

Bachelor Thesis

Shakashaka

Stefan Pedratscher (01518338)
Stefan.Pedratscher@student.uibk.ac.at

4 October 2018

Supervisor: Prof. Dr. Aart Middeldorp

Eidesstattliche Erklärung

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

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum

Unterschrift

Abstract

Shakashaka is a pencil and paper puzzle published by Nikoli. The main focus of this bachelor thesis is about implementing a user friendly tool which allows to solve such Shakashaka puzzles, generate random ones, create own instances and provide an opportunity for the user to solve the puzzles on his own. Moreover, this paper will present an NP-completeness proof of Shakashaka.

Contents

1	Introduction	1
1.1	Objective	1
1.2	Rules	2
2	Solver	3
2.1	Human-based Solver	3
2.1.1	Basic Idea	3
2.1.2	Implementation	4
2.2	SMT Solver	7
2.2.1	Basic Idea	7
2.2.2	Implementation	8
2.3	Experiments	10
3	Generator	15
3.1	Idea	15
3.1.1	Generate a Random Puzzle	15
3.1.2	Making a Puzzle Unique (Approach 1)	16
3.1.3	Making a Puzzle Unique (Approach 2)	19
3.1.4	Remove Triangles and Dots	19
3.2	Conclusion	20
4	NP-completeness	22
5	Web Application	28
5.1	System Features	28
5.1.1	Solve Puzzle	28
5.1.2	Select a Puzzle	29
5.2	Play Mode	30
5.3	Edit Mode	31
6	Conclusion	33
6.1	Future Work	33
	Bibliography	34

1 Introduction

Shakashaka (also known as Proof of Quilt) is a pencil and paper, logic-based puzzle published by the Japanese puzzle company Nikoli [2]. While playing, the user does not need any arithmetical skills to obtain a valid solution. The puzzle is therefore suitable for children and adults.

1.1 Objective

A Shakashaka instance consists of a $n \times m$ rectangular field of squares. Each of them has one out of twelve different values.



Figure 1.1: Twelve possible cell values.

An unsolved puzzle is a rectangular grid containing white and black cells. To solve a puzzle the player has to fill triangles in white areas of the board. Some white cells may stay white at the end. An instance is solved if all remaining white areas have a rectangular or square shape, independent of their rotation (horizontally, vertically or diagonally within a 45 degree rotation. As an example, consider Figure 1.2). A black dot can be placed by the user to indicate that no triangle is possible in this spot.

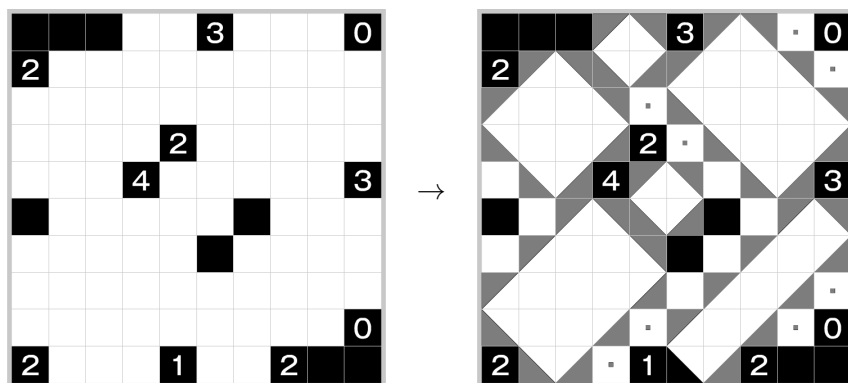








Figure 1.2: Shakashaka instance with a valid solution.

1.2 Rules

The rules of Shakashaka are quite simple:

- 1 A user is only allowed to place , , , ,  or  inside the white fields of the board.
- 2 A number inside a black cell (hint block) indicates how many triangles have to touch this rectangle. A black cell without a number can have one, two, three, four or even zero triangles surrounding it.

A basic tactic for solving Shakashaka puzzles is to not create acute angles. If a triangle reaches the side of the board or a side of an internal rectangle it obviously needs to be a corner of the newly formed rectangle.

2 Solver

The rules of Shakashaka are quite simple: add triangles into white areas and form rectangles or squares. Nevertheless, solving them within a reasonable time is not always as easy as it seems. A simple approach would be to recursively apply assumptions to the puzzle and check if they lead to a valid solution. If they do not, backtrack and try other decisions until a correct solution is obtained or, all possible assumptions are tested and the puzzle is classified as not solvable. This is a quite simple, but very expensive approach. To get quick solutions and to guarantee a good performance it is essential to make as few assumptions as possible. In this chapter, we will analyse two different approaches on how to solve Shakashaka puzzle instances by providing information about the underlying ideas, the game rule modellings, and experiments on how the solvers behave on different puzzles, varying in their levels of difficulty.

2.1 Human-based Solver

2.1.1 Basic Idea

The idea behind the following solver (see Figure 2.1) is to fill all white areas, where only one triangle or no triangle (\square) is possible, by checking its surrounding cells. This process is very fast and in normal practice most of the puzzle is solved. Assumptions are only made if cell values depend on more than their surrounding ones and if the puzzle is not already finished.

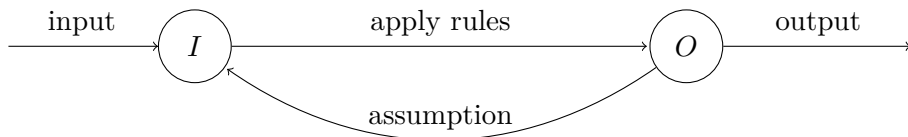


Figure 2.1: Main solver idea: with an input puzzle I look at the surroundings of each non-white cell and fill the puzzle with as many triangles as possible to produce O . If needed, make assumptions and recursively apply the basic rules until the puzzle is solved or declared as non-solvable.

With this method only a few assumptions should be made to get a fast solution for every input puzzle. In the following sections we will analyse the process of *apply rules* and *assumption* in detail.

2.1.2 Implementation

First of all we need to model the game rules of Shakashaka. A cell can have one out of twelve different values. The rules are represented by sets for every possible cell value, except for the empty, white one. These sets indicate the possible cell values on the top, right, bottom and left side according to the selected one in the middle. If there are less than four surrounding cells, for example if a cell touches the border, only possible ones are specified. The white value will be ignored because it signalizes that the field is not set and every cell value is possible around this one. If a cell is clearly identified as a white one it will be indicated with a dot. We distinguish in this section between rules without any restriction, where cell values can be set independent of other ones around the centre cell, and restriction rules where the correctness vary from the other surrounding ones. Let us first focus on the cell rules without restrictions:

nr	T_{nr}	R_{nr}	B_{nr}	L_{nr}
■	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }
■	{ ■, □ }	{ ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }
■	{ ■, ■, □ }	{ ■, □ }	{ ■, □ }	{ ■, ■, □ }
■	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, □ }	{ ■, □ }
■	{ ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, □ }
□	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }	{ ■, ■, □ }
0	{ □ }	{ □ }	{ □ }	{ □ }
4	{ ■, ■ }	{ ■, ■ }	{ ■, ■ }	{ ■, ■ }

Table 2.1: Shakashaka rules without any restriction. T_{nr} , L_{nr} , R_{nr} and B_{nr} (top, left, right and bottom) are sets of valid cell values in its specific location.

These rules have the advantage that each side can be set individually at any time. In contrast, the correctness of the cell values according to the rules with restriction vary at least on one other cell. In this case the rules are defined as sets of quadruples.

r	(t_r, r_r, b_r, l_r)
□	{ (■, □, □, ■), (■, ■, □, □), (□, ■, ■, □), (□, □, ■, ■), (■, ■, ■, ■), (■, ■, ■, ■), ... }
1	{ (■, □, □, □), (■, □, □, □), (□, ■, □, □), ... }
2	{ (■, ■, □, □), (■, ■, □, □), (■, □, □, ■), ... }
3	{ (■, ■, ■, □), (■, ■, ■, □), (■, ■, ■, □), ... }

Table 2.2: Shakashaka rules with restriction. The sets of quadruples indicate the possible cell values on the top, right, bottom and left (t_r , r_r , b_r and l_r) side, depending on the one in the middle. In some cases corner cells may also be considered.

Each corner cell of a 3×3 field may be limited by the other eight cells, including the one in the middle. Considering $r = \blacksquare$ and $l_r = t_r = \square$ leads to the only possible value \blacklozenge of the cell on the left top corner.

We note that only the dot cell (\square) has a restriction and a non-restriction rule. All others have either restriction or non-restriction rules. Please keep in mind that there may be different rules for cells at borders. In special cases the cell rules may also change from restriction to non-restriction or may even be missing completely to signal an invalid cell.

t_r		r	(t_r, r_r, b_r, l_r)
r	r_r	1	$\{ (\blacksquare, \square, \square, /), (\square, \blacksquare, \square, /), (\square, \blacklozenge, \square, /), (\square, \square, \blacklozenge, /) \}$
b_r		2	$\{ (\blacksquare, \blacklozenge, \square, /), (\blacksquare, \blacksquare, \square, /), (\blacksquare, \square, \blacklozenge, /), (\square, \blacklozenge, \blacklozenge, /), (\square, \blacksquare, \blacklozenge, /) \}$

Table 2.3: Shakashaka rules with restriction at the left border.

Comparing Table 2.3 with Table 2.2 we note that the black square containing three and the dot value rules are missing. This modification is done because these cells do not depend any more on other surrounding ones. Moreover the possible cell values in the restriction rules are also reduced.

T_{nr}		nr	T_{nr}	R_{nr}	B_{nr}	L_{nr}
nr	R_{nr}	3	$\{ \blacksquare \}$	$\{ \blacksquare, \blacklozenge \}$	$\{ \blacklozenge \}$	\emptyset
		\blacksquare	$\{ \blacksquare, \square \}$	$\{ \blacksquare, \blacklozenge, \square \}$	$\{ \blacklozenge, \square \}$	\emptyset
		\blacklozenge	$\{ \blacklozenge \}$	$\{ \blacklozenge, \square \}$	$\{ \blacklozenge, \square \}$	\emptyset
		\square	$\{ \blacksquare, \square \}$	$\{ \blacksquare, \square \}$	$\{ \blacksquare \}$	\emptyset
		\square	$\{ \blacksquare, \square \}$	$\{ \blacksquare, \blacklozenge, \square \}$	$\{ \blacklozenge, \square \}$	\emptyset
		0	$\{ \square \}$	$\{ \square \}$	$\{ \square \}$	\emptyset

Table 2.4: Shakashaka rules without restriction at the left border.

The black cell with the number four does not have any restriction or non-restriction rule because it is not possible at the border. This also applies to two of the four possible triangles. In the case where a cell touches two borders the rules are even smaller and therefore easier to use for the solver.

In the following two subsections these rules will be used to illustrate the basic idea of the solver.

Apply rules

To fill the puzzle with as many triangles as possible we look at all non-white cells and set the surrounding ones, if only one cell value is possible. If a cell has a non-restriction rule we can set each side where the valid block size is one:

$$M \in \{T_{nr}, R_{nr}, B_{nr}, L_{nr}\} \bigwedge \#(M) = 1 \quad (2.1)$$

The restriction rules have to be considered differently. Let us define R as the set of quadruples, indicating the possible cell values around a specific one. If at least one of these four surrounding cells is not valid we remove the quadruple from the set. After that step we check if the size of R is one and set the cell values according to the quadruple:

$$\#(R) = 1 \quad (2.2)$$

If a cell has both, a restriction and a non-restriction rule, we need to check additionally if we can exclude one of these. Considering only one cell and check if the size of valid cell values is one works mostly only for cells touching the border. In other cases we need to look at the intersection of the surrounding ones (see Figure 2.2).

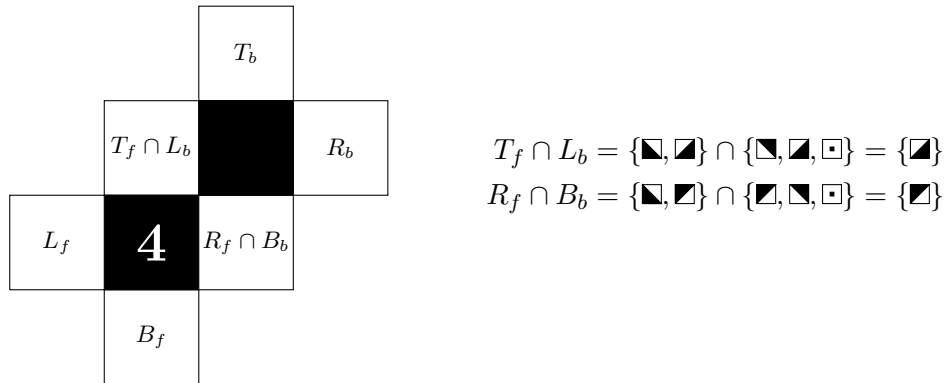


Figure 2.2: Determine cell values by looking at intersections.

This process will be repeated until each cell is checked and no further white cells can be set. As mentioned in the previous section, this process is very fast in relation to the *assumption* step and in normal practice a big part of the puzzle is solved using this method.

Assumptions

If the puzzle is not completely finished after the *apply rules* step, we make assumptions. At this point it is clear that some cells depend on more than only their surrounding ones. Figure 2.3 shows such an example. Black triangles and dots are all set within the *apply rules* process. An assumption may be the green cell. We continue filling the puzzle with

correct triangles and dots. The red cells indicate that something is wrong in the puzzle and indeed the only possible cell value for the green one is a dot.

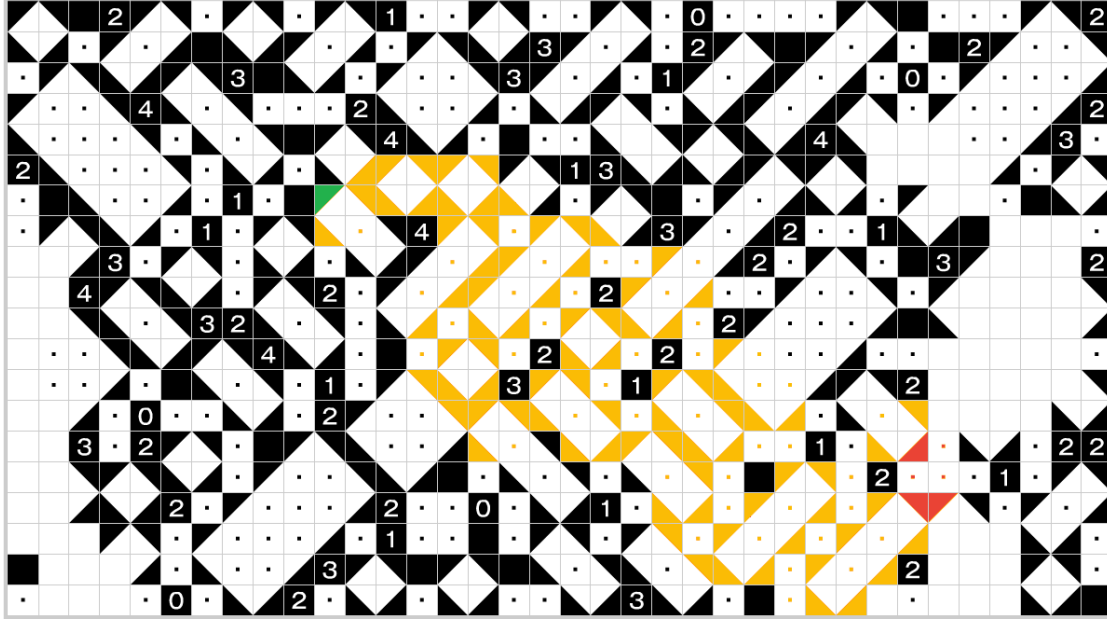


Figure 2.3: Long distance assumption.

The figure shows that the correctness of an assumption can depend on other cells, far away from the premise. Most likely more assumptions are made and all of them need to be backtracked if the first one is wrong. Nevertheless, it still makes sense to try assumptions instead of looking at a very big radius around each cell, which would be more time-consuming. After an assumption is applied, the solver continues filling the puzzle within the *apply rules* step. If the puzzle is finished, it will be returned, otherwise a new assumption is made. Assumptions are always made in the same order (■, ■, ■, ■ and □) to guarantee a deterministic algorithm.

A puzzle can be classified by inspecting the amount of correct assumptions which are needed to get a correct solution of the puzzle. Moreover, we consider the size of the instance as an additional factor for the classification of the difficulty of a puzzle.

2.2 SMT Solver

2.2.1 Basic Idea

The idea behind the following logic-based solver (developed with my supervisor) is to define the rules of Shakashaka by adding logical formulas according to the board and

passing them to an existing SMT solver to find a satisfiable model of the instance. This solver will use Z3, a state-of-the art SMT solver from Microsoft Research. The solver therefore transforms the puzzle and its rules to a Z3 readable format, calling Z3, which finds a satisfiable model, and then transform the output again to display the correct solution of the puzzle.

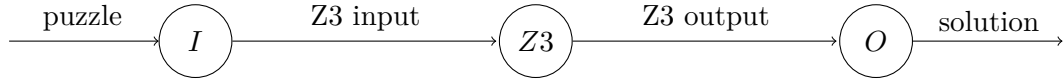


Figure 2.4: Main Idea: Given an input puzzle I , transform all fixed constraints (all black cells) and the rules of Shakashaka to a Z3 readable format, pass them to Z3 to find a valid solution and transform its output to a displayable format O .

2.2.2 Implementation

First, all 13 possible cell values of a Shakashaka puzzle are defined in a new data type to simplify the encoding. At this point all fields of the current input puzzle can be defined as variables: the fixed ones (all black ones with and without a number) are set to their corresponding values while the others stay undefined by now. Now we can start adding constraints to the puzzle to add more restrictions and therefore to limit the amount of possible cell values in their specific locations. According to the Shakashaka rules, the only possible cell values for the corners, which can be placed by the user, are the following:

$$\begin{aligned}
 \text{LeftTopCorner} &= \square \vee \blacksquare \\
 \text{LeftBottomCorner} &= \square \vee \blacksquare \\
 \text{RightTopCorner} &= \square \vee \blacksquare \\
 \text{RightBottomCorner} &= \square \vee \blacksquare
 \end{aligned} \tag{2.3}$$

Next we define the sides of our Shakashaka puzzle. This works as before: If a cell is white a user can only set a certain amount of valid cell values. Corner cells are already defined and therefore not considered within this step (they already have a larger restriction).

$$\begin{aligned}
 \text{Left} &= \square \vee \blacksquare \vee \blacksquare \\
 \text{Right} &= \square \vee \blacksquare \vee \blacksquare \\
 \text{Top} &= \square \vee \blacksquare \vee \blacksquare \\
 \text{Bottom} &= \square \vee \blacksquare \vee \blacksquare
 \end{aligned} \tag{2.4}$$

At this point, let us focus on the possible triangle shapes. According to the rules of Shakashaka also here only certain triangle shapes are valid for a correct solution. As described earlier, at the end of the solving process all remaining fields need to form

rectangles or squares by adding triangles to the field. Let us focus within the next figure on the internal cells of a Shakashaka puzzle instance. Additionally, let us note that the corner and side cells have to be considered differently because there we have more restrictions and therefore less cell values valid, which is good for the solving process.

	lt	t	rt	
	l	c	r	
	lb	b	rb	

$$c = \blacksquare \implies (lt = \blacksquare \oplus t = \blacksquare) \wedge (rb = \blacksquare \oplus r = \blacksquare) \wedge (rt = \square \vee rt = \blacksquare) \wedge ((b = \square \wedge l = \square) \implies \neg lb = \square)$$

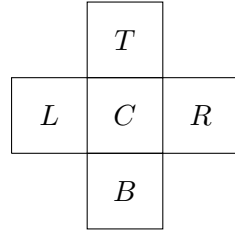
$$c = \blacksquare \implies (lb = \blacksquare \oplus l = \blacksquare) \wedge (rt = \blacksquare \oplus t = \blacksquare) \wedge (lt = \square \vee lt = \blacksquare) \wedge ((b = \square \wedge r = \square) \implies \neg rb = \square)$$

$$c = \blacksquare \implies (lt = \blacksquare \oplus l = \blacksquare) \wedge (rb = \blacksquare \oplus b = \blacksquare) \wedge (lb = \square \vee lb = \blacksquare) \wedge ((t = \square \wedge r = \square) \implies \neg rt = \square)$$

$$c = \blacksquare \implies (lb = \blacksquare \oplus b = \blacksquare) \wedge (rt = \blacksquare \oplus r = \blacksquare) \wedge (rb = \square \vee rb = \blacksquare) \wedge ((l = \square \wedge t = \square) \implies \neg lt = \square)$$

Figure 2.5: Triangle shape constraint, while t, b, r, l denotes the cells at the top, bottom, right and left of the centre cell c and lt, rt, lb, rb denotes the corner cells at the left top, right top, left bottom and right bottom. The symbol \oplus denotes an *exclusive or*, a logical operation which results true if either of the two inputs are true, but not both.

Last, the number constraints (black cells with numbers) for the different sides, corners and internal cells need to be defined. Also here we will look at the internal cells and point out that the corners and sides are quite similar but they add more restrictions to the puzzle. A black cell with a three on a side of the board means that on each possible side there needs to be a triangle. The same cell in the middle of the field denotes that one side is not covered by a triangle.



$$\begin{aligned}
T' &:= (T = \blacksquare \vee \blacklozenge) \\
B' &:= (B = \blacksquare \vee \blacklozenge) \\
R' &:= (R = \blacksquare \vee \blacklozenge) \\
L' &:= (L = \blacksquare \vee \blacklozenge)
\end{aligned}$$

C	Asserts
0	$\neg T' \wedge \neg B' \wedge \neg L' \wedge \neg R'$
1	$(T' \wedge \neg B' \wedge \neg R' \wedge \neg L') \vee (\neg T' \wedge B' \wedge \neg R' \wedge \neg L') \vee$ $(\neg T' \wedge \neg B' \wedge R' \wedge \neg L') \vee (\neg T' \wedge \neg B' \wedge \neg R' \wedge L')$
2	$(T' \wedge B' \wedge \neg R' \wedge \neg L') \vee (T' \wedge \neg B' \wedge R' \wedge \neg L') \vee$ $(T' \wedge \neg B' \wedge \neg R' \wedge L') \vee (\neg T' \wedge B' \wedge R' \wedge \neg L') \vee$ $(\neg T' \wedge B' \wedge \neg R' \wedge L') \vee (\neg T' \wedge \neg B' \wedge R' \wedge L')$
3	$(T' \wedge B' \wedge R' \wedge \neg L') \vee (T' \wedge B' \wedge \neg R' \wedge L') \vee$ $(T' \wedge \neg B' \wedge R' \wedge L') \vee (\neg T' \wedge B' \wedge R' \wedge L')$
4	$T' \wedge B' \wedge L' \wedge R'$

Table 2.5: Number constraints.

By simply negating all the variables and combining them with a logical *or* (\vee) we can eliminate a specific, generated solution of the puzzle and therefore it is possible to get the next solution or to detect if a puzzle has only one. Each cell j has its own variable (c_j) and corresponding value (v_j) which represents the solution of a puzzle.

$$(c_0 = \neg v_0) \vee (c_1 = \neg v_1) \vee \dots \vee (c_n = \neg v_n) \quad (2.5)$$

A puzzle will be classified by using the provided statistics function from Z3. Especially the conflicts and decisions to find a valid model and therefore a correct solution of a Shakashaka puzzle instance will be used for the classification.

2.3 Experiments

This section will examine the behaviour of both solvers. We will look at different puzzles, varying in their levels of difficulty and artificial instances of Shakashaka.

	human-based solver						SMT solver	human winner
Puzzle	a	b	c	d	e	f	g	h
Nikoli Puzzle 1	0	0	0	0.031	0.0	0.031	0.047	43
Nikoli Puzzle 2	0	0	0	0.026	0.0	0.026	0.053	47
Nikoli Puzzle 3	0	0	0	0.061	0.0	0.061	0.055	47
Nikoli Puzzle 4	0	0	0	0.076	0.0	0.076	0.058	93
Nikoli Puzzle 5	0	0	0	0.080	0.0	0.080	0.053	107
Nikoli Puzzle 6	0	0	0	0.092	0.0	0.092	0.059	103
Nikoli Puzzle 7	0	0	0	0.213	0.0	0.213	0.098	188
Nikoli Puzzle 8	0	0	0	0.215	0.0	0.215	0.124	236
Nikoli Puzzle 9	1	1	2	0.499	0.027	0.526	0.272	577
Nikoli Puzzle 10	5	20	25	0.325	0.367	0.692	0.288	603
Cs 2012	31	46	77	0.525	1.968	2.493	0.593	965
Cs 2013 Jun	5	7	12	0.674	0.132	0.806	0.663	603
Cs 2013 Dec	20	11	31	0.618	0.818	1.436	0.628	587
Cs 2014 Jun	10	15	25	0.580	0.632	1.212	0.586	721
Cs 2014 Dec	4	0	4	0.778	0.099	0.877	0.675	540
Cs 2015	37	299	336	0.668	7.164	7.832	0.592	543
Cs 2016 Apr	10	34	44	0.555	0.702	1.257	0.599	431
Cs 2016 Dec	32	121	153	0.627	3.052	3.679	0.677	805
Cs 2017	22	45	67	0.611	2.256	2.867	0.605	452
Cs 2018	18	84	102	0.633	1.587	2.220	0.661	406

Table 2.6: Experimental results for instances at **nikoli.com**.

Cs := Championship

Let us first focus on easy, medium, hard and championship puzzles. On the official Nikoli website we can find *ten sample problems* of Shakashaka. Four of them are classified as easy, three as medium and other three as hard ones. The easy ones have the size 10×10 and 18×10 . Each of them is solvable in less than 0.080 seconds by both solvers. The medium ones are bigger (up to 24×14) and a correct solution is obtained in less than a quarter of a second for the human-based solver and less than one tenth of a second for the SMT solver. Table 2.6 illustrates the total time, in seconds, for the human based solver (column *f*), the SMT solver (column *g*) and the human winners (column *h*) to solve the puzzle instances. The easy and medium ones are solved without making any assumptions. For two out of three hard ones the human-based solver makes assumptions to get a valid solution. Table 2.6 provides additional information about the number of correct assumptions (column *a*), wrong assumptions (column *b*), total assumptions (column *c*) as well as the time taken by the apply rules step (column *d*) and the assumptions step (column *e*), both in seconds. All ten puzzles are solved in less than 0.7 seconds for the human-based and in less than 0.3 seconds for the SMT solver. Big differences between the solver can be seen by looking at more difficult puzzles. *Championship puzzles* have the size 45×31 . Most of them are solvable within three seconds by the human-based solver. We can see clearly that puzzles where a lot of assumptions are needed, take a lot of time. The championship puzzle from 2015 takes the most time because the algorithm makes a lot of wrong assumptions. The SMT solver manages to solve all championship puzzles within 0.7 seconds and we notice that it does take always quite similar time for this specific size of puzzles. Moreover, it has to be mentioned that each puzzle from Nikoli has a unique solution. Comparing the average solving time of Nikoli winners with this implementation we notice that the human-based solver is about 200 times, while the SMT solver is about 1000 times faster.

The artificial puzzles (see Figure 2.6) are of size $2n \times 2n$, where for each $n = 1, 2, \dots$ the board consists of $4 \times \sum_{i=1}^{n-1} i = 2n(n-1)$ black squares, and $4 \times (n-1)$ black squares containing the number two. Experimental results for $n = 2, 3, \dots, 40$ for both solvers are shown in Figure 2.6. We note a fast growth considering the puzzle size for the human-based solver. All instances of this kind are solved without making any assumptions. Nevertheless, for big puzzles the solver takes quite long. Dot cells take the most time to be set because they are the only cell values with possible restriction and non-restriction rules. Before setting a cell value, one of these rules need to be excluded. Each increase of n results in $4n$ additional cells, whereby $2n$ of these are dot values. If we consider the size of these puzzles regarding the championship ones we may focus on the puzzle with $n = 18$, which has the size 36×36 and therefore almost the same as the championship ones. Championship ones are solved in an average time of 2.47 seconds. Compared to them, the artificial puzzle with about 100 cells fewer, is solved in almost half of the time. Also in this experiment the SMT solver is much faster than the human-based one.

All measurements were made using a Lenovo Yoga 710.

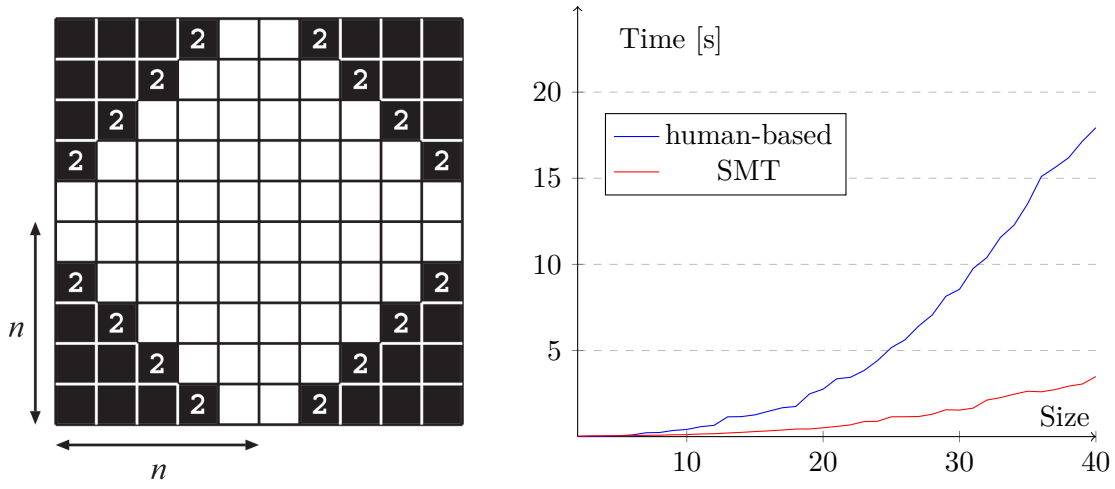


Figure 2.6: Experiment results of artificial instances with size $2n \times 2n$ for $n = 2, 3, \dots, 40$ for both solvers[1].

The idea of *apply rules* within the human-based solver sounds rather bad for puzzles without any numbered black values because at the beginning only one non-restriction rule for the black square can be considered. Nevertheless, if these squares are set in a regular distance this simple rule is very efficient and the solver can set dots and triangles. The rules of these newly set cell values are then used to continue filling the puzzle. The instance of size 10×10 from Figure 2.7 is solved in 0.012 seconds and the algorithm does not need to make any assumptions. The SMT solver obtains the valid solution of this numberless instance in 0.03 seconds.

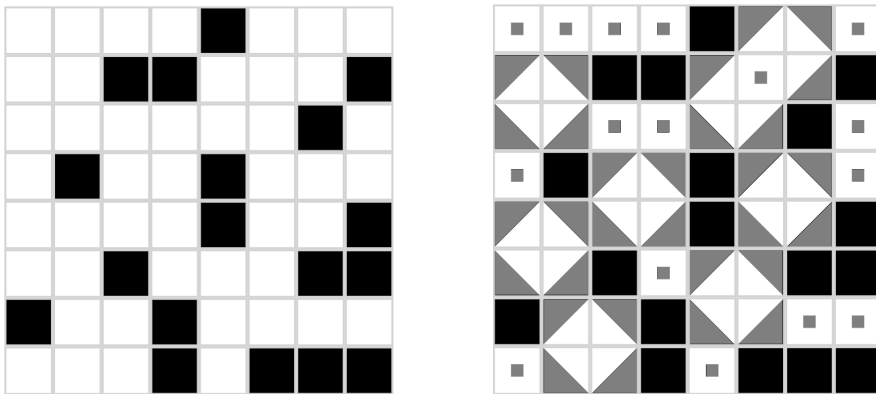


Figure 2.7: An instance of a Shakashaka puzzle without numbers[1].

Since the human-based solver makes assumptions, and the SMT solver searches a valid model, both provide a solution for puzzles where multiple solutions are possible (see Figure 2.8).

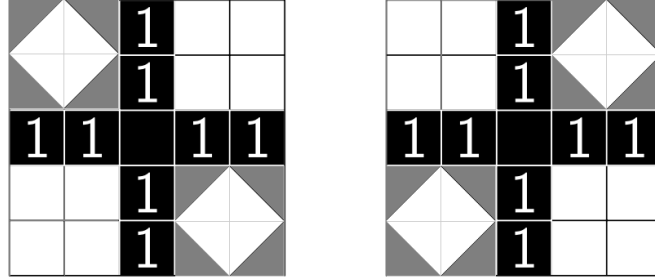


Figure 2.8: An instance of a Shakashaka puzzle with two possible solutions.

For this specific puzzle the human-based algorithm makes one assumption to get a valid solution. For a completely white board of size 10×10 the human-based algorithm finds a solution within 0.114 seconds by making 25 correct assumptions, while the SMT solver obtains a solution in 0.04 seconds.

3 Generator

In this chapter we will focus on generating random puzzle instances varying in their level of difficulty. Shakashaka puzzles from the official Nikoli website are crafted by hand due to the fact that generating random puzzles with adequate difficulty classification is a very difficult task.

3.1 Idea

Let us first define the basic idea of the puzzle generator. The core idea consists of the following three steps:

Step 1: Generate a random valid puzzle which may have multiple possible solutions.

Step 2: Make the generated solution unique by setting more restrictions to the puzzle (if desired by the user).

Step 3: Remove all triangles and dots from the puzzle, to let the user fill them by herself.

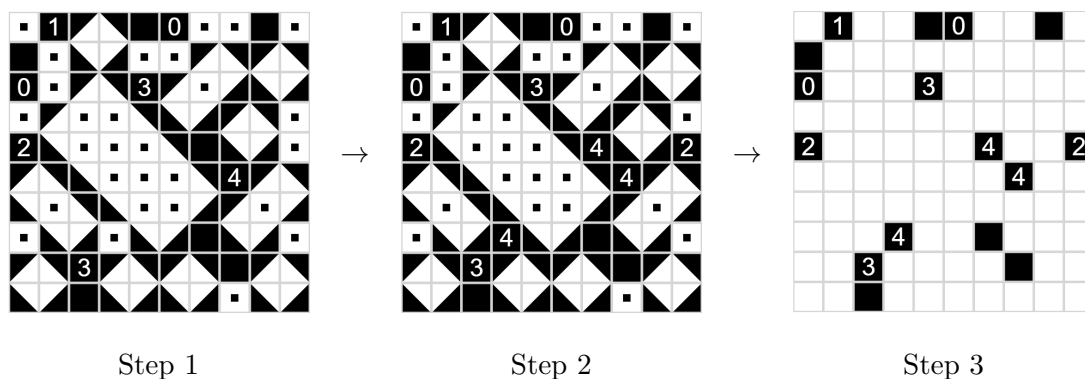


Figure 3.1: Basic puzzle generation idea illustrated with an example.

The following subsections will describe each step in detail.

3.1.1 Generate a Random Puzzle

The generator starts with an empty $n \times m$ field of white cells and looks for empty (white) fields. The first empty cell which is found is the white cell at the top left corner. This

cell will be inspected in terms of possible cell values in this specific location. A randomly chosen valid cell value will be set. At this point the generator uses the solver to fill the puzzle with triangles to a certain extent, without making any assumptions. This step guarantees that each cell value which is set needs to be in this location and the generator does not need to try other cell values because there are no other possible values. If there are still empty areas left, the algorithm searches the next white cell and repeats this process until a valid puzzle is generated. If a cell value leads to an invalid solution we simply backtrack and set another one.

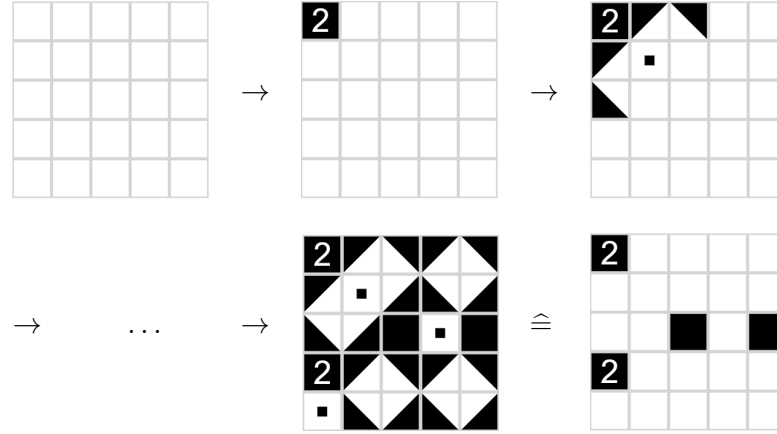


Figure 3.2: The generator starts with an empty board, sets a random valid cell value at the first white area and solves the puzzle without making any assumption. This step will be repeated until a valid solution is obtained. In this example the generator crafted a unique Shakashaka puzzle instance.

Difficulty can be controlled within this step by reducing or increasing the likelihood of placing numbered cells in the puzzle. If there are bigger white areas in the puzzle it is obviously more difficult for the user to solve it.

Generating a puzzle with multiple possible solutions is of course much faster than creating one with a unique solution. Nevertheless, if a user desires puzzles with unique solutions, he or she has the possibility to do so.

3.1.2 Making a Puzzle Unique (Approach 1)

In the previous section we examined the process of generating a valid Shakashaka puzzle instance which may have multiple solutions. So, within step 1 we got a valid puzzle and one possible solution of it. The goal of step 2 is to make the generated solution unique. This subsection will describe an approach on how to make a input puzzle unique by using the human-based solver (Algorithm 1). First of all we add all triangles which need to be in a specific location by applying the solver, again without making any assumptions. The puzzle is now solved to a certain extent, but not completely. If the puzzle still contains white cells we need to assume that there are multiple solutions possible because

at that location we can not set a specific cell value. At this point we look at cells with dots or black cell values around the white ones to replace them with numbered cells and therefore add more restrictions to the puzzle. We know if a surrounding cell has a dot or black cell value by looking at the previous generated valid solution. This step will be repeated until there are no more white cells to consider (as an example consider Figure 3.4).

Input: valid puzzle solution
Output: unique puzzle
 solve puzzle without making assumptions;
while *not all white cells considered* **do**
 get next white cell;
 if *one surrounding cell is a dot or black cell* **then**
 replace found cell with numbered one;
 solve puzzle without making assumptions;
 end
end

Algorithm 1: Make puzzle unique.

If there is no dot or black cell around a white one which can be replaced at the end of the puzzle the simplest approach is to generate a new puzzle because that signals that this specific solution cannot be unique (Algorithm 2).

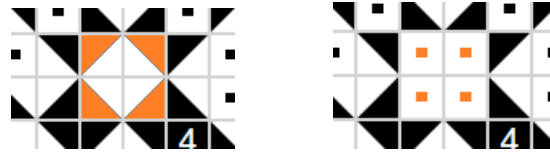
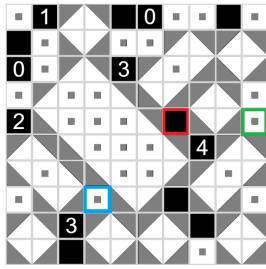


Figure 3.3: Part of a Shakashaka puzzle where the solution cannot be made unique.

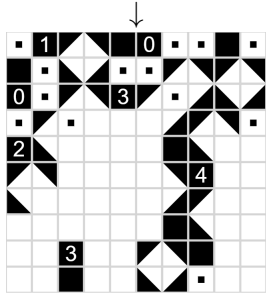
Input: potential unique puzzle solution from algorithm 1
if *no white cell in puzzle* **then**
 solution is unique
else
 solution cannot be made unique
end

Algorithm 2: Check if solution is unique.

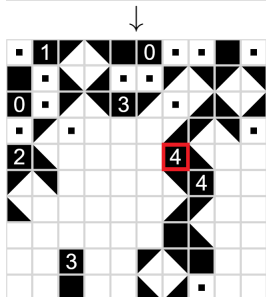
The difficulty will be affected within this step because new numbered cells will be added to the puzzle, but nevertheless in normal practice only few cells are changed to guarantee a unique solution.



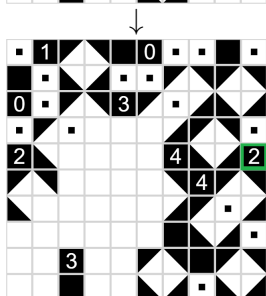
From step one we got a valid Shakashaka puzzle instance with a possible solution (coloured in grey). The goal is now to make the generated solution unique by adding more restrictions to the puzzle.



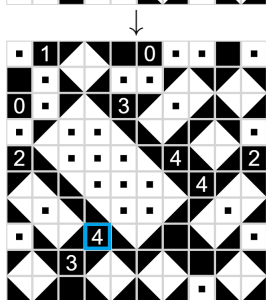
First of all the solver is applied to the generated puzzle without making any assumptions. It is now filled to a certain extent. All cells which are not set by now (all white ones) are not unique and there are other ones possible.



At this point we look at the first white cell with a dot or a black cell value to change that to a numbered black cell and therefore adding more restrictions to the puzzle. To determine which cell can be changed we look at the generated solution (coloured in red). The change of the black cell to a black cell with a four does not help to solve the puzzle by now, but it may help at the end of this process.



The next cell which will be replaced is a dot cell by a black two. We know also here that there is a dot cell at that location by looking at the solution we want to make unique. This replacement helped to solve the right side of the puzzle, but still the puzzle is not solved completely.



The change of the blue marked cell to a black four helped to solve the puzzle completely and therefore the solution is unique. The algorithm stops because there are no white cells left (the puzzle is now solved and unique).

Figure 3.4: Making a puzzle unique

3.1.3 Making a Puzzle Unique (Approach 2)

The idea of the second approach to make a puzzle unique using the SMT solver is quite similar: generate two solutions of the input puzzle and add restrictions that only one of them is valid (Algorithm 3). The valid solution will be kept and a new will be generated until there is only one solution possible and the puzzle has therefore a unique solution. To make only one solution valid we add more restrictions to the puzzle by comparing each cell of both solutions and looking at their differences. Also in this approach we look at cells where dots or black cells are around them to replace these blocks with numbered cells.

```

Input: empty input puzzle
Output: unique puzzle
solution1, solution2 = get solutions for puzzle;
while two solutions exist do
    | find difference in solutions;
    | add more restrictions to one solution to exclude other;
    | generate one new solution;
end

```

Algorithm 3: Make puzzle unique (SMT solver).

A big advantage of having two valid solutions is that we can replace either a cell in the first, or if that is not possible, we can replace a cell in the second solution. In the previous section we had to generate a whole new puzzle to make it unique. In some special cases also here both valid solutions cannot be made unique, but that case is very unlikely.

Also here the number of black cells inside a puzzle will be changed and therefore the difficulty of a puzzle will be affected to some extent. Nevertheless, only few cells are changed to make a unique puzzle.

To add an additional factor for random puzzles we set a random seed for the Z3 SMT solver. With this small modification it is possible to get different, unique instances on the same input puzzle.

As an example of how the algorithm makes a puzzle unique consider Figure 3.5.

3.1.4 Remove Triangles and Dots

In this step we simply remove the triangles and dots from the generated solution. If a user wants to see a solution we do not need to apply the solver because we already have a valid one.

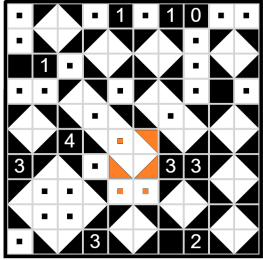
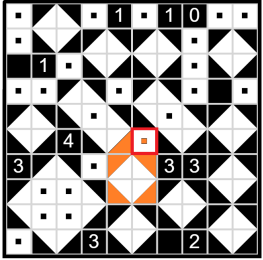
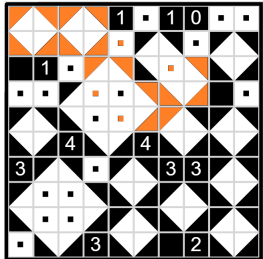
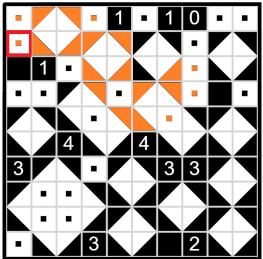
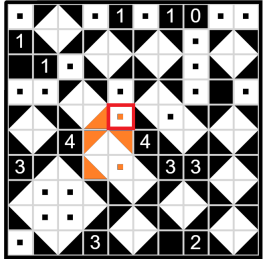
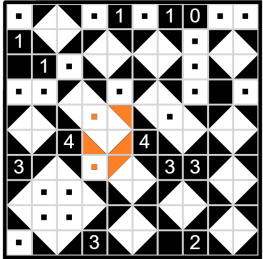
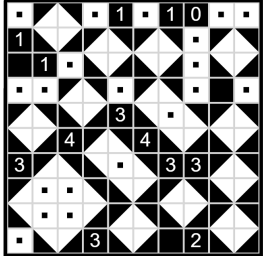
Puzzle solution 1	Puzzle solution 2	Description
		At the beginning the solver generates two valid solutions of the given input puzzle. These solutions are different (marked in orange) and therefore the solver is able to find a cell which can be replaced (indicated with a red square) by a numbered black one to eliminate puzzle solution 1 as a valid solved instance.
		Puzzle solution 2 will be kept and one new puzzle solution 1 needs to be generated. Again, we search for differences (marked in orange) and add more restrictions to the puzzle.
		Also in this step we can keep the solution 2 and generate a new solution 1. The cells differentiate now by the six orange ones, which can be made unique by changing the marked red cell to a black numbered square.
	/	Finally the algorithm does not find any other solution, which signals that the puzzle has now a unique solution.

Figure 3.5: Making a puzzle unique using the SMT solver.

3.2 Conclusion

A big advantage of separating the generation of a puzzle with possibly multiple solutions and making a puzzle unique is that the user can use these parts independently from each

other. A user has for example the possibility to make a puzzle he or she created unique. It is therefore easy to create artificial puzzles with a unique solution (see Figure 3.6).

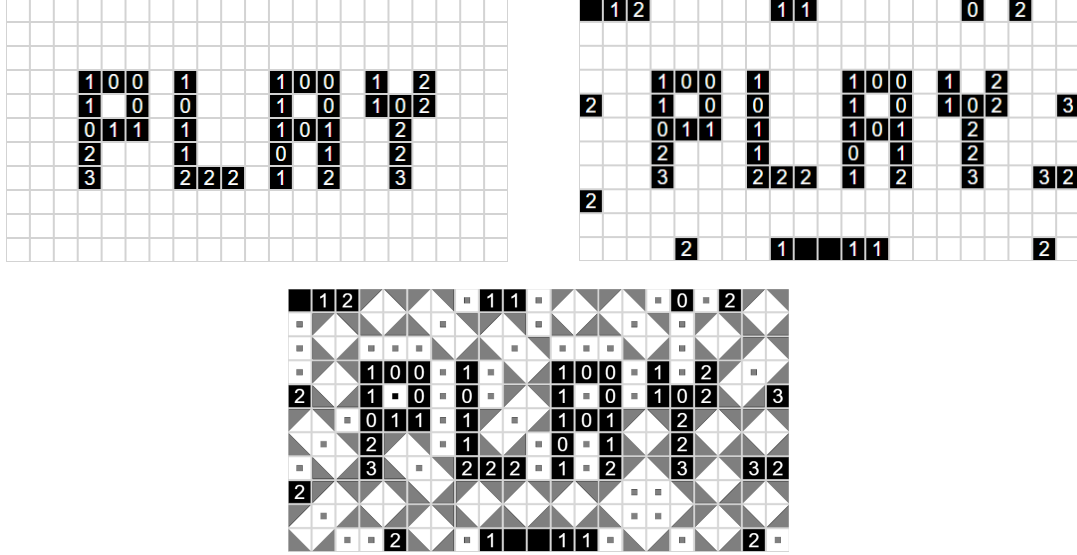


Figure 3.6: Left: input puzzle. Right: unique puzzle (*PLAY* consists only of numbers to keep the areas around it white). Bottom: solved puzzle.

Moreover, it is also possible for the user to generate a puzzle with multiple solutions if he or she wants to.

4 NP-completeness

This section will present the NP-completeness proof of Shakashaka from [1]. In more detail, we are going to present a bigger restriction: Shakashaka is NP-complete even if each black square is either empty or contains the number one. The proof is done by a reduction from a well known NP-completeness problem, planar 3SAT. Please note that most of the figures from this section were adopted from [1].

To show that a problem is NP-complete we need to show that it is NP-hard and in NP. The NP-membership is done by checking whether an assignment is valid or not. This step is easy and can be done in polynomial time. The challenging part is to prove NP-hardness by transforming the input of a planar 3SAT problem to an instance of Shakashaka. Five gadgets will be introduced to do that.

Let F be an instance of planar 3SAT, which consists of a set $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ of m clauses over n variables $\mathcal{V} = \{V_1, V_2, \dots, V_n\}$. Each clause C_i consists of three literals and the graph $G = (\mathcal{C} \cup \mathcal{V}, \mathcal{E})$ is planar, where \mathcal{E} contains an edge $\{C_i, V_i\}$ if and only if the literal V_i or \bar{V}_i is in the clause C_i .

Wire gadget

This gadget is used to join the literals and clauses. The main idea is to use the following pattern:

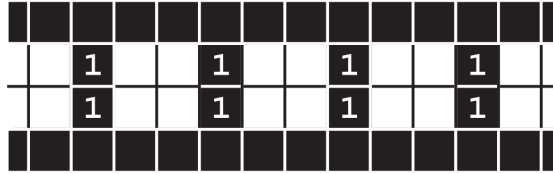


Figure 4.1: Empty wire gadget.

The pattern works as a wire propagating a signal. We have two possibilities to fill the white areas. Depending on the input (clause or variable value) the pattern will be filled and the signal will be propagated.

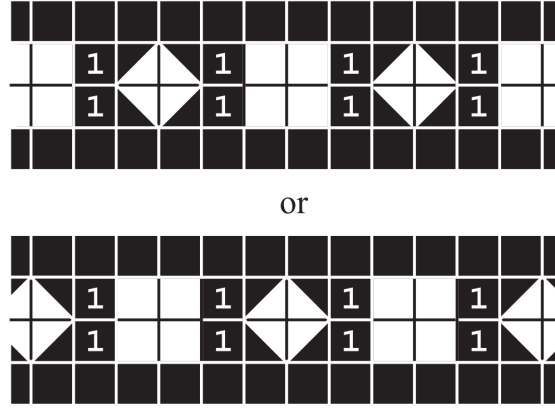


Figure 4.2: Filled wire gadget.

We define a 2×2 square containing four white squares as a ‘0’ and a diamond formed by four triangles as a ‘1’.

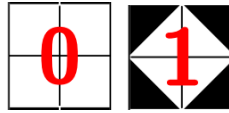


Figure 4.3: Zero and one representation.

Variable gadget

Also here we have two possible ways to fill the pattern. The value can be propagated using the wire gadget introduced in the previous subsection. Moreover, it is easy to obtain the negation of the variable by placing the wire at another position.

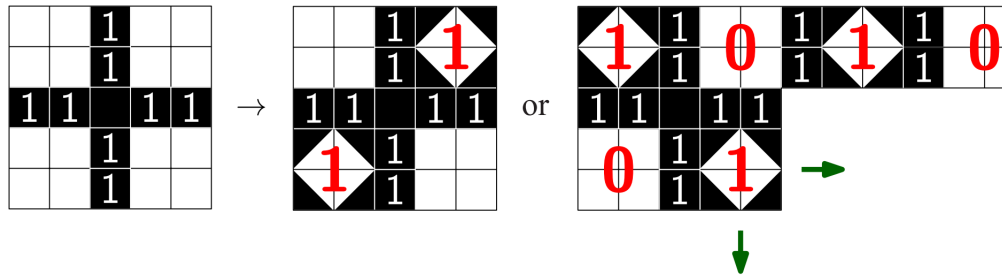


Figure 4.4: Variable gadget.

Split and corner gadgets

Using these gadgets we can increase the degree of a variable gadget. We use the corner gadget to propagate the signal through corners and the split gadget to duplicate a

variable. Both gadgets are based on the wire gadget.

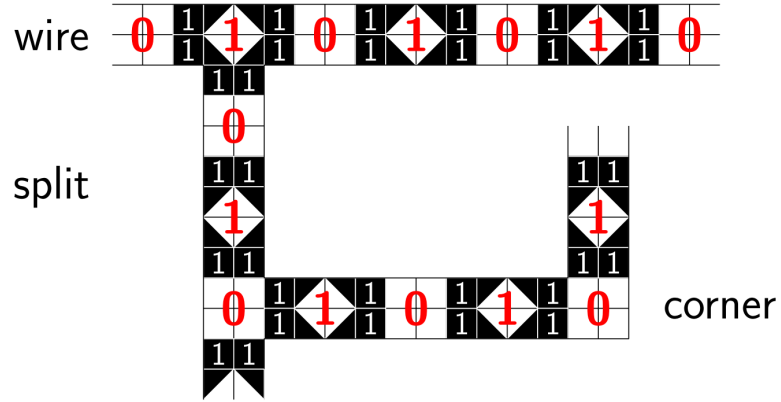


Figure 4.5: Split and corner gadgets.

Clause gadget

Figure 4.6 shows the clause gadget for a clause $C = \{x, \bar{y}, z\}$, combined with three variable gadgets. Within this pattern the variable x is responsible for the field x' , y for the field y' and z for the field z' . In this example, we have one variable gadgets for each variable. The negation of y is obtained by taking the variable value at another position.

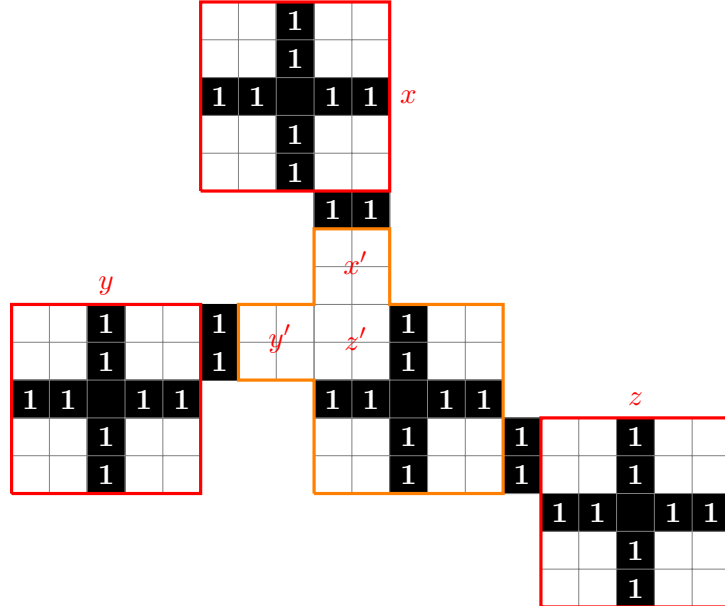


Figure 4.6: Clause gadgets (coloured in orange), combined with three variable gadgets (coloured in red).

According to the values of x , y and z we have eight possible cases. Only the case $x = z = 0$ and $y = 1$ violates the conditions of Shakashaka.

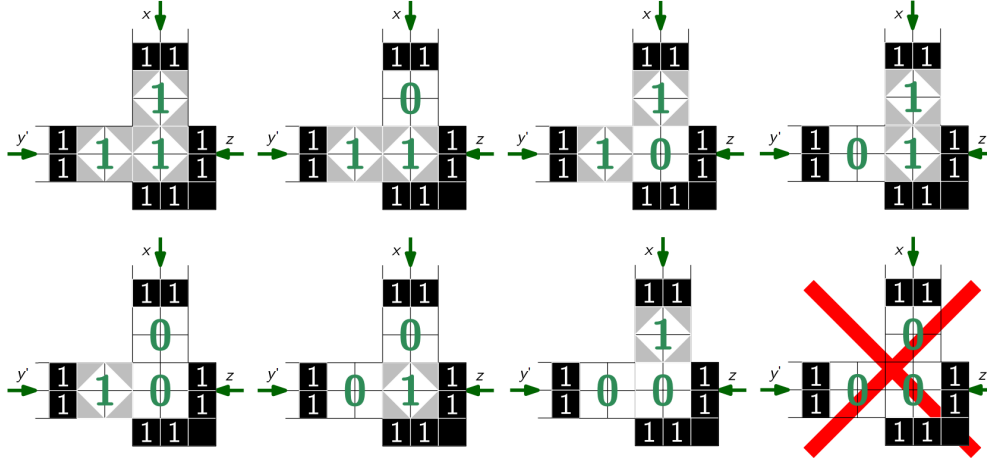


Figure 4.7: Cases of the clause gadgets.

Parity gadget

The wire, variable, corner and split gadgets are designed to fit into a 3×3 square tiling. The clause gadget does not and therefore we need to shift the position of the wires to fit the gadget by using the parity gadget.

The wire gadget is a sequence of units with size three. To shift the position we need a gadget which can be integrated in the wire gadget and meanwhile shift the position of the signal. The parity gadget with size five does this job.

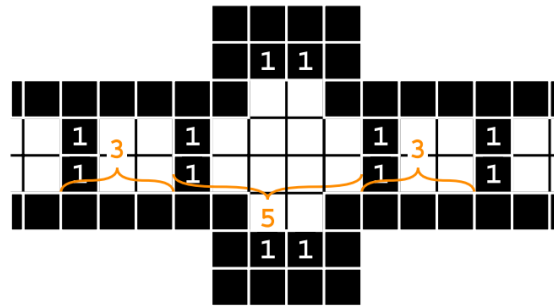


Figure 4.8: Parity gadgets.

To change the position of the wire we can simply join two copies of the parity gadget.

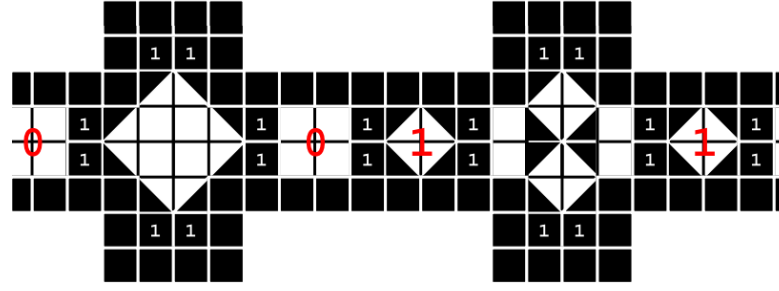
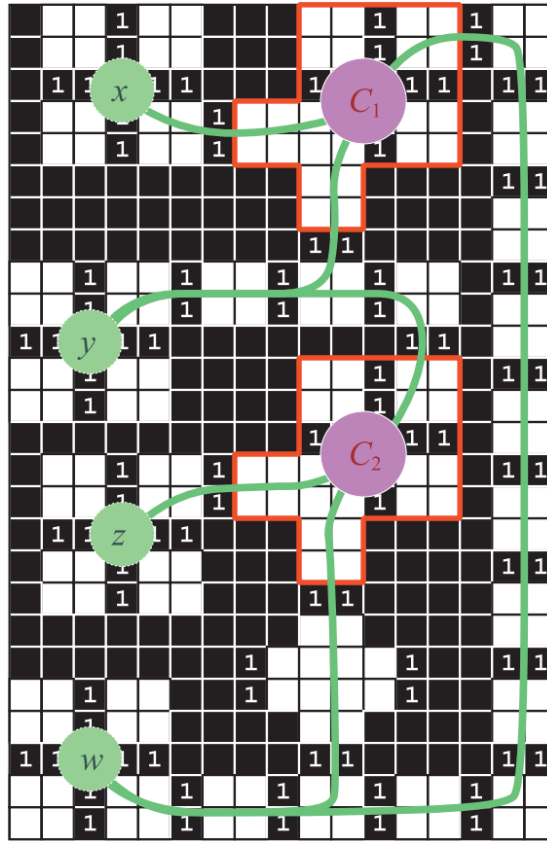


Figure 4.9: Parity gadgets.

Using these gadgets and exploiting the fact that G is planar we can arrange variable and clause gadgets on a sufficiently large board to join them without crossing by using the wire, split and corner gadget. Please note that each gadget is surrounded by black squares. At a clause gadget we need to arrange the position by using the parity gadget. Remaining gaps are filled with black "neutral" squares. A Shakashaka puzzle instance can be constructed in polynomial time. As an example, consider Figure 4.10.

Figure 4.10: Example for $f = C_1 \wedge C_2$, where $C_1 = \{x, \bar{y}, w\}$ and $C_2 = \{y, \bar{z}, \bar{w}\}$.

The resulting Shakashaka has a solution if and only if the formula F is satisfiable.

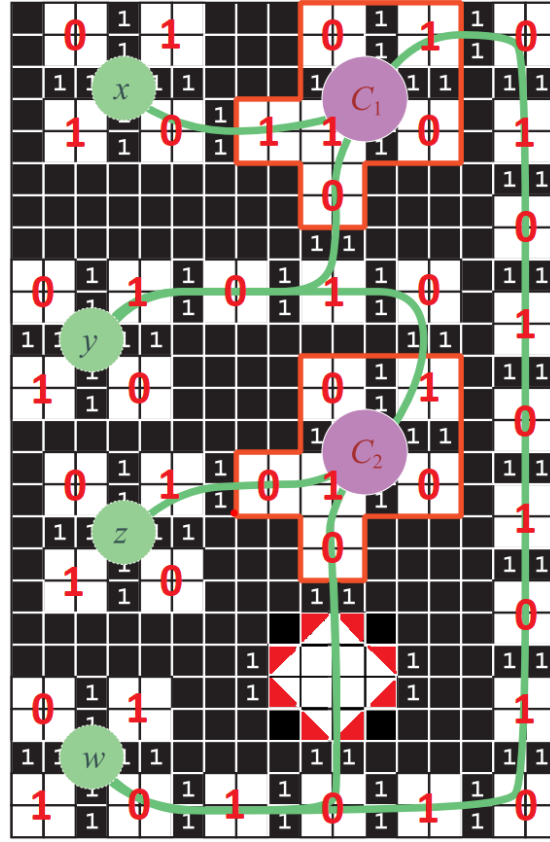


Figure 4.11: Filled example for $f = \mathcal{C}_1 \wedge \mathcal{C}_2$, where $\mathcal{C}_1 = \{x, \bar{y}, w\}$ and $\mathcal{C}_2 = \{y, \bar{z}, \bar{w}\}$ with $w = x = y = z = 1$.

We showed that we can represent an instance of planar 3SAT as an instance of Shakashaka in polynomial time. Moreover, we know that Shakashaka is in the class NP. Therefore, we can conclude that Shakashaka is NP-complete.

5 Web Application

Developing a user friendly tool, which allows to solve, generate and make new puzzles was a part of this bachelor project. The tool was implemented using *JavaEE* and *Java servlet faces* in the form of a web application. This section will give an overview of the tool, its most important features and the two available modes a user can use.

5.1 System Features

5.1.1 Solve Puzzle

At any time a user can choose between the human-based or the SMT solver to get a valid solution of the currently displayed puzzle. If a puzzle is not solvable the tool will display a corresponding error message. A puzzle will be classified after the solving process in four different categories: easy, medium, hard and extreme. Additionally a user can use the ‘Count solutions’ button which will calculate the amount of correct solutions of the displayed puzzle by using the SMT solver as described in section 2.2.

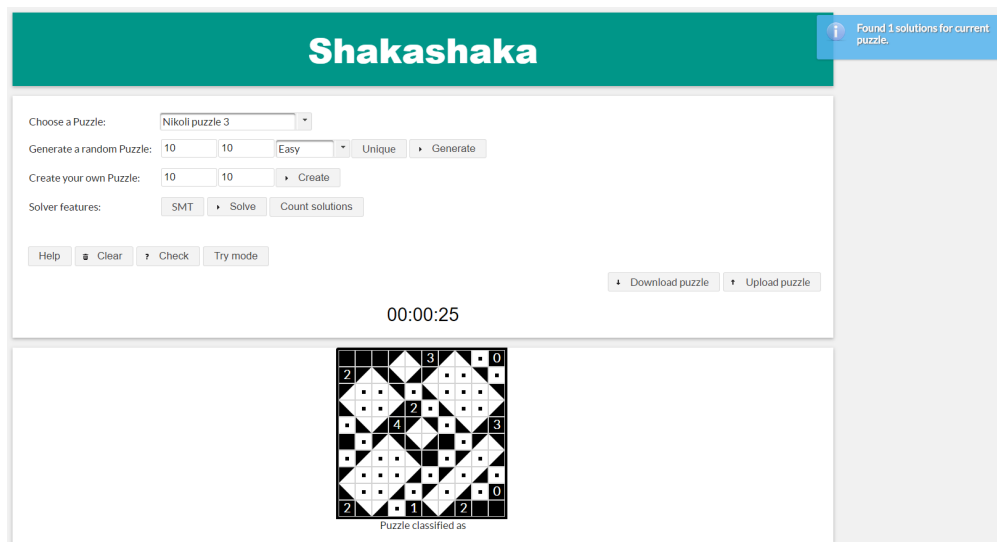


Figure 5.1: Solved puzzle.

5.1.2 Select a Puzzle

Generating a Random Puzzle

To generate a random puzzle the user can simply click on the ‘Generate’ button after specifying the size of the puzzle, the difficulty and if the puzzle should be unique.

Generate a random Puzzle:

Figure 5.2: Generate random puzzle.

According to the selected values the algorithm may take more or less time to generate a correct and valid puzzle. The generated puzzle will be displayed immediately and the user can start solving it.

Upload/Download Puzzle

To upload an instance of a Shakashaka puzzle the user has to follow a specific encoding. A valid instance is stored in a *plain ASCII* file, starting with the rows and columns of the board, followed by the actual board. The representation of the cells is shown in Table 5.1.








Cell	Encoding
	.
	b
	N
	A
	B
	C
	D

Table 5.1: Encoding table.

A black cell with a number inside will be encoded by using its value. Clicking the ‘Download puzzle’ button will convert the puzzle to its corresponding encoding. As an example, consider Figure 5.3, a small 5×5 Shakashaka puzzle instance.

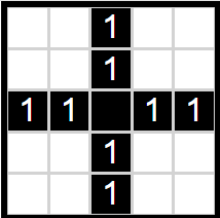
	5 5
	..1..
	..1..
	11b11
	..1..
	..1..

Figure 5.3: Encoding example.

Select a Nikoli Puzzle

A user has the possibility to select one of the ten Nikoli sample problems as well as one out of ten championship puzzle or other artificial instances of Shakashaka.



Figure 5.4: Generate random puzzle

5.2 Play Mode

Calling the web application puts the user into play mode immediately. This mode allows to place triangles inside the currently selected board and to remove them with a simple right click. Moreover the user has the possibility to go into a trial mode. Within that mode a user can solve problems just like in the regular mode with the only difference that he or she can undo all of them if his or her decisions were wrong or fix them if the assumptions are correct. Triangles which are set within the trial mode are coloured in red.

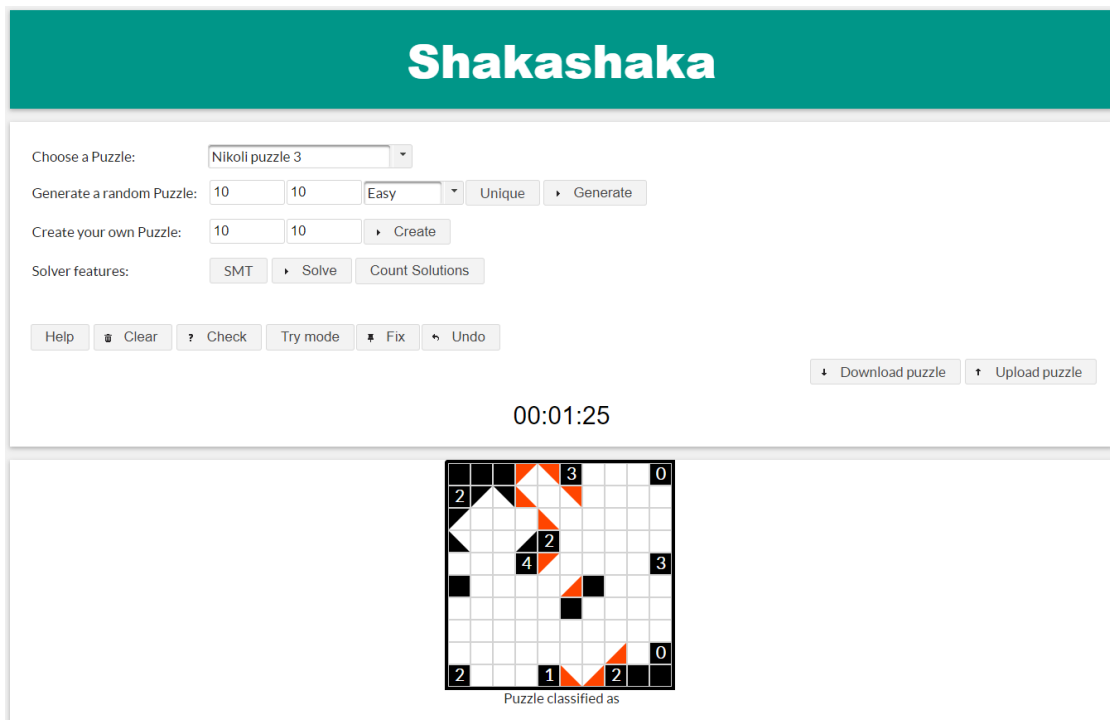


Figure 5.5: Graphical user interface.

A solution can be checked for validity by a simple click on the ‘Check’ button. If the puzzle is correctly solved the tool will open a pop-up with the time taken by the user to solve it (Figure 5.6). Otherwise the tool will highlight the incorrect block. All triangles of the board can be erased by using the ‘Clear’ button. A timer will always be started when a new puzzle is selected to keep track of how long a user takes to solve it.

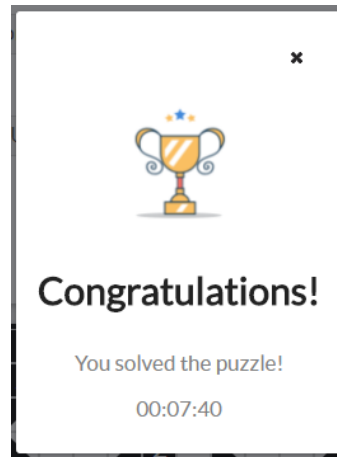


Figure 5.6: Congratulation message.

5.3 Edit Mode

If a user wants to create an own puzzle he or she has the possibility to do so. First, the size of the puzzle need to be specified and then he or she can click the ‘Create’ button. An empty board will appear and the user can click on a possible cell to place it into the board.

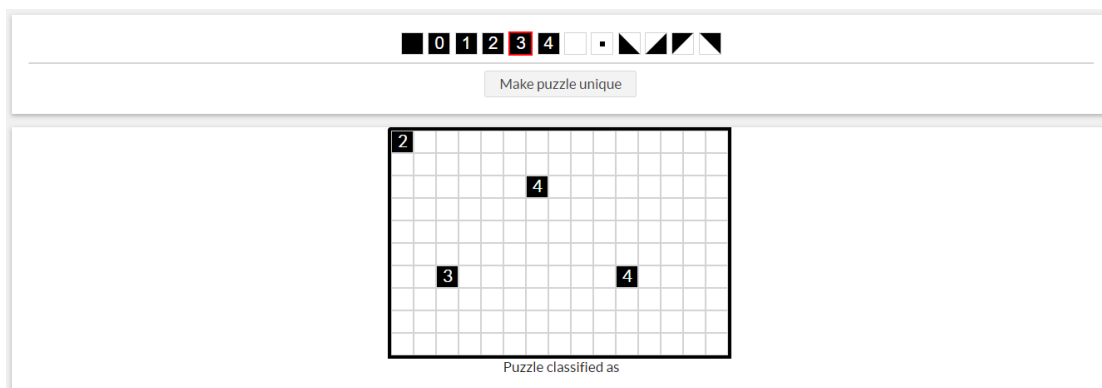


Figure 5.7: Play mode.

Additionally a user can make the puzzle unique, which means that the algorithm will add additionally cells to exclude other solutions until only one is valid. A created puzzle can be stored by clicking on the save button which will transform the puzzle to its correct encoding.

6 Conclusion

This thesis introduced the logic-based, paper and pencil puzzle Shakashaka. First of all the basic rules and the objective of the puzzle were explained to guide the reader closer to the topic. Chapter 2 presents two different solvers for Shakashaka puzzle instances. The first approach solves puzzles as a human would do, while the second one uses an SMT solver to find a satisfiable model of the instance. Even though the SMT solver is much faster than the human-based one, both solvers outperform human players. Before the introduction of the generator both solvers are defined, due to the fact that the generator is based on them. Moreover, both solvers manage to classify puzzles in their level of difficulty. The human-based solver does so by counting the number of assumptions which are needed to get a correct solution of the puzzle, while the SMT solver takes the decisions and the number of conflicts for finding a satisfiable model into account. Chapter 4 presented the NP-completeness of Shakashaka with easy understandable figures. Last but not least a web app was developed for letting players solve Shakashaka puzzle instances, generate random puzzles and even use one of the solvers to solve the puzzles. The tool has additionally features like a trial mode and an option to let the user create puzzles on his own.

6.1 Future Work

There are a lot of improvements which can be addressed in the future. First of all the generation of puzzles with predefined level of difficulty is quite hard and does not perform good in every case. Moreover, generating small puzzles work quite fast, but for bigger puzzles the algorithm may take very long.

Bibliography

- [1] E. D. Demaine, Y. Okamoto, R. Uehara, and Y. Uno. Computational complexity and an integer programming model of Shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 97(6):1213–1219, 2014.
- [2] Nikoli Co. Ltd. Nikoli.com, 2018. <http://www.nikoli.com/en/>.