of literals but also other things, like the ordering of the clauses can have an effect. The following sections describe what methods can be used to improve the performance of the solving process.

4.8.1 Borders and Arrow Cells as Neighbour

When computing the formula for an empty cell, the goal is to encode that the cell is black or that there is a correct line. For many cells some cases can be ignored right away, resulting in only a few allowed directions for the line. The

allowed directions can be computed by just looking up all the four neighbour cells, and only if this cell is existing and is an empty cell a line to this neighbour is valid. Let's illustrate this on the example puzzle shown in Figure 4.6.

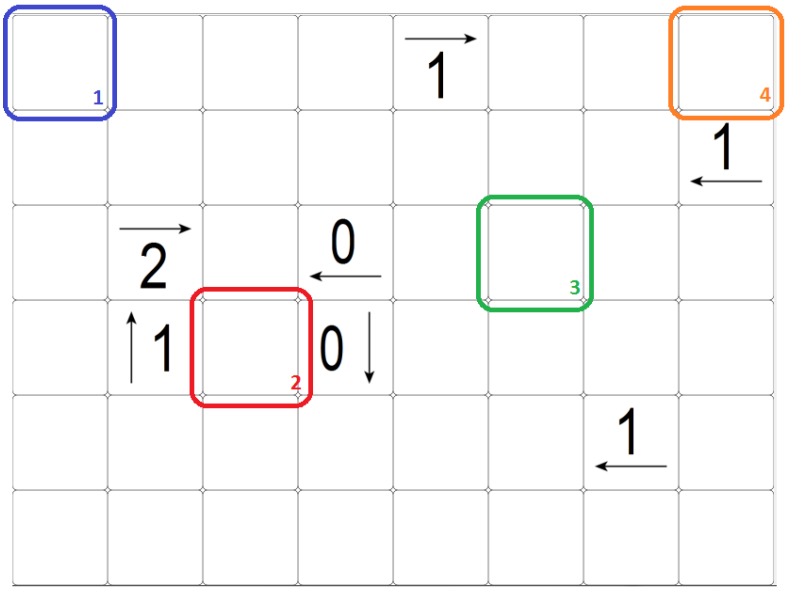


Figure 4.6: Example puzzle

For the marked cells the following can be learnt and considered within the formula:

1. The cell in the left upper corner has only two allowed directions, which are right and down. The directions left and up are not allowed because there is no existing empty cell at these positions. In this case where exactly two directions are allowed Formula 4.1 can be shortened resulting in the following formula for this cell:

$$\begin{aligned}&(B_{0,0} \wedge \neg L^r_{0,0} \wedge \neg L^d_{0,0} \wedge \neg L^l_{0,0} \wedge \neg L^u_{0,0}) \vee \\ &(\neg B_{0,0} \wedge L^r_{0,0} \wedge L^d_{0,0} \wedge \neg L^l_{0,0} \wedge \neg L^u_{0,0})\end{aligned} \tag{4.10}$$

2. For this second case the cell has arrow cells as right and left neighbours. Therefore there are exactly two directions possible which results in a similar reduction of the formula as in the previous case.

3. The third case shows the cell at position $(2, 5)$. Here all the neighbours are existing empty cells and all the possibilities have to be considered, see Formula 4.1.

4. The cell in the upper right corner now has only one possible direction

for a line, which is to the left. This means that there cannot be a valid line and the cell has to be black. Therefore all other possibilities can be omitted and only the following formula is needed:

$$B_{0,7} \wedge \neg L^r_{0,7} \wedge \neg L^d_{0,7} \wedge \neg L^l_{0,7} \wedge \neg L^u_{0,7} \tag{4.11}$$

It can be seen that by looking at the neighbours it is possible to shorten the formula and for big puzzles this can have a quite big impact on the solving performance.

4.8.2 Arrow Cells: Lookahead

An arrow cell means that the combinations of black cells in a row/column with the correct count have to be encoded in the formula, as was explained in Section 4.3.3.
For large puzzles and especially in cases where there are lots of cells to consider in the specified direction the result is a huge amount of possible combinations. But in many cases it is possible to reduce this amount due to the existence of other arrow cells pointing in the same direction within the same row/column. Based on the following examples some cases can be pointed out:

Figure 4.7: Arrows with same direction and number

In this first case two arrows having the same number and direction within the same row are encountered. This means that all the computation of possible combinations can be omitted for the left arrow because it would result in the exact same combinations. All the cells between the two arrows are not allowed to be black so this can be encoded directly when finding such a case. Furthermore if the arrows would be right one after another nothing at all has to be done for the left arrow.

Figure 4.8: Arrows with same direction but different numbers

Figure 4.8 shows an example where two arrows within the same row have the same direction but not the same count. This means that for the arrow with the higher count it is enough to look only at the cells between the arrows and the number can be decreased by the number of the other arrow. In the example

this means that the right arrow with number 4 can be handled like an arrow with number 3 where only combinations of cells between the two arrows are considered.

Figure 4.9: Arrows with same direction and only one cell in between

Figure 4.9 now shows a special case of the above case. Here only one cell is between the arrows and the left arrow has exactly the number of the right arrow decreased by one. This means that the cell in between has to be black, which can be encoded directly.

Hence for many of the arrows encountered in the puzzles such cases can be found the formula for the whole puzzle becomes a lot more compact and the performance of the solving process increases notably.

4.8.3 Avoiding Loops

As mentioned earlier the constraint of the single loop is not encoded in the boolean formula for the puzzle. This means that the line is checked by another procedure after the SAT solver has found a possible model. For large puzzles now the solver may find many solutions consisting of all possible combinations of small loops all over the puzzle.

To avoid the computation of thousands of possible solutions it is possible to give the solver clauses which prevent loops. After finding the first solution with more loops these loops can be encoded as a boolean formula. The negation of this formula is then added to the solver as it tries to find the next solution and the solver will not find solutions with these loops again. The resulting performance improvement is huge, especially for larger puzzles.

Figure 4.10 illustrates a quite large puzzle and a possible solution with more loops the solver finds first. This puzzle has a lot of empty space which makes many small loops possible. The puzzle has been taken from the Nikoli website (Sample problem 4).[3]

Now all the eleven loops will be encoded as a formula. For example the small loop in the lower right corner spanning over cells at position $(9, 16)$, $(9, 17)$,

[3] `http://www.nikoli.com/en/puzzles/yajilin/`

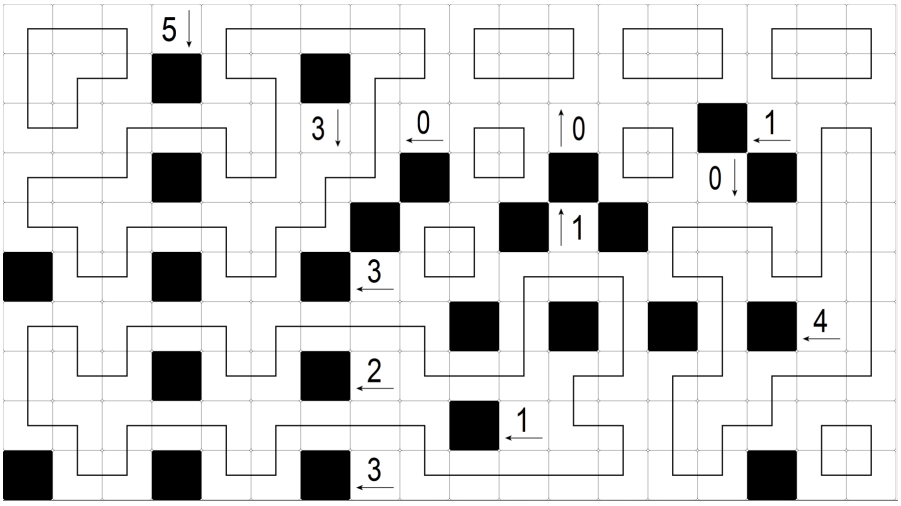


Figure 4.10: A solution with more loops

$(8, 16)$ and $(8, 17)$ is encoded as follows:

$$L^u_{9,16} \wedge L^r_{9,16} \wedge L^l_{9,17} \wedge L^u_{9,17} \wedge L^d_{8,17} \wedge L^l_{8,17} \wedge L^r_{8,16} \wedge L^d_{8,16} \tag{4.12}$$

The negation of the above formula now tells the solver that this loop is not allowed for further models. Comparing the time needed to solve the puzzle shows that it really speeds up the solving process. If the loops are encoded and given to the solver the correct solution is found after 11 models and it takes 0.2 seconds. Otherwise the solver finds 205920 models before it encounters the correct one which results in a runtime of 203 seconds. Taking other puzzles with less loop possibilities the difference will not be that big as in this example but it is really crucial for the solver performance in general.

How the solver performs in practice on many different puzzles will be further discussed in Section 6.1.

5 Generator

The topic of this chapter is the generation of new Yajilin puzzles. Generation of correct puzzles is an expensive tasks, because it involves a lot of randomness and it can be time-consuming to determine if a puzzle has only one correct solution. The general approach is to first create puzzles based on some random algorithms and then test the puzzles using the solver. The solver has to determine if the puzzle is solvable and if it has exactly one solution. The latter fact can be assured by letting the SAT solver model all possible solutions. If the SAT solver finishes and has found exactly one solution with a single loop a valid Yajilin puzzle has been found.

Two generators based on different approaches have been developed throughout this work. The *Line Generator* tries to construct a line first and then fills the rest of the puzzle with arrows. The *Arrow Generator* randomly puts arrow cells into an empty puzzle. The following sections describe the generation process behind both generators in detail.

5.1 Line Generator

The line generator tries to generate a new puzzle based on a random line. To accomplish that, at first a random line is generated. If the creation of this line succeeds, the rest of the grid is filled with black cells and arrows. The advantage of this generation is that a puzzle generated this way is always solvable. As a drawback it can take very long to create bigger puzzles with many line tiles. The reason is that the creation of the line can easily fail if a situation occurs where the start and the end cannot be connected any more.

5.1.1 First Attempts

The first ideas were to directly construct the line instead of the cells which contain the line. One of these attempts was to make, with a 50% probability, either a curve or a straight line. This attempt did not work out well because the line quickly ran into problems. These occurred mostly because the line split the puzzle in two halves and made it impossible to connect start and end.

The second idea was to draw the lines randomly. In this case the line had less problems with splitting the field but with surrounding itself with other lines. Because it is not possible to get rid of these errors without adding an enormous

overhead (lookahead or backtracking), the final version ignores these problems and tries to be quick. Another problem is that the overhead used in this case is much more expensive than just removing the puzzle and try another one. With backtracking it is not guaranteed that the puzzle does not have more solutions and also it is not possible to ensure that the problem is much bigger and cannot be solved by just backtracking a few steps. In such a case it is often necessary to go back to the beginning which means time is lost with that action where in the meantime a new puzzle could already be created. Some ideas which should make the creation better and how they worked out can be found in Section 6.2.1.

The final version of the algorithm creates a random path. But it does not use the line as a base but the tiles. That means only the x and y coordinates are changed and the construction algorithm does not know anything about directions.

5.1.2 The Algorithm

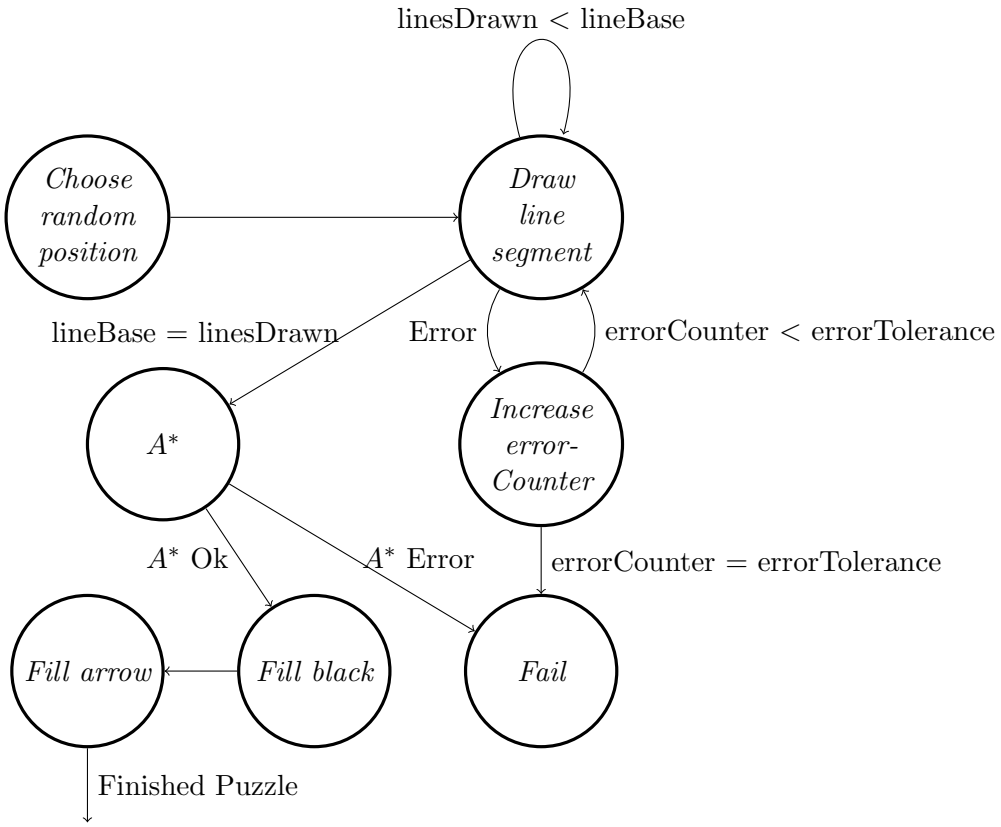


Figure 5.1: The linedraw algorithm schema

Figure 5.1 shows the algorithm based on line drawing. The basic idea is to draw a random line of a given length and afterwards connect the ends of this line with the A^* algorithm. After creating the line the puzzle has to be filled with black cells and arrows. During the first step (the drawing of the line) many errors can occur whilst during the second part where the puzzle is filled with black cells and arrows no error can occur.

There are a few variables in this algorithm:

`lineBase` The variable `lineBase` describes the number of line tiles which should be drawn before the A^* algorithm starts. This number is fixed at the start of the algorithm and cannot be changed later.

`linesDrawn` For every placed line tile `linesDrawn` is incremented. If during the placement of a tile an error occurs it is decremented. If `linesDrawn` is equal to `lineBase` the random part is finished and the A^* algorithm can start.

`errorTolerance` This number has to be fixed at the start of the algorithm. It defines the number of errors which are allowed to occur while creating the random line before the algorithm fails.

`errorCounter` Whenever an error occurs when placing a line the `errorCounter` is being incremented. The algorithm terminates and returns `false` if `errorCounter` reaches the same value as `errorTolerance`. This means that no correct puzzle could be created.

The algorithm starts by choosing a random position in the puzzle which acts as the starting point for the line generation. As a base it does not use the line itself but the cells which afterwards contain the line. With this abstraction the algorithm itself looses the overhead of correctly connecting line segments. Furthermore the line creation does not loose any possible paths because the four directions the line could go are also possible with this approach. The difference can be seen in Figure 5.2. In this example black cells denote marked cells for the line generation.

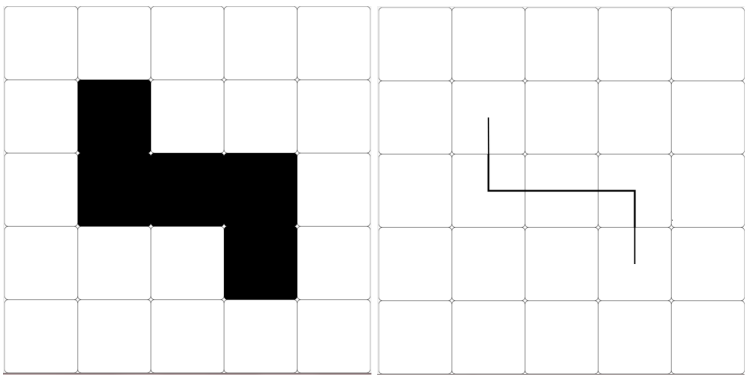


Figure 5.2: Difference between using only cells and real lines

After choosing the first cell, the algorithm computes a set of possible neighbours which could contain the next line segment. This set consists of all cells next to the current cell which are not already in the current path. If this set is not empty, a cell is being chosen and added to the path, marked as the current cell and `linesDrawn` will be incremented. Otherwise, if the set is empty, an error occurs. In this case `errorCounter` is incremented, `linesDrawn` is decremented and the last placed cell is deleted in case there is another possible path. In many cases there does not exist such a path and to prevent the algorithm from trying infinitely often, `errorTolerance` defines the maximum number of tries before aborting. If `linesDrawn` equals `lineBase` then this part is finished and the current path is handed over to the A^* algorithm.

The A^* algorithm is a basic pathfinding algorithm which finds the shortest path between two points in a graph. In this case the nodes of the graph are the cells which have not been selected as line cells and the edges are transitions to the neighbour cells.
The current line has now two loose ends, one where the generation started and one where it stopped. The algorithm's job is now to connect these two ends. In some cases this is impossible due to the fact that the randomly generated line may have divided the puzzle into two halves. In this case the whole generation fails and `false` is returned. If the A^* algorithm is able to connect the two ends then the line is finally finished and now the black cells can be filled in. This happens according to the simple algorithm shown in Listing 5.1. For every empty cell it is checked if there is a neighbour which is already black, and if not, a black cell is placed.

```
for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
    if(no neighbour is black){
      puzzle.setTileAt(i,j,new BlackTile());
    }
  }
}
```

Listing 5.1: Black Cell Algorithm

The last part of the line generation places an arrow on every free field with equal probabilities for every direction. The arrow is able to pick the correct amount because the black cells have been placed already in the step before. In the rare case that an arrow is placed at the border pointing outwards the arrow is simply flipped. When all arrows are placed the created puzzle is finished and can now be tested by the solver whether it has a unique solution.

How the line generator performs in practice will be described in Section 6.2.1.

5.2 Arrow Generator

The second generator developed in this project is the arrow generator. The basic idea of this generator is to take an empty puzzle and fill in some arrows randomly and then test the puzzle using the solver. Two techniques to produce puzzles containing some arrows have been realized, and will be described in the following two sections.

5.2.1 Create Whole Puzzle and Test

This is a really simple and primitive way to create a puzzle. The generator is just iterating over all the cells of a puzzle and with a given probability a cell is chosen. Now it is necessary to estimate which directions and numbers for an arrow are allowed and pick them randomly. If the iteration over all cells is finished then the whole puzzle is passed to the solver. The solver is now used to determine if there is only one correct solution and if so then the process is finished, otherwise it starts from the beginning.

5.2.2 Test After Each Arrow

Unlike the first approach this one tests the puzzle using the solver after each new arrow that is inserted. The algorithm works as follows:

Step 1 Select a random empty cell.

Step 2 Estimate the possible arrow directions for the cell and pick one.

Step 3 Estimate the minimum and maximum allowed number for the arrow and pick a random allowed number.

Step 4 Place the arrow and try to solve the puzzle. Now three cases can be distinguished:

- The puzzle is satisfiable and has **only one solution**. The generator was successful and stops.
- The puzzle is **solvable but ambiguous**. Continue with step 1.
- The puzzle is **not solvable**. Continue with with step 5.

Step 5 Delete the previously inserted arrow and continue with step 1.

Furthermore a step counter is incremented after each selection of a random arrow. If this counter reaches a certain limit all arrows will be deleted and the algorithm starts off with a new empty puzzle. This is necessary because otherwise puzzles with mostly arrow cells would be produced.
As can be seen, there is a lot of randomness involved, which means that the time needed to generate a puzzle can vary a lot and also the quality of the

puzzles is of course not always the same. But especially the quality factor for a Yajilin puzzle is really hard to measure.

Comparing the two ways of creating puzzles has shown that the first one is suitable only for small puzzles. Up to a size like 8×8 it works quite good, but then the performance is really bad and therefore not suitable for practical use.

The generator which tests after each insertion performs much better and offers also some parameters which may be changed to affect the generated puzzles. Details about the realization are presented in the following sections.

5.2.3 Choosing Direction and Number

It is crucial that the inserted arrow cells have a valid direction and number. Choosing a direction is trivial. It is only necessary to check if a neighbour cell in a certain direction is existing. If not then the cell is at a border and an arrow pointing out the puzzle is not desirable, even though it would be allowed with number zero.
To choose a number for the arrow it is necessary to compute the maximum and minimum number that is valid for this row or column and chosen direction. These numbers are defined by the position of the cell and other arrows in the same row or column and their numbers. Therefore it is required to look at the whole row or column to compute them.

To visualize this let's look at the following example:

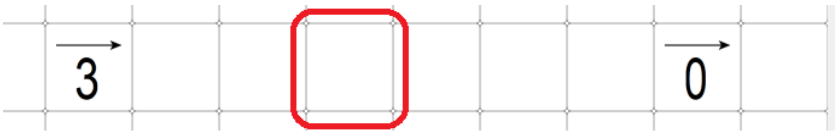


Figure 5.3: Trying to insert an arrow pointing right

The goal is to insert an arrow pointing right into the marked cell. Now there are two other arrows within the same row that affect the possible number of the arrow. The rightmost arrow defines the minimum number. In this case it is zero and therefore not relevant, hence zero is the default minimum.
It is now important to look at the space between the rightmost arrow and the spot where the new arrow should be. The number of empty cells defines how many black cells there may be. Having n empty cells means that there are by the definition of the rules at most $\lceil \frac{n}{2} \rceil$ black cells.
Applying this to the example it follows that there can only be two black cells between the new and the rightmost arrow and therefore the maximum has to be set to 2. Furthermore it is also necessary to check if this conforms with the leftmost arrow. The leftmost arrow limits the maximum number to 3 and the minimum number is affected by the space between the new and the leftmost

arrow. In this case there is at most 1 black cell which defines the minimum as $3 - 1 = 2$.
The conclusion for the above example is that the maximum and minimum number are equal, hence there is no further choice to make.

The following example shows a similar scenario, where the rightmost arrow has now a count of 1 instead of 0. This changes the maximum number to 3 because now that the rightmost arrow has a count of 1 it is enough to have 2 black cells between them. The minimum number is again 2, defined like described for the previous example.
For this scenario it now is possible to randomly choose if the new arrow will have a count of 2 or 3.

Figure 5.4: Trying to insert an arrow pointing right scenario 2

Figure 5.5 shows another similar example but with a clash. Here it is not possible to insert any arrow at all in the marked spot, because it has to be black in order to have a result with 3 black cells between the other two arrows. Such a scenario means that this spot will be discarded and another cell to insert an arrow will be chosen.

Figure 5.5: Trying to insert an arrow pointing right with clash

5.2.4 Parameters for the Arrow Generator

To control and change the behaviour of the generator, some parameters can be adjusted. Some of them are also accessible through the user interface of the program. The following three parameters are given as floating point numbers. The concrete value is then computed by multiplying with the total number of cells of the puzzle. For example having a minimum arrows parameter of 0.02 and a puzzle with 100 cells results in a concrete minimum of $100 \cdot 0.02 = 2$ arrows.

Minimum Arrows Defines how many arrows the puzzle should have at least. In general the number should be really low or even left at zero, because

valid puzzles with a low number of arrows are usually interesting to play and hard to generate. Convenient values are between 0 and 0.1.

Maximum Arrows Defines a desired maximum of arrows. Note that the concrete implementation allows more arrows, but if the maximum is already reached and the puzzle is not a correct one, the previously inserted tile will be removed with a high probability. In order to be more flexible it may also be allowed and then the number will be exceeded and therefore the value is more like a guidance value. By choosing a low number of maximum arrows it will be much harder for the generator to find a valid puzzle and the resulting puzzle may look more natural and be more interesting to play. Good values are between 0.15 and 0.20.

Steps to try Defines how many steps the generator works on the same puzzle, as described in Section 5.2.2. By trying really long on the same puzzle the generator may run into scenarios where hardly any move is correct or the puzzle may become less interesting because of the high amount of arrow cells. In practice good values are between 1.0 and 3.0.

6 Performance Analysis and Benchmarks

The previous chapters explained the realization of solvers and generators for Yajilin puzzles. Within this chapter the performance of the various components will be analysed. The different solvers and generators will also be compared to each other.

The computer used for the benchmarks has a 3.2 GHz quad-core CPU with 8 GB RAM and runs Windows 7 Professional 64 bit as operating system.

6.1 Solver Benchmarks

Section 4.8 already showed some effective methods to improve the performance of the Yajilin solver. Now some concrete benchmark results will be presented. The measurements are based on 50 different Yajilin puzzles with sizes ranging from 3×3 up to 31×45. A list of the sources for these puzzles can be found in Chapter 7.

As already explained in Section 4.7 two different solvers, the `ImmediateCnf` solver and the `AutoTransform` solver have been implemented. The solvers were tested such that every puzzle has to be solved 100 times and the average of all the solving times will be computed.
The results show a quite large difference between the two solvers:

ImmediateCnf 485 ms average

AutoTransform 1323 ms average

The difference arises mostly from a few huge and hard to solve puzzles. This can be seen by looking at solving times for single puzzles. Table 6.1 shows some benchmark results of both solvers for a single puzzle.

The results presented in the table show that the solver using the automatic transformation is always slower, even though the difference is practically not really noticeable for the first five results. For big and hard to solve puzzles the difference can be quite large. Puzzles like the one taken for measurement #6 need a huge formula for solving and therefore the `AutoTransform` solver has to do lots of time consuming transformations.

#	Size	Description	ImmediateCnf	AutoTransform
1	6 × 6	Own Creation	32 ms	62 ms
2	10 × 10	Nikoli No1	94 ms	172 ms
3	14 × 18	Generated	234 ms	312 ms
4	14 × 24	Nikoli No7	470 ms	500 ms
5	20 × 36	Nikoli No10	530 ms	671 ms
6	20 × 36	Janko No19	17470 ms	49810 ms
7	31 × 45	MathgrantBlog	1972 ms	2432 ms
8	31 × 45	Nikoli Championship	2863 ms	13455 ms

Table 6.1: Solver Performance Comparison

Another drawback of the `AutoTransform` solver is that the memory consumption is higher. The reason once again is the transformations in the case of huge formulas. Concretely the `ImmediateCnf` solver is able to solve all the puzzles tested so far having only 256 megabytes of memory available. The `AutoTransform` solver on the other hand needs approximately 1024 megabytes to do so. This has been tested by using the `-Xmx` parameter for the Java VM which controls how much memory the virtual machine may use.

It is generally notable that the time needed to solve does not depend only on the size of the puzzle but also on the placement of the arrows.
Like Table 6.1 shows, a puzzle with size 31 × 45 might be solved faster than one with size 20 × 36. The reason can be arrows for which an immense number of combinations of black cell placements have to be considered. This is the case if for example the arrow is at the beginning of a row or column and within this row or column many cells have to be considered for the black cell placement. Figure 6.1 visualizes such a scenario:

Figure 6.1: A row with an arrow and lots of cells to consider

For the arrow pointing right now all correct placement of black cells within the row have to be considered. Note that this concrete size with 14 cells to consider will not be a time consuming problem for the actual solvers. But if the number of relevant cells is bigger, e.g. 30, there are over a billion possible combinations for the black cells. Out of these combinations the ones with the correct number of black cells and no adjacent black cells have to be filtered.

6.1.1 Scaling

The performance measurements so far have been done on a quite fast 3.2 GHz processor with 4 cores. The following table shows how the solver performs using different processor core speeds:

Puzzle	Size	1600 Mhz	2400 Mhz	3200 Mhz
Nikoli No9	10 × 10	856 ms	650 ms	470 ms
Nikoli Champ.	31 × 45	5250 ms	3730 ms	2863 ms
Janko No19	20 × 36	33070 ms	22540 ms	17470 ms

Table 6.2: Solver Scaling

For the test the same processor has been used but the clock has been adjusted. As solver the `ImmediateCnf` solver has been used.
The results show that the solving process scales nicely with the processor speed.

6.2 Generator Benchmarks

Within Chapter 5 the generation of Yajilin puzzles has been discussed and two concrete generators have been described. It has already been pointed out that generation is a quite hard task. This section now presents concrete results of the generators. Furthermore the two generators will be compared to each other.

6.2.1 Line Generator Performance

Within this section the performance of the line generator presented in Section 5.1 will be analysed by performing some benchmarks.

To measure the speed a simple benchmark is created which performs the algorithm a fixed number of times. After finishing the average time taken is computed.

Puzzles created in this benchmark were divided into three different dimensions, with different values for the `lineBase` parameter. By increasing the value the huge impact of this parameter on the performance of the algorithm can be seen. In Table 6.3 the results are presented.

Approaches to Generate Puzzles Quicker

To make the algorithm perform better some tricks can be used which are described within this section.

Size	`lineBase`	# runs	Average time
6 × 6	0.60	500	34 ms
6 × 6	0.65	500	92 ms
6 × 6	0.70	500	343 ms
8 × 8	0.60	500	902 ms
8 × 8	0.65	500	3584 ms
8 × 8	0.70	50	60517 ms
10 × 10	0.60	50	24455 ms

Table 6.3: Benchmark results for line generation

One idea is to add backtracking. This attempt should make the generation of the initial line quicker and more error resistant. Backtracking will not guarantee that the A^* algorithm succeeds. As mentioned before backtracking is only useful if only a few steps are tracked back. Our version only backtracks one step if an error occurs and tries other directions. To prevent making the same error again a set with past directions is kept and as long as the same direction occurs a counter is incremented. With every step in the same direction the probability to make such a step again becomes smaller. Table 6.4 shows the results of a benchmark comparing the results with simple backtracking and without.

Size (`lineBase`)	# runs	without backtracking	with backtracking
6 × 6 (0.65)	500	67 ms	97 ms
8 × 8 (0.65)	500	2335 ms	2931 ms

Table 6.4: Benchmark of backtracking

As can be seen backtracking does not help and makes the creation of new puzzles slower rather then faster. Therefore we chose the default setting to be `backTracking = false`.

Another idea to generate puzzles quicker was to run the A^* algorithm more times if an error occurs. Every run one end of the path is cut off and A^* tries to connect them again. If the cutting is performed too often then the puzzle will not be very interesting because it will mostly consist of black cells and arrows. To prevent the generator from cutting too many lines a variable keeps track of the cuttings and if the value becomes too high the generation is stopped and rejected. Some results of benchmarks are presented in Table 6.5.

Size (`lineBase`)	# runs	rerun A^*	run A^* once
6 × 6 (0.65)	500	28 ms	33 ms
8 × 8 (0.65)	500	800 ms	2179 ms

Table 6.5: Benchmark of rerunning A^* (max. 3 times)

As can be observed this strategy makes the algorithm faster, even if only three ends are cut. But chances are very high that the generated puzzle will not look very good and mostly consists of arrows and black cells. This happens even with only three reruns of the A^* algorithm.
The user interface has the option to enable such a generation but as a default it is deactivated. When setting the retry number higher the creation of bigger puzzles becomes very quick but they will look more like a chess-field than a Yajilin puzzle.

6.2.2 Arrow Generator Performance

To show how the arrow generator performs in practice some benchmarks are presented in this section. The arrow generator can be configured using the parameters described above and therefore it is possible to test with lots of different configurations.

For the first measurement the standard parameters are 0 for minimal arrows, 0.18 for maximum arrows and 1.5 for the steps to try have been used. Using these values usually results in interesting puzzles.
The results are shown in Table 6.6.

Size	**# Puzzles**	**Average time**
6×6	500	35 ms
8×8	500	198 ms
9×9	500	473 ms
10×10	500	843 ms
10×12	100	1781 ms
12×12	100	5120 ms
14×14	100	29725 ms

Table 6.6: Generating puzzles with the arrow generator

It can be seen that the generator is quite fast for medium sized puzzles with a size like 10×10, where it needs around one second. For larger puzzles the time to generate increases really fast and so it is already necessary to wait about 30 seconds for a 14×14 puzzle.

Changing the parameters changes the look of the generated puzzles and also affects the time needed. How different changes of the parameters affect the performance can be seen in Table 6.7.

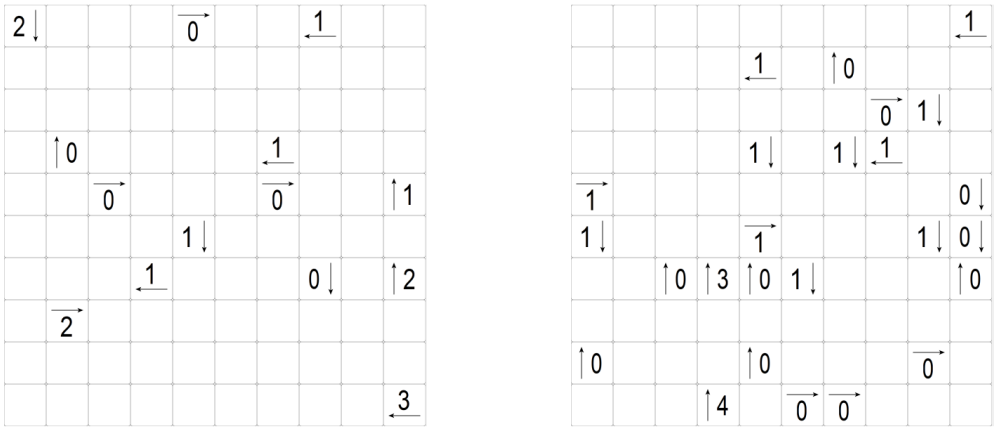
The results show that the minimum arrow parameter has hardly any effect on the performance. It will just be avoided that puzzles with very few arrows are generated, but they are hard to generate.

Different is the situation for the maximum number of arrows parameter. It can

Size	Min	Max	Steps	# Puzzles	Average time
10 × 10	**0.1**	0.18	1.5	500	879 ms
10 × 10	0.0	**0.12**	1.5	500	5950 ms
10 × 10	0.0	**0.15**	1.5	500	1764 ms
10 × 10	0.0	**0.18**	1.5	500	843 ms
10 × 10	0.0	**0.20**	1.5	500	690 ms
10 × 10	0.0	**0.25**	1.5	500	550 ms
10 × 10	0.0	0.18	**3.5**	500	932 ms
10 × 10	0.0	0.18	**2.5**	500	792 ms
10 × 10	0.0	0.18	**1.0**	500	938 ms
10 × 10	0.0	0.18	**0.6**	500	1620 ms

Table 6.7: Using different parameters for the arrow generator

be seen that changing it affects the running time a lot. The explanation is that if many arrows are allowed, the generator's work becomes much easier. But the puzzles generated that way are usually not really interesting nor challenging because the course of the line and the black cells are too obvious. So this parameter should really not be too high. The difference between a typical puzzle generated with factor 0.12 and factor 0.25 is depicted in Figure 6.2.



(a) Max Arrows = 0.12

(b) Max Arrows = 0.25

Figure 6.2: Two puzzles generated with different parameters

Furthermore it can be observed that the third parameter may also have an influence on the runtime. As described earlier this parameter controls the number of steps the generator tries on the same puzzle before restarting with a fresh puzzle. If the generator tries many different arrow placements on the same puzzle then the result may be quite good, but after a certain number of steps it usually is cheaper to just start again with a fresh puzzle. This changes with the size of the puzzle, because on really large puzzles it is harder to reach a certain level of generation and it may be good to try lots of possibilities before

restarting.

Scaling

For all the benchmarks presented so far, the same hardware setup with a quite fast processor and lots of memory has been used. Now let's have a look at how the performance scales if a less powerful processor is used together with memory limitations for the Java VM.

For the following benchmark the generator has to produce 250 puzzles with a size of 12 × 12. Table 6.8 shows how long this takes using different hardware configurations. The tests were performed using the same machine as previously but the processor clock has been adjusted to different speeds.
To limit the memory available for the Java VM the parameter `-Xmx` has been used.

CPU Core Speed	Memory	Average time
3200 Mhz	1024 MB	5150 ms
3200 Mhz	512 MB	4502 ms
1600 Mhz	512 MB	8963 ms

Table 6.8: Generating 250 puzzles with size 12 × 12

Observing the results shows that it is not necessary to have lots of memory. The concrete benchmark was even faster with less memory, but this is just due to the variance when randomly generating puzzles. As the puzzle size gets larger the memory might have a bigger effect. The processor speed on the other hand has a big influence on the generation time. In fact a linear scaling can be observed. But the performance is quite satisfying even with only a 1600 Mhz CPU and 512 MB memory.

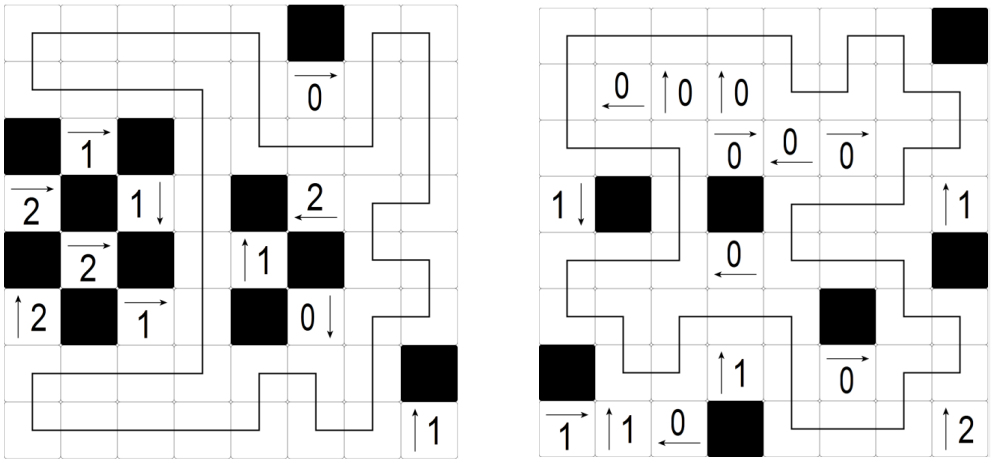
6.2.3 Comparison of the Generators

In Sections 5.1 and 5.2 two completely different approaches to generate new Yajilin puzzles have been described and their performance has been analysed in the previous sections. Now let's compare their characteristics and performance.

Puzzle Characteristics

A generated puzzle should be interesting to play. Generally both generators are able to produce nice challenging puzzles, but due to the randomness this may occur really infrequently.
The line generator here has a major drawback because the insertion of arrows

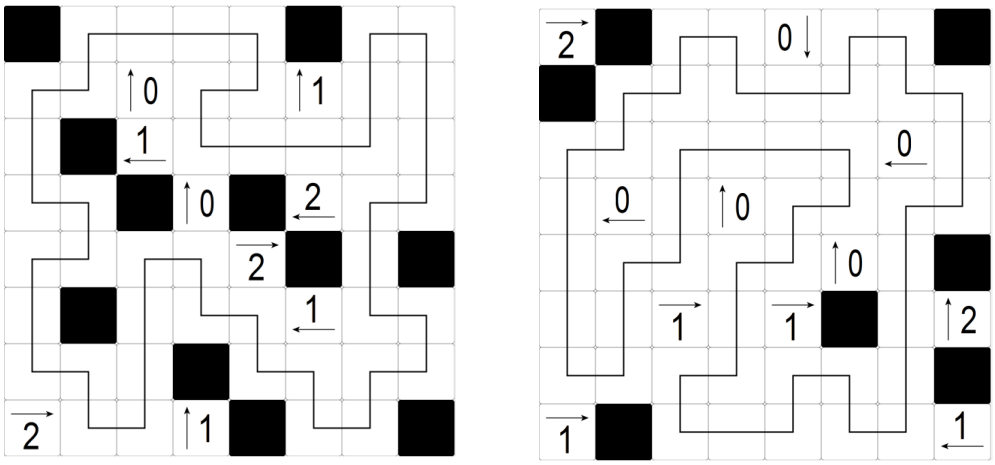
after the creation of the line can often lead to spots crowded with lots of arrows. As a result the solution may be really obvious. An example is depicted in Figure 6.3(a). The arrow generator usually manages to produce more interesting puzzles, but with some probability it may also create such spots with many arrows. Especially in bigger puzzles this occurs more often. Figure 6.3 shows two puzzles from the generators which can be considered as puzzles with a rather bad quality.



(a) Line Generator (b) Arrow Generator

Figure 6.3: Two rather "bad" puzzles generated by the different generators

More interesting generated examples are the ones shown in Figure 6.4.



(a) Line Generator (b) Arrow Generator

Figure 6.4: Two good puzzles generated by the different generators

These two puzzles do not contain too obvious arrow spots and are therefore also more challenging. Note that generally the line generator produces such

nice results with a lower frequency than the arrow generator.

Performance

The performance of the two generators has already been described within Sections 6.2.1 and 6.2.2. It has been shown that both generators have their limits. To get good puzzles the line generator is usable up to like 8×8 puzzles. The arrow generator on the other hand is also able to produce puzzles with a size like 16×16 in a quite short time. Even larger puzzles are possible but here the generation time can vary a lot and it may take several hours.

Table 6.9 compares the average time needed of the two generators. The line generator has been used with a `lineBase` of 0.65 (see Section 5.1.2). The parameters for the arrow generator where 0.0 for the minimum arrow number, 0.15 for the maximum arrow number and 2 for the steps to try (see Section 5.2.4 for the explanation). Note that the parameters for both generators have been chosen such that the work becomes quite hard but the chance to get interesting puzzles is much higher.

Size	# Puzzles	Avg. LineGenerator	Avg. ArrowGenerator
6×6	100	99 ms	60 ms
8×8	100	3538 ms	505 ms
10×10	10	519750 ms	1963 ms

Table 6.9: Generator Performance Comparison

As can be seen the line generator already takes about 9 minutes to produce a 10×10 puzzle with the selected parameters. The arrow generator on the other hand manages to do so in about 2 seconds.

6.2.4 Threading

The performance of both generators can be improved by using more threads and therefore taking advantage of processors with more cores or machines with multiple processors.

The principle to realize usage of more threads when generating is as follows:

1. A central coordinator starts multiple generators
2. All generators try to produce a puzzle
3. If one generator finds a correct one it announces this to the coordinator
4. The coordinator stops the other generator threads

Using this implementation the probability to find a correct puzzle increases due

to the fact that there are more generators trying.
Table 6.10 shows how the average time needed to generate a 12 × 12 puzzle changes with the amount of worker threads used. The test was run on a quad core processor. It can be observed that using two instead of one thread improves the performance notably. Up to the number of cores available the performance improves with more threads. Using more workers than cores available slows the generation down because a single core is pretty much used to capacity with one generator worker.

Threads	**Average time**
1	4683 ms
2	2713 ms
3	2113 ms
4	1959 ms
8	2253 ms

Table 6.10: Generating 500 puzzles with size 12 × 12

The current implementation uses one thread less then available cores, if there are more cores available. For example on a quad core processor three worker threads will be started. The advantage is that the whole machine is not that busy during the generation.

7 Application & GUI

7.1 Used Technology and Basics

For the program itself the programming language Java has been used. This choice allows to run the program on many different systems like Windows, Linux or Mac OS. The whole program only uses Java and the SAT4J libraries to ensure that there are no special dependencies and it works on every machine with Java 1.5 or newer installed. Our graphical user interface (GUI) is completely built from scratch and uses only Java Swing and AWT. This choice was also made to allow many different machines to run the program.

7.2 GUI Overview

We tried to keep the GUI as simple as possible to let the player focus on playing the puzzle instead of scrolling through menus (see Figure 7.1). Except loading and saving a puzzle, all options are chosen in the same window and no new ones are opened. To play the player only needs to draw a line and place black tiles. This is done by pressing the left mouse button and directly draw the line into the puzzle. With the right button the player can clear cells and place black tiles.

There is only one menu bar at the top of the puzzle where all necessary options can be found. There are three main menus with a few sub menus. In the file menu the following items can be found:

Save State Some of the bigger puzzles can easily take more than one hour to solve by hand. If the user wants to save such a puzzle he can use this function. The current state of the puzzle is saved, completely with all lines and placed black cells. The user can choose the location where the puzzle will be saved. It will be saved in the XML based format described in Chapter 3.

Load State If the user wants to restore a puzzle which he previously saved using the "Save State"-function this item can be used. The puzzle is restored with all lines and black cells. This item is also needed to load puzzles previously created with the "Create from Scratch"-function.

Puzzle List During the work on this thesis we collected puzzles from various sources. The program comes with a list of these puzzles already collected

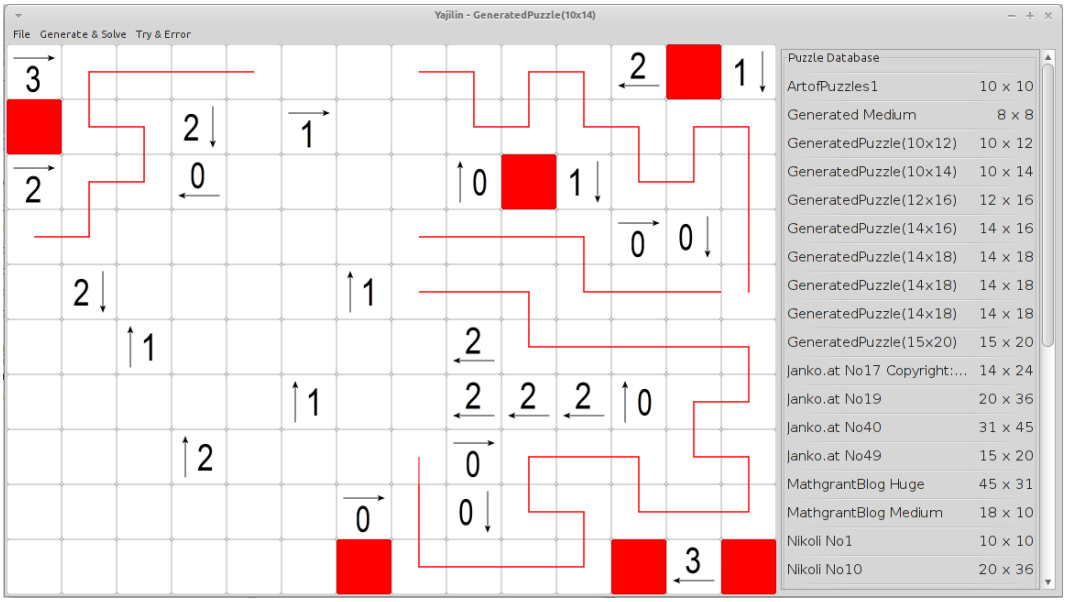


Figure 7.1: The program running under Linux Mint 11

in a database. This database can be called using this menu item. Furthermore the list also contains good puzzles which were created by our generators. The sources for these puzzles are:

- Nikoli samples (`http://www.nikoli.com/en/puzzles/yajilin/`)
- Nikoli Championship Puzzles (`http://www.nikoli.com/en/event/puzzle_hayatoki.html`)
- Art of Puzzles Blog (`http://motris.livejournal.com/`)
- janko.at (`http://www.janko.at/Raetsel/Yajilin/index.htm`)
- Mathgrant Blog (`http://mathgrant.blogspot.com/`)
- Line & Arrow Generators

Exit This item simply closes the application and all running background threads.

The next sub menu is the Generate & Solve menu which is used to solve puzzles and generate new ones. The different items in this menu are:

Hint If the user is stuck during solving a puzzle this button gives the user a hint. The hint starts from the upper left corner line by line to the lower right corner and corrects the first false placed tile. This button only works if the solver was able to solve the puzzle.

Check This item is used to check if the current puzzle is already solved. The function does not need the solver that was started in the background to be finished. It is completely independent and uses a set of methods to

check if the rules from Chapter 2 are fulfilled.

Solve Whenever a puzzle is loaded a solver is started in the background which tries to solve the puzzle. If the solver has finished the solved puzzle can be loaded by pressing this button. If the solving process is not finished then a message telling the user so is shown.

Generate The program comes with two built-in generators. By pressing this button the user can open a menu which allows him to generate a new puzzle. When generating a puzzle the player has the option to create a puzzle with three different difficulties or choose custom dimensions. It is also possible to adjust some parameters for the generators.

Create from Scratch At last there is the button which allows the user to create puzzles from scratch. After pressing the user is asked if he wants to alter an existing puzzle. Such a puzzle has to be in the XML-based format for Yajilin puzzles described in Section 3.2.
If the user wants to create a totally new puzzle he can choose the dimensions and a new puzzle only consisting of empty cells is shown. Now he can place arrows on the puzzle field and try if the puzzle is already a real puzzle according to the rules described in Chapter 2. A puzzle can be saved with the "Save State"-menu item. If the user made a correct puzzle then a new button appears which allows the user to play the puzzle.

Finally there is the Try & Error sub menu which has only the two following items:

Clean Puzzle This menu item removes all lines and black tiles from the puzzle.

Try & Error To help the user solve a puzzle this function was implemented. While the Try & Error mode is active all newly drawn lines and black cells are painted red. The user has now the option to make such lines permanent or to return to the state the puzzle was before starting this mode.

7.3 GUI Architecture

The architecture of the GUI is shown in Figure 7.2. The base of the whole architecture is the class `MainFrame`. All other elements are contained in this frame. At the top of the frame there is the menu bar, at the right side, if needed, the generation panel or puzzle list and in the middle the puzzle itself. The `MainFrame` contains a `PuzzlePanel` which contains the tiles in form of a two-dimensional array of tiles. These tiles are represented by the class `TileFrame`. Every one of these frames has its own `PuzzleMouseListener` which is responsible for drawing the line and placing other cells inside the puzzle.
The panel on the right side of the `MainFrame` is invisible most of the time. But if the player chooses to generate a puzzle or wants to see the puzzle list the panel on the right side is opened. The class `RightPanel` coordinates if either

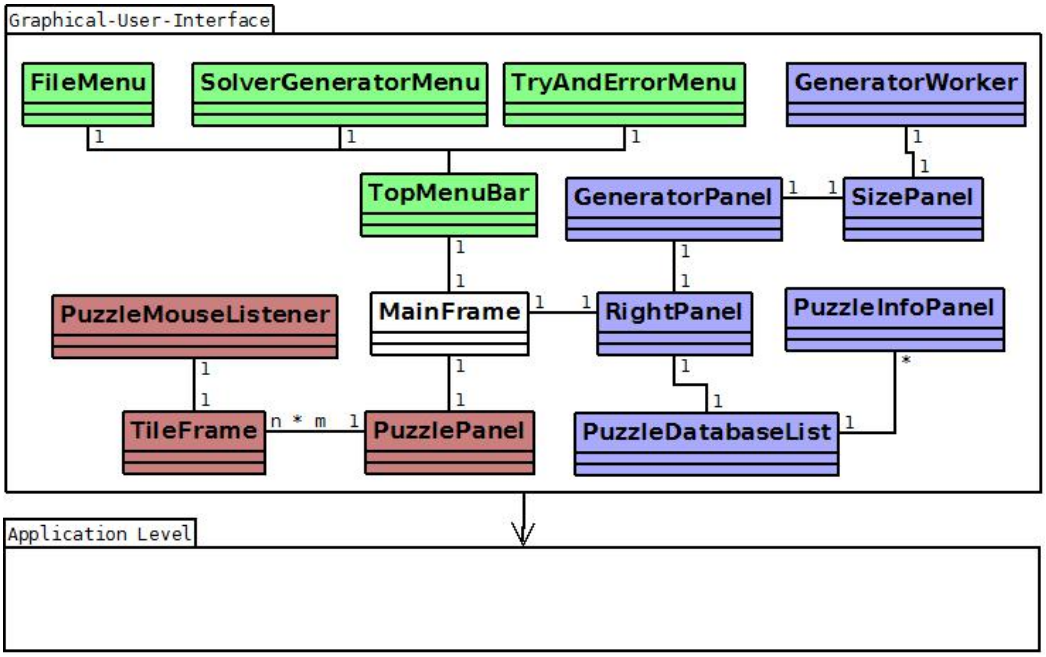


Figure 7.2: UML class diagram for the GUI

the `GeneratorPanel` or the `PuzzleDatabaseList` is shown.
The `TopMenuBar` contains the three menus `FileMenu`, `SolverGeneratorMenu` and `TryAndErrorMenu`. They consist of simple `JMenuItems` with the according functionality.

If the user wants to generate a puzzle it may take very long. During this time the user can continue to play because the `GeneratorWorker` is running in a background thread. When the generation is finished the player can either choose to use the newly generated puzzle or ignore it and continue to play.

If the user chooses to use the "Create from Scratch" function the `TileFrames` get another listener attached which is responsible to create the arrows in the cells. During this mode some of the functions of the menu are disabled because they make no sense while creating a puzzle.

8 Conclusion

This bachelor thesis covers the Japanese logical puzzle Yajilin. The three major goals have been fulfilled. A fast solver was developed which can detect puzzles that are not correct and puzzles which are correct but have more than one solution. Also two different generators were developed. The generator based on placing arrows is far better than the generator which relies on drawing a random line. The resulting puzzles are of equal quality but the arrow generator is much faster and therefore able to produce good puzzles more frequently. At last a graphical user interface was developed. It allows to play the puzzle and also has a try & error mode. The solver and both generators are built-in and the program also comes with a database of puzzles from various sources. Furthermore the user has the possibility to create puzzles and check if they are valid.

8.1 Future Work

Even with the arrow generator puzzles with sizes bigger then 15 x 15 become very time consuming to generate. Maybe there can be more research to make a quicker generator or improve the two developed during this thesis. Sadly there were no other solvers available at the time this thesis was written, so it was not possible to compare our implementation. Although the solver is very fast even on big puzzles, a comparison would show how much room for improvement exists.

Bibliography

[1] DIMACS. Satisfiability Suggested Format, 1993. Available online at: `http://www.satlib.org/ubcsat/satformat.pdf`[Last accessed: 30 June 2011].

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

[3] D. Le Berre and A. Parrain. The SAT4J library, release 2.2, 2010. Available from: `http://jsat.ewi.tudelft.nl/content/sd/JSAT7_4_LeBerre.pdf` [Last accessed: 20 June 2011].

[4] G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. 1968.