Opleiding Informatica

Solving and generating

Fobidoshi puzzles

Niels Heslenfeld

Supervisors:
Hendrik Jan Hoogeboom & Dalia Papuc

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

16/07/2024

Abstract

Fobidoshi is a Japanese logic puzzle. In this thesis, a backtracking solving algorithm for Fobidoshi is implemented that uses frequently occurring patterns of the puzzle to improve the algorithm. Using this solving algorithm, two different algorithms are constructed that can generate new Fobidoshi puzzles. The first algorithm randomly generates starting configurations and the second algorithm randomly generates solutions and then tries to decrease these to starting configurations. Experiments are conducted that show that the detection of patterns significantly improves the solving algorithm and that the generating algorithm that randomly generates starting configurations is more successful than the other generating algorithm. Also, this thesis describes how a Satisfiability Modulo Theories (SMT) instance can be created for a Fobidoshi puzzle and how this instance can be solved using an SMT solver.

Contents

1 Introduction **1**
1.1 Related work . . . 1

2 Description of Fobidoshi **3**

3 Solving Fobidoshi puzzles **5**
3.1 Storing puzzle configurations . . . 5
3.2 Backtracking . . . 5
3.3 Improvements to the solving algorithm . . . 7
3.3.1 Barriers of crosses . . . 7
3.3.2 Detecting good moves . . . 9
3.3.3 Empty cells with neighbouring circles . . . 11

4 Generating Fobidoshi puzzles **13**
4.1 Randomized starting configuration . . . 13
4.2 Randomized solution . . . 15
4.2.1 Randomizing solutions . . . 16
4.2.2 Preventing loops . . . 18
4.2.3 Reducing solutions to starting configurations . . . 20

5 SMT **21**
5.1 Solving Fobidoshi puzzles . . . 21

6 Experiments **27**
6.1 Solving . . . 27
6.2 Generating . . . 29

7 Conclusions and further research **33**

References **35**

1 Introduction

Fobidoshi is a logic puzzle invented by Naoki Inaba. Inaba is a Japanese puzzle author who has created over 400 original puzzles [Ina] making him one of the most creative puzzle authors in the world. In Japanese, Fobidoshi means “Forbidden Four” which is a reference to one of the rules: no stripes of four or more connected circled cells are allowed. The aim of the puzzle is to connect all the circles on the playingfield. Every puzzle should have a unique solution which is enforced by the rules of the puzzle that will be explained in Section 2. An example of a Fobidoshi puzzle is given in Figure 1.

O		O	O
O			
			O
O	O		O

O		O	O
O	O	O	
	O	O	O
O	O		O

Figure 1: An example of a Fobidoshi puzzle and its corresponding solution.

This thesis contains two different methods for solving Fobidoshi puzzles. Section 3 discusses the approach to the construction of a backtracking algorithm and how this algorithm is later improved. On the other hand, Section 5 explains a completely different approach to solving by rewriting the constraints of the puzzle into a formula and then using a solver to find the solution. Two different methods for generating Fobidoshi puzzles are described in Section 4 and in Section 6, experiments are conducted to measure and compare statistics such as the runtime and the number of recursive calls of the algorithms. Section 7 concludes the thesis.

The research question for this thesis is: “How can Fobidoshi puzzles be solved and generated?”. This bachelor thesis is supervised by Hendrik Jan Hoogeboom and Dalia Papuc, both affiliated to the Leiden Institute of Advanced Computer Science (LIACS).

1.1 Related work

A hot topic in the world of research is “combinatorial game theory”. This is defined as the mathematical study of combinatorial games – typically two-player games. One-player combinatorial games like Sudoku, Tetris, and Fobidoshi are also combinatorial games. To make a clear distinction between one-player and two-player games, the first are referred to as puzzles. There exist many different puzzles of which only a small part has been studied so far.

Fobidoshi is an example of one of the puzzles that, to the best of our knowledge, has not been studied before which is why it is chosen for this thesis. However, this does complicate the search for related work. Luckily, there are many games that are, in some way, similar to Fobidoshi as they also have a connectivity constraint. Games like these have been researched before by, for example, Van

der Knijff [dK21]. Van der Knijff studied 6 games with a connectivity constraint. For each of these games, Van der Knijff constructed an instance of the "Satisfiability Modulo Theories" problem. This has been the inspiration for Section 5.

Zantema and Joosten [ZJ15, p.7] proposed a method for enforcing the connectivity constraint. Van der Knijff creates SMT instances using the SMT-LIB syntax [BFT21] in which this method is implemented. The SMT solver Z3 [dMB08] is used to solve these instances. In Section 5, these techniques and theories are applied to Fobidoshi.

Projects like this one have been previously done by other students from Leiden University – like Meili Hegeman [Heg22] and Rob Mourits [Mou22] – for different puzzles. The project from Rob Mourits titled "Solving and Generating the Nurimeizu puzzle" is used for some inspiration for this project and to get some ideas about the possible structure of the thesis.

2 Description of Fobidoshi

Fobidoshi is a game that is always played on an $N \times N$ playingfield with $N > 3$. Cells in the playingfield are either empty or filled with a circle or a cross. In the introduction, it is mentioned that the aim of Fobidoshi is to connect all the circles on the playingfield where, for each puzzle, a unique solution has to exist that is enforced by the rules of the puzzle. Below, the two rules for Fobidoshi are given:

- All circled cells in the playingfield must form an orthogonally contiguous area. This is the case when one can travel from any circled cell to any other circled cell only traversing over circled cells without diagonal moves.
- It is not allowed to have horizontal or vertical stripes of more than three adjacent circled cells.

Since the game only has two basic rules, one could think that this is an easy game. However, nothing is further from the truth. With only two rules, there are not many possibilities at each point in time and ruling out certain moves can turn out to be quite challenging as well. Thus, looking ahead is often required when trying to solve Fobidoshi puzzles. An example of the manual solving process of a Fobidoshi puzzle is given in Figure 2.

O		O	O
O			
			O
O	O		O

(a) Starting configuration.

O	O	O	O
O			O
O			O
O	O	O	O

(b) Incorrect moves.

O	X	O	O
O			X
X			O
O	O	X	O

(c) Possible solving approach.

O	X	O	O
O	O	O	X
X	O	O	O
O	O	X	O

(d) Solution of the puzzle.

Figure 2: An example of a Fobidoshi puzzle and the process of solving it. Green symbols are correct moves, red ones are incorrect moves.

The starting configuration is given in Figure 2a. All symbols – in this specific case only circles – present in this starting configuration cannot be modified. The cells of interest are the empty cells. In each of these cells, one can place a circle or a cross. Placing crosses in a cell only functions as a tool to show that this cell should not contain a circle. Figure 2b shows where circles cannot be placed without violating the second of the two rules mentioned above. Having determined this, Figure 2c shows how one could approach this puzzle. By first placing crosses in all cells that should not be filled with a circle, some possible moves are eliminated leaving less room for error. The corresponding solution is given in Figure 2d.

Of course, this example shows one of the easiest puzzles out there but it is a good way to get familiar with the puzzle and its rules. In this thesis, puzzles ranging from $N = 4$ until $N = 12$ are used. As the size increases, the number of possible configurations grows exponentially. All of this (and more) will be explained in the rest of the thesis.

3 Solving Fobidoshi puzzles

Writing a solving algorithm for a puzzle can be done in many different ways and by using different types of algorithms. In many cases, a backtracking algorithm is used for this purpose. Backtracking is often used in combination with brute force, a method that tries all possible combinations until a valid solution is found. As one can imagine, brute force is in many cases not the most efficient way of solving a problem. This part of the thesis describes how a backtracking algorithm, that is an improved version of brute force, capable of solving Fobidoshi puzzles, is implemented. This algorithm is written from scratch in the C++ programming language.

3.1 Storing puzzle configurations

Since the goal of the algorithm is to find the solution to a provided Fobidoshi puzzle in the shortest amount of time possible, it is necessary to think of a way to "feed" starting configurations of puzzles to the C++ program and a way to store these configurations. Providing puzzle configurations by hand takes a lot of time which is not the intention. It turns out that the already available puzzles on the website from Angela and Otto Janko [JJ] are stored in a fairly simple way that can also be used in this case. Every empty cell is represented by "-", every circle is represented by "o", and every cross is represented by "x". Because of its simplicity, this representation is copied for storing puzzle configurations and then feeding them to the backtracking algorithm. A bonus is that this makes it possible to retrieve the configurations from the website in an automated way.

To solve a puzzle, a text file with contents in the form as described above is given to the program. The first line of these text files consists of one number, which is the size (N) of the playingfield (only one number is necessary since it is an $N \times N$ playingfield). This number is followed by $2N$ lines of information of which the first N lines contain the representation of the starting configuration and the last N lines the representation of its solution. The solutions are also retrieved from the website and are, for example, used for comparison with the solution found by the solving algorithm to see if they match. The program reads all the contents from such a text file and stores the information in two separate 2D-vectors. Vectors are used since the size of a vector does not have to be specified beforehand. A big advantage of this is that the program can first read the size of the playingfield from the text file and after that specify the size of the vector, making it a very convenient datastructure for this purpose. In each vector, the value at an index stores the type of character that is currently present on that cell.

3.2 Backtracking

Using the puzzle configurations stored as explained in the previous section, a backtracking algorithm is created to solve these Fobidoshi puzzles. First, it is important to note that puzzles of smaller sizes are significantly easier to solve than puzzles of larger sizes because the amount of possible configurations grows exponentially with the increase of the playingfield size. The approach to solving this problem is to first implement a very simple backtracking algorithm capable of solving the smallest and easiest puzzles and to then improve this algorithm until it can also solve the largest and hardest puzzles.

The first version of the algorithm is a very basic backtracking algorithm that, to some extent, uses brute-force. This algorithm works recursively and sequentially executes the following steps in each recursive call:

1. Check if the configuration in this recursive call meets the requirements of a solved puzzle. If so, return `true`.
2. Insert crosses in all the cells that "need" a cross. These are the cells having either a connected stripe of three circled cells on one of the four neighbouring sides or a connected stripe of two circled cells on one of the four neighbouring sides and at least one circled cell on the opposite neighbouring side. In other words, if placing a circle in a cell would create a stripe of more than three circled cells, this cell has to be filled with a cross.
3. Check if there is still an empty cell in the playingfield (iterating over all cells from top left to bottom right) and, if so, retrieve the coordinates of this cell and do two consecutive "guesses":
   - Fill this cell with a circle and call the function of this algorithm recursively. If the return-value of this recursive call is `false`, the algorithm backtracks by deleting the circle and doing a second guess. Otherwise, the solution is found and the algorithm returns `true`.
   - Fill this cell with a cross and call the function of this algorithm recursively. If the return-value of this recursive call is `false`, the algorithm will backtrack by deleting the cross and returning to a higher recursive level. Otherwise, the algorithm returns `true`.
4. If `true` is never returned, the function will return `false` by default.

As one can see from this, the algorithm is not entirely brute-force since it already uses a pattern to cut-off parts of the state space tree of the puzzle. Cutting is done by inserting crosses whenever possible since a cross in a cell can never be overwritten by a circle. The first big advantage of this is that quite a lot of faulty configurations are ignored. However, a bonus is that this also eliminates the need to, in each recursive call, check if the current configuration is still a valid configuration – in the sense that it should not contain horizontal or vertical stripes of more than three connected circled cells. Inserting crosses already eliminates the possibility of a configuration like that ever occurring.

Figure 3 shows the process executed by this backtracking algorithm. The starting configuration is shown at the top left. For this starting point, the backtracking algorithm is called. The algorithm will execute the enumeration above until it reaches the solution or until all possible configurations have been tried and no solution is found. Backtracking will happen when an incorrect configuration is reached. An example of this is given at the top right. For this configuration, no possible moves are left even though the circle in the bottom left cell is not connected to the rest of the circles. Thus, the algorithm will backtrack. After backtracking and reaching incorrect configurations a couple of times in a row, the algorithm will eventually backtrack to the configuration shown at the bottom right. The symbol in the yellow cell is changed from a circle to a square. The algorithm then expands this configuration to the solution shown at the bottom left.

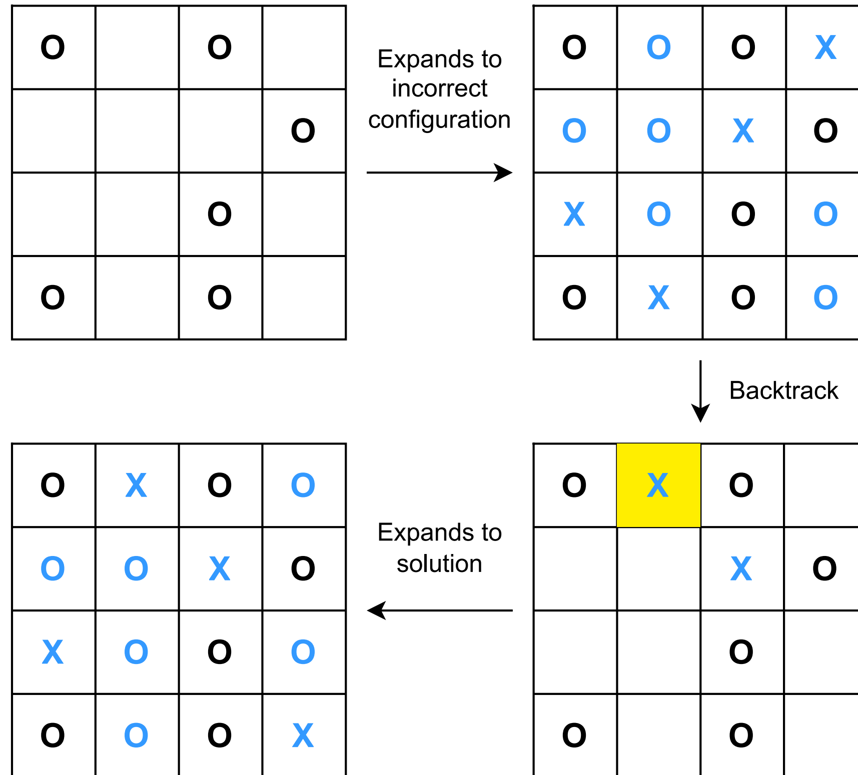


Figure 3: Visualisation of the steps executed by the backtracking algorithm.

Testing this algorithm only on the simplest and smallest puzzles gives promising results as it never takes longer than seconds to find the solution. Nevertheless, for puzzles with $N \geq 7$, this algorithm is already struggling to get to the solution in under a minute. This is not odd, since this backtracking algorithm is still very close to “brute forcing” its way to the solution and because it is only designed for the easiest puzzles, for which it serves its purpose. With the aim being to solve even the largest and hardest puzzles on the website from Angela and Otto Janko [JJ], this algorithm must be improved.

3.3 Improvements to the solving algorithm

On paper, Fobidoshi is quite a simple puzzle with only two basic rules. What is challenging, however, is to find specific patterns of interest that occur regularly while solving a Fobidoshi puzzle. For example, being able to say for sure that a certain cell should be filled with a cross depends on some patterns. Besides these, there are also three other patterns that occur relatively often. All of these patterns are related to the rule that all circles in the playingfield must be orthogonally connected.

3.3.1 Barriers of crosses

The first pattern of interest is when two (clusters of) circles are cut-off by crosses. These crosses could either be known to be correct for that specific configuration or they could simply be part of a

randomly generated configuration. An example of a barrier of crosses is given in Figure 4. If such a barrier exists at any moment in the process of solving the puzzle, it can be concluded that this configuration does not have to be explored any further since it is impossible to connect these two cut-off parts at any future configuration derived from it. As a result, the connectivity constraint is left unsatisfied. In this case, the algorithm goes back to a configuration where no such barrier exists – this is the main characteristic of backtracking.

	X	O	O	O	X
X	O	X	O	O	X
X	O	O	O	X	
	O	X	X		O
X	X	O	X	O	
O		O			O

Figure 4: Puzzle configuration with a barrier of crosses.

In order to perform backtracking in such a case, these patterns first have to be recognised. A way to do this is to "walk over clusters of crosses" and to see if such a cluster has a cross placed in a row or column at one of the edges of the playingfield and another cross placed in a row or column at one of the other edges or the same edge (having at least one cell in between) of the playingfield. Although the theory behind this is quite straightforward, converting this to code is not. Meeting the requirements specified above does not necessarily mean that there is indeed a barrier of crosses that actually has circles on both sides of the barrier. In addition to that, the constraint as mentioned above does not detect all possibly present barriers since barriers do not need to have a cross in a row or column on one of the edges of the playingfield. Two examples of this are shown in Figure 5. For both examples it holds that, when present in the center of the playingfield without touching any of the edges, the circled cell in the center cannot be connected to potential other circled cells in the playingfield.

X	X	X
X	O	X
X	X	X

	X	
X	O	X
	X	

Figure 5: Two examples of a barrier of crosses that could be present in the center of a playingfield.

Another possible method to recognise barriers of crosses depends on a different principle. In the previous method only information of filled cells is considered. For this new method, all the empty cells in the playingfield are also of interest. These empty cells have the possibility of being filled with a circle in the solution – of course they can also be filled with a cross but for this method only the possibility of them being filled with a circle is relevant – while cells with a cross in them do not. As non-connected (clusters of) circles in a configuration will always have to be connected in a future configuration derived from it, one can conclude that this connection has to be established using the empty cells. Thus, if treating all empty cells as circles still does not connect all circles already present in the playingfield, it can be concluded that there is a barrier of crosses.

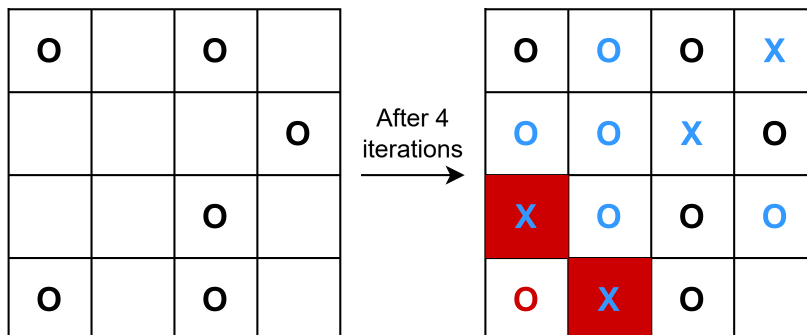


Figure 6: An example of a formed barrier of crosses.

An example of such a case is given in Figure 6. This figure shows that, for the starting configuration on the left, the algorithm reaches the configuration on the right after four iterations of the backtracking algorithm. In this configuration, the circle on the bottom left of the playingfield (colored red) is cut-off by the two crosses in the red colored cells – one above it and one to its right. Thus, this configuration cannot be expanded to the solution of this puzzle.

The algorithm uses the method specified in the paragraph above to recognise such cases. Every time a cross is placed in the playingfield, the algorithm performs a barrier check. This check starts by iterating over the playingfield (from top left to bottom right) until it finds a circle. Starting at this circled cell, it will perform the Flood Fill Algorithm to visit all connected circled cells and empty cells in a Depth-First Search (DFS) manner filling them with the symbol "$*$". This symbol has no function besides marking a cell as visited. After this process is finished, the playingfield is again searched for a circle. If a circle is found, it means that there is a barrier of crosses present in the playingfield, therefore `true` is returned. The algorithm will then backtrack to a state where the barrier is not present anymore.

3.3.2 Detecting good moves

The second pattern of interest is related to the pattern of existing barriers. However, this pattern is not so much about existing barrier detection but more about barrier prevention. In some cases, barriers can be prevented by checking whether placing a cross in a certain cell would form a barrier. Such a situation can occur when there is a circled cell with only one empty neighbouring cell. If placing a circle in the empty neighbouring cell is the only possible way to connect this circled cell

(and, potentially, its connected cluster) to the rest of the circled cells, then the empty neighbouring cell must be filled with a circle.

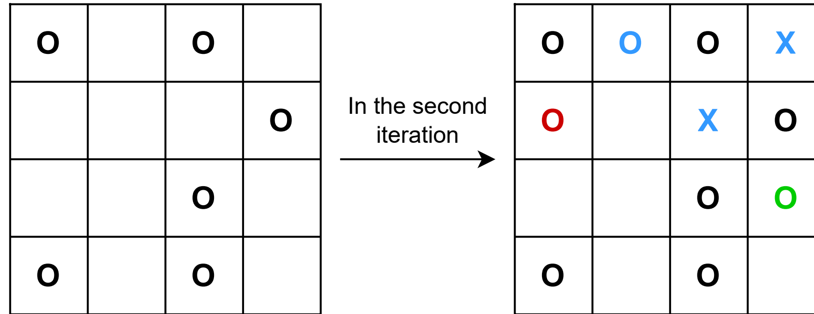


Figure 7: An example of a puzzle for which a good move exists.

Figure 7 shows such a situation for which this is the case and a “good move” exists. The red-colored circle and the green-colored circle are two alternative moves. The red-colored circle shows the standard guess and the green-colored circle the good move. Without checking for the good move pattern, the algorithm will, when starting with the configuration on the left, place the red-colored circle in the second iteration (after placing the cross in the top right cell). Implementing good move detection will, instead of placing the red-colored circle, place the green-colored circle. This is a good move because substituting the green-colored circle by a cross would form a barrier as the black-colored circle in the right column will be cut-off from the rest of the circles. Thus, this cell must not contain a cross and therefore placing a circle in this cell is a good move.

However, it turns out that this is not the only possibility. Barriers could potentially also be formed somewhere in the middle of the playingfield. In this specific algorithm, this will not happen but, since it would still be valid behaviour, it does offer the possibility of detecting such a pattern beforehand. So, for example, say there is an empty cell – this could be any empty cell in the playingfield – that, when being filled with a cross, would form a barrier that cuts-off unconnected circles, then this cell has to be filled with a circle. This includes the situation explained above but, since it is a more general approach, it also holds for many more situations.

For humans, this is a relatively easy pattern to spot. Nevertheless, it is slightly more complicated to implement a check for this pattern in code. The method that is used here is to iterate over the playingfield searching for empty cells. When an empty cell is found, this cell is temporarily filled with a cross. Subsequently, the playingfield is checked for any existing barriers (good to note, this check was also performed before starting the process of this optimisation). If the playingfield suddenly contains a barrier of crosses, it is certain that this barrier was formed by filling the previously empty cell with a cross. Knowing that this cell should not be filled with a cross, it has to be filled with a circle. Therefore, this is a move that is guaranteed to be good for the current configuration. In the case that no barrier is formed, the cell will be emptied again after which the same process is executed for the next empty cell until every cell in the playingfield has been checked. Figure 8 shows the steps of this process starting (top left) with the configuration that is present at that point and ending (bottom left) with a detected good move.

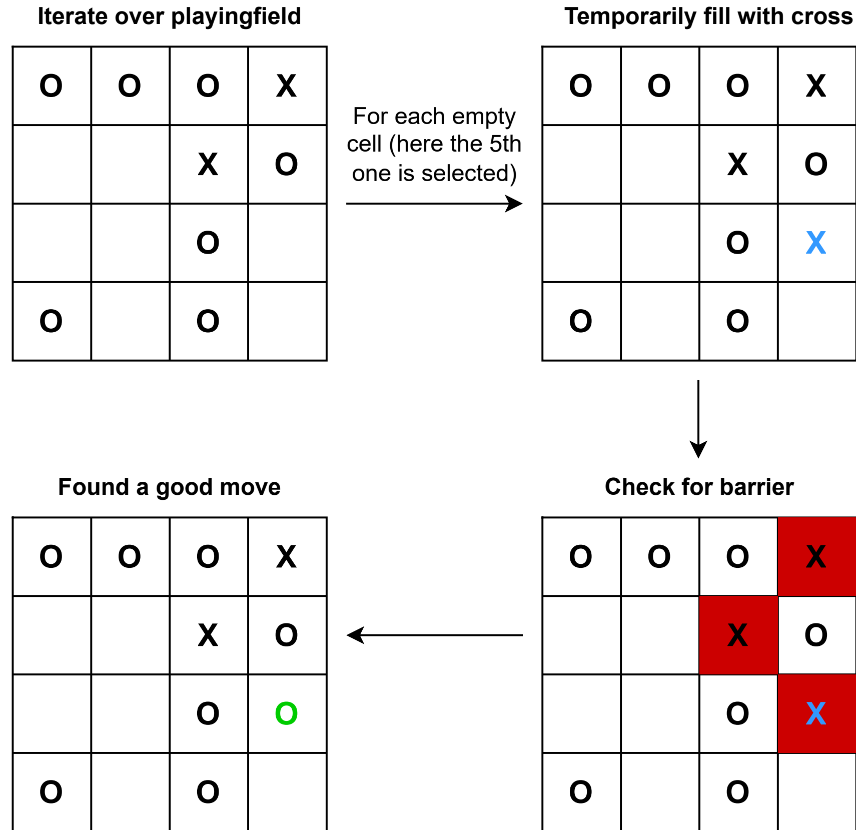


Figure 8: An example of the process of checking if a good move exists.

One could be fooled to think that, having implemented this, the need for barrier checks is eliminated because the general approach is to prevent barriers from being formed. This is, however, not true. In some cases, placing a circle in one cell to prevent a barrier from being formed can enforce a cross to be placed in another cell which does create a barrier. The barrier check will then detect this barrier in the next iteration of the algorithm. Therefore, the barrier checks as explained in Section 3.3.1 will stay present.

3.3.3 Empty cells with neighbouring circles

The two previously explained improvements both, in some way, make use of barrier forming, which is something that occurs quite often in Fobidoshi puzzles. This third improvement, however, does something completely different. Step three of the backtracking algorithm – as explained in the enumeration in Section 3.2 – performs a search for the first empty cell in the playingfield by iterating over all cells from top left to bottom right. Thus, this operation is using brute-force to select an empty cell.

As mentioned earlier, brute-force is not often the optimal way of solving a problem. To remove part of the brute-force, an improvement is implemented for selecting an empty cell in this step of the

backtracking algorithm. This improvement uses the connectivity constraint. Since, in the solution, all circles have to be connected to each other, it suffices to only consider empty cells that have a circle in one of their neighbouring cells – instead of all empty cells. This way, the backtracking algorithm will never introduce unconnected (clusters of) circles that, after some iterations, can turn out to be wrong. Of course, only considering cells that have circles in one of their neighbouring cells is far from guaranteed to always result in a good move but it does eliminate quite some possibilities and, because of that, increases the chances of selecting a useful cell.

To implement this improvement, the algorithm keeps track of the number of neighbouring circled cells for each cell in the playingfield. When getting to step three of the backtracking algorithm from Section 3.2, the algorithm will still iterate over all the cells of the playingfield in the same way. However, when finding an empty cell, it now checks its number of neighbouring circled cells. If this number is larger than 0, this cell is selected. Otherwise the algorithm will continue iterating over the playingfield until it finds an empty cell that does have a neighbouring circled cell. If such a cell does not exist, no circles can or should be placed anymore and thus the algorithm returns `false`. This last case should, in practice, never occur in the third step of the backtracking algorithm since it is already handled in an earlier step.

With this improvement, empty cells are not anymore selected sequentially but with some form of heuristic which increases the chance of selecting a useful cell faster than without this improvement. This is, however, not something that has a massive impact on the efficiency of the algorithm but it can eliminate part of the incorrect configurations that would be visited otherwise. For that reason, this is a third improvement of the backtracking algorithm.

4 Generating Fobidoshi puzzles

Using the algorithm constructed for solving Fobidoshi puzzles, it is also possible to generate new ones. Even though solving and generating are tasks that are related to each other – as one cannot generate puzzles without using a solving algorithm – they require different approaches. For example, when solving an already existing puzzle known to have a unique solution, one can use this knowledge to create an optimised solving algorithm. This algorithm can contain some small tricks that decrease computation time by cutting-off partial solutions that are unable to be expanded to the solution (as explained in Section 3.3). This way, the algorithm will find the solution much faster than when it tries to brute-force its way to the solution. Then, after finding the solution, the algorithm can immediately exit since all possible configurations following it are irrelevant.

When generating Fobidoshi puzzles, one has to be able to generate a valid starting configuration that is solvable in one unique way. This means that, after having generated a possible starting configuration, the solving algorithm has to be applied to this puzzle. Though this time, the solving algorithm cannot exit once it has found a solution since now the number of possible solutions is not known and has to be determined. For that reason, it is important that the solving algorithm is implemented efficiently and that the generating algorithm does not add too much complexity to this because, in many cases, the solving algorithm is given much more work than when it would simply try to solve a puzzle that has a unique solution. This section explains how these problems are tackled using two different approaches.

4.1 Randomized starting configuration

The first approach to generating Fobidoshi puzzles is to start with an empty $N \times N$ playingfield and, from there, work towards a possible starting configuration. As has become standard by now, the theory sounds straightforward while the practice is not. In this case, the problem is that starting configurations do not really have to adhere to many rules other than the standard rules already mentioned in Section 2. There is, for example, no standard measurement for the amount of circles that have to be present or for the locations of these circles, all of which sounds quite random. Thus, having noticed that randomness seems to play a big part, it seems fitting to also use randomization for creating possible starting configurations. This is the first method that is studied.

To randomly generate starting configurations of Fobidoshi puzzles, an algorithm is written that is only supplied the desired size of the playingfield (N). The algorithm then initializes an $N \times N$ empty playingfield. The next step is to randomly add circles to the playingfield meaning that a (semi-)random percentage of randomly picked cells is filled with circles while keeping into account the basic rules of the puzzle – no vertical or horizontal stripes of more than 3 connected circles. However, since there are no extra rules that have to be taken into account when generating a starting configuration of a Fobidoshi puzzle, it is necessary to establish a stop-condition.

As mentioned, the goal is to fill a (semi-)random percentage of cells with circles. If a completely random percentage is chosen, it will probably become very hard to create an algorithm that generates valid starting configurations because for many percentages, generating a starting configuration is close to impossible. Complete randomness is therefore not an option. Luckily, there are quite

some examples of already generated starting configurations available online. By looking at the 240 puzzles available on the website from Angela and Otto Janko [JJ], for example, it is possible to get an idea of some statistics that could be helpful for determining a stop condition. It turns out that, for these 240 puzzles ranging from $N = 4$ to $N = 12$, the average percentage of filled cells – circles and/or crosses – in the starting configuration is 38.2691%. However, the starting configuration containing the least amount of filled cells is just 22.2222%.

When only looking at the circles present in the starting configuration, it appears that the average percentage of filled cells drops to 34.2309% with a minimum percentage of 17.1875%. The minimum percentage gives a distorted picture due to some very strategically placed crosses that eliminate all possible solutions that would exist in their absence but one. Nevertheless, percentages like these do have an influence on the average percentage that could, as a result, turn out to be lower. Since the algorithm will just be placing circles in the playingfield, only the average percentage of circle filled cells is used as a reference for the stop-condition. This is done by taking 29% as a starting point and then adding a random number between 0 and 12 to it. The result of this is a percentage that lies somewhere between 29% and 41%, averaging around 35%. The average here is slightly higher than the average that is observed to compensate for the influence of puzzles with a very low percentage of circles, as mentioned before. Generating smaller puzzles, like 4×4 ones, becomes very hard otherwise.

After a possible starting configuration is created, the next step is to check if this configuration is indeed solvable and, if so, how many possible solutions exist for this puzzle. This means that the algorithm from Section 3.2 has to be applied to the puzzle. However, this time it has to try all possible configurations of interest – skipping configurations whenever possible, as explained in Section 3.3 – instead of exiting directly after finding a solution. Running this algorithm shows that the generated configurations are either solvable in many ways or not solvable at all. In exceptional cases, a configuration is generated that has a unique solution. If a starting configuration has no solution at all, it is useless. While on the other hand, starting configurations with one or more solutions are useful. The only obstacle is that a starting configuration can be approved only if it has a unique solution. Hence, a way has to be found to modify generated configurations with multiple solutions to configurations with only one possible solution.

Since starting configurations are allowed to have crosses in them and cells filled with crosses cannot contain a circle, this poses an elegant way of doing so. Though, this leads to the question in which cells to place crosses since placing a cross in some cells will reduce the amount of possible solutions while placing a cross in some other cells will not. The following process takes care of this problem.

First, the solving algorithm is slightly modified to store the solution containing the least amount of circles present in the playingfield. The idea behind this is that any solution including more circles than this stored solution could just be an expansion of the solution with the least amount of circles. As one can conclude, this would lead to multiple possible solutions because the extra circle(s) – with respect to the solution with the least amount of circles – could either be placed or left out which would lead to multiple valid solutions.

Therefore, the next step in the algorithm is to, for all empty cells and cells with a cross in the stored solution, copy a cross to the starting configuration. This leaves only one possible path to connect all circles in the puzzle: a unique solution. Obviously, there is no point in generating such a puzzle having only "correct" moves. Thus, the algorithm randomly deletes a cross that is present in the starting configuration and checks whether the starting configuration still has only one possible solution. If this is the case, the cross is left out. Otherwise, the removal of the cross will be made undone. This process is repeated until all crosses are tried. In general, this algorithm works for every starting configuration having multiple solutions but, naturally, some cases lead to more aesthetically pleasing configurations with better distributed circles and less crosses than others.

In the end, the generated configuration can be written to a file in the same format as the configurations from the website. Figure 9 shows an example of a puzzle that is generated by this algorithm. This puzzle can be solved by hand.

	O				X
O			O		O
O	O			X	O
O			X		
			O		O
O			O		

	O	O	O		X
O	O		O	O	O
O	O	O		X	O
O		O	X		
	O	O	O		O
O	O		O	O	O

Figure 9: Puzzle generated by the randomized starting configuration algorithm and its corresponding solution.

4.2 Randomized solution

The second approach to generating Fobidoshi puzzles is one that attempts to generate starting configurations without (or with close to no) crosses in them. The previous algorithm is only slightly trying to limit the amount of crosses present in a starting configuration which, sometimes, still leads to a configuration containing an unwanted large amount of crosses. Achieving this is more challenging because, as mentioned in the previous section, crosses can be used to eliminate possible solutions and are therefore very useful to make sure that there is only one possible solution. To overcome this problem, this algorithm works "the other way around". What is meant here is that this algorithm starts by generating a configuration that could be the solution to a starting configuration and then "reduces" this solution to a matching starting configuration.

Accomplishing this is quite challenging since a solution has to satisfy the connectivity constraint and, at the same time, it cannot have horizontal or vertical stripes of more than three connected circles. With a starting configuration known to have a unique solution, the process of reaching the

solution is straightforward as one can just place circles and crosses in a "trial and error" manner until arriving in the state of the solution. Randomly generating a configuration of a solution is, however, totally different.

Just like with the previous method, this algorithm only gets the desired size of the playingfield as input. The stop-condition is determined in a similar way as for the algorithm in Section 4.1. This time, however, the focus is on the average percentage of circles present in the solutions of the puzzles on the website of Angela and Otto Janko. After solving the 240 puzzles and importing their solutions, it turns out that the average percentage of circled cells in the solutions is 66.8944%, with a minimum of 57.8125%. It is clearly visible here that the percentage point difference between the average and the minimum value for the solutions is a lot smaller than with the starting configurations. Using this result, the same strategy as in the previous section is applied by, this time, taking 61% as a starting point and then adding a random number somewhere between 0 and 12. For this case, it would always result in a percentage that lies somewhere between 61% and 73% and averages around 67%.

Even though this is a useful stop-condition, it is not yet clear how to start with generating a solution and, thus, when to use this stop-condition. For generating a solution, it seems to be a good idea to start at a random position in the playingfield, add a circle to this cell and, from this starting point, "spread-out" over the rest of the playingfield placing circles in cells that are already connected to previously placed circles. The next section explains the development for this process as it has quite some snags.

4.2.1 Randomizing solutions

As said before, the goal is to fill the playingfield with circles beginning at a random starting position and then expanding from there, which is easier said than done. The first step of this algorithm is to add a circle to the randomly picked starting position in the playingfield. Subsequently, the x- and y-coordinate of this position are added to a list as one element – a pair. Thereafter, the algorithm enters a loop in which it executes the following steps as long as the list of coordinates still contains elements:

1. Generate a random index of the list with pairs of x- and y-coordinates. Access this element, retrieving and storing the coordinate pair (x,y) and then deleting the element from the list.
2. Loop over all adjacent neighbours of this coordinate pair (above, right, below, left) and, for each neighbour, check if its corresponding cell is empty.
3. If empty, a circle is placed in this cell in 90% of the cases – this percentage is referred to as the threshold parameter. This, by itself, enforces some form of randomness.
4. If a circle is placed, a pair of the x- and y-coordinate of this cell is stored as an element in the list. After that, crosses will be inserted in all the cells that "need" a cross (in the same way as with the algorithm in Section 3.2).
5. Check if the configuration has fulfilled the stop-condition and, if so, store the solution and exit the algorithm with `true` as return value.

A visualisation of the steps from the enumeration above is shown in Figure 10. In this figure, the leftmost playingfield shows the configuration when entering the loop for the first time. All four neighbouring cells have a 90% chance of being filled with a circle. The second playingfield from the left shows the configuration after the first iteration of the loop. Three out of four neighbouring cells have been filled with a circle and now their empty neighbouring cells have the potential of being filled with a circle. In the end, the configuration on the right is reached.

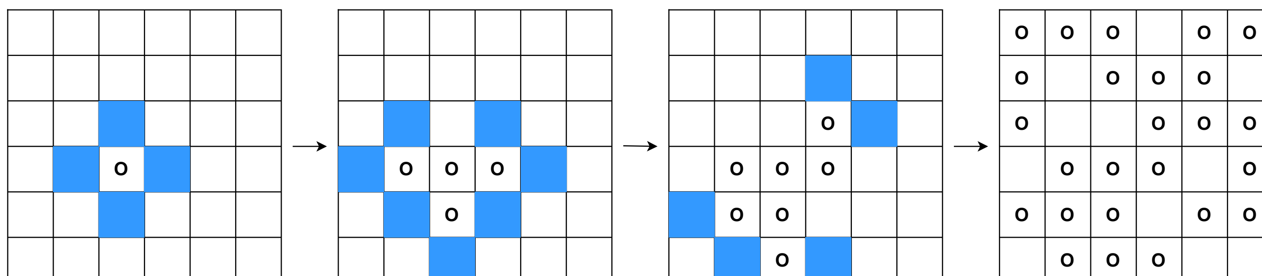


Figure 10: Visualisation of the steps executed by the randomized solution algorithm. Blue cells are cells that have the potential of being filled with a circle in the corresponding iteration of the algorithm.

When running the algorithm, it appears that there are different sorts of outcomes. In some cases, the algorithm quickly generates a solution while in other cases this process takes a while. What stands out is that a lot of the solutions contain loops of circles and that some solutions are ambiguous. To clarify, a loop of circles in this case means that there is a connected cluster of circles that encapsulates one or more non-circled cells in the playingfield. Ambiguity is present when solutions still contain empty cells in which circles can be "legally" connected to the rest of the circles, sometimes this also introduces a loop. Having loops in a solution can be valid when enforced by the starting configuration but, if this is not the case, a loop leads to ambiguity since there is always the possibility to take out one of the circles in such a loop without breaking the connectivity constraint. Hence, generating solutions becomes a bit more complicated than initially anticipated. Section 4.2.2 explains how this problem is tackled.

For the case where a solution still contains empty cells in which circles can be "legally" connected to the rest of the circles, the algorithm is extended so that it, after exiting the loop in which the enumeration from above is executed, walks over all empty cells and adds a circle if possible or a cross if necessary. These crosses are forced to stay present in the starting configuration. If one were to leave such a cross out in the end, it can again result in an ambiguous puzzle as it could be solved by placing a circle in this cell as well as by not placing a circle in this cell.

In the meantime, the algorithm also has to be sped up and, perhaps, be a bit more random than it currently is. With one little tweak to the algorithm, both of these problems are tackled at once. In the enumeration above, the four neighbours of circle filled cells are checked in the same order every time (above, right, below, left). An undesired side issue of this is that two runs of the algorithm that, by accident, have the same random starting position and random stop-condition have a fair chance of generating the same result. At this point, the only randomization comes from the first and third step as described in the enumeration at the start of this section.

To improve this, some more randomization has to be introduced. This is done between the first and the second step of the enumeration that is at the start of this section. Instead of having a list of the neighbouring cells that is always in the same order (above, right, below, left), the order is randomized so that every neighbouring cell has the same chance of being the first visited cell. This really is a significant change because, in many cases, it can happen that when a circle is placed in one neighbouring cell, a cross can (or has to) be placed in another neighbouring cell which prevents it from being initialised with a circle. Now that the order in which the neighbouring cells are visited is randomized, the possibility of something like this happening to some cells more often than to other cells is eliminated. This extra randomization appears to be a valuable improvement because distinct solutions are generated more often than before and because the algorithm is quicker with generating valid solutions.

4.2.2 Preventing loops

Preventing loops from being formed is one of the biggest problems to be solved for generating random solutions. The conception is that loops are undesired and, for that reason, must be avoided. As said earlier, loops can lead to ambiguity when not enforced by the starting configuration since there is always the possibility to take out one of its circles without unsatisfying the connectivity constraint. Loops come in many different shapes, an example is given in Figure 11. When this loop of connected circled cells is present anywhere in a playingfield and one is allowed to delete one of these circles, the connectivity constraint remains satisfied. This means that there are multiple valid solutions to the puzzle.

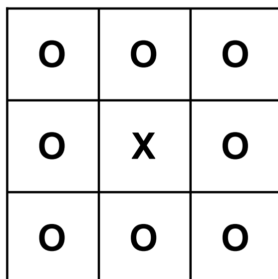


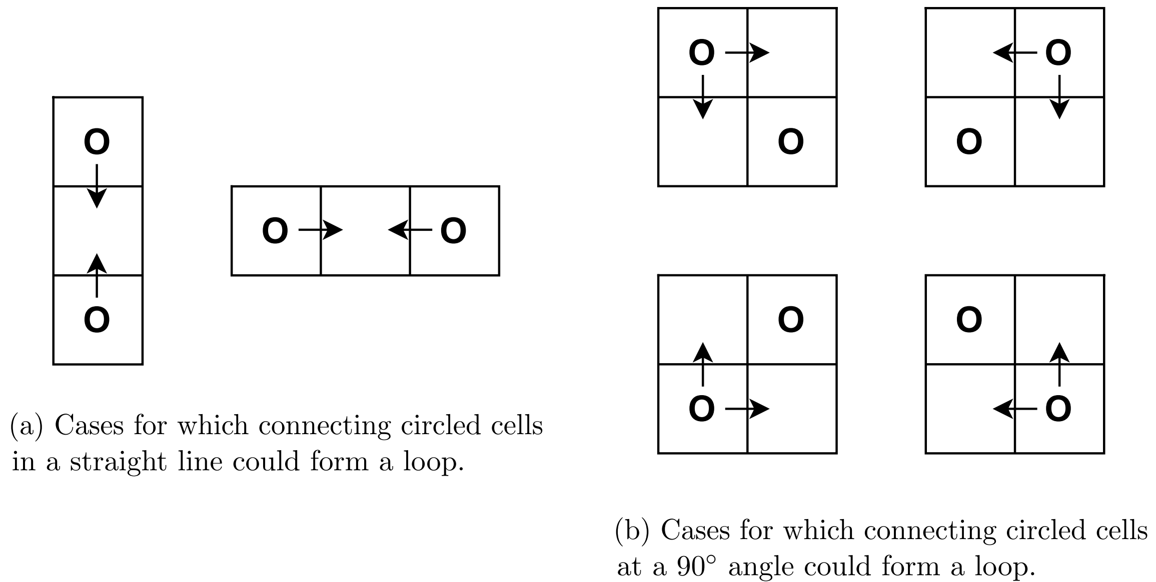
Figure 11: An example of a loop that could be formed during the process of generating a solution.

The two possible approaches to this problem are to either prevent loops from being formed or to eliminate formed loops in the end, when a solution is already formed. Of these two, the first seems to be the best option as prevention is often better than cure and because the second option means that circles have to be eliminated after the stop-condition evaluated to `true` – which is based on the presence of a certain percentage of circles in the playingfield. Prevention has only one hurdle to be taken: detecting that a loop is going to be formed.

In total, there are 12 cases divided over two groups (4 in one group and 8 in the other) in which a loop can be formed by placing one more circle in the playingfield. These cases are all based upon the same principle. Since newly added circles are only placed in cells adjacent to (at least) one other circled cell, there is, at any given point in time, at most one cluster of circles – in other words,

all circles are always connected. Therefore, a loop could be introduced when placing a circle in an empty cell that would connect two other circles. The 12 possible cases mentioned above are presented in Figure 12. To keep it simple, only the interesting parts of the playingfield are displayed and the rest is ignored.

Figure 12a represents the 4 cases that are in the same group where every arrow represents one case. For example, the left diagram in this figure shows that, when either the top or bottom cell in this diagram – both containing a circle – is the cell corresponding to the index retrieved in step 1 from Section 4.2.1, then placing a circle in the middle cell would reconnect those two circles – these circles already have to be connected in some way. When these two circles are connected by a stripe of circles in either the column directly on the left or the column directly on the right of this one, no loop will be introduced. If this is not the case, then one can conclude that a loop will be introduced because the algorithm encapsulates an empty cell. The exact same holds for the right diagram in Figure 12a but then the row directly above and the row directly below this row are inspected.



(a) Cases for which connecting circled cells in a straight line could form a loop.

(b) Cases for which connecting circled cells at a 90° angle could form a loop.

Figure 12: Visualisation of the 12 possible cases in which loops can be formed.

Figure 12b represents the other 8 cases where, again, every arrow represents one case. Assume that, for the top left diagram in this figure, the top left cell in this diagram – containing a circle – is the cell corresponding to the index retrieved in step 1 from Section 4.2.1. If, as shown in the diagram, the bottom right cell contains a circle, then placing a circle in either the top right or bottom left cell would reconnect those two circles – these circles have to be already connected in some way. When the cell opposite to the "target cell" – the cell that is planned to be filled with a circle – is also empty, it can be assumed that a loop is introduced because it is very likely that this empty cell (but in some cases also another cell) will be encapsulated when placing a circle in the target cell. In the opposite case, no loop will be introduced because if beforehand there was no loop present yet, placing one circle that completes a 2×2 block of circle filled cells can never encapsulate an empty cell.

Though, there is one exception in which the case above does not form a loop of circles. When the target cell itself is encapsulated by circles, except for one cell – the cell opposite to the target cell, then placing a circle in the target cell will not form a loop. However, this case is very undesirable since placing a circle in the target cell would create a 3×3 block of which all cells but one are filled – the target cell is in the center of this grid. This leaves very little room for the configuration to be expanded to a solution because, when a 3×3 block of circle filled cells is created, there is no way to expand as there are only stripes of 3 circles – it is not allowed to expand these any further. For this reason it is easier to just assume that a loop will be formed because by that the (undesired) exception is ignored and computation time is saved.

4.2.3 Reducing solutions to starting configurations

The previous sections explain how the generation of solutions is done. However, these solutions have to be reduced to starting configurations. During the generation of a solution, all coordinates of circled cells are stored in a list – the "coordinate list". For reducing the solution, the algorithm enters a loop. Within each iteration of this loop, a random element of the coordinate list is selected. For this element, the cell corresponding to its coordinates – the "selected cell" – is emptied. Subsequently, the solving algorithm is called for the current configuration. If the solving algorithm still only finds one solution, then emptying the selected cell does not lead to a violation of the unique solution constraint. Otherwise, the selected cell will, again, be filled with a circle.

The randomly picked element from the coordinate list is deleted from the list, regardless of the two cases mentioned in the previous paragraph. Thereafter, it is checked whether the stop-condition is fulfilled. Here, the range of percentages for the stop-condition is the same as for the generating algorithm from Section 4.1. This time, however, the percentage of circled cells in the playingfield must drop below this stop-condition for it to be fulfilled, since it is derived from the solution. The algorithm exits when the stop-condition is fulfilled or when the coordinate list is empty.

Once a valid starting configuration has been generated, it can be written to a file in the same format as the configurations from the website. Figure 13 shows an example of a puzzle that is generated by this algorithm. This puzzle can be solved by hand.

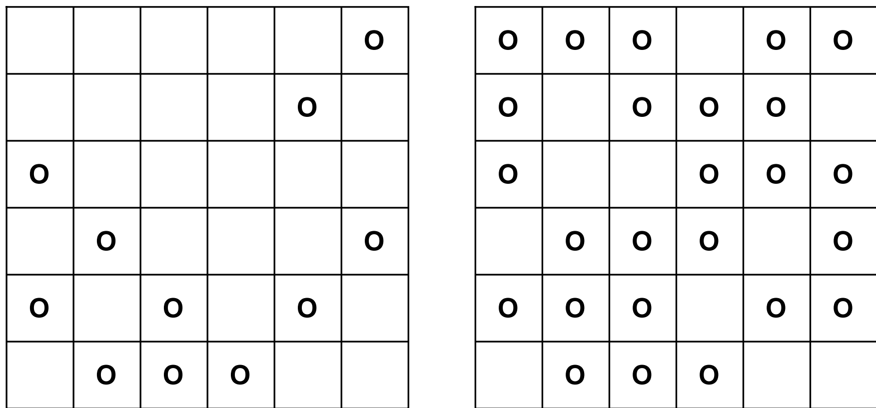


Figure 13: Puzzle generated by the randomized solution algorithm and its corresponding solution.

5 SMT

So far, it is described how Fobidoshi puzzles can be solved and generated using the C++ programming language. This is, however, only one of the many methods that can be used for this goal. Another method is to combine the rules of the puzzle and its configuration into a formula with variables. This formula is then checked to see if there exists an interpretation that satisfies this formula. One of the best known examples of this is the "Boolean Satisfiability Problem" (also referred to as SAT).

As the name already suggests, SAT instances have to be Boolean formulas. Such formulas use exclusively Boolean variables to which a SAT solver – an algorithm for solving SAT – then tries to assign either the value `TRUE` or `FALSE` in such a way that, in the end, the formula evaluates to `TRUE`. If such an assignment of values exists, the formula is called satisfiable. When no such interpretation exists, the formula is called unsatisfiable. Since SAT instances can only be Boolean formulas, its potential is relatively limited. This is where "Satisfiability Modulo Theories" (SMT) comes into play.

Where SAT is the problem of determining whether a Boolean formula is satisfiable, SMT is an extended version of this for mathematical formulas. This way, many more possibilities are introduced by allowing more data types like integers and data structures like lists. The idea remains the same: try to assign values to the variables in such a way that the formula evaluates to `TRUE`. Tools that have been created for this purpose are, similar to those for SAT, called SMT solvers.

For logic puzzles like Fobidoshi, it is possible to create both SAT and SMT instances that can then be checked for satisfiability by one of their respective solvers. In this paper, the decision is made to formulate Fobidoshi puzzles as SMT instances inspired by the previous work done by Van der Knijff [dK21]. One of the most popular SMT solvers is Z3, which is also the solver that is used here. Z3 is an efficient SMT solver from Microsoft Research used in several software verification and analysis applications [dMB08]. The input format for Z3 is an extension of the one defined by the SMT-LIB format [BFT21] which is similar to the syntax of the LISP programming language.

Before describing how the SMT instance is constructed, it is important to note that all the following operations are executed by a Python script. This Python script reads and stores puzzle configurations from text files that are in the same format as explained in Section 3.1. It is also responsible for writing the different parts of the SMT instance to another file in the SMT-LIB format and then subsequently running Z3 for this instance. The output of Z3 is always sent to a text file to be used in a later step meaning that only the output generated by the Python script is printed.

5.1 Solving Fobidoshi puzzles

To be able to use an SMT solver to solve Fobidoshi puzzles, an SMT instance of Fobidoshi has to be created. In an SMT instance, the rules of the puzzle as well as the starting configuration of the puzzle are encoded. As mentioned earlier, Fobidoshi has only two, relatively easy to understand, rules: horizontal or vertical stripes of more than 3 connected circled cells are not allowed and all the

circled cells in the playingfield have to be connected to each other using only orthogonal directions. In the starting configuration of a Fobidoshi puzzle there can be empty cells and cells filled with a circle or a cross. These are the main things to take into account.

An SMT instance is a formula with variables. For Fobidoshi, the variables are the individual cells of the playingfield – these variables are referred to as "cell-variables" and are of Integer type. There are three possible assignments for these cell-variables at each point in time – empty, circle, or cross. Within the SMT instance, it is chosen to encode these values with 0, 1, and 2 (respectively). The cells that are already filled with a circle or a cross in the starting configuration of a puzzle can never be emptied or filled with the opposite symbol, they have to keep their assigned value. Thus, for each cell-variable corresponding to such a cell, a clause is introduced that limits its possible values to 1 – for a circle – and to 2 – for a cross. For each cell-variable corresponding to an empty cell in the starting configuration, another clause is introduced that limits its possible values to 0 and 1.

Each cell of the playingfield must have a unique cell-variable. The name of each cell-variable is chosen in such a way that it directly refers to the coordinates of its corresponding cell in the playingfield. Here, the coordinates match the indices of the arrays in which the configuration of the playingfield and the configuration of the solution are stored. The coordinates of each cell in the playingfield are shown in Figure 14.

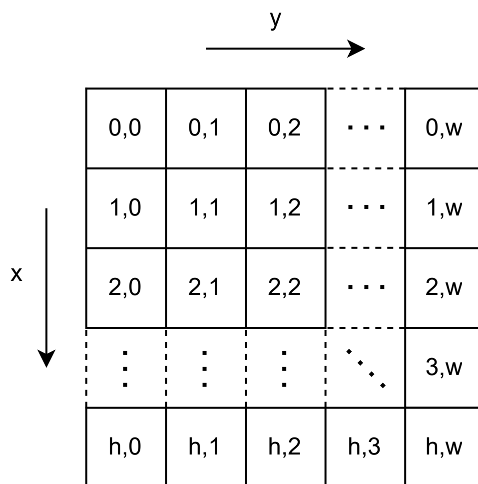


Figure 14: Coordinates of the cells in the playingfield where h stands for "height" and w stands for "width".

Using the assignment of coordinates to the cells in the playingfield, the constants representing the cell-variables are declared using the following SMT-LIB syntax:

```
(declare-const x{x}y{y} Int)
```

In this example, `x{x}y{y}` represents the name of the constant. Here, `x` and `y` are characters that are part of the name, and `{x}` and `{y}` must be substituted by the x- and y-coordinate (respectively)

of the cell corresponding to this constant. This results in a unique name for each cell-variable. The above notation is used here to generalize the examples in this paper. In the actual SMT instance, the cell-variables are only referred to with their exact name – for example `x0y0`, `x0y1`, and `x0y2`. The declaration of these variables in the file with the SMT instance looks as follows:

```
(declare-const x0y0 Int)
(declare-const x0y1 Int)
(declare-const x0y2 Int)
...
```

As mentioned earlier, the possible range of values that can be assigned to the cell-variables is specified next. This gives the following (generalized) SMT-LIB syntax for the three possible options – empty, circle, or cross (respectively):

```
(assert (or (= x{x}y{y} 0) (= x{x}y{y} 1)))
(assert (= x{x}y{y} 1))
(assert (= x{x}y{y} 2))
```

For the first statement above – when the cell corresponding to that cell-variable is empty, the value 2 is purposely left out because the SMT instance does not contain any rules for where to place crosses as such rules do not exist. Placing crosses can help humans and algorithms by not having to visit parts of the state space tree but that is not the case here. When expanding the clause with the value 2, Z3 could randomly assign this value to some cell-variables purely because it is given the chance to do so, not because it is a smart thing to do.

Currently, Z3 still has no idea what the rules of Fobidoshi are. The only thing it knows is the range of values for each cell-variable. For that reason, the SMT instance is expanded starting with the rule saying that horizontal or vertical stripes of more than 3 circled cells are not allowed. This rule is enforced by introducing a clause for each possible horizontal and vertical quartet of cell-variables that correspond to four neighbouring cells. Each clause imposes that not all four of its cell-variables may be assigned the value 1 – which corresponds to a circle. After introducing these clauses, Z3 will not create stripes of 4 or more circled cells. For a 4×4 grid, this results in a total of 8 clauses – 4 for stripes in horizontal direction and 4 for stripes in vertical direction. Below, two examples of these clauses in SMT-LIB syntax are given – one for the horizontal and one for the vertical direction (respectively):

```
(assert (not (and (= x{x}y{y} 1) (= x{x}y{y+1} 1)
                  (= x{x}y{y+2} 1) (= x{x}y{y+3} 1))))

(assert (not (and (= x{x}y{y} 1) (= x{x+1}y{y} 1)
                  (= x{x+2}y{y} 1) (= x{x+3}y{y} 1))))
```

What is lacking still is the constraint and, with that, the need for Z3 to connect all the circles in the puzzle. The connectivity constraint is quite a special constraint. Implementing clauses that impose this constraint is not as straightforward as implementing the clauses that impose the horizontal and vertical stripe rule. Implementing this connectivity constraint was also part of Van der Knijff's

research [dK21] who constructed SMT instances for 6 different puzzles. This paper formed the inspiration for this part of the project. The connectivity constraint in Van der Knijff's paper is implemented in a way that is proposed by Zantema and Joosten [ZJ15, p.7]. They presented a general manner to check if the connectivity constraint is satisfied. A more detailed version of this is given below in the lemma and its corresponding proof. This is also what is used to enforce connectivity for this SMT instance.

Lemma. A graph $G = (V, E)$ is connected iff there exists a mapping $f : V \to \mathbb{N}$ such that (1) there is a unique vertex $r \in V$ with $f(r) = 0$, and (2) for each $v \in V \setminus \{r\}$ there exists $w \in V$ such that $(v, w) \in E$ and $f(w) < f(v)$.

Proof. Assume G is connected. Choose a spanning tree for G and take arbitrary vertex r. Assign to each vertex its distance to r via the spanning tree. This mapping satisfies the requirements.
Conversely assume that there is a mapping $f : V \to \mathbb{N}$ with the two requirements. Each connected component of G must have a minimal element under the assignment of f. For each such minimal element, no neighbour with a smaller value can be found. Hence the graph must have only a single component.

The lemma above assigns a value – in the form of a natural number – to each vertex of the graph. Here, the vertices correspond to the cells of the playingfield. Each cell of the puzzle already has a cell-variable that stores whether it is an empty cell or a cell with a cross or a circle. As these cell-variables cannot store two values, a second variable is assigned to each cell – these variables are referred to as "vertex-variables" and are also of Integer type. For each vertex-variable, a clause is introduced to specify which values can be assigned to it. All vertex-variables can be assigned a positive number greater than 0, except the vertex-variable for the unique vertex r. This vertex-variable is fixed to the value 0. For simplicity purposes, the unique vertex r corresponds, in this case, to the first circled cell that is encountered when iterating over the starting configuration from top left to bottom right. The following SMT-LIB syntax is used for declaring the constants that represent the vertex-variables:

```
(declare-const Vx{x}y{y} Int)
```

As one can see, the names are almost identical to those of the constants for the vertex-variables except for the "V" at the front of the name – which stands for "vertex" – to be able to distinguish them. The possible range of values that can be assigned to a vertex-variable is specified as follows:

```
(assert (= Vx{x}y{y} 0))
(assert (> Vx{x}y{y} 0))
```

The first of the two above statements is for the unique vertex r, since this vertex-variable is fixed to the value 0. The second statement is for all of the other vertex-variables. Each of these variables can be assigned a positive number greater than 0.

Having specified this, the lemma is used to construct a clause for the connectivity constraint. Assume that there is a cell c in the playingfield. For cell c, there exists a literal of the clause that imposes that the cell-variable corresponding to cell c is either assigned a value not equal to 1 –

empty or a cross – or the value 1 – a circle. If the first case is true, the literal evaluates to `TRUE` and the next literal for the next cell will be checked. In the second case, the lemma has to be satisfied. To validate this, the cell-variables corresponding to the 4 (or less) neighbouring cells of cell c are checked. For one of these cells, its corresponding cell-variable has to be assigned the value 1 and its corresponding vertex-variable has to be assigned a value that is less than the value assigned to cell c's corresponding vertex-variable. Only if such a neighbour is found, the literal evaluates to `TRUE`. Similar literals exist for all cells in the playingfield except for the cell that corresponds to the unique vertex r, as this is the minimal element. If one of these literals evaluates to `FALSE`, the connectivity constraint is not satisfied.

All that is explained above is combined into one big clause of conjunctions where each literal of the conjunction contains the clause for one cell in the playingfield. This clause enforces that the connectivity constraint is indeed satisfied for each circled cell. Part of the clause, with an example of one of the literals of the conjunction, is given here in SMT-LIB syntax:

```
(assert (and
          (or (distinct x{x}y{y} 1)
              (and (= x{x}y{y} 1)
                   (or (and (= x{x-1}y{y} 1) (< Vx{x-1}y{y} Vx{x}y{y}))
                       (and (= x{x}y{y+1} 1) (< Vx{x}y{y+1} Vx{x}y{y}))
                       (and (= x{x+1}y{y} 1) (< Vx{x+1}y{y} Vx{x}y{y}))
                       (and (= x{x}y{y-1} 1) (< Vx{x}y{y-1} Vx{x}y{y}))
                   )
              )
          )
        ...
        ))
```

This almost completes the SMT instance for Fobidoshi puzzles. To be able to use this instance to find the solution for given starting configurations, the written Python script that generates the SMT instance adds the line `(check-sat)`. This tells Z3 to check the satisfiability of the formula. If the formula is satisfiable, the output is `sat`. If it is not satisfiable, the output is `unsat`. Z3 also has a functionality that allows for printing of the model that satisfies the formula to the standard output. This is done by adding the line `(get-model)`. For this instance, it outputs all the values assigned to the cell-variables and the vertex-variables. Instead of printing this to the terminal, the output is sent to a text file so that the output is preserved.

At the beginning of the Python script, also the solution of the puzzle is stored in a 2D array. This data is now used to check whether the model found by Z3 actually matches the solution. The Python script retrieves the data that was generated by `(get-model)` and filters out all the coordinates of the cells and their corresponding value – 0, 1, or 2. The values are then converted to its matching symbol – empty, circle, or cross (respectively) – and stored in another 2D array. These 2 arrays are then compared to see if all the coordinates of their circled cells are matching. If this is not the case, the Python script exits with an error message, otherwise it continues.

Since each Fobidoshi puzzle also has to have a unique solution, it is necessary to check if a model that satisfies a formula is in fact the only possible satisfying model for this formula. Unfortunately, Z3 does not have a built-in functionality for this. However, checking if a model is unique can be achieved by executing Z3 again for the same SMT instance but this time with an extra clause. This clause states that not all variables – for this instance only the cell-variables – can be equal to their assigned value in the satisfying model. If the model was indeed unique, running Z3 again for this formula should produce the output `unsat`. This means that no errors have been detected and the model is the correct one. If the output is `sat`, there exists another model for the formula which means that the model is not unique. In this case, the algorithm exits with an error message.

Part of the clause that checks for uniqueness can look as follows in SMT-LIB syntax:

```
(assert (not (and
                (= x{x}y{y} 1)
                (= x{x}y{y} 0)
                 ...
             )))
```

For this clause, each cell-variable with its respective value in the satisfying model will be a literal of the negated conjunction as shown above. This way, the previously found model is prohibited which forces Z3 to search for another interpretation that satisfies the formula.

6 Experiments

The previous sections discuss how the different parts of this project have been implemented. To get a better understanding of what they are capable of and how they are functioning, this section conducts experiments.

6.1 Solving

To start with, a few experiments are conducted for the solving algorithm from Section 3.2. This algorithm has a very basic frame to which some improvements have been applied – as explained in Section 3.3. The goal of these experiments is to show how the algorithm's runtime, the recursive calls of the backtracking function, and the maximum depth of the recursive tree are influenced by the size of the puzzle that has to be solved.

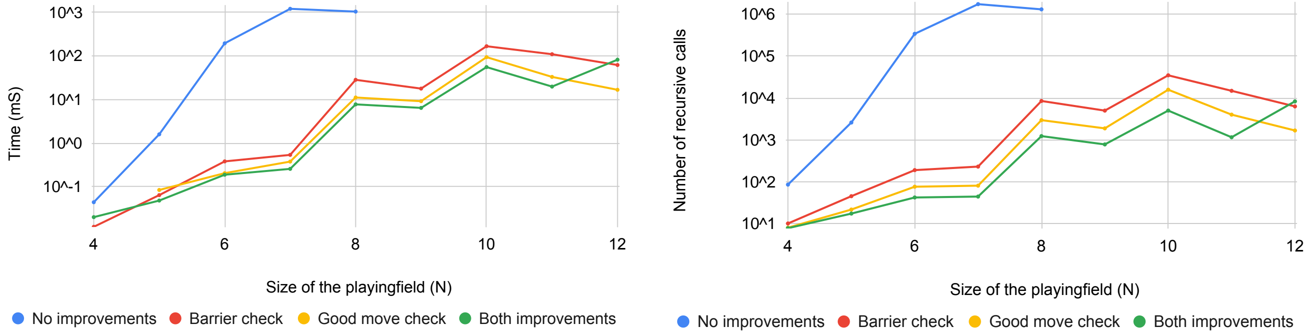
Before being able to conduct these experiments, it is necessary to be in possession of a decently-sized set of valid puzzles. As mentioned before, the website from Angela and Otto Janko [JJ] has 240 Fobidoshi puzzles ranging from $N = 4$ until $N = 12$ – with $N = 11$ excluded – already available. However, this set of puzzles is not balanced well since there are significantly more puzzles available for some sizes – like for $N = 8$, 10, or 12 – than for other sizes – like for $N = 4$ or 5. Because these experiments focus on the influence of the puzzle size on the parameters mentioned above, it is necessary to balance this subset. This is done by randomly generating puzzles – using the algorithm from Section 4.2 – for the puzzle sizes that are in the minority until there are at least 50 available puzzles for each puzzle size. It may be that some of these puzzles are duplicate. This is not a problem, since the algorithm cannot learn from previously solved puzzles and the chances of having duplicates are quite low, except for $N = 4$, because there is only a very limited number of configurations possible for this size.

The function running these experiments loops over $N = 4$ until $N = 12$. For each size, the algorithm does 5 repetitions where, for one repetition, the first 50 puzzles of this size are solved. This is decided since, for some puzzle sizes, there are more than 50 puzzles available on the website of Angela and Otto Janko. To make sure that the results are constant and reliable, only the first 50 are solved in each repetition. For each puzzle, the solving algorithm is called four times with four different combinations of parameters.

First, the solving algorithm is called without any of the improvements described in Section 3.3. The second time, the algorithm is only allowed to check for barriers – as explained in Section 3.3.1, while the third time, it may only make use of the improvement that detects good moves – see Section 3.3.2. The last call to the solving algorithm is with both of these improvements enabled. Here, the improvement that is explained in Section 3.3.3 is ignored and, for that reason, also disabled. Tests showed that activating this improvement decreased the runtime with as little as 30%. Thus, the effect of this improvement on the efficiency of the algorithm does not come close to that of the other two improvements and is therefore ignored here. To make sure that the results of the experiments can be obtained without having to wait for days, the decision is made to exit the execution of the solving algorithm after 5 seconds. The boundary should not limit the majority of the executions of the algorithm – finishing within a second – while making sure that running the

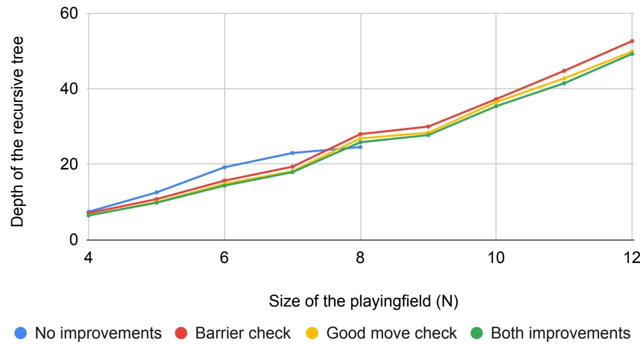
algorithm does not get too expensive for the other cases – which can happen (especially) when no improvements are activated. The boundary of 5 seconds does exactly this.

During the execution of these experiments, the values for the average runtime in milliseconds, the average number of recursive calls, and the average depth of the recursive tree are stored. These values are, within each repetition, averaged over the number of puzzles and, after finishing all repetitions, over the number of repetitions. The results of this are visualised in the three diagrams in Figure 15 where each line represents one of the four configurations of the improvement parameters passed to the solving algorithm – as can be seen from the legend below each diagram.



(a) Average time it takes the solving algorithm from start to finish when finding a valid solution.

(b) Average number of recursive calls executed by the solving algorithm when finding a valid solution.



(c) Average depth of the recursive tree when finding a valid solution.

Figure 15: Statistics gathered by running the solving algorithm from Section 3.2 for sizes $N = 4$ until $N = 12$.

What stands out is that, in all three diagrams, the blue line – matching the call to the solving algorithm without any of the improvements – differs (a lot) from the other three lines and has no values after $N = 8$. The explanation for this is that, for this configuration of parameters, the solving algorithm takes a long time to solve these puzzles. For $N = 7$, on average, still 76% of the puzzles are solved within 5 seconds where for $N = 8$, this number already drops to 16%.
This small subset of puzzles that is still solvable is not producing reliable results anymore because these puzzles have a starting configuration that, as it turns out, is quite optimal for the solving

algorithm. The fact that these puzzles are solved within the time limit of 5 seconds is just a matter of “lucky shots”. As a result, the time in Figure 15a and the number of recursive calls in Figure 15b are slightly lower for $N = 8$ than for $N = 7$ while 84% of the puzzles are not even solvable within 5 seconds. As one can imagine, including these statistics would influence these results drastically. Therefore, no results beyond $N = 8$ are taken into account for this configuration.

The other three lines follow patterns that are quite similar to each other. All of these configurations perform much better compared to the configuration with no improvements activated. In general, having only the barrier check activated gives the worst results out of these three configurations while having both improvements activated is the fastest and “cheapest” – when it comes to the recursive calls and the depth of the recursive tree.

An exception to this rule is for $N = 12$. Activating both improvements actually makes the algorithm slower and more expensive in this case. The other two configurations, each with only one improvement activated, do not show this behaviour and they both have better results. The algorithm performing best in this case is the algorithm that only uses good move detection.

Overall, the results in these three diagrams show that activating less improvements and increasing the size of the puzzle is correlated with an increase in runtime, recursive calls, and depth of the recursive tree. Good to note, the solving algorithm manages to solve close to all the puzzles for all puzzle sizes within the boundary of 5 seconds when one or more improvement is activated.

6.2 Generating

Generating Fobidoshi puzzles is, just like solving puzzles, a big part of this project. Solving Fobidoshi puzzles is, within this project, always done by the same backtracking algorithm. For generating, however, two different algorithms have been implemented. Section 4.1 explains the first algorithm – which tries to randomly produce possible starting configurations – and Section 4.2 the second one – which tries to randomly produce possible solutions and then works its way back to a matching starting configuration. To distinguish these two algorithms, they will, within this section, be referred to as “first (generating) algorithm” and “second (generating) algorithm” respectively. To show how both algorithms perform and to highlight their similarities and their differences, experiments are conducted. The performance of these algorithms is measured by the percentage of validly generated configurations, the average time it takes to generate a valid configuration, and the average occupation of crosses per playingfield (in percentages).

For the first algorithm, the experiments are executed by a function that loops over $N = 4$ until $N = 9$. For each size, the algorithm does 5 repetitions where, for one repetition, 500 iterations of the generating algorithm are executed. These parameters are chosen because they are believed to generate reliable and representative results while not causing extremely high runtimes. Again, the decision is made to terminate the execution of the algorithm after a predefined amount of time. However, randomly generating a (potential) starting configuration is not such a time consuming process. On the other hand, trying to solve this configuration in all possible ways and subsequently checking for uniqueness when deleting crosses is. Thus, it is decided to exit the execution of the algorithm if the part following the random generation of a starting configuration takes more than 5

seconds. During the execution of these experiments, the values for the number of validly generated configurations, the time per generated configuration in milliseconds – this is for the entire algorithm that first generates a starting configuration and then modifies it to have a unique solution, and the occupation of crosses per playingfield are stored. These values are, within each repetition, averaged over the number of iterations and, after finishing all repetitions, over the number of repetitions. The results of this are shown in Table 1.

Size (N)	Valid configurations	Average time per configuration (ms)	Average occupation of crosses per playingfield
4	55.48%	0.110214	16.5162%
5	52.08%	0.479135	13.3479%
6	47.36%	2.56269	11.937%
7	43.72%	22.2286	11.8154%
8	38.28%	238.846	11.385%
9	29.64%	714.703	10.523%

Table 1: Average percentage of validly generated configurations, average time it takes to generate a valid configuration in milliseconds, and average percentage of occupation of crosses per validly generated playingfield gathered by running the generating algorithm from Section 4.1 for $N = 4$ until $N = 9$.

This table shows that increasing the size of the puzzle is correlated with a decrease of validly generated puzzles and the occupation percentage of crosses present in these configurations. One may be fooled to think that this directly implies a decrease in the number of crosses but that is not true. Since the percentages are only slightly decreasing as the size increases, it is very likely that the number of crosses increases when the size does. This shows that, for all sizes, crosses are needed to eliminate the possibility of having starting configurations with multiple solutions.

An increase in size also means an increase in runtime for validly generated configurations, as can be seen in the table above. A larger size means that there are more cells in the playingfield and thus more possible configurations. This generating algorithm uses the solving algorithm from Section 3.2 multiple times to be able to, in the end, generate a valid starting configuration. As shown in Figure 15a, the solving algorithm has a higher runtime when the size increases. This correlation is also shown here.

For the first generating algorithm, running these experiments with $N = 9$ already leads to high runtimes. The results showed that already more than 10% of the runs were interrupted because they exceeded the time limit of 5 seconds. It is thus decided to not experiment with sizes bigger than this. The general trend can already be grasped from the current results.

For the second algorithm, a function is written that loops over $N = 4$ until $N = 12$. For each size, the algorithm does 5 repetitions where, for one repetition, 2000 iterations of the generating algorithm are executed. These parameters are chosen because this algorithm executes less computationally heavy processes than the first generating algorithm meaning that its runtimes are much shorter. To still get reliable results, a higher iteration number is chosen which increases the number of results used to compute the average values of the statistics. In theory, the results that are obtained this way should be even more reliable and representative while still not causing extremely high runtimes. It is decided to not run the algorithm with more than 2000 iterations as it is unlikely that choosing

a higher value will still have an influence on the reliability of the results. Also this time the decision is made to exit the function if it takes too long to generate a result. Since randomly generating a solution is not very time consuming, only the part of the algorithm that reduces the solution to a starting configuration is limited to 5 seconds. When it reaches this boundary, the execution of the algorithm is stopped. During the execution of these experiments, the same values are stored in the same way as for the first generating algorithm. The results of this are shown in Table 2.

Size (N)	Valid configurations	Average time per configuration (ms)	Average occupation of crosses per playingfield
4	0.3%	0.188889	0.138889%
5	0.55%	0.518679	0.602393%
6	2.09%	2.40665	0.197345%
7	1.86%	8.97021	0.173252%
8	1.29%	43.0763	0.278354%
9	0.89%	172.376	0.198133%
10	0.79%	250.595	0.362771%
11	0.25%	397.917	0.112161%
12	0.19%	561.233	0.532407%

Table 2: Average percentage of validly generated configurations, average time it takes to generate a valid configuration in milliseconds, and average percentage of occupation of crosses per validly generated playingfield gathered by running the generating algorithm from Section 4.2 for $N = 4$ until $N = 12$.

This table shows relations that are identical to those in Table 1. However, the last column in this table – the average occupation of crosses per playingfield – has only values below 1.0%. This is an expected result since the second generating algorithm is written with the goal to minimize the number of crosses present in the starting configuration. Nevertheless, the difference is quite big which shows that the two algorithms really do use different methods that produce different results.

When comparing the other two columns with those of Table 1, the difference between the percentage of validly generated configurations catches the eye. In Table 1, this value was not lower than 29% with a maximum of 55.48%. In Table 2, its maximum value is 2.09% with a minimum of 0.19%. These differences are big in favour of the first generating algorithm. However, once the second algorithm does have a run that produces a valid configuration, it is, in most cases, faster than the first algorithm and it remains fast until $N = 12$. It can also produce puzzles for sizes larger than $N = 12$ but experimenting with this is not very relevant for this project since Table 2 already gives enough information.

Another interesting parameter to experiment with here is the threshold parameter – this parameter is mentioned in the third step of the enumeration in Section 4.2.1. The standard value for this is set to 90% because this seems to produce satisfiable results. It is not known, however, if this is also the percentage for which the best results are generated. To measure this, different values for the threshold parameter are compared by looking at the average percentage of validly generated configurations.

This experiment is executed by a function that loops over $N = 4$ until $N = 12$. For each puzzle size, it loops over 5 different threshold values – 60%, 70%, 80%, 90%, and 100%. Per threshold value, 5

repetitions of 2000 iterations are executed. This function is, in the same way as the previous one, forced to exit if it takes more than 5 seconds to generate a configuration. The average percentage of validly generated configurations is, within each repetition, averaged over the number of iterations and, after finishing all repetitions, over the number of repetitions. Table 3 shows these results.

Size \ Threshold	60%	70%	80%	90%	100%
4	0.12	0.24	0.22	0.16	0
5	0.06	0.18	0.43	0.5	0.07
6	0.02	0.17	0.77	1.96	4.13
7	0.01	0.1	0.41	1.85	5.43
8	0	0.04	0.13	1.14	3.64
9	0	0	0.16	0.85	3.31
10	0	0	0.05	0.81	3.22
11	0	0	0.06	0.28	2.01
12	0	0	0	0.25	1.36

Table 3: Average percentage of validly generated puzzles when running the generating algorithm from Section 4.2 with threshold parameter values ranging from 60% up to, and including, 100% for sizes $N = 4$ until $N = 12$.

This table shows that, in general, the lower threshold values perform worse than the higher threshold values. For $N \geq 8$, little to no valid solutions are generated when setting the threshold value to 60% or 70%. When setting this value to 80%, the algorithm is capable of producing a small number of configurations for all puzzle sizes except for $N = 12$. The two highest values show the best results. Interesting to see is that, with a threshold value of 100%, no valid configurations for $N = 4$ are generated. For $N = 5$, valid configurations can be generated but the average percentage is still very low. For these two sizes, the threshold values of 70% and 90% (respectively) perform best. For $N \geq 6$, the algorithm generates, by far, the most valid configurations when setting the threshold value to 100%.

One might say that, for this reason, it would be best to initialise the threshold parameter to 100%. However, if this is done, little to no valid configurations would be generated for $N = 4$ and $N = 5$. For that reason, the standard threshold value is left at 90% since this generates valid configurations for $N = 4$ until $N = 12$.

7 Conclusions and further research

The first part of this thesis gives an introduction, and a description of, the logic puzzle Fobidoshi. The goal is to connect all the circled cells present in the puzzle without creating horizontal or vertical stripes of more than 3 connected circled cells. In the end, a unique solution has to be found. A backtracking solving algorithm is implemented capable of doing exactly that. Basic backtracking in combination with brute-force does not find the solution fast enough which is why two big improvements and a third, smaller, improvement are implemented. The two big improvements try to detect or prevent barriers of crosses in the playingfield which eliminates a big part of the state space tree. The smaller improvement enforces that only circles will be placed in cells that have a neighbouring circled cell. In the end, the algorithm is capable of solving each individual puzzle on the website of Angela and Otto Janko [JJ] in less than five seconds – most puzzles being solved within a second.

Using this solving algorithm, two algorithms are constructed that can generate new Fobidoshi puzzles. The first algorithm randomly places circles in a playingfield to create a potential starting configuration. This algorithm often has to add crosses to the starting configuration to enforce that there is only one possible solution, which is a constraint of the puzzle. The second algorithm randomly generates a possible solution to a puzzle and tries to reduce this solution to a matching starting configuration. To enforce some randomness, this algorithm contains a threshold parameter. When a cell is selected to be filled with a circle, the value of this threshold parameter is the chance of actually filling this cell with a circle. This algorithm was constructed to try to minimize the number of crosses present in the starting configuration. For this second algorithm, there was the possibility of loops being formed in the solution. As this is not desirable, this is prevented by an embedded check.

Another way of solving Fobidoshi puzzles is implemented in the way of a “Satisfiability Modulo Theories” (SMT) instance that can be solved by an SMT solver. The SMT instance of a Fobidoshi puzzle represents its rules and the starting configuration in the form of a formula after which, in this case, Z3 [dMB08] will try to find an interpretation for which the formula evaluates to `TRUE`. For all SMT instances of the puzzles that are available on the website of Angela and Otto Janko, Z3 manages to find a satisfying model in well under a second.

Experiments are done for the backtracking solving algorithm and the two generating algorithms. It turns out that, in general, the best solving algorithm is the algorithm for which the two big improvements are activated. Nevertheless, enabling only one of these two improvements already creates an algorithm that can solve puzzles in under five seconds. This is not the case for puzzles with $N \geq 8$ when no improvements are activated.
Of the two generating algorithms, the algorithm creating random starting configurations is clearly the more successful algorithm as it generates way more valid starting configurations than the other algorithm. A downside of this algorithm is that on average, for $N \geq 9$, each valid starting configuration is generated in about 0.7 seconds while many of the failed attempts run for 5 seconds before being terminated. For $N = 9$, the algorithm fails to generate a valid starting configuration in over 70% of the cases thus resulting in high runtimes. Besides, a lot of the valid starting configurations generated by this algorithm contain many crosses. The other algorithm shows better

results for these statistics. Thus, it cannot be concluded that one of the two algorithms is superior to the other one since this depends on the situation.
The value of the threshold parameter is also experimented with. It turns out that, in general, the best value for this parameter is 100%. However, for this value, little to no valid puzzles are generated for $N = 4$ and $N = 5$. This is not the case for the smaller values of the threshold experiment. Thus, again, saying which value is best depends on the situation.

There are still enough things to be considered for future research. For example, the computational complexity of the puzzle could be studied, which was not feasible within this thesis. This information is useful to get an idea of the amount of resources required for solving Fobidoshi. Another possibility is to improve the backtracking algorithm used for solving. A way to do this could be by introducing some heuristics for finding empty cells that have a high chance of being a correct cell to be filled with a circle. Also, since the different sizes of the playingfield have different optimal values for the threshold parameter, one can also devote future research to exploring whether it is beneficial to optimise the threshold parameter for each playingfield size.

References

[BFT21] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. *https://smt-lib.org/*, 2021. Accessed 19-06-2024.

[dK21] G. Van der Knijff. Solving and generating puzzles with a connectivity constraint. Bachelor's thesis, Radboud University, Nijmegen, The Netherlands, jan 2021.

[dMB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.

[Heg22] M. Hegeman. Generating Pipes puzzles using maze-generating algorithms. Bachelor's thesis, Leiden Univeristy, Leiden, The Netherlands, may 2022.

[Ina] N. Inaba. Puzzle Laboratory. *http://inabapuzzle.com/*. Accessed 19-06-2024.

[JJ] A. Janko and O. Janko. Fobidoshi. *https://www.janko.at/Raetsel/Fobidoshi/index.htm*. Accessed 16-06-2024.

[Mou22] R. Mourits. Solving and Generating the Nurimeizu puzzle. Bachelor's thesis, Leiden Univeristy, Leiden, The Netherlands, jul 2022.

[ZJ15] H. Zantema and S.J.C. Joosten. Latin squares with graph properties. Paper, Eindhoven University of Technology and Radboud University, Eindhoven and Nijmegen, The Netherlands, oct 2015.