Bachelor thesis
Computing Science

Radboud University

Solving and generating puzzles with a connectivity constraint

Author:
Gerhard van der Knijff
s1006946
c.vanderknijff@student.ru.nl

First supervisor:
Prof. dr. H. Zantema
H.Zantema@tue.nl

Second supervisor:
Prof. dr. J.H. Geuvers
herman@cs.ru.nl

January 6, 2021

Abstract

Many solutions of pen-and-paper puzzles have a connectivity constraint. In this thesis, we will show a new way to solve this kind of puzzles using SMT. The connectivity constraint is implemented using a graph property. We elaborate on six examples of puzzles: Slitherlink, Masyu, Shingoki, Nurikabe, Hitori and Hashi. For the Shingoki puzzle, we also prove that this puzzle is NP-complete.

Contents

Chapter 1

Introduction

A lot of people like to do a pen-and-paper puzzle once in a while. Some of these puzzles are more difficult than others. One element that can make a puzzle more difficult, is a connectivity constraint. This constraint says: puzzle elements in the solution should be connected to each other. For example, all line parts in the solution of a Slitherlink puzzle should be connected to each other, and have to form a loop.
In other related work, some solving methods for separate puzzles were presented. Our solution presents one, efficient way that can be used for all kinds of puzzles with a connectivity constraint. We use SAT/SMT solving to solve and generate the puzzles. We show how the connectivity constraint can be encoded in SMT. Furthermore, we will give 6 examples of puzzles with a connectivity constraint: Slitherlink, Masyu, Shingoki, Nurikabe, Hitori and Hashi. We will see how we can solve them with the new encoding of the connectivity constraint. Furthermore, we will show how we can generate Slitherlink and Masyu puzzles.

For Shingoki, we didn´t find a proof of NP-completeness. Since this puzzle is very similar to Shingoki and Slitherlink, we created the NP-completeness proof based on the proof of Masyu. The last chapter of this paper consists of this proof.

First, we describe the related work in chapter 2. After that, we describe a general approach to solve puzzles using SMT in chapter 3. In chapter 4, we see how the connectivity constraint can be implemented. After these general approaches, we will show for each separate puzzle how this puzzle can be solved and for some of them how they can be generated (chapter 5). The last chapter, chapter 6 gives the NP-completeness proof of Shingoki.

Chapter 2

Related Work

Satisfiability Modulo Theories
We use Satisfiability Modulo Theories to solve our puzzles. We won't go into the details of the SMT language, and we will use only the basics of the SMT-LIB standard. There has been a lot of research on this. A standard work on this topic is a chapter in the book 'Handbook of Model Checking'. The chapter is called 'Satisfiability Modulo Theories', written by Barret and Tinelli [3].

Graph properties
Zantema and Joosten wrote an article about Graph properties[22], in which we can also find the idea for our property. For this paper, theorem 2 is the most interesting. This is the proof why our connectivity approach works.

Solving Slitherlink using SMT
Westreicher wrote a bachelor thesis about solving Slitherlink using SMT[20]. He used another way to solve the 1-loop problem. However, his approach works only for Slitherlink and is not applicable to puzzles like Hitori. He uses a technique that puts cells in the loop, and outside of the loop. Then, he checks if there is a path to a cell outside or a cell inside the loop. This way works, but is less efficient and compact as the way described in section 4.

On the NP-completeness of puzzles
When doing the research on implementing the connectivity of puzzles, we came across a paper, written by Yato in 2000 [21]. This paper gives a proof of the NP-completeness of the Slitherlink puzzle. It proves that determining if a given Slitherlink puzzle has a solution, is NP-complete. In their approach to prove this, they take the Hamiltonian circuit problem and reduce that in polynomial time to a Slitherlink puzzle. The Hamiltionian circuit problem is proven to be NP-complete, for example in Garey and Johnson in 1979[14, p. 56-60]. When researching Masyu, we also found a proof for that to be NP-complete by Friedman[12]. This proof is also based on a Hamil-

tonian circuit. This proof turned out to be very useful for our own proof of Shingoki. We will use this proof to prove that Shingoki is also NP-complete. Friedmann also wrote some other NP-completeness proofs for puzzles, for example Corral Puzzles[11], Cubic[10] and Spiral Galaxies[13].
There is also an other paper on NP-completenss of other variants of Slitherlink, written by Kölker[18]. It uses also an approach containing Hamiltonian Circuits.

Chapter 3

Solving puzzles using SMT

We first give a short introduction on SMT, and on the strategy we used to solve the different puzzles. In this chapter, we will not go into detail on the rules of the puzzles, but we will give an abstract overview of our approach.

3.1 Satisfiability Modulo Theories

The problem of Satisfiability Modulo Theories (SMT) is checking whether a given first-order logic formula is satisfiable, in the context of some background theory [2]. The question that has to be solved is: is there a model that assigns the variables in a way such that the formula is true. An SMT problem consists of variables, ranges of the variables, and constraints. Satisfiability Modulo Theories Problem is a form of the Constraint Satisfaction Problem. SMT is an extension of Boolean Satisfiability (SAT). With SAT, the instances can only consist of boolean variables. With SMT, also linear (in)equalities are also possible. Variables can thus also have other ranges then boolean values. We can for example use integers now.

SMT procedures to solve the problem are often (informally) called SMT-solvers [3, p. 738]. A lot of these solvers are available online. The most famous ones are Microsoft Z3, Yices and OptiMath. We want to use SMT-LIB[2], so we need a solver that supports that. We choose to use Z3, because Z3 is very often used, user-friendly and it supports all the functionality that we need. It also has an online portal[1], which makes it easy to run small tests very fast.

3.2 Standard SMT-LIB syntax

We now know which solver we are going to use. We have to give Z3 correct input files, so that the solver can read in the file and can give a correct

[1] https://rise4fun.com/Z3

output. A standard syntax for these files is SMT-LIB. In this section we will describe which basic parts of the SMT-LIB syntax we need, and what they mean. We use the documentation of version 2.6 for this[2].
All functions in SMT-LIB syntax have to be written in prefix notation. The first thing we have to use is the declaration of functions. In SMT syntax, this is written as follows:

(declare-fun f ($\sigma_1 \cdots \sigma_n$) σ)

In this formula f, $\sigma_1 \cdots \sigma_n$ are the types of the n variables of the function, and σ is the type of the result of the function. In fact a constant is just a function with no variables. We could write this using the declare-fun syntax, but in SMT-LIB we also have some syntactic sugar for this:

(declare-const f σ)

To specify our constraints, we use the *assert* keyword:

(assert t)

Formula t has to be satisfied. Multiple asserts are possible in one file. We use the basic combinators *and*, *or*, *xor* and *implies* to combine (parts of) formula's. At the bottom of the bottom of the SMT-file, we put two commands:

(check-sat)

and

(get-model)

The keyword $check - sat$ tells Z3 that is has to start looking if the formula is satisfiable. If the formula is not satisfiable, we get *unsat* as output. If the formula is satisfiable, we use $get - model$. We get *sat* as output, and after that we get the model that satisfies the formula: a assigning of a value of each variable in the formula. These are the basic parts that we need to solve our puzzles using SMT.

3.3 General approach on solving a puzzle

Now that we have the basic parts, we can describe our general approach on solving a puzzle. Every puzzle has different rules, but the puzzles we considered have a lot in common. All puzzles we considered are created on a rectangular grid. Some have numbers in a part of the fields (Slitherlink, Hashi, Shingoki, Nurikabe), one puzzle has only black and white dots in a part of the field (Masyu), and one puzzle is completely filled with numbers (Hitori). For each puzzle, we have to describe the puzzle rules and put them in the SMT solver.
We use a standard coordinate system to describe the point in the grid. The left bottom of the grid is point (1,1). We choose this rather than (0,0)

because it describes the first cell, and it is more intuitive to start at 1 when counting the cells. For width w of the puzzle, the x-axis ranges from 1 to w. For height h, the y-axis ranges from 1 to h. The bounds are inclusive here. We have two different kinds of grid puzzles. Puzzles for which the information is in the intersections of the grid (Masyu, Shingoki and Hashi) and puzzles for which the information is in the center of the cells (Slitherlink and Hitori). We first describe a grid for a puzzle with information in the intersection of the line. The intersections are just the coordinates. So the leftmost corner of the grid has coordinate (1,1). The information is also in coordinate (1,1). The line between (1,1) and (1,2) is (1,1,1,2). All lines are described from bottom to top if vertical, and from left to right if horizontal. If we put it in a picture, we get the following grid:

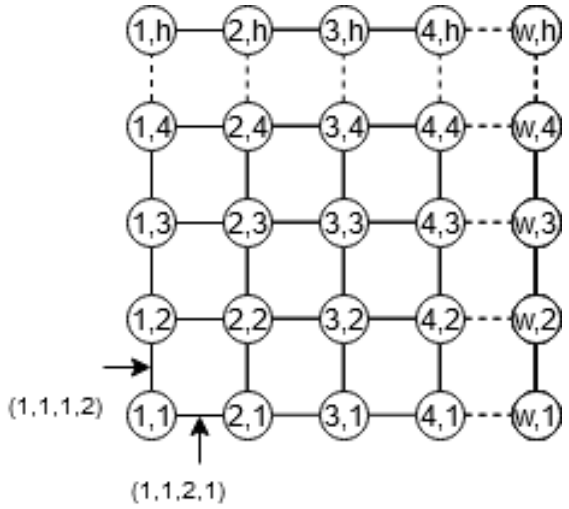


Figure 3.1: Grid of size wxh with information in the intersections

For puzzles with information in the cells, we have a slightly different grid. The coordinates are the same, but now the cells are more important than the intersections. The cells have the same index as their bottom-left corner. We get the following grid:

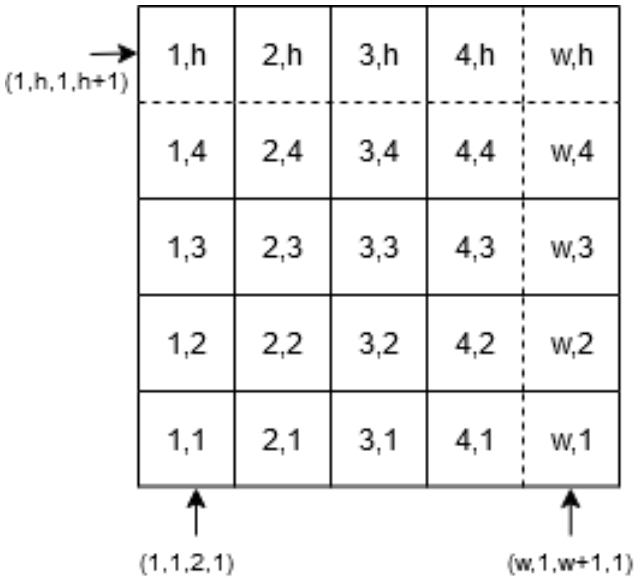


Figure 3.2: Grid of size wxh with information in the cells

We use these grid also with Hitori, although we don't need the lines in that puzzle. For each puzzle, we introduce certain functions. If the intersection points or cells have numeric values, we use a function with the coordinates as input variables and an integer result. In the Masyu puzzles, they result is a string, namely 'Black' or 'White'. In some cases, we also add a boolean function for the lines. For example, in Slitherlink only some of the lines have to be drawn. We will elaborate more on the details in chapter 5, where we will see more details on each puzzle.
Now since we have functions, we can add the puzzle specifications. For example, if we want to represent the value v in cell (x, y) of the puzzle, we describe this as follows in mathematical notation:

$$\text{Cell (x,y)} == \text{v}$$

In SMT, we write this as:

```
(= (Cell x y) v)
```

Or if we want to indicate if a line has to be drawn, we write:

$$\text{Line (x1,y1,x2,y2)} == \text{true}$$

and in SMT:

```
(= (Line x1 y1 x2 y2))
```

In this way, we can describe the properties of the puzzles. After the puzzle specifications, we add the constraints of the puzzle. We will see more about

the constraints in chapter 5, for each separate puzzle. A special constraint is the connectivity-constraint, which will be explained in the next chapter. At the end of the file, we put the (check-sat) and the (get-model) lines (as described in section 3.2), to let Z3 give the correct solution. The model that Z3 gives us back, tells us which inputs of the different functions should give which outputs. Since the inputs are often in a random order, and we don't see directly if the solution for the puzzle is correct, we need to parse the output of the Z3 output file to get a nice solution for our puzzle.

3.4 Uniqueness of the solution

We now have our solution, but how do we know that this solution is unique? We can easily check this with Z3. We use the same SMT file we generated, and add the negation of the conjunction of the variables that are true in the solution. In this way, Z3 will have to look for another, different solution. If Z3 founds another satisfying assignment of variables, multiple solutions are possible, and our original solution was not unique. If Z3 gives unsat, the first solution was the only possible solution, and our puzzle has an unique solution.

Chapter 4

Implementing connectivity using SMT

One of our main point of this paper is describing a special constraint. This constraint, the connectivity constraint, is used in all the puzzles that we considered. We want to create one, generic, way to implement this, and apply this on all the puzzles. The constraint is not equally strong for every puzzle. For example, in Hitori, each white cell has to be connected with each other white cell. But in Slitherlink, the constraint is that every line must be in one big loop, and the loop may not touch itself. But the common property is, that some parts of the puzzle have to be connected with each other.
One way to solve this is presented by Westreicher. He calls this the '1 Loop Problem'[20, p.10]. In his paper, he only focusses on Slitherlink. Therefore, his way to solve the '1 Loop Problem' is not applicable for puzzles like Hitori and Hashi. The way we present is more general, and can be applied on more puzzles. Our approach uses natural numbers. Each part of the puzzle that has to be connected to the other parts, gets a number. The solution to make sure that each part is connected with all other parts, is quite simple: each part (except for a starting point) should be connected with another part that has a natural number less than its number. So for example, if part A has the number 4, there must be some part B, for which it holds that A is connected to B, and B has a lower number than A.
We use Theorem 2 of Zantema and Joosten[22, p.7] to proof that this is correct. The theorem is the following:

Theorem 1. *Graph (V,E) is connected if and only if* $f : V \to Int$ *exists such that for all* $i \in V \setminus 1$ *a node* $j \in V$ *exists such that* $(i, j) \in E$ *and* $f(j) < f(i)$.

We assume here that there is one node that has the value 1. They prove in their paper that this is correct. The proof uses the fact that an infinite

decreasing sequence of natural numbers is not possible. Now, if we can convert an arbitrary solution of a puzzle to a graph, we know that the solution is connected if and only if the constraints with the numbers are satisfied.
For Slitherlink, Masyu and Shingoki we need an extra constraint, to assert that the solution is not only connected, but also forms exactly one loop. We do this by adding the constraint that every node in the puzzle must have degree zero or two. If a node has degree zero, there are no lines to that node, and the node is not in the solution. If a node has degree 2, it is in the solution. Since every node in the solution has degree two, there cannot be open ends, and all nodes in the solution must be in a loop. Due to the every connectivity constraint, that asserts that every node is connected, it cannot be that that there are multiple loops. So these two constraints assert that the solution is exactly one closed loop.

4.1 Implementing in SMT

We now have to implement these constraints in SMT. We use a Number function, which has as input the coordinates of the points. The function assigns a number to each part that has to be connected:

```
(declare-fun Number (Int Int) Int)
```

Then, we two for variables for the coordinates of the starting point. That point must get the number zero. In SMT, we write

```
(declare-const StartPointX Int)
(declare-const StartPointY Int)
(= 0 (Number StartPointX StartPointY)
```

Now, for each point we state that if no parts adjacent from or to that point are true, the node is not in the solution and gets the value -1 assigned by the number function. Otherwise, the Number function assigns a positive number with the given constraints.
In practice, we use a script to generate the SMT code corresponding to this. We will give an example of what the constraint looks like for just one point. We take a point (x,y) which is not in the corner or on the edge, because such a point has the most adjacent parts and it thus most suitable for an example. In case a point is close to the edge of the puzzle, the connections that are not in the puzzle, are just omitted.

```
(implies
    (and
        (not (Connection x y-1 x y))
        (not (Connection x-1 y x y))
        (not (Connection x y x y+1))
        (not (Connection x y x+1 y))
    )
```

```
    (= (Number x y) -1 )
)
(implies
    (and (Connection x y x y+1) (Connection x y x+1 y) )
    (and
        (> (Number x y) 0 )
        (or
            (> (Number x y) (Number x y+1))
            (> (Number x y) (Number x+1 y))
        )
    )
)
...
(implies
    (and (= true (Connection x y-1 x y))(= true (Connection x-1
        y x y)) )
    (and
        (> (Number x y) 0 )
        (or
            (> (Number x y) (Number x y-1))
            (> (Number x y) (Number x-1 y))
        )
    )
)
```

This constraint has thus to be added for each point in the puzzle, except for the starting point.
For the puzzles that need the 0-or-2-degree constraint, we also encode that in SMT. Again, we just give an example of created code using point (x,y). We assume point (x,y) is not on the edge. If a point is on the edge, just the lines that are outside the field of the puzzle are omitted. This results in the following SMT code:

```
(or
    (and (not (Line x y-1 x y)) (not (Line x-1 y x y)) (not (
        Line x y x y+1)) (not (Line x y x+1 y)) )
    (and (Line x y-1 x y) (Line x y x+1 y) (not (Line x-1 y x y)
        ) (not (Line x y x y+1)) )
    (and (Line x y-1 x y) (Line x y x y+1) (not (Line x-1 y x y)
        ) (not (Line x y x+1 y)) )
    (and (Line x y-1 x y) (Line x-1 y x y) (not (Line x y x+1 y)
        ) (not (Line x y x y+1)) )
    (and (Line x-1 y x y) (Line x y x+1 y) (not (Line x y-1 x y)
        ) (not (Line x y x y+1)) )
    (and (Line x-1 y x y) (Line x y x y+1) (not (Line x y x+1 y)
        ) (not (Line x y-1 x y)) )
    (and (Line x y x y+1) (Line x y x+1 y) (not (Line x-1 y x y)
        ) (not (Line x y-1 x y)) )
)
```

This is also done for each other point in the puzzle. In the next chapter, we will dive deeper into the different example puzzles and use this connectivity constraint to solve them.

Chapter 5

Implementing the connectivity on different puzzles

In this chapter, we will describe the different puzzles that we have chosen in more detail. We have chosen five different puzzles, with one thing in common. That common property is that they all have a form of connectivity in their solutions. For Slitherlink, Masyu and Shingoki, this means that the solution must be exactly one loop of lines. For Hitori, this means that all white fields in the solution must be connected. For Nurikabe, it means that all black cells must be connected. For Hashi it means, that all islands have to be connected in the end.
For each puzzle, we will give a small introduction, the rules of the puzzle and some examples. Furthermore, we will describe how the unique solution of a puzzle can found using a SMT-solver. For Slitherlink and Masyu, we also managed to generate some puzzles ourselves.

5.1 Slitherlink

The first puzzle we look at, is Slitherlink. The puzzle is also known as Fences, Takegaki, Loop the Loop, Loopy, Ouroboros, Suriza and Dotty Dilemma, and also known in Dutch as 'Kamertje verhuren'. Slitherlink is created by the Japanese puzzle publisher Nikoli around 1989[8]. Slitherlink is proven to be NP-complete by Yato in 2000 [21][17, p.24].
The goal of the puzzle is to connect dots horizontally and vertically, so that one single loop with no loose ends is created. The loop may not touch itself. The numbers (0,1,2 or 3) in the cells indicate how many of the four sides of that cell have to be drawn.

5.1.1 Examples

In the figures 5.1, 5.2 and 5.3 we see examples of Slitherlink puzzles.[1] Figure 5.1 is just a normal puzzle instance, figure 5.2 shows that a puzzle with only zeroes and ones is also possible, and figure 5.3 shows the smallest possible puzzle with at least one number that has a solution.

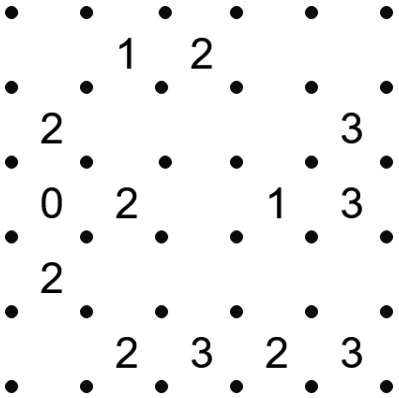


Figure 5.1

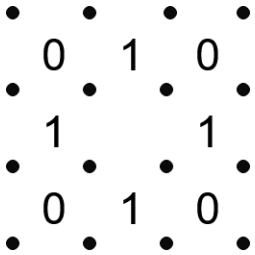


Figure 5.2

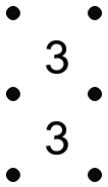


Figure 5.3

In the next figures, we see the solutions of the examples:

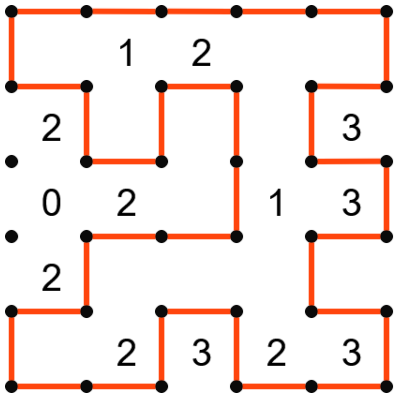


Figure 5.4

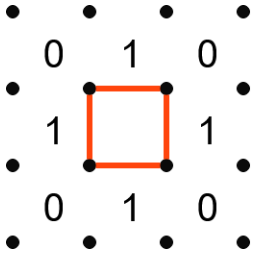


Figure 5.5

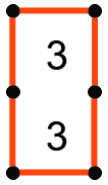


Figure 5.6

5.1.2 Find a solution using SMT

To find a solution, we will use the grid as described in figure 3.2, because the data in the Slitherlink puzzle are in the cells. The first and most important constraints we have, are the numbers in the cells. We will use the Cell

[1] Puzzle 5.1 is taken from https://www.puzzle-loop.com/ on 12-11-2020. All the puzzles on this site have a unique solution. Puzzles 5.2 and 5.3 are self-created, and those are edge cases

function to describe what is in a cell. The input variables are the coordinates of the cell, and the output is a number. We use the function Line to express whether a side is drawn or not. The input variables are the two coordinates of the line. It can either be a horizontal line, which is represented as (x, y, $x+1$, $y+1$) or a vertical line, which is represented as (x, y, x, $y+1$).

```
(declare-fun Cell (Int Int ) Int)
(declare-fun Line (Int Int Int Int) Bool)
```

For the cells that are given in the puzzle, we write the value in the SMT file. For the cells with no number, we don't add a number and we let Z3 free to fill in a number fitting to the solution. We write in SMT for example:

```
(= (Cell x y) 2)
(= (Cell x y+1) 0)
...
```

The next thing we add, is the constraint for each cell. If a cell has value 0, than all sides must be false. We write:

```
(implies
    (= 0 (Cell x y ))
    (and
        (not (Line x y x y+1))
        (not(Line x y+1 x+1 y+1))
        (not (Line x+1 y x+1 y+1))
        (not (Line x y x+1 y))
    )
)
```

For value 1, we have to assert that exactly one of the lines is true. We use the or operator for that:

```
(implies
    (= 1 (Cell x y ))
    (or
        (and (Line x y x y+1) (not (Line x y+1 x+1 y+1)) (not
            (Line x+1 y x+1 y+1)) (not (Line x y x+1 y)) )
        (and (not (Line x y x y+1)) (Line x y+1 x+1 y+1) (not
            (Line x+1 y x+1 y+1)) (not (Line x y x+1 y)) )
        (and (not (Line x y x y+1)) (not (Line x y+1 x+1 y+1))
             (Line x+1 y x+1 y+1) (not (Line x y x+1 y)) )
        (and (not (Line x y x y+1)) (not (Line x y+1 x+1 y+1))
             (not (Line x+1 y x+1 y+1)) (Line x y x+1 y) )
    )
)
```

For 2, we assert that one of the combinations of two lines must be true. We again use the or for that:

```
(implies
    (= 2 (Cell x y ))
    (or
        (and (Line x y x y+1) (Line x y+1 x+1 y+1) (not (Line
            x+1 y x+1 y+1)) (not (Line x y x+1 y)) )
```

```
        (and (Line x y x y+1) (not (Line x y+1 x+1 y+1)) (Line
            x+1 y x+1 y+1) (not (Line x y x+1 y)) )
        (and (Line x y x y+1) (not (Line x y+1 x+1 y+1)) (not
            (Line x+1 y x+1 y+1)) (Line x y x+1 y) )
        (and (not (Line x y x y+1)) (Line x y+1 x+1 y+1) (Line
            x+1 y x+1 y+1) (not (Line x y x+1 y)) )
        (and (not (Line x y x y+1)) (Line x y+1 x+1 y+1) (not
            (Line x+1 y x+1 y+1)) (Line x y x+1 y) )
        (and (not (Line x y x y+1)) (not (Line x y+1 x+1 y+1))
            (Line x+1 y x+1 y+1) (Line x y x+1 y) )
    )
)
```

For the number 3, we use the same approach as with the value 1, but now exactly one line must be false:

```
(implies
    (= 3 (Cell x y ))
    (or
        (and (Line x y x y+1) (Line x y+1 x+1 y+1) (Line x+1
            y x+1 y+1) (not (Line x y x+1 y)) )
        (and (Line x y x y+1) (Line x y+1 x+1 y+1) (not (Line
            x+1 y x+1 y+1)) (Line x y x+1 y) )
        (and (Line x y x y+1) (not (Line x y+1 x+1 y+1)) (Line
            x+1 y x+1 y+1) (Line x y x+1 y) )
        (and (not (Line x y x y+1)) (Line x y+1 x+1 y+1) (Line
            x+1 y x+1 y+1) (Line x y x+1 y) )
    )
)
```

We add these constraints for each cell (x,y). If we would now run Z3 on these constraints, the solver would keep the numbers in consideration, but not that the solution must be exactly one loop. We first add the constraint that each point must have degree 0 or 2, as described in section 4.1. If we would only add this constraint, the solution could possibly consist of multiple loops apart from each other. To prevent this, we also add the connectivity constraint as presented in section 4.1. Now we are sure that we get a correct solution of the puzzle. If no solution exists, the solver gives *unsat* as output. If a solution exists, it gives exactly the lines that have to be drawn, or which line don't have to be drawn. That depends on whether the number of true lines is bigger than the number of false lines. For example:

```
(define-fun Line ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (ite (and (= x!0 x) (= x!1 y) (= x!2 x) (= x!3 y+1)) true
    (ite (and (= x!0 x) (= x!1 y+1) (= x!2 x+1) (= x!3 y+1))
        true
    ...
```

If we parse these Z3 output, and draw the lines in the puzzle, we have our solution. Z3 is a powerful tool, and it can solve quite large puzzles. The largest puzzle that we tested was size 25x30. The largest puzzles on puzzle websites are about that size. Z3 found the solution in a less than a few seconds.

5.1.3 Generating puzzles

The next thing we want to try is generating puzzles ourselves. To do this, we first specify a height and width, generated a line in the grid of that size. That line would in the end become the solution of our puzzle. We tested two options to do so.

Creating a random line

The first option we considered was creating random parts of a line, and let Z3 generate a line that connects those parts. By a trial-and-error approach we see that we have to generate about 15 percent of the line to get a solution by Z3. The advantage is that we create a very different line each time. The disadvantage is that it is not guaranteed that we find the solution of the line. It also doesn't guarantee that the line fills the whole puzzle grid. Therefor, we consider a second approach which guarantees us that we will get a line.

Creating an expanding line

The second option starts with a small square loop (see figure 5.7). The strategy is to extend the sides of the loop. In each step, we do a possible extension. We remove one line, and add one cell to the cells inside the loop. See figure 5.8 for the possible extensions. If an extension conflicts with the line, or the extension is outside the grid, we do not consider this extension as possible. In that way, the loop will never touch itself and will never go outside the grid. We choose randomly where the line is extended. Since the loop is closed at the start, and the loop is not broken with each step, we will always have one closed loop. We repeat the extension step until no extensions are possible anymore. In figure 5.9, we see an example of a generated line.

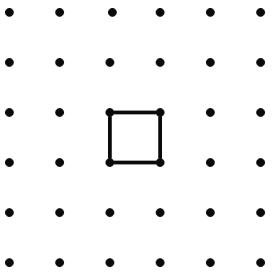


Figure 5.7

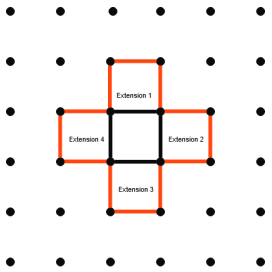


Figure 5.8

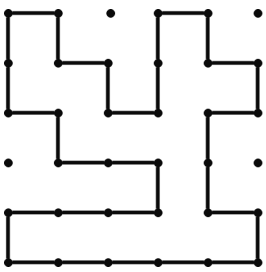


Figure 5.9

Putting and removing numbers

The next thing we do, is putting all numbers in the grid (figure 5.10). We now have a puzzle, but we want to make the puzzle more difficult. We want to remove numbers, in such a way that we keep a puzzle with a unique solution. So for each number, we hide it, and run Z3 on the puzzle. If the

puzzle still has a unique solution, we remove the number from the puzzle. If removing this number causes that there are multiple solutions of the puzzle, we keep the number. After looping randomly over each number, we have removed some of the numbers and kept some numbers. We now have our own puzzle, with a unique solution, namely the line we created. Figure 5.11 is the generated puzzle, figure 5.12 shows the solution.

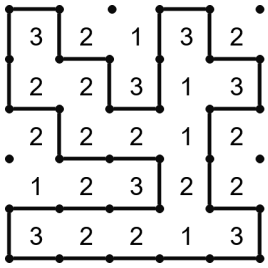


Figure 5.10

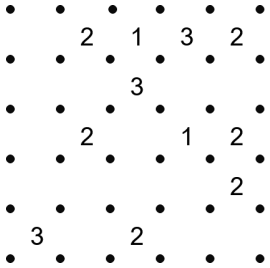


Figure 5.11

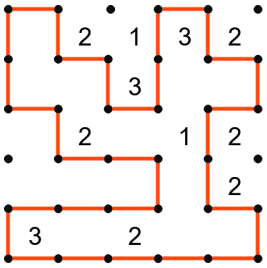


Figure 5.12

Since removing each number and running Z3 on the puzzle again and again on the puzzle is quite a heavy job, the time it takes to generate a puzzle grows exponentially with the size of the puzzle. We tested the generation for different sizes. For 5x5 puzzles, it takes less than a minute to generate the puzzle, for 10x10 puzzles it takes about 5 minutes and for 15x15 it takes already about 25 minutes.

Another remark on this way of generating Slitherlink puzzles is that we do not know on forehand if the puzzles that are generated are also solvable for human. For small puzzles it is often possible, but for bigger puzzles, more and more backtracking is needed, and it becomes very difficult for a human to solve it. In Appendix 6.1, the reader can find some generated puzzles.

5.2 Masyu

The second puzzle that we will consider, is Masyu. This puzzle is also designed by Nikoli, and is a variant of Slitherlink[6]. An other name for this puzzle is Pearl. The Masyu (= Pearl) puzzle is proven to be NP-complete by Friedman in an unpublished manuscript in 2002[12]. This paper is will later be used as one of the bases for our Shingoki proof.
The goal of the puzzle is the same as Slitherlink: draw one, continous loop through the grid such that all rules are respected. The big difference between Slitherlink and Masyu is that Masyu uses white and black circles (pearls), instead of numbers. An other difference is that the lines in Masyu are crossing through the cells, instead of over the edges of the grid.
The white and black pearls have their own rules. The white must be crossed straight through, but the line must turn in the previous or the next cell of the path. The black pearls must be turned upon, but then the line must go straight through the previous and the next cell of the path.

5.2.1 Examples

Below we find some examples of Masyu puzzles[2]. Figure 5.13 and 5.14 are just normal puzzles. The puzzle in figure 5.15 has the smallest possible size, if the puzzle has at least one white and one black cell.

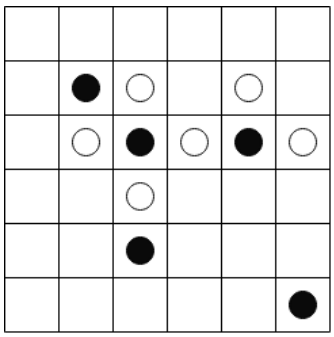


Figure 5.13

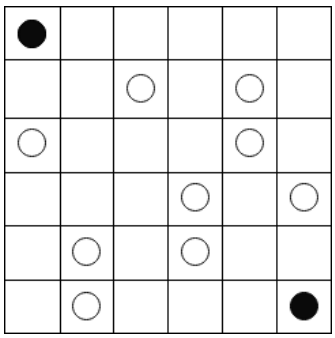


Figure 5.14

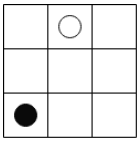


Figure 5.15

And here we see the solutions of our examples:

[2]The puzzles 5.13 and 5.14 are taken from https://www.puzzle-masyu.com/

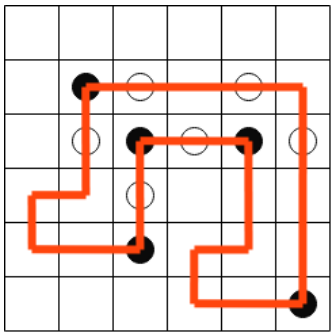


Figure 5.16

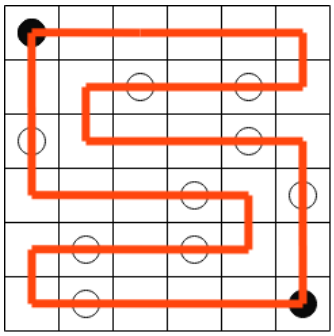


Figure 5.17

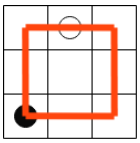


Figure 5.18

5.2.2 Solve the puzzle using SMT

The grid of this puzzle is different from the Slitherlink grid, because the lines are drawn through the cells. We use the grid from figure 3.1, because in fact, the information of the puzzles is in the corners. The grid used to draw the puzzle is not used for the solution, but is just to give more structure to the puzzle.
The only constraints we get from the puzzle are the white and the black pearls. We use the Pearl function to describe a white or a black pearl. The inputs are the coordinates, the output is a string: "black" or "white". Just as in Slitherlink, we use the same function Line again to indicate whether a line is drawn or not.

```
(declare-fun Cell (Int Int ) String)
(declare-fun Line (Int Int Int Int) Bool)
```

We write the given pearls in the SMT file, for example:

```
(= (Pearl x y) "Black")
(= (Pearl x y+1) "Black")
...
```

The next things we add, are the constraints for the white and black pearls. For the black pearls, we have to assert that for exactly two sides $\in$ {up, down, left, right}, a line of length two goes that side. We list all combinations, and then use the or operator to assert that exactly two are true. For and example pearl with coordinates (x,y), we write in SMT:

```
(implies
    (= "Black" (Pearl x y ))
    (or
        (and
            (Line x y x y+1) (Line x y+1 x y+2)
            (Line x y x+1 y) (Line x+1 y x+2 y)
            (not (Line x y-2 x y-1)) (not (Line x y-1 x y))
            (not (Line x-2 y x-1 y)) (not (Line x-1 y x y))
        )
        (and
```

```
            (Line x y x y+1) (Line x y+1 x y+2)
            (Line x-2 y x-1 y) (Line x-1 y x y)
            (not (Line x y-2 x y-1)) (not (Line x y-1 x y))
            (not (Line x y x+1 y)) (not (Line x+1 y x+2 y))
        )
        (and
            (Line x y-2 x y-1) (Line x y-1 x y)
            (Line x y x+1 y) (Line x+1 y x+2 y)
            (not (Line x y x y+1)) (not (Line x y+1 x y+2))
            (not (Line x y x-2 y)) (not (Line x y-1 x y))
        )
        (and
            (Line x y-2 x y-1) (Line x y-1 x y)
            (Line x-2 y x-1 y) (Line x-1 y x y)
            (not (Line x y x y+1)) (not (Line x y+1 x y+2))
            (not (Line x y x+1 y)) (not (Line x+1 y x+2 y))
        )
    )
)
```

For the white pearls, we only have two options, because the line crosses it either horizontally, or vertically. However, for the white pearl, we have to constraint that the loop turns either in the next cell, or in the previous cell. For both horizontal and vertical, we list the turns, and state that at least one turn must be true using the or operator.

```
(implies
    (= "White" (Pearl x y ))
    (or
        (and
            (= true (Line x-1 y x y))
            (= true (Line x y x+1 y))
            (or
                (= true (Line x-1 y-1 x-1 y))
                (= true (Line x-1 y x-1 y+1))
                (= true (Line x+1 y-1 x+1 y))
                (= true (Line x+1 y x+1 y+1))
            )
        )
        (and
            (= true (Line x y-1 x y))
            (= true (Line x y x y+1))
            (or
                (= true (Line x-1 y-1 x y-1))
                (= true (Line x y-1 x+1 y-1))
                (= true (Line x-1 y+1 x y+1))
                (= true (Line x y+1 x+1 y+1))
            )
        )
    )
)
```

To assert that the solution is exactly one loop, we have to add the constraints as described in section 4.1. Again both constraints are needed, because the

loop may not split at a certain point, and there must be exactly one loop. If we now run Z3 on the generated SMT file, we get a line that gives a correct solution of the puzzle.

5.2.3 Generating Masyu puzzles

Since Masyu is quite similar to Slitherlink, it is interesting to look if we can generate Masyu puzzles in about the same way. For Masyu, we also need a line for a solution, so we will use the same way to generate a random line as with Slitherlink. Whenever we have a line, we can draw all possible pearls in the line (figure 5.19). If we would leave out the line and keep the pearls, we have a puzzle with a solution. However, we don't know if this solution is unique, and we want a puzzle with a unique solution. So first we check if there is only one solution, by running Z3 on the puzzle file with the negation of the line in it. If the puzzle has a unique solution, we can go a step further and try to minimize the number of pearls in the puzzle. This makes the puzzle more difficult. We remove the pearls in the same way as we removed the numbers in Slitherlink. We remove a pearl, and then we check if the puzzle still has a unique solution. If so, we can leave the pearl out. If not, we have to keep the pearl in the puzzle. In the end, we have a puzzle with some of the pearls left. Figure 5.20 shows the generated puzzle, figure 5.21 shows the solution.

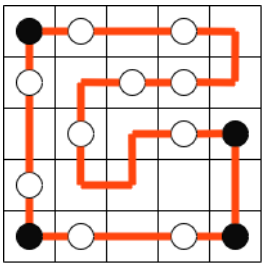


Figure 5.19

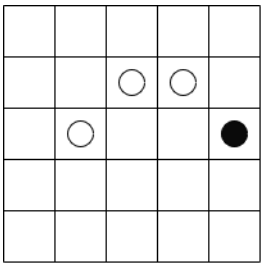


Figure 5.20

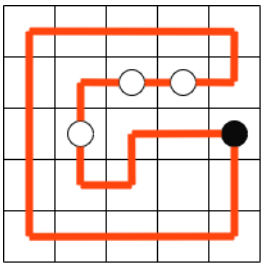


Figure 5.21

Remarks

There are some remarks that we have to add to the generation of Masyu puzzles. In the first place, we noticed that it is very often the case that a puzzle must have a certain number of black pearls in it. If not, the puzzle often has multiple solutions.

Furthermore, we noticed that it is very difficult to create large Masyu puzzles. For example, a 10x10 puzzle took already very long. It is difficult because in almost all cases the puzzle has multiple solutions. A solution to this could be to add certain pearls to the puzzle that don't match with the line. In that case, we would have to change the line. That would be a way

to create bigger puzzles.
In Appendix A, we have added some puzzle that the reader can solve. All puzzles are human solvable and have a unique solution.

5.3 Shingoki

Shingoki is the third puzzle we consider. The origin of this puzzle is not really clear. The puzzle is also sometimes called Semaphores. The puzzle is a variant of Masyu, and therefore also a variant of Slitherlink. There is no proof of NP-completeness of Shingoki yet, but we will prove that this puzzle is NP-complete in chapter 6. Just like Masyu, the puzzle also contains black and white pearls. The goal is again to create exactly one loop. The white pearls must be crossed straight, and the black pearls must be turned on. Unlike Masyu, we do not have rules about turns in the next or previous cells here. Instead, we have a new rule. Each pearl has a number in it. This number describes the sum of the lengths of the two straight, outgoing lines. It does not matter if there is another Pearl on that straight line. Because of the single loop we have to create, this puzzle is also suitable to implement the connectivity constraints.

5.3.1 Examples

We will now see three examples of the puzzles[3]. There are not many puzzles available, because the puzzle is very new. Figure 5.22 and 5.23 are just normal puzzles, figure 5.24 is an edge case with only one black and one white pearl.

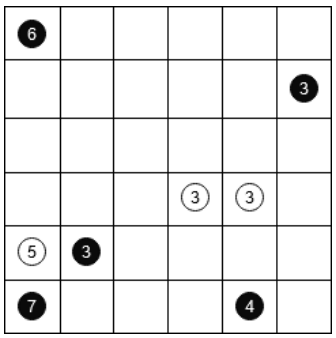


Figure 5.22

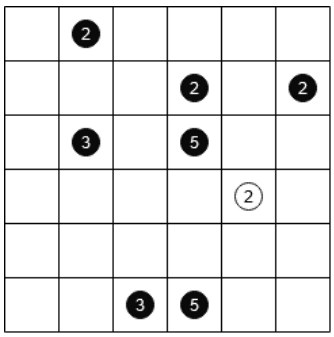


Figure 5.23

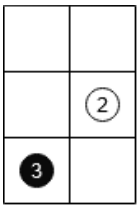


Figure 5.24

And here we see the solutions of our examples:

[3]Figure 5.22 and figure 5.23 are taken from https://www.puzzle-shingoki.com/

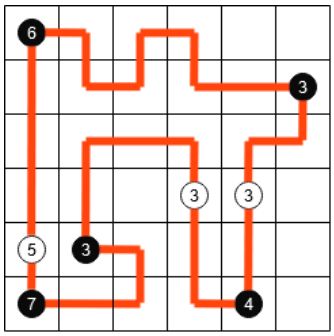


Figure 5.25

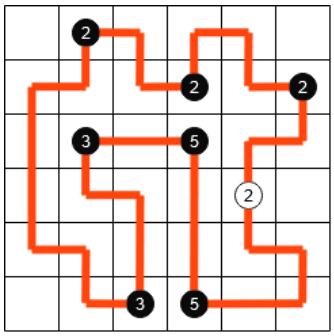


Figure 5.26

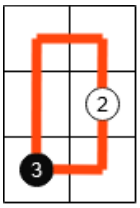


Figure 5.27

5.3.2 Solve the puzzle using SMT

Just as we did with the previous puzzles, we will explain in this section how we can solve this puzzle using SMT. In the first place, we consider the grid. The grid is just the same as used with Masyu. The information is in the corners of the line, and the grid of the puzzle has no function in the solution. We have different kinds of information in the puzzle. In the first place, we have the numbers in the pearls. We use the Pearl function for this. It has the coordinates as input variables, and an integer as output. We also need a PearlColor function, to describe if the pear is black or white. That function has a string "Black" or "White" as output. To describe the line, we just use the same Line function as we used with the previous puzzles. We write in SMT:

```
(declare-fun Pearl (Int Int ) Int)
(declare-fun PearlColor (Int Int) String)
(declare-fun Line (Int Int Int Int) Bool)
```

We write the given pearls in the SMT file, for example:

```
(= (Pearl x y) 5)
(= (PearlColor x y) "Black")
...
```

Now, we have two different kinds of constraints in this puzzle. The numerical constraints, and the color constraints. We can combine them, in the following way. First we consider the white pearls. We know from the puzzle which pearls are white. So for that pearls, we write that there must be a horizonal or vertical line through this pearl. One every side must be at least a straight line of length one. Let n be the number in the pearl. We already have two parts of the line, and we have to divide the other $n-2$ parts to extend the two straight lines to meet the two constraints. In figure 5.28, we see an example of the possibilities of a white pearl with number 3 in it.

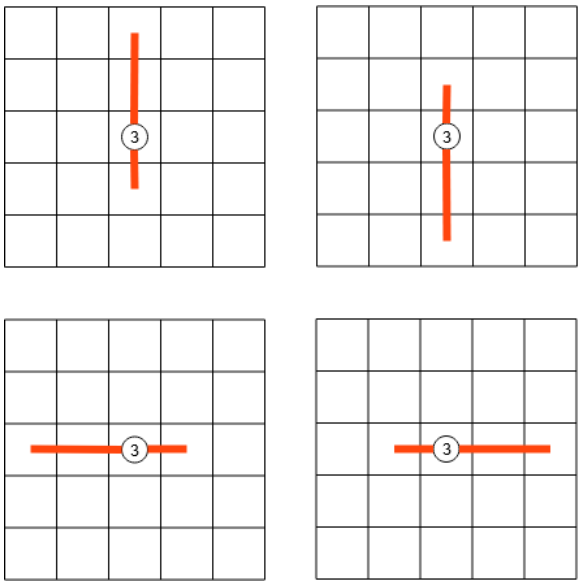


Figure 5.28

We list all the possible combinations, and write that exactly one of them must be true using the or. For a white pearl on coordinates (x, y) with number 3 in it, we write:

```
(or
    (and
        (not (Line x y-2 x y-1))
        (Line x y-1 x y)
        (Line x y x y+1)
        (Line x y+1 x y+2)
        (not (Line x y+2 x y+3))
    )
    (and
        (not (Line x y-3 x y-2))
        (Line x y-2 x y-1)
        (Line x y-1 x y)
        (Line x y x y+1)
        (not (Line x y+1 x y+2))
    )
    (and
        (not (Line x-3 y x-2 y))
        (Line x-2 y x-1 y)
        (Line x-1 y x y)
        (Line x y x+1 y)
        (not (Line x+1 y x+2 y))
    )
    (and
        (not (Line x-2 y x-1 y))
        (Line x-1 y x y)
        (Line x y x+1 y)
        (Line x+1 y x+2 y)
```

```
        (not (Line x+2 y x+3 y))
    )
)
```

For the black pearl, we use the same approach, but now we have more possibilities because the line must turn. We see the possibilities for the same pearl in a black color in the next figure:

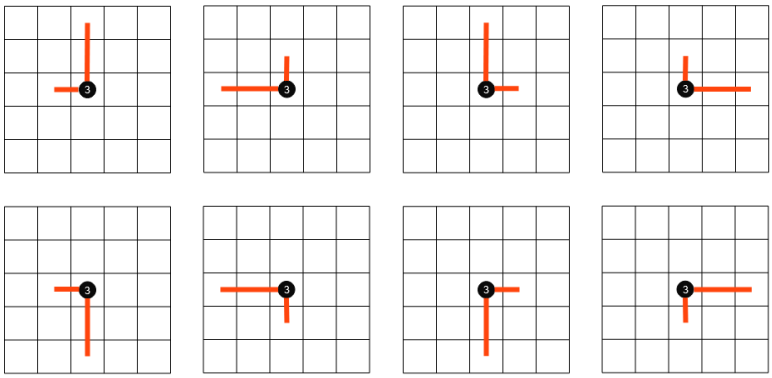


Figure 5.29

We list the possibilities also in SMT:

```
(or
    (and
        (not (Line x-2 y x-1 y))
        (Line x-1 y x y)
        (Line x y x y+1)
        (Line x y+1 x y+2)
        (not (Line x y+2 x y+3))
    )
    (and
        (not (Line x-3 y x-2 y))
        (Line x-2 y x-1 y)
        (Line x-1 y x y)
        (Line x y x y+1)
        (not (Line x y+1 x y+2))
    )
...
)
```

The same approach can be used for numbers higher then three, because we just list all the possibilities and then put an or operator around them. The last constraint is the connectivity constraint. We can use the both constraints as described in section 4.1. We need the first constraint to make sure that all parts are connected with each other. The second constraint (degree 0 or 2) is needed to make sure that there are no loose ends in the loop. These constraints work perfectly for this puzzle. If we run Z3 on the SMT file with the constraints, we get all parts of the line, which is our solution.

5.3.3 Generating Shingoki

Just as with Masyu, we didn't generating Shingoki in this research. The same approach as with Slitherlink could be possible, by creating all possible diamonds on a given line, with the numbers in it. After that one could remove the diamonds to get a new puzzle. This could be done in later research.

5.4 Hitori

The fourth puzzle we consider is Hitori. This puzzle is also designed by Nikoli[5]. The puzzle is proven to be NP-complete by Demaine and Hearn[16]. Hitori is quite different from the puzzles that we have seen already. The puzzle does not involve any lines, but it contains numbers in white cells and black cells.
The puzzle is quite simple. The puzzle starts with a grid, filled completely with numbers. The goal of the puzzle is to get a grid where no number appears twice in a row or column. This is done by making some cells black. There are two more rules. No black cells may be adjacent to each other, and the remaining white cells must be connected to each other. For the last rule, we can use our designed connectivity constraint.

5.4.1 Examples

We will first look at some examples of Hitori puzzles[4]. Figure 5.30 and figure 5.31 are just normal puzzles, figure 5.32 is a very small puzzle with at least one duplicate number and an unique solution.

2	3	2	1	5
2	4	1	3	5
2	1	2	2	5
1	5	2	4	3
2	2	5	5	2

Figure 5.30

4	2	5	3	5
5	2	4	1	2
4	5	1	1	4
2	1	1	4	5
4	4	2	4	3

Figure 5.31

2	1
2	2

Figure 5.32

And here we see the solutions:

[4]The puzzles 5.30 and 5.31 are taken from https://www.puzzle-hitori.com/

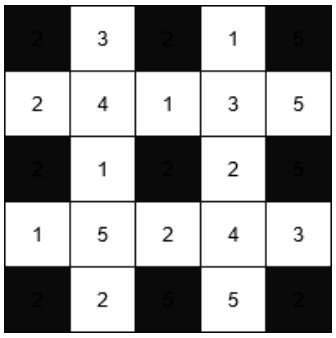


Figure 5.33

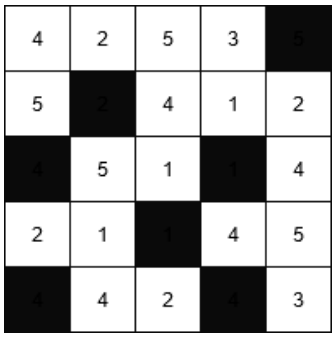


Figure 5.34

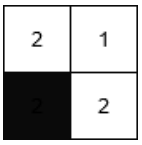


Figure 5.35

5.4.2 Solve Hitori using SMT

Since the information is in the cells, and we don't do anything with lines, we use the grid of figure 3.2. The constraints we have are based on the numbers in the puzzle, and on the color of the cell: black or white. We need one function to describe the number in the cell. The inputs are the coordinates of the cell. The result is a number. We also need a function for the color of the cell. The inputs are again the coordinates, the result is a string "Black" or "White. In SMT:

```
(declare-fun Cell (Int Int ) Int)
(declare-fun CellColor (Int Int) String)
```

We write the given numbers in the SMT file, for example:

```
(= (Cell x y) 3)
(= (Cell x y+1) 2)
(= (Cell x y+1) 3)
...
```

We can now start with the constraints. We must make sure that there are no two white cells in a row or column. This is easy to constrain in SMT. We state that it may not happen that two cells have the same x coordinate or y coordinate, have the same result for the Cell function and are both white. So for a certain row r and two different cells (x, r) and (y, r) in SMT, we get:

```
(not
    (and
        (= (Cell x r) (Cell y r )
        (= (CellColor x r) "White")
        (= (CellColor y r) "White")
    )
)
```

For columns, we do the same. For a certain column c and two different cells (c, x) and (c, y) we write:

```
(not
    (and
        (= (Cell c x) (Cell c y )
        (= (CellColor c x) "White")
        (= (CellColor c y) "White")
    )
)
```

We do this for all combinations of two cells in every row, and for every combination in every column. The next constraint we have, is that no two black cells may be adjacent to each other. For each cell (x, y) we write that if that cell is black, the four adjacent cells may not be black. In SMT syntax:

```
(implies
    (= (CellColor x y) "Black")
    (and
        (not (= (CellColor x-1 y) "Black"))
        (not (= (CellColor x+1 y) "Black"))
        (not (= (CellColor x y-1) "Black"))
        (not (= (CellColor x y+1) "Black"))
    )
)
```

The last constraint we have to add is the connectivity constraint. Since we do not have to deal with one single loop, using the first constraint of section 4.1. We need an extra constraint here, because we don't want that the starting point of the connected part is black, because all white cells have to be connected. We write in SMT:

```
(not (= (CellColor StartPointX StartPointY) "Black"))
```

If a cell is black, it gets the number -1, because it is not part of the connected cells. If a cell is white, there is a connection with another cell if that cell is adjacent and also white. For one of the connected cells, it must hold that it has a lower number. This is an implementation of the general approach as described in section 4.1.
If we now run Z3 on the file, we get a solution which exactly describes which cells should be black, and which cells should be white. This is the solution of our puzzle.

5.4.3 Generating Hitori puzzles

Generating Hitori puzzles is very different from generating Slitherlink puzzles. You cannot create a puzzle easily by leaving out information, because all information is already there. One way to do this could be to assign random numbers to a grid, and try whether that puzzle has a solution or not. However, after a small manual test, we observed that we almost never got a nice puzzle by random assigning numbers. It was not in the scope of this research to find out another way to generate those puzzles. This could be done in future research.

5.5 Nurikabe

The fifth puzzle we consider is Nurikabe. This puzzle is also designed by Nikoli[7]. The puzzle is proven to be NP-complete by McPhail and Fix[19]. Just like Hitori, Nurikabe contins white and black cells with numbers. However, only some of the white cells contain a number.
The puzzle is less simple than Hitori. The puzzle starts with a grid, filled with some numbers. The goal of the puzzle is to get color some of the grid cells black and some white. A number n in a cell denotes that that cell is part of an island with n cells. Each white island must consist exactly one numbered cell. The rest of the cells, the black cells must form exactly one see. That means that all black cells must be connected with each other. Therefore, we can use our connectivity property. One more restriction is that 2x2 black cells are not allowed.

5.5.1 Examples

We will now look at some examples of Nurikabe puzzles[5]. Figure 5.36 and figure 5.37 are just normal puzzles, figure 5.38 is a very small puzzle with at least one number and one black cell.

		10						2	
									3
				2					
					2				
				2			2		
	1					2			
		3							
		11							

Figure 5.36

3				2
		2		
1				2

Figure 5.37

1	

Figure 5.38

And here we see the solutions:

[5]The puzzles 5.36 and 5.37 are taken from https://www.puzzle-nurikabe.com/

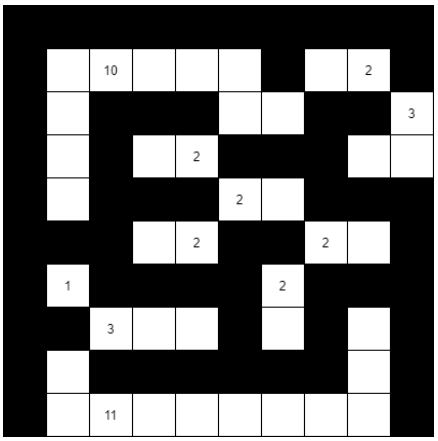


Figure 5.39

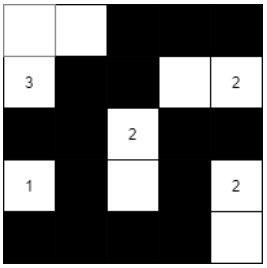


Figure 5.40

Figure 5.41

5.5.2 Solve Nurikabe using SMT

Just like with Hitori, we use the grid of figure 3.2. The constraints we have are based on the numbers in the puzzle, and on the color of the cell: black or white. We need one function to describe the number in the cell. The inputs are the coordinates of the cell. The result is a number. We also need a function for the color of the cell. The inputs are again the coordinates, the result is a string "Black" or "White. We need one more function. This function describes to which numbered cell a white cell is connected. We use a function with the coordinates of the numbered cell and the other cell as input, and the result is a boolean. This function denotes thus if a cell is connected to that numbered cell. In SMT:

```
(declare-fun Cell (Int Int ) Int)
(declare-fun CellColor (Int Int) String)
(declare-fun ConnectedWithNumberedCell (Int Int Int Int) Bool)
```

We write the given numbers in the SMT file, for example:

```
(= (Cell x y) 1)
(= (Cell x y+1) 3)
(= (Cell x y+1) 2)
...
```

We can now start with the constraints. We must make sure that there are exactly n cells connected to a cell with the number n. We use a way to assert that exaclty n variables are true, using the plus operator. If a field is connected, we add 1. If not we add 0. In this way, we can exactly reach the desired number. So for a certain numbered field (nX, ny) and all possible connected cells (within a range of n of the numbered field) in SMT, we get for example:

```
(=
(+
```

```
(ite (ConnectedWithNumberedCell nX nY nX nY) 1 0)
(ite (ConnectedWithNumberedCell nX nY nX-1 y) 1 0)
(ite (ConnectedWithNumberedCell nX nY nX+1 y) 1 0)
(ite (ConnectedWithNumberedCell nX nY nX-2 y) 1 0)
(ite (ConnectedWithNumberedCell nX nY nX+2 y) 1 0)
...
)
n
)
```

The next constraint we add, is that the numbered fields must have the white color. This is an easy constraint in SMT. For all numbered fields, we write:

```
(= (CellColor x y) "White" )
```

Now, we state that each white cell should be connected to exactly one numbered cell. A numbered cell have to be trivially connected to itself. All non-numbered cells should have one other cell that they connect with. All black cells must not be connected with any numbered cell. For the numbered cells, we write:

```
(ConnectedWithNumberedCell x y x y)
```

For all other cells, we write:

```
(implies (= (CellColor x y) "White")
    (or
        (ConnectedWithNumberedCell n1X n1Y x y) (not (
            ConnectedWithNumberedCell n2X n2Y x y) ) ...
        (ConnectedWithNumberedCell n2X n2Y x y) (not (
            ConnectedWithNumberedCell n1X n1Y x y) ) ...
        ...
    )
)
(implies (= (CellColor x y) "Black")
    (not
    (or
        (ConnectedWithNumberedCell n1X n1Y x y)
        (ConnectedWithNumberedCell n2X n2Y x y)
        (ConnectedWithNumberedCell n3X n3Y x y)
        ...
    )
    )
)
```

We can state that cells have to be connected to a numbered cell, but how do we assert that they are actually adjacent in the puzzle? To assert this, we use the connectivity constraint, but than in a small setting. For each island, we will assert that every cell that is connected to the numbered cell in the island, is reachable from the numbered cell. The numbered cell gets value 0, and all other cells in that island must have a neighbour that has the value of the cell minus one. This is the same idea as our constraint in section 4.1. For each numbered cell, we write in SMT:

```
(= (Number x y) 0 )
```

For each other cell, we write:

```
(implies
    (= (Color x y) "Black")
    (= (Number x y) -1 )
)
(implies
    (= (Color x y) "White")
    (and
        ( > (Number x y) 0 )
        (= (Number x-1 y) (- (Number x y) 1))
        (= (Number x+1 y) (- (Number x y) 1))
        (= (Number x y-1) (- (Number x y) 1))
        (= (Number x y+1) (- (Number x y) 1))
    )
)
```

However, we have not constraint that the islands must be separated from each other by black cells. They may not touch each other. We do this by stating that if a cell is white, and the right or the top neighbour is also white, that cell must be connected to the same numbered cell. So they must have the same values for the function connectedWithNumberedCell. For example, we write in SMT:

```
(implies (Color x y)
    (implies (= (Color x+1 y) true)
        (= (connectedWithNumberedCell n1X n1y x y) (
            connectedWithNumberedCell n1X n1y x+1 y))
        (= (connectedWithNumberedCell n2X n2y x y) (
            connectedWithNumberedCell n2X n2y x+1 y))
        (= (connectedWithNumberedCell n3X n3y x y) (
            connectedWithNumberedCell n3X n3y x+1 y))
        ...
    )
)
...
(implies (Color x y)
    (implies (Color x y+1)
        (= (connectedWithNumberedCell n1X n1y x y) (
            connectedWithNumberedCell n1X n1y x y+1))
        (= (connectedWithNumberedCell n2X n2y x y) (
            connectedWithNumberedCell n2X n2y x y+1))
        (= (connectedWithNumberedCell n3X n3y x y) (
            connectedWithNumberedCell n3X n3y x y+1))
        ...
    )
)
```

The next constraint we add is the constraint that no 2x2 areas may be all black. We do this by stating that for all cells the top, right diagonal, and right adjacent cell may not be all black if that cell is black. We write in SMT:

```
(not
    (and
        (= (CellColor x y) "Black")
        (= (CellColor x y+1) "Black")
        (= (CellColor x+1 y+1) "Black")
        (= (CellColor x+1 y) "Black")
    )
)
```

The last constraint we have to add is the connectivity constraint. Since we do not have to deal with one single loop, using the first constraint of section 4.1. We need an extra constraint here, because we don't want that the starting point of the connected part is white, because all black cells have to be connected.

```
(= (CellColor StartPointX StartPointY) "White"))
```

If a cell is white, it gets the number -1, because it is not part of the connected cells. If a cell is black, there is a connection with another cell if that cell is adjacent and also white. For one of the connected cells, it must hold that it has a lower number. This is an implementation of the general approach as described in section 4.1.
If we now run Z3 on the file, we get a solution which exactly describes which cells should be black, and which cells should be white. This is the solution of our puzzle.

5.5.3 Generating Nurikabe puzzles

Just like Hitori, the most trivial approach to generate these puzzles is to put random numbers in a grid and check if there is a solution. However, this will not be the most smart way to generate these puzzles. It was not in the scope of this research to find out another way to generate those puzzles. This could be done in future research.

5.6 Hashi

The last puzzle we consider is Hashi. Hashi is also known as Hashiwokahero, Bridges or Chopsticks. The puzzle is invented by Nikoli in 2004[4]. The puzzle is proven to be NP-complete by Andersson in 2009[1]. Hashi contains lines and connectivity but the lines do not have to form a loop. The puzzle consist of islands, with numbers on them. The numbers are in the range 1 to 8. The goal of the puzzle is to connect all the given islands with lines. Each line must be horizontal or vertical. Lines may not cross each other. Between two islands, at most two lines may be drawn. The number on the island states how many lines must be connected to that island. In the end, there must be a path between all pairs of islands. Therefore, we will use the connectivity constraints.

5.6.1 Examples

In the following figures we have some Hashi examples[6]. The first two are just regular examples. Figure 5.44 is an edge case with only two islands.

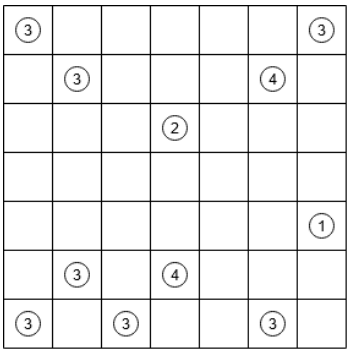


Figure 5.42

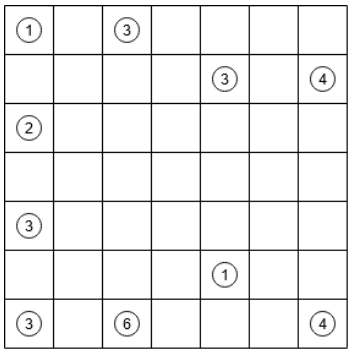


Figure 5.43

Figure 5.44

And here we see the solutions:

[6]The examples 5.42 and 5.43 are taken from https://www.puzzle-bridges.com/

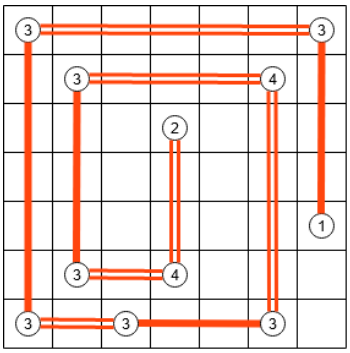


Figure 5.45

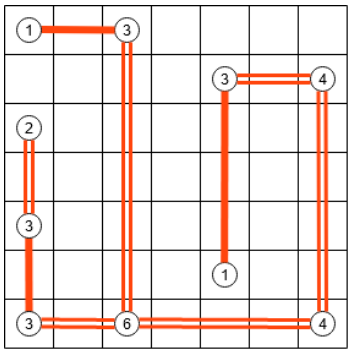


Figure 5.46

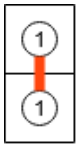


Figure 5.47

5.6.2 Solving using SMT

For this last puzzle, we will also show how this puzzle can be solved using SMT. We first define a function to describe the islands. The Island function has the coordinates of the island as input, and the number in it as an integer output. We will use a Line function that is slightly different from the line function we used in the previous puzzles. The line function has the coordinates of the first island and the second island which the line connects as input. The result of the function is an integer, more precisely zero, one or two. Zero means that no line is drawn at all. One means that exactly one line is drawn. Two means that two lines are drawn between the two islands. In SMT we define the following functions:=

```
(declare-fun Island (Int Int ) Int)
(declare-fun Line (Int Int Int Int) Int)
```

We use a script to generate the SMT file. In this SMT-file, we create a list of all possible lines. If two cells are in the same row, and no cell is between them, there can possibly be one or more lines between them. The same holds for the column. For each line, we make sure that it gets value zero, one or two.

```
(or
    (= (Line x y x+1 y) 0)
    (= (Line x y x+1 y) 1)
    (= (Line x y x+1 y) 2)
)
```

Now for each island, we check which possible lines can reach this island. We make a list of the possible lines that start or end in those island. The sum of those lines must add up to the number inside the island. To show this with an example, assume that we have an island with the number 5 on it. There are other islands above, below and right to that island. Then we write in SMT: (= (Island x y) (+ (Line (x y x $y+1$)) (Line (x y $x+1$ y)) (Line (x $y-1$ x y))))

The next constraint we have is that the lines may not cross each other. Two lines cross each other if two islands are in the same row, and there are some other islands in a column which the horizontal line crosses and one of the islands is above the horizontal line and the other is below the horizontal line. We can easily check when this is the case using the script. For each of these cases, we have to make sure that one of the lines has value zero. In that way, the lines cannot cross. For two possible crossing lines $(x1, y1, x2, y1)$ and $(x3, y2, x3, y3)$ we write in SMT:

```
(or
    (= 0 (Line x1 y1 x2 y1))
    (= 0 (Line x3 y2 x3 y3))
)
```

The last constraint we have is the connectivity constraint. We only need one constraint from section 4.1 because it is not necessary to create a loop as solution. We add the first number-constraint to make sure that all islands are connected. When we run Z3 on the generated SMT file, we see exactly which possible lines get value zero, one or two with respect to all the constraints.

5.6.3 Generating Hashi puzzles

We did not dive into an approach to generate Hashi puzzles. It is not so straight-forward as with Slitherlink, so we cannot use the same technique for that. This subject could be addressed in later research.

Chapter 6

Shingoki is NP-complete

In this chapter, we will show that the question: 'does a given Shingkoi puzzle have a solution' is NP-complete. The proof is very similar to the proof for Slitherlink by Yato[21] and the proof for Masyu by Friedman[12]. Since Masyu is very closely related to Shingoki, we will follow the structure of the proof for Masyu. We do this by creating Shingoki puzzles corresponding to arbitrary cubic planar graphs. We will show that the following theorem is holds:

Theorem 2. *A cubic planar graph G(V,E) has a Hamiltonian circuit if and only if the corresponding Shingoki puzzle has a solution*

We know that the question: 'does a given cubic planar graph have a Hamiltonian circuit' is NP-complete. That is proven by Garey, Johson and Tarjan in 1976[15]. The problem is also contained in the overview of Garey and Johnson[14, p.199]. So if we can construct a Shingoki for an arbitrary graph, we know that the decision is NP-hard. If we have that, we can complete the proof by verifying that a solution of a Shingoki puzzle can be checked in polynomial time.

For the proof, we will use the same example graph as Friedman. The graph G and its rectilinear representation can be seen in figure 6.1.

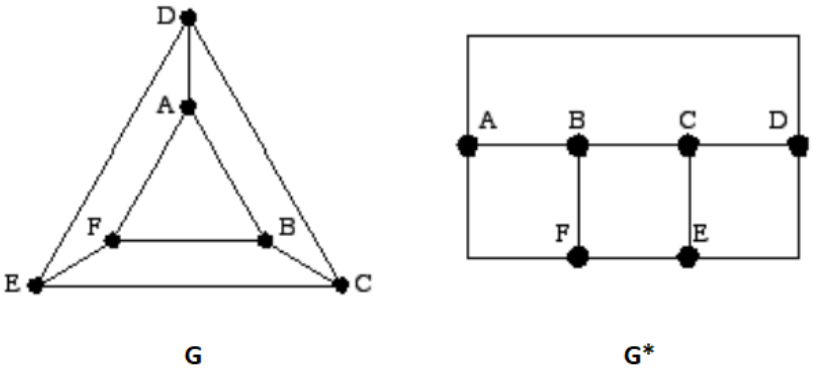


Figure 6.1

The Shingoki puzzle will contain only white pearls, with numbers in them. We follow the idea of Friedman to use walls. Those walls can be arranged to form rooms, separated by passageways. The passageways will correspond to the edges of the graph. Some of the rooms will correspond to the vertices of the graph.

6.1 Walls

In the proof for Masyu, Friedman uses the idea of walls. Friedmann shows that for $n >= 3$, there are only two local solutions for a 2 x n rectangle of white pearls. He uses the following figure:

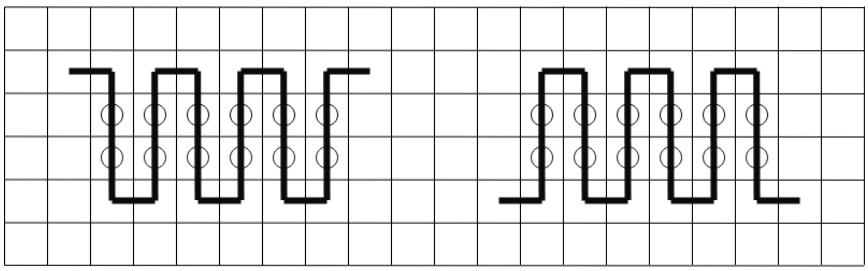


Figure 6.2

For the Shingoki puzzle, we can also create the same walls for which only two local solutions exist. We do this by putting the number 3 into all white pearls:

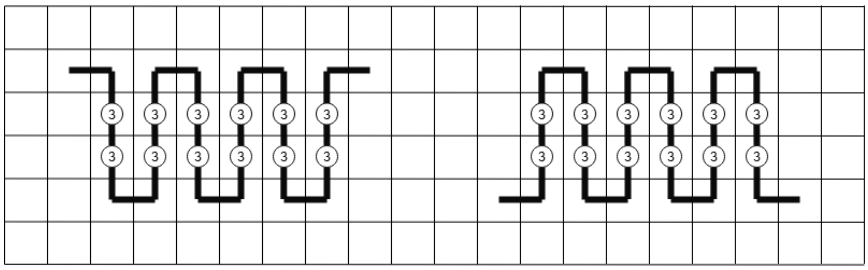


Figure 6.3

If two walls are diagonally adjacent, only one local solution exists. So if we put these rectangles together, we can create walls with a unique solution. In each face of G*, we will put a wall unit. The wall units are very flexible in size, andAn example of a wall unit can be seen in figure 6.4. Note that the loop has to go further outside the wall unit, because otherwise it will touch itself some time.

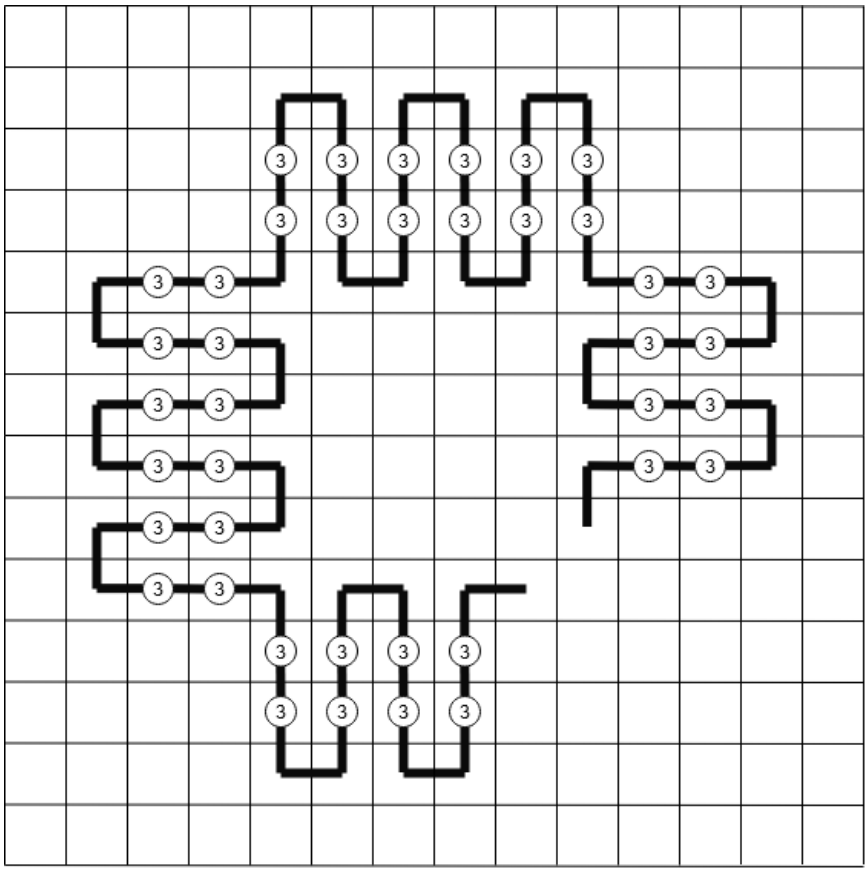


Figure 6.4

The wall units will be placed next to each other, with a passageway between them. Every passageway is just one field wide, which ensures that the loop can only go once through each passageway. The passageways correspond to the edges of G*. The small blank areas that are created at the intersections of passageways are called rooms. Each vertex corresponds to one of the rooms. We can use figure 6 of Friedman to give an example of the rooms and passageways:

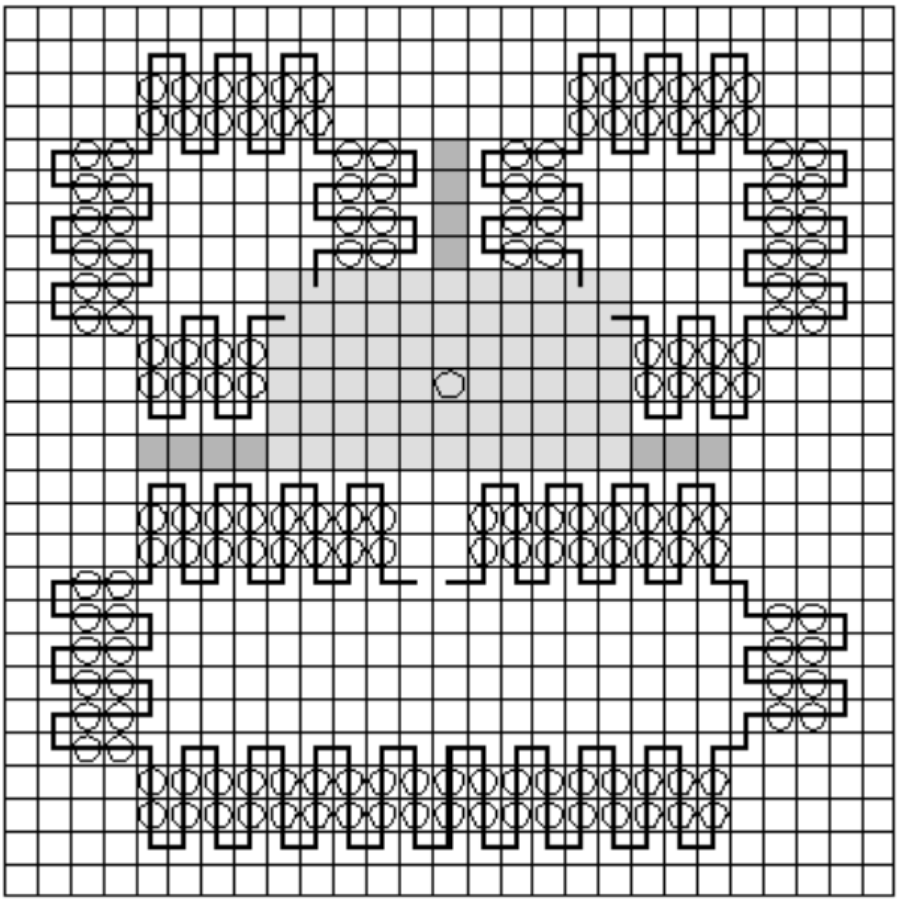


Figure 6.5

The light grey area is a room, the dark grey area's are passageways. In our case, the only difference until now will be that in the white pearls of the walls, the number 3 is used.
In each room corresponding to a vertex of G*, we put a white pearl. We put the number two in it, so that the rooms can be as small as possible. These pearls assert that the loop will come to each room. Since G* is cubic and the passageways can only be passed once, the loop must enter each room exactly once. Friedman adds to this that the ends of the paths of a wall

unit must be in a room which corresponds to a vertex in G*. This asserts that that wall unit will be part of the loop.
Now since we have that each room that corresponds to a vertex needs to be visited exactly once, and all passageways correspond to edges, the puzzle will have a solution if and only if the graph has a Hamiltonian circuit.
Now, for the puzzle G, we can create the following puzzle:

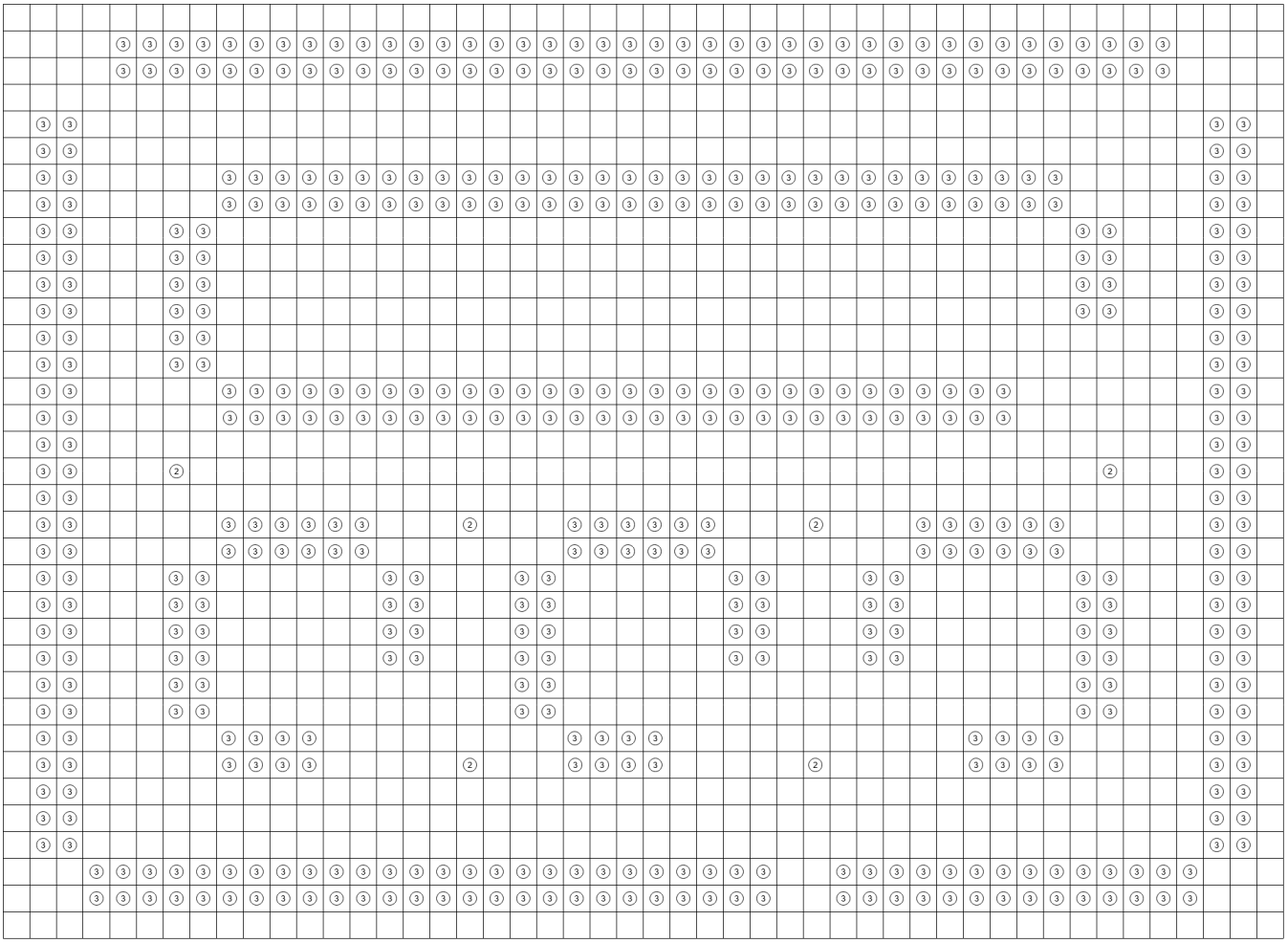


Figure 6.6

We see that the puzzle is very similar to the Masyu puzzle that Friedman uses. Instead of empty white pearls, we now have pearls with the number 3 for the walls, and the number 2 for the pearls in the rooms. In the following figure, we see a solution for the Shingoki puzzle in figure 6.6:

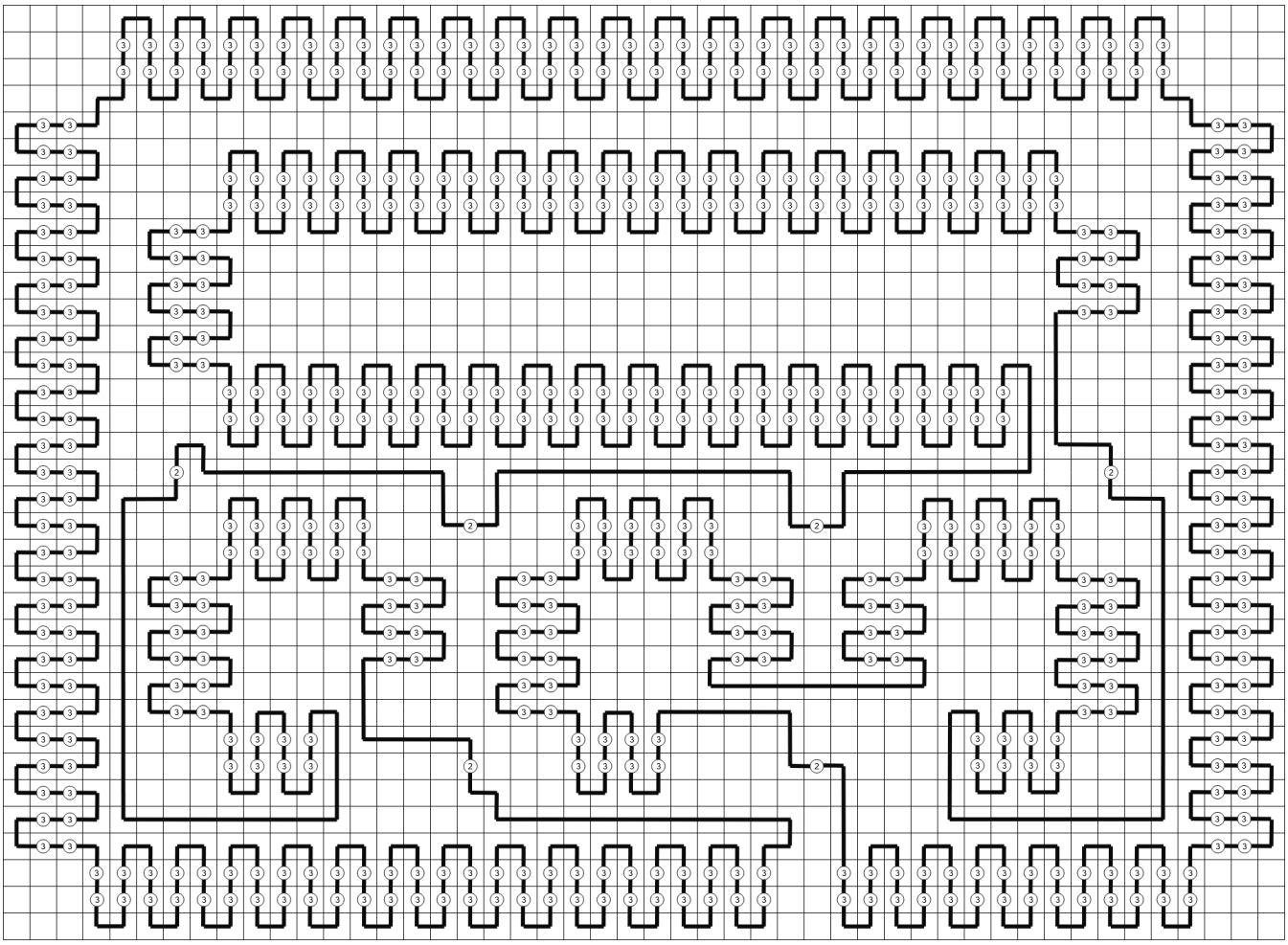


Figure 6.7

Friedman cites a paper by de Fraysseix, Pach, and Pollack [9] to state that a planar graph with n vertices can be realized using $O(n^2)$ area. That means that the mapping we created is polynomial.
To complete the proof, we have to show that a possible solution of a Shingoki puzzle can be checked in polynomial time. This is the same as for Masyu, so we use the same argumentation as Friedman. We need to check if the line goes through each pearl correctly. This can be done in $O(n^2)$, since the number of pearls is at most n^2. Furthermore, we need to check that the line consists of a single loop. This can be done in $O(n^4)$, because the number of possible lines is at most $O(n^2)$, and it suffices to check all the pairs of those lines. This completes the proof that the Shingoki puzzle is NP-complete.

Bibliography

[1] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19):1145 – 1146, 2009.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, 2017. Available at `www.SMT-LIB.org`.

[3] C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. 2018.

[4] Wikipedia contributors. Hashiwokakero, 2020. Online; accessed 19 November 2020.

[5] Wikipedia contributors. Hitori, 2020. Online; accessed 18 November 2020.

[6] Wikipedia contributors. Masyu, 2020. Online; accessed 17 November 2020.

[7] Wikipedia contributors. Nurikabe, 2020. Online; accessed 4 December-2020.

[8] Wikipedia contributors. Slitherlink, 2020. Online; accessed 12 November 2020.

[9] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[10] E. Friedman. Cubic is NP-complete. *preprint*, 2001.

[11] E. Friedman. Corral puzzles are NP-complete. *Unpublished manuscript, August*, 2002. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.579&rep=rep1&type=pdf`.

[12] E. Friedman. Pearl puzzles are NP-complete. *Unpublished manuscript, August*, 2002. `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.3324&rep=rep1&type=pdf`.

[13] E. Friedman. Spiral galaxies puzzles are NP-complete. *Unpublished manuscript, March*, 2002. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.6267&rep=rep1&type=pdf.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[15] M. R. Garey, D. S. Johnson, and R. E. Tarjan. The planar hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976.

[16] R. A. Hearn and E. D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

[17] G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[18] J. Kölker. Selected slither link variants are NP-complete. *Journal of information processing*, 20(3):709–712, 2012.

[19] B. McPhail and J. Fix. Nurikabe is NP-complete. In *Poster Session of the 6th CCSC-NW Annual Northwestern Regional Conference of the Consortium for Computing Sciences in Colleges*, 2004.

[20] D. Westreicher. Slitherlink reloaded. 2011. https://david-westreicher.github.io/static/papers/ba-thesis.pdf.

[21] T. Yato. On the NP-completeness of the slither link puzzle. 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL74-3.pdf.

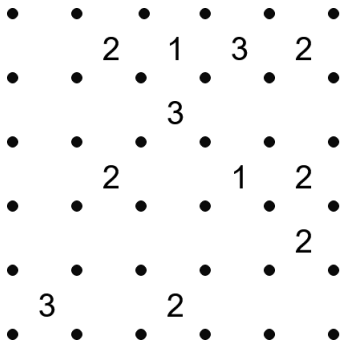
[22] H. Zantema and S.J.C Joosten. Latin squares with graph properties. https://www.win.tue.nl/~hzantema/lsqj.pdf, 2015.

Appendix

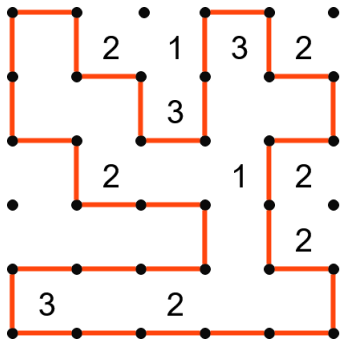
In this appendix, the reader can find some puzzles that are generated during this thesis. The puzzles have different sizes. The bigger puzzles are more difficult to solve. The solutions are also given.

A.1 Slitherlink

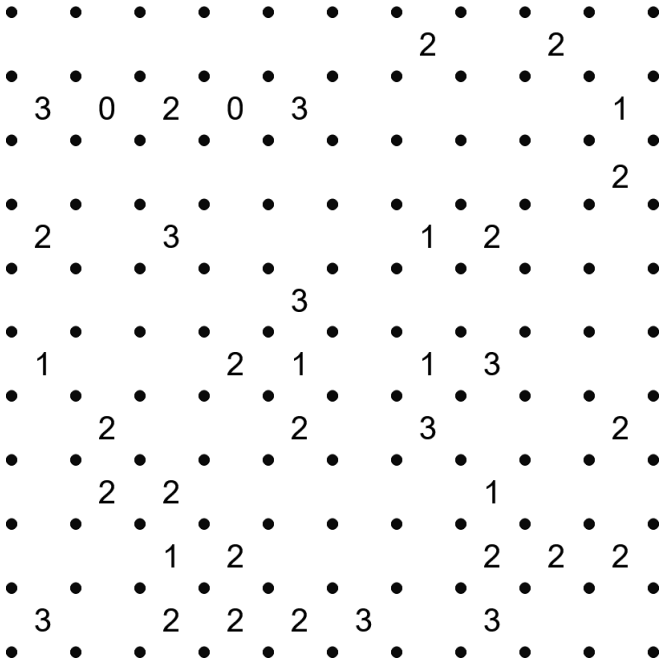
Slitherlink 5x5



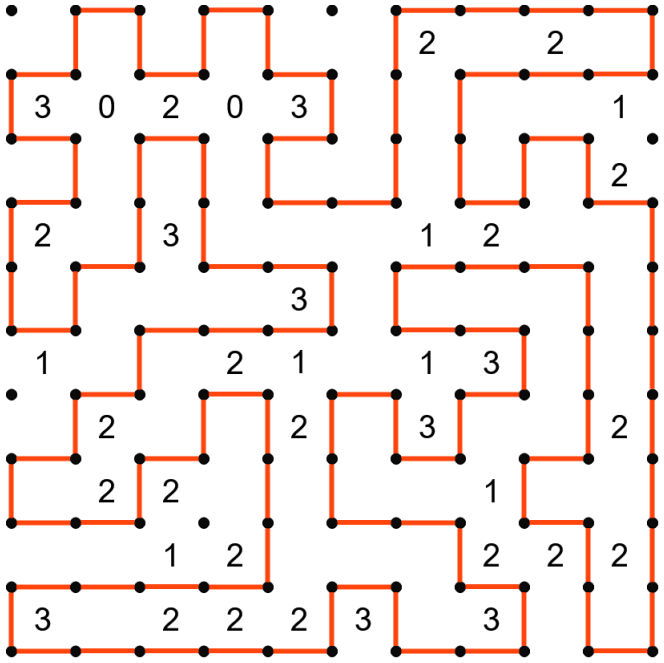
Solution Slitherlink 5x5



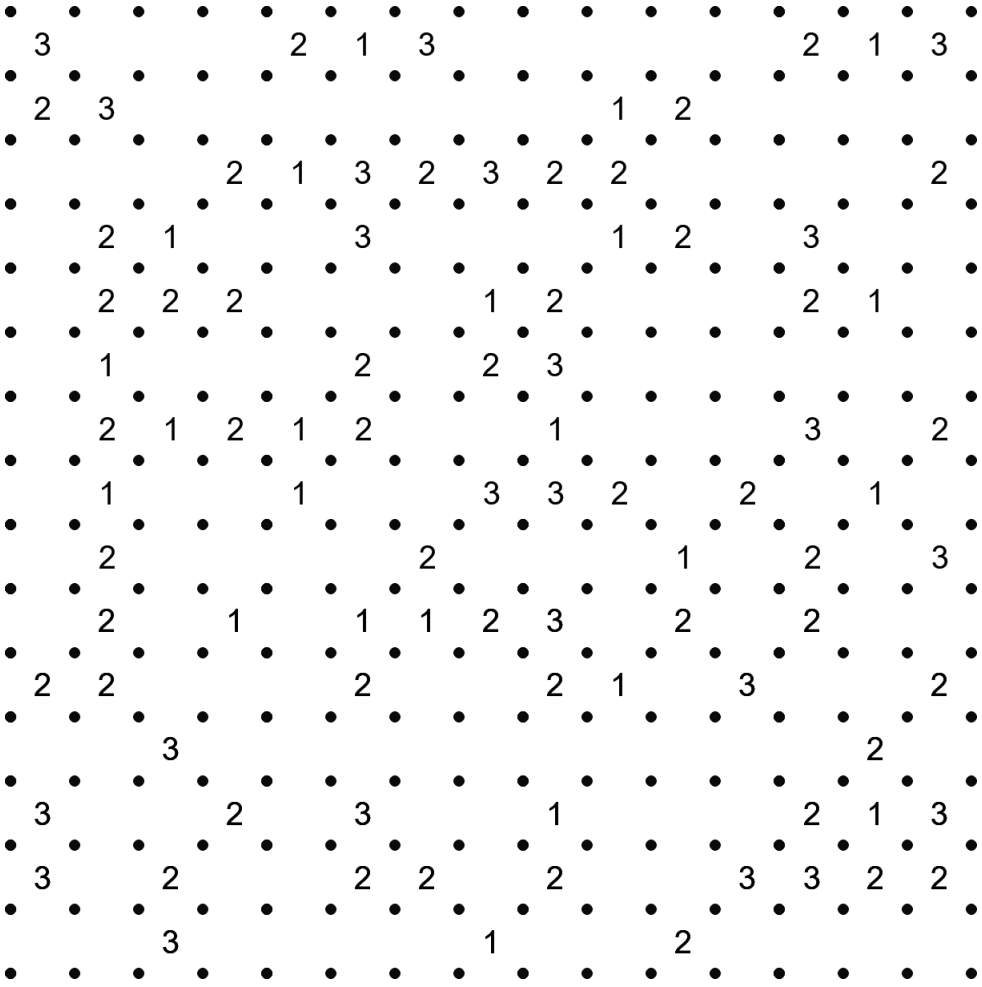
Slitherlink 10x10



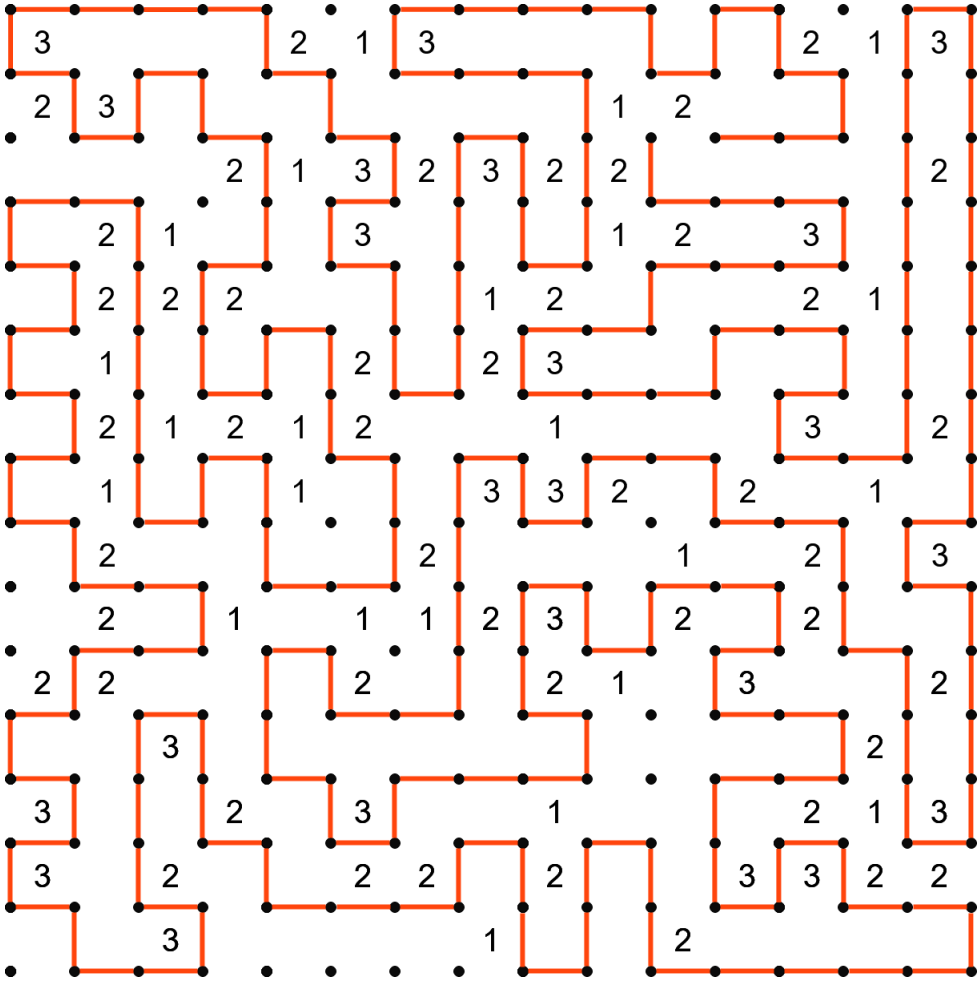
Solution Slitherlink 10x10



Slitherlink 15x15

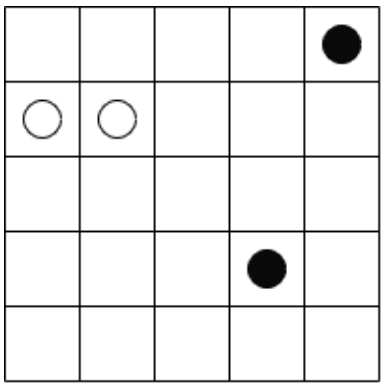


Solution Slitherlink 15x15

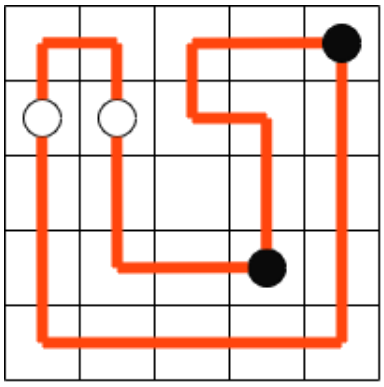


A.2 Masyu

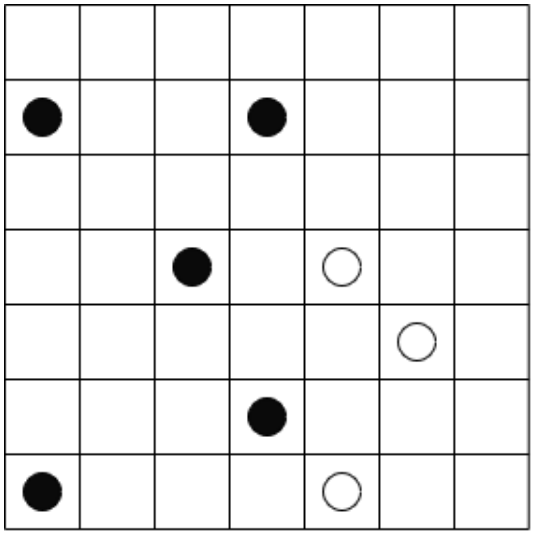
Masyu 5x5



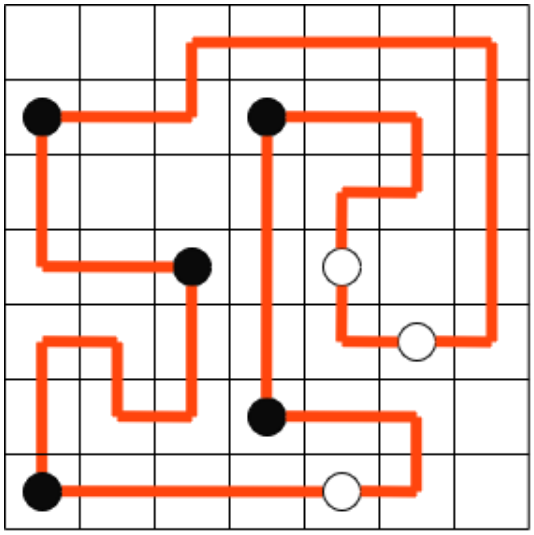
Solution Masyu 5x5



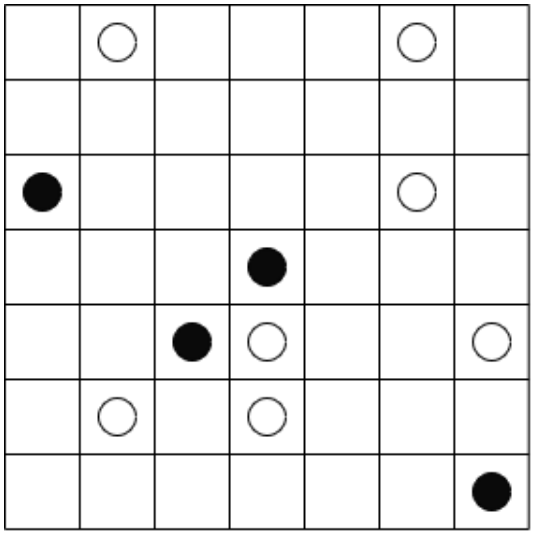
Masyu 7x7



Masyu 7x7 Solution



Masyu 7x7 - 2



Masyu 7x7 - 2 Solution