Opleiding Informatica

Solving and generating Kuroshuto puzzles

Hanna Straathof

Supervisors:
Hendrik Jan Hoogeboom & Jeannette de Graaf

BACHELOR THESIS

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

28/01/2025

Abstract

Kuroshuto is a logical puzzle where squares in a grid have to be coloured black or white based on numbers that occur in some of the squares. We discuss two different methods for solving the puzzle. One uses an algorithmic approach and applies backtracking for difficult puzzles. The other translates the rules into CNF formulas and applies a SAT solver. The method to implement connectivity uses two different trees that span the board. This method has not been used before for connectivity in grid-like puzzles. Furthermore, we have developed a program for generating new puzzles. This thesis concludes with some experiments to compare the two solving methods and using the solvers to determine the difficulty of a puzzle.

Contents

1 Introduction

The Kuroshuto puzzle is a logic puzzle played on a $n \times n$ grid, typically between 6×6 and 14×14. Initially, the board consists of empty tiles and numbered tiles. Numbers are between 1 and 9 and do not exceed $n - 1$ if n is less than 10. The aim of the puzzle is to colour all the empty tiles either white or black according to the rules. Furthermore, numbered tiles are considered white tiles. Figure 1 presents an example of a puzzle and its corresponding solution, where the grey tiles represent empty tiles.

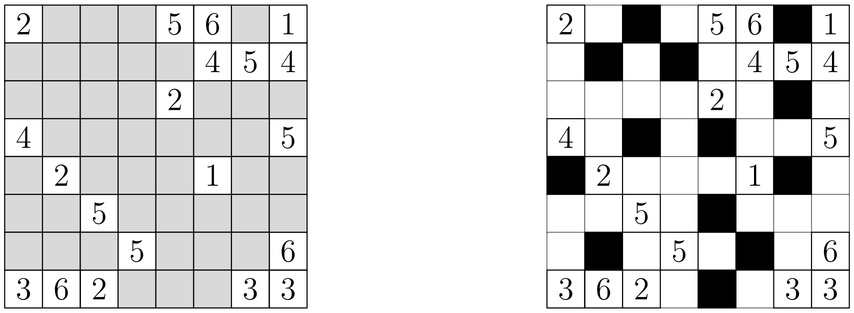


Figure 1: Kuroshuto puzzle and its corresponding solution

We will first provide some background information in Section 2. In Section 3 and 4 we discuss two different solving methods. The first uses an algorithmic approach and applies backtracking for difficult puzzles when necessary. The second translates the puzzle rules and board into CNF formulas and then applies a SAT solver. To implement connectivity for the SAT solver we use a new method where we construct two trees that span the board. In Section 5 we explain the generator we developed and in Section 6 we show our experiments. Finally, Section 7 contains our conclusion and our thoughts on future research.

This bachelor thesis has been commissioned by LIACS and supervised by Hendrik Jan Hoogeboom and Jeannette de Graaf.

1.1 Puzzle rules

The Kuroshuto puzzle has only three rules. The first rule states that black tiles cannot be each other's neighbour horizontally or vertically. This implies that black tiles can only have white neighbours. The second rule is that every numbered tile having value m must reach precisely one black tile in m steps in either horizontal or vertical direction. This means that in Figure 2 the cell with the circled number 2 reaches empty tiles in all possible directions at distance 2 and exactly one of those tiles must be coloured black. However, for the circled number 3 tile there are only two possibilities for a black tile. One of the other directions is outside the scope of the board and the tile three rows up is a numbered tile, which is considered a white tile.

The final rule is that all white tiles must be connected through a path of white tiles neighbouring each other horizontally or vertically. This means that the black tiles are not allowed to cut off one or more white tiles from the others. For example, in Figure 3 the white tile in the bottom-right corner is cut off from all other white tiles because of the two black tiles, which is not allowed.

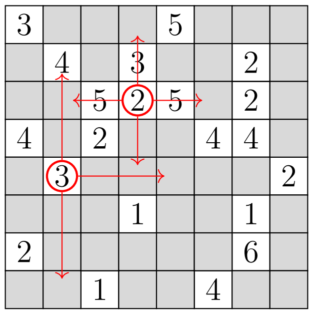


Figure 2: Arrows point towards tiles that the circled numbers reach

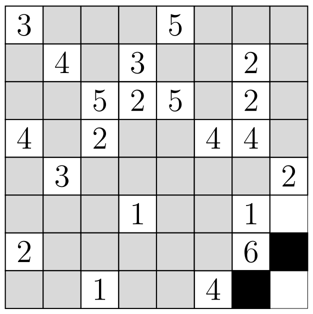


Figure 3: Black tile placement leads to a white tile being cut off

1.2 Origin

Most sources on the Kuroshuto puzzle are websites where you can play the puzzle. They contain only a brief explanation on the origin which usually consists of only one or two sentences. We sketch some background information based on the little information we have.

Many of the sources are conflicting on what the exact origin of the Kuroshuto puzzle is. Most agree that it was created in Japan or by a Japanese author and according to the GridPuzzle site the word "Kuroshuto" stems from the Japanese word "Kurochute", which means "black shot" [1]. However, who the exact author is, is unclear. The puzzle company Nikoli [2] is mentioned many times as the creator but the Kuroshuto puzzle cannot be found on their website. The Janko puzzle website believes that the puzzle might be inspired by puzzles from Naoki Inaba [3]. Naoki Inaba is a Japanese puzzle author.

2 Related Work

The Kuroshuto puzzle is a one-player combinatorial puzzle which falls under the combinatorial game theory. Many puzzles that fall under this theory, such as Sudoku, have been thoroughly researched. At the LIACS centre of Leiden University similar papers have been written for other puzzles. For example, Niels Heslenfeld analysed the Fobidoshi puzzle [4] and Rob Mourits [5] the Nurimeizu puzzle. However, to our knowledge, there has not yet been carried out any research on the Kuroshuto puzzle, which makes it interesting to study. In this section we will expand on the method we use for the connectivity constraint.

The Kuroshuto's rule that the white tiles must be connected implies that the black tiles cannot cut off some of the white tiles on the board. A lot of puzzles contain a connectivity constraint. In other scientific work, such as this study on pipe puzzles [6], the most common technique to enforce this rule using a SAT solver is as follows. First, a root tile is selected. Then, for every other tile, it is specified that the tile must be able to reach the root in at least $i + 1$ steps by requiring that one of its neighbours can reach the root in i steps [7].

This method requires you to estimate a maximum number of steps (i_{max}) that a tile could need to reach the root tile and this number grows as the size of the board grows. If a puzzle is of length and width n then its size is $N = n \times n$. The worst case occurs when a tile can only reach the root through every other tile on the board and, thus when $i_{max} = N$. In total, for a board of size N you need $N \times i_{max} = N^2$ variables in the worst case and the number of variables grows quadratically with the size of the board with this method. This number could be decreased by using binary to represent each variable. Then the number of variables necessary is $\log_2(N^2)$. This is a significant difference, but this method requires much more clauses to implement connectivity. We did not pursue this direction.

In [8] a different encoding is proposed which encodes a partial order between tiles, ensuring transitivity and forbidding cycles. This avoids variables stating the distance to the root. Still we need $O(N^2)$ variables to encode "there exists a path from u to v" between any pair of tiles u, v.

In this thesis we decided to build upon the idea of having two separate trees that enforce the connectivity constraint, which will be explained in detail in Section 4.2.1. This technique is introduced by Klaus Reinhardt [9], where he uses it to prove that a language of pictures is recognisable. In our case, we can apply the technique to ensure that all white tiles are connected. This allows us to only have a constant number of variables per tile, or $O(N)$ in total. As far as we know, this is the first time this method is used for connectivity in grid-like puzzles.

3 Algorithmic Solver

In this section we use an algorithmic method for solving the Kuroshuto puzzle and for this we developed a program in C++. It initially carries out the puzzle rules and specific patterns, without any guessing. If some tiles remain uncoloured after doing so, then it applies a backtracking technique. We will first explain how the puzzle has been translated into our program and describe the existing patterns, followed by an explanation of our techniques for solving this puzzle.

3.1 Initialization

3.1.1 The board

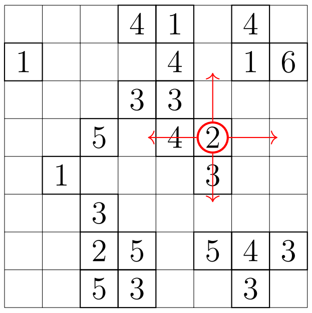
The program takes a text file that describes a puzzle and we have used the puzzles created by Otto and Angela Janko to test our program [3]. The source code of each puzzle can be found on their website and we chose to use the same representation as they use for convenience. Empty tiles are represented by a "-", black tiles by an "x", numbered tiles by their corresponding number and the remaining white tiles by an "o".

We represent the board with a two-dimensional character array. This allows us to assign the necessary character, as described above, to each tile. Numbers will thus be stored as a character and not an integer variable. Furthermore, we use an array of Booleans to assess whether a numbered tile reaches a black tile. When every tile of the board is coloured, we can confirm it is a correct solution if every Boolean in the array is set to true. We do not need to check if any other rule is violated since we only colour a tile if it does not violate any of the rules.

3.1.2 Lists

We use lists to store relations between tiles. For every numbered tile there exists a list of all the possible positions for a black tile. If one of those positions is coloured white, we remove it from the list. In Figure 4a the circled number 2 tile reaches four positions that could be a black tile and those are all added to the list of that number 2. We also want to know for every empty tile which numbered tiles reach that empty tile. Thus, we store for every empty tile a list of every numbered tile that can reach that empty tile. For example, in Figure 4b the four tiles that have an arrow pointed towards them all reach the circled tile and that circled tile could therefore be black. The benefit of using these empty tile lists is that when we colour a tile, we can immediately find which numbered tiles are affected instead of every time having to search the complete board to find them. The set of numbered tile lists and the set of empty tile lists are each other's inverse.

The numbered tile lists get updated whenever an empty tile is coloured white. That tile will be removed from every numbered tile list as it is no longer a possible position for a black tile that a numbered tile can reach. It is not necessary to update the numbered tile lists whenever a tile is coloured black. Once a number reaches a black tile, the corresponding cell in the Boolean array is updated and we do not need to look at that number any more to continue solving the puzzle. The Boolean array is used to assess whether the numbered tile reaches a black tile. The empty tile lists are never updated since this information never changes when solving the puzzle.


(a) Numbered tile (2) points to four empty tiles

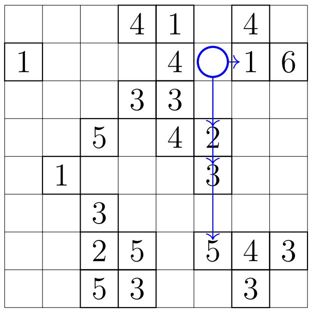

(b) Circled empty tile points to four numbered tiles

Figure 4: Relation between numbered and empty tiles

We use two two-dimensional arrays of vectors to store all these lists. If the puzzle is of size $n \times n$ then the arrays are also of size $n \times n$. In doing so we can easily find the list of position (i, j) on the board because it will be at position (i, j) in the array. One array stores all the numbered tile lists and the other all the empty tile lists. The lists themselves are stored as vectors because those have the advantage that we can modify their size. For the numbered tiles we know that there are a maximum of four positions, but we would like the list to be able to shrink so that we can immediately find how many possible positions are left. With the empty tile list we do not know the maximum of numbered tiles that could reach an empty tile. Each vector consists of pairs of integers that correspond to the coordinates of a tile on the board.

3.1.3 Patterns

Patterns can be derived from the rules of the puzzle. These patterns can contribute to finding the solution quicker. We have found two patterns for the Kuroshuto puzzle which we will call *diagonal position* and *diamond structure*.

We noticed that when a tile with number 1 has only two possible positions for a black tile and those options are diagonally adjacent from each other then we know for sure that the tile neighbouring both these options, and not being the number 1 tile, needs to be coloured white. This is because either one of the options will be black and the horizontal and vertical neighbours of a black tile must be white. Figure 5 shows that the circled number 1 tile has only two possibilities for a black tile, and therefore the tile marked red in the figure must be coloured white.

To limit the number of times we need to iterate over the complete field to check if any white tiles are cut off, we can search for what we call a diamond structure. A diamond structure consists of three black tiles surrounding a white tile. The fourth neighbour of the white tile cannot be black, otherwise it isolates the white tile. The example in Figure 6 shows you that the empty tile marked red cannot be coloured black or the black tiles form a diamond around a white tile. In this instance the empty tile is positioned at the left side of the diamond, but it could also be on the top, bottom or right.

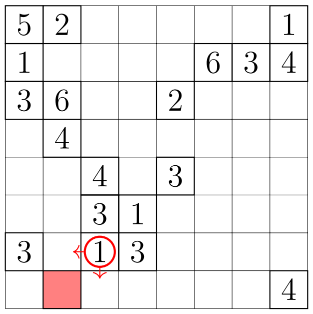


Figure 5: Red tile must be white because of diagonal position

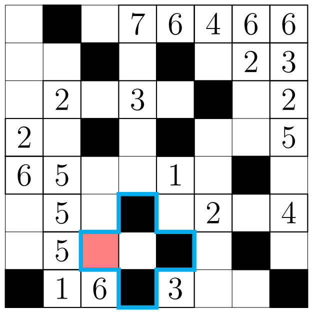


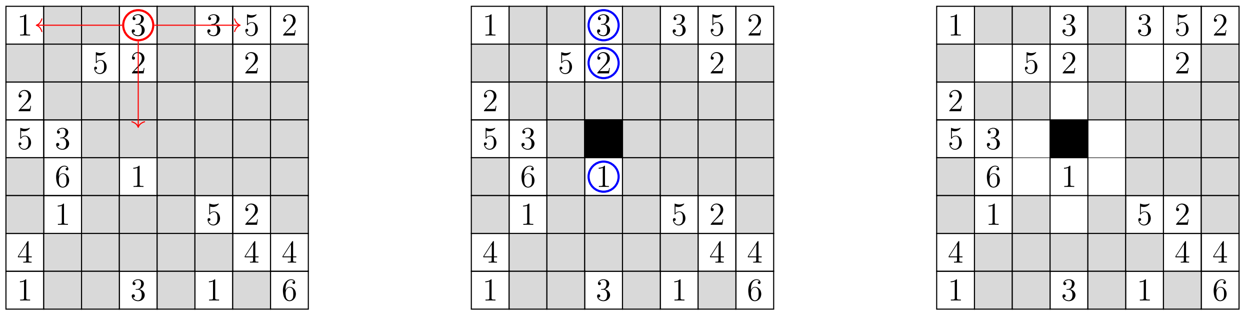
Figure 6: Red tile must be white because of diamond structure highlighted with blue

There exist more patterns in the Kuroshuto puzzle, but those do not occur as frequently as the patterns mentioned above. Searching for those other patterns in each puzzle may take more time than the potential speed-up they provide. Therefore, we have decided to focus only on the mentioned patterns.

3.2 Solving without guessing

We first try to solve the puzzle without making use of backtracking. Backtracking is more time-consuming, so it is preferable to avoid applying it. Overall, most puzzles and especially simpler puzzles can be solved without using backtracking. To solve puzzles this way we iterate through the board from top to bottom and left to right and only colour a tile when we are sure that it must be that colour. We stop reiterating through the board when we found the solution or when there are no longer any changes being made.

If we find a numbered tile when iterating, we use the Boolean array to check if it reaches a black tile. If not, we can verify if that tile has only one possibility left for a black tile using that tile's numbered tile list. In that case, we colour that tile black. There could be other numbered tiles that reach the newly coloured black tile and this information is kept in the empty tile list of that black tile. Since numbered tiles must reach precisely one black tile we can colour from every mentioned numbered tile, including the original numbered tile, the other tiles that they reach white. Subsequently, we can colour the horizontal and vertical neighbours of the new black tile white due to the rule that black tiles cannot be neighbours. Whenever we colour a tile white with this method, we update the numbered tile lists. This process is illustrated in Figure 7.



(a) Circled number (3) has one option left for a black tile

(b) Circled numbers are affected by black tile

(c) Affected tiles are coloured white

Figure 7: Execution flow when encountering a numbered tile

Upon encountering an empty tile during the board iteration, we assess if a numbered tile reaches that empty tile utilising its empty tile list. If that is not the case, we will colour that tile white. This is only allowed if the puzzle has a unique solution and then we assume that a tile can only be black if at least one numbered tile reaches it. If there are numbers that do reach the empty tile, we cannot be certain yet if the tile must be coloured black or not. However, we can assess whether some white tiles would be cut off from the others if the tile would be coloured black, in which case we can discard the tile as an option for the numbered tiles that reach it.

To check if one or more white tiles are cut off from the other white tiles when colouring the empty tile black, we first evaluate if the empty tile is part of a diamond structure. If not, then we use the flood fill algorithm to look further on the board. The flood fill algorithm is an algorithm that performs a breadth first search across the grid. It evaluates for every tile its reachable neighbours. With the Kuroshuto puzzle we want to know if all white tiles are connected and therefore, we only evaluate a horizontal or vertical neighbour if it is white. Eventually all white tiles must have been examined to be sure that they are all connected. When we are considering colouring a tile black, we temporarily colour it black and then use the flood fill algorithm on one of the white tiles to determine whether all tiles are still reachable. If there are still empty tiles while using flood fill then they are considered white tiles during that process.

3.3 Backtracking

If we are not able to solve the puzzle completely with the method described in the previous section, we apply a backtracking method to the puzzle. With the backtracking function we recursively guess the colour of a tile until we have found the solution. It takes much more time solving the puzzle with this method because the algorithm will try many possibilities to find the solution. Therefore, every time we guess a colour, we run the algorithm from the previous section. This reduces the number of recursive calls, making the algorithm quicker, and we can accurately determine how often backtracking is necessary to solve the puzzle.

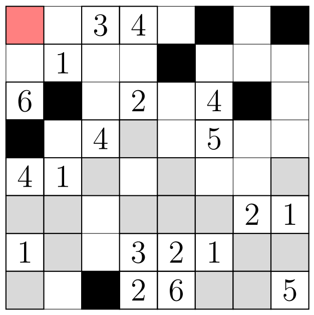
We start the algorithm with the tile in the top-left corner and move from left to right on each row. In each recursive call we first check if every tile on the board is coloured. If this condition is met, we have found a solution to the puzzle and our program is finished. Otherwise, the program continues and we will explain in the following paragraphs what happens next in each recursive call.

For every numbered tile we encounter, we assess how many possibilities it has that reach a black tile. In case there are no options left, we return false, since every numbered tile must reach one black tile. This is done to attempt to stop the recursion as soon as possible whenever the current colouring is no longer viable. After that, we move to the next tile.

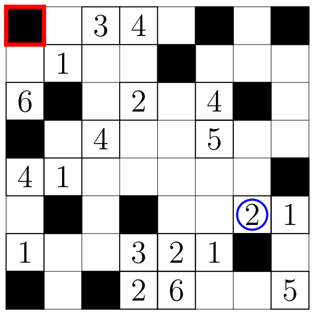
If tile (i, j) in the current recursive call is an empty tile, we execute the following steps:

1. Colour tile (i, j) black and all affected tiles white. The affected tiles are all the neighbours of the black tile as well as the other tiles that can be reached by every numbered tile that reaches the just coloured black tile as illustrated in Figure 7c. Following that, we solve the puzzle with the algorithm discussed in Section 3.2.
   (a) If the puzzle is solved, return true.
   (b) If not all tiles are coloured yet, we recursively call the function with the next tile.
      - If this call returns true, we have found a solution and the recursive call for tile (i, j) returns true as well.
      - If this call returns false, we continue to step (c).
   (c) If it does not lead to a correct solution, we will make tile (i, j) empty again. Furthermore, we also make all tiles empty that we had coloured white because they were affected by tile (i, j). Then we proceed to step 2.
2. Colour tile (i, j) white. Subsequently, we solve the puzzle with the algorithm discussed in Section 3.2.
   (i) If the puzzle is solved, return true.
   (ii) If not all tiles are coloured yet, we recursively call the function with the next tile.
      - If this call returns true, we have found a solution and the recursive call for tile (i, j) returns true as well.
      - If this call returns false, we proceed to step (iii).
   (iii) If it does not lead to a correct solution, the current colouring of the board is not part of the solution, regardless of the colour of tile (i, j). We make tile (i, j) empty again and return false.

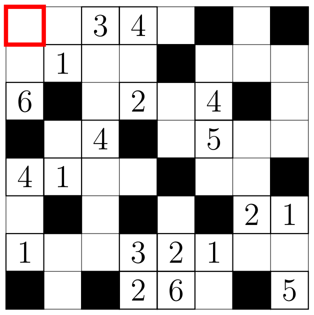
The algorithm from Section 3.2 colours every empty tile that is not reached by any numbered tile white. The backtracking algorithm is only applied after the other method and therefore we can assume that we are left with empty tiles that are reachable. Furthermore, we only apply backtracking if following the rules does not help us any further. This implies that every empty tile on the board is allowed to be black or white and thus we do not need to check in our backtracking algorithm whether we may colour a tile black or white.



(a) Empty tile marked red can be both black and white



(b) Colouring the red outlined tile black leads to incorrect solution



(c) Colouring the red outlined tile white leads to the correct solution

Figure 8: Finding the solution using backtracking

To illustrate the process we show an example in Figure 8. Figure 8a depicts the puzzle after we have tried solving using only the rules. In this state every empty tile could be black or white. The tile marked red will be coloured black to find out if that is part of the solution. After consecutively applying the algorithm from Section 3.2 and the backtracking method we are left with the board in Figure 8b. This is an incorrect solution because the number 2 tile circled with blue does not reach a black tile. Consequently, we eventually backtrack to the black tile marked red and colour that tile white. This is the correct colouring for that tile and thus we find the solution as shown in Figure 8c.

4 SAT Solver

In this thesis we will use a SAT solver as a second method to solve the Kuroshuto puzzle. The satisfiability problem, also known as SAT, is the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true, i.e. is satisfiable [10]. In this chapter we will first translate the board and its rules into logical formulas. Some of them will not yet be in the SAT solver's required form due to readability. After that we transform those formulas into the specific format the SAT solver can use and show how we have assigned 'numbered' variables recognised by the SAT solver to the abstract variables we have used in the construction below. We will conclude this chapter with an example of a CNF file. It is important to note that all grey tiles in the figures in this section correspond to black tiles in a puzzle, for clarity.

4.1 Initializing the board

The initial board of length and width n consists of $n \times n$ tiles, which each may contain a number. We decide that variable $t_{i,j}$ corresponds to a tile at position (i, j) starting from the upper left corner. Furthermore, $t_{i,j}$ is true whenever the tile is white and $t_{i,j}$ is false whenever the tile is black. Since numbered tiles count as white tiles, we initialise those tiles to true. If the outcome of the SAT solver is satisfiable, then the values of the variables for the tiles will show us the solution to the puzzle.

4.2 Converting the rules

The first rule is that no black tiles can be neighbours horizontally or vertically. This means that for every two neighbouring tiles at least one of them must be white. If we look for every tile (except for the last column) to the right and for every tile (except for the last row) below, then we cover every pair of tiles. For this we can use formula 1 which is a direct translation, except for omitting one of the two disjunctions at either the last column or the last row.

$$(t_{i,j} \vee t_{i,j+1}) \wedge (t_{i,j} \vee t_{i+1,j}) \tag{1}$$

The second rule is that for every numbered tile there should be precisely one black tile at distance m horizontally or vertically where m is the number of the tile. Take an arbitrary numbered tile t at position (i, j) with number m and assume that D is the set of possible black tile positions $(i-m, j), (i, j-m), (i+m, j)$ and $(i, j+m)$. The positions are in the direction up, left, down, right respectively. Precisely one black tile means that there should be at least one and not more than one. This can be translated into the formula underneath where the first clause explains that there cannot be more than one and the second clause that there should be at least one. We only use this formula on numbered tiles and we restrict D to the tiles within the boundaries of the board.

$$\bigwedge_{p,q \in D | p \neq q} (t_p \vee t_q) \wedge \bigvee_{p \in D} \neg t_p \tag{2}$$

4.2.1 Connectivity

The third rule is that all white tiles should be connected and the black tiles cannot cut off some of the white tiles from the board. We need a more extensive approach to implement this rule. To ensure all white tiles are connected we want to create a tree that connects all white tiles, called the *tile tree*. We use a tree structure because we need an acyclic graph with a root. Every node of the tree corresponds to a tile. An edge can exist between two tiles if they are horizontal or vertical neighbours. In this section we will explain how the tree is constructed.

We want to prevent any cycle from occurring in the tile tree because if a cycle exists then that either means that the black tiles cut off some of the white tiles or all white tiles are connected on the board but we would not know because they are not connected in the tree. To solve this we use the idea of adding a second tree to connect all vertices [9], which we will call the *vertex tree*. The vertex tree is similar in structure to the tile tree but now each tree-node corresponds to a board-vertex. We will not allow the arrows of each tree to cross with arrows from the other tree. The vertex tree prevents cycles in the tile tree because if there exists a cycle then there is at least one vertex inside that cycle that will have an outward arrow that needs to cross an arrow of the cycle. In Figure 9 the red dot marks the vertex that is within such a cycle. Vice versa, the vertex tree is also not allowed to contain any cycles because then some of the tiles cannot be included in the tile tree as shown in Figure 10.

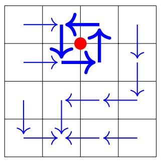


Figure 9: Red dot marks the vertex that exists in a tile tree cycle

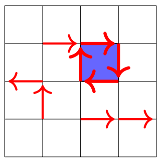


Figure 10: Blue tile enclosed by a cycle in the vertex tree

An example of a puzzle and a possible construction of the two trees is shown in Figure 11. The blue arrows are part of the tile tree and the red arrows are part of the vertex tree. The green tile in the figure marks the root of the tile tree and the grey fields show the solution to the puzzle.

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

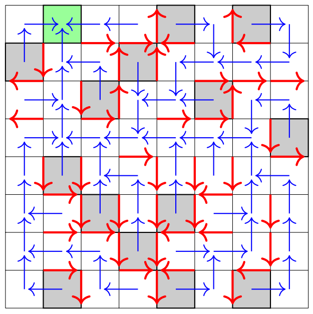


Figure 11: Puzzle and a possible construction of the two trees

To denote the formulas that describe both trees we need to declare some variables. First, we need to clarify the positions of tiles and vertices. As mentioned before, variable $t_{i,j}$ corresponds to tile (i,j) on the board starting from the upper left corner. Similarly, the vertices are also numbered from the upper left corner. Figure 12 shows the relation between the positions of tiles and vertices (see also Figure 16). On top of that we will need variables for the arrows. We will use the set P of variables $p_{i,j,d}$ to describe the outward arrow of a tile. An arrow $p_{i,j,d}$ points from tile $t_{i,j}$ to a neighbour in direction d. There are four possible directions d for an arrow; up, left, down, right. We will use the set D' with numbers 0, 1, 2 and 3 to represent those directions, respectively. Analogously, we will use the set Q of variables $q_{i,j,d}$ to specify the outward arrow of a vertex. Our method will only involve arrows that point within the board. A summary of all mentioned variables can be found in Table 1.

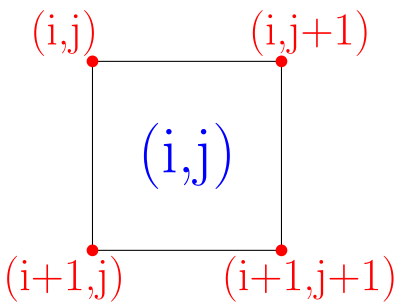


Figure 12: Relation between tile variables and vertex variables

Variable	**Description**
$t_{i,j}$	Tile at (i,j) is white.
d	Direction up, left, down, right correspond to 0, 1, 2, 3 respectively.
$p_{i,j,d}$	Tile tree has an outward edge in direction d from tile (i,j).
$q_{i,j,d}$	Vertex tree has an outward edge in direction d from vertex (i,j).

Table 1: Summary of used variables

Both trees have their own root and we need to choose this root wisely. Every tile in the tree needs to be able to reach the root and for the tile tree this means that the root must be a white tile. With the Kuroshuto puzzle we have the convenience of being able to choose a numbered tile for the root. We know at initialization that numbered tiles are white, thus, this makes them a fitting solution for the root. The numbered tile chosen as root also must be positioned at one of the edges of the board and we will explain later why this is important. Furthermore, we exclude the root tile from every formula in this method as it is not necessary to include it.

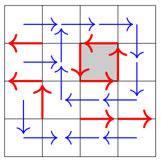
To choose a root for the vertex tree we could choose a vertex at one of the edges of the board. That being said, when we do this all vertex arrows on the edges could point along the edges toward the root and this way they do not add value to the solution of the puzzle. As a result, we choose to discard the vertex arrows on the edges of the board and view all vertices on the edges as one root.

To construct the two trees, we have to clarify that every tile and every vertex should have exactly one outward arrow. If there is no outward arrow, that tile or vertex is not part of the tree. The only exception to this is the root of each tree as the root only has inward arrows. It is important that all other tiles and vertices are part of their respective tree to make certain that no cycles can occur. If, for example, a vertex arrow is missing, it creates an opportunity for a cycle in the tile tree (see Figure 9), and vice versa (see Figure 10). The formulas we need for this rule are 3 and 4, one for each tree, and they are similar in structure to formula 2.

$$\bigwedge_{k,\ell \in D' \mid k \neq \ell} (\neg p_{i,j,k} \vee \neg p_{i,j,\ell}) \wedge \bigvee_{k \in D'} p_{i,j,k} \tag{3}$$

$$\bigwedge_{k,\ell \in D' \mid k \neq \ell} (\neg q_{i,j,k} \vee \neg q_{i,j,\ell}) \wedge \bigvee_{k \in D'} q_{i,j,k} \tag{4}$$

We need to prevent that a cycle in the vertex tree occurs around a black tile as this allows a cycle in the tile tree. The solution to this is to also include the black tiles in tile tree. Figure 13a shows a simplified example of how such a cycle could occur, leaving many tiles unconnected to each other and in Figure 13b this is solved by adding an arrow for the black tile. Black tiles can have an outward arrow to both black tiles and white tiles. However, black tiles will never connect directly in the tile tree in the Kuroshuto puzzle because the puzzle does not allow them to be next to each other horizontally or vertically. Therefore, they can only exist as leaves of the tree. White tiles, on the contrary, are only allowed to have an outward arrow to other white tiles. This is to make sure that we can still enforce the rule that all white tiles are connected on the board. We express this in formula 5 which consist of four clauses that declare that if a tile at position (i, j) is white and it has an outward arrow in a certain direction, then the tile where the arrow points towards must be white too.


(a) Black tile has no arrow, thus allowing cycles in both trees

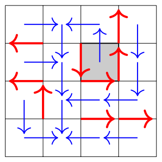

(b) Black tile has arrow

Figure 13: Example of how adding arrows for black tiles prevents vertex tree cycles

$$\begin{aligned}((t_{i,j} \wedge p_{i,j,0}) \longrightarrow t_{i-1,j}) \wedge ((t_{i,j} \wedge p_{i,j,1}) \longrightarrow t_{i,j-1}) \wedge \\ ((t_{i,j} \wedge p_{i,j,2}) \longrightarrow t_{i+1,j}) \wedge ((t_{i,j} \wedge p_{i,j,3}) \longrightarrow t_{i,j+1})\end{aligned} \tag{5}$$

Both trees cannot allow nodes to have any inward arrows coming from the same direction as where their outward arrow is going to as this would be a small cycle between two nodes. It could occur if the solution does have some white tiles that are separated by black tiles. In that case, at least one white tile would have an inward arrow from the same direction as their outward arrow. In Figure 14 an example of this is shown and the opposing arrows are highlighted with a blue circle. We use the two formulas 6 and 7 to express this rule. The first clause in the rule states that when there is an arrow pointing downwards from a tile $t_{i,j}$ there cannot be an arrow upwards from tile $t_{i+1,j}$, which is the tile below $t_{i,j}$. The second clause is similar but explains that there cannot be an arrow from the left to tile $t_{i,j}$ if tile $t_{i,j}$ already has an arrow to the right.

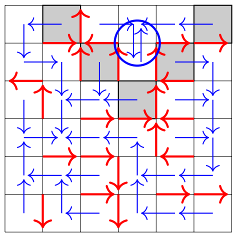


Figure 14: Incorrect solution causes opposing arrows because of vertex tree cycle

$$(p_{i,j,2} \longrightarrow \neg p_{i+1,j,0}) \wedge (p_{i,j,3} \longrightarrow \neg p_{i,j+1,1}) \tag{6}$$

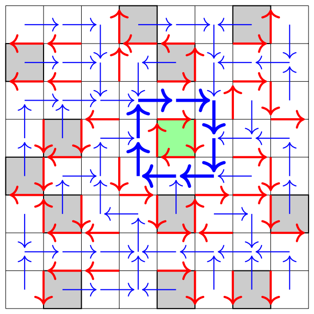
$$(q_{i,j,2} \longrightarrow \neg q_{i+1,j,0}) \wedge (q_{i,j,3} \longrightarrow \neg q_{i,j+1,1}) \tag{7}$$

Finally, to finish the construction of the two trees, we have to specify that tile arrows and vertex arrows cannot cross each other and we use formula 8. The formula consists of four parts, for each possible direction. Each part declares that if a tile arrow exists in a certain direction, then there can be no vertex arrow crossing it from either possible side. It is not necessary to use an analogue formula for the vertex arrows since formula 8 already forces that if a vertex arrow exists, a tile arrow cannot cross it.

$$\begin{aligned}&(p_{i,j,0} \longrightarrow \neg(q_{i,j,3} \vee q_{i,j+1,1})) \wedge (p_{i,j,1} \longrightarrow \neg(q_{i,j,2} \vee q_{i+1,j,0})) \wedge \\ &(p_{i,j,2} \longrightarrow \neg(q_{i+1,j,3} \vee q_{i+1,j+1,1})) \wedge (p_{i,j,3} \longrightarrow \neg(q_{i,j+1,2} \vee q_{i+1,j+1,0}))\end{aligned} \tag{8}$$

To prove that both trees do not contain any cycles and therefore all white tiles are connected, we use a proof by contradiction. If one or more cycles exist in either tree, we look at every innermost cycle in case there are nested cycles. The innermost cycle is either around one tile or vertex or more tiles and vertices. First, when it is around one tile or vertex then that tree-node could not have an outward arrow because that arrow would have to cross one of the arrows of the cycle. Since every tree-node must have an outward arrow, this situation could not occur. Second, if the cycle is around more tiles or vertices, at least one tile or vertex would have an inward arrow from the same direction as their outward arrow because every tree-node must have an outward arrow. However, opposing arrows between tree-nodes are not allowed in our method. Both situations lead to a contradiction therefore no cycle can exist in either tree and this method using two trees can be used for connectivity.

There is one exception that would render our proof invalid. That is when one or more cycles occur around the tile tree root. Figure 15 shows two examples of this situation where the green tile marks the root tile. We solve this by requiring that the numbered tile chosen for the tile tree root is positioned at one of the edges of the board. If there does not exist such a numbered tile, then we cannot solve the puzzle with this method. By placing the tile tree root at the edges, a cycle in the tile tree cannot go around the root. If a cycle in the vertex tree goes around the root, then it would prevent some of the white tiles to be connected to the root. This results in at least one pair of tree-nodes with opposing arrows between those cut off tiles which is not allowed. In conclusion, demanding that the tile tree root is positioned at one of the edges prevents cycles and makes our method realisable.


(a) Cycle directly around root

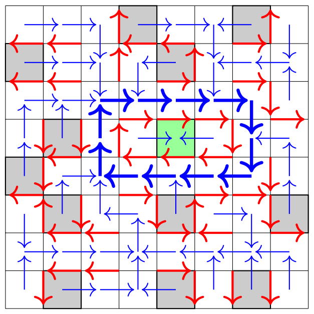

(b) Multiple cycles around root

Figure 15: Examples of cycles around the tile tree root

4.3 CNF form

We use the MiniSAT solver [11] as our SAT solver. MiniSAT "requires that its input be in conjunctive normal form (CNF)" [12]. CNF is built from expressions consisting of one or more clauses that are connected through AND operators and each clause consist of one or more literals connected through OR operators. Some of the formulas from Section 4.2.1 still need to be converted into CNF formulas since implication is not allowed in CNF. We use that $a \longrightarrow b$ is equivalent to $\neg a \vee b$. Furthermore, we also need De Morgan's law for the negation of a conjunction which states that $\neg(a \wedge b)$ is equivalent to $\neg a \vee \neg b$. For formula 5 we first remove every implication and then we apply De Morgan's law to convert it into formula 9. Similarly, removing the implication from formulas 6 and 7 leaves us with formulas 10 and 11, respectively, that are in CNF.

$$(\neg t_{i,j} \vee \neg p_{i,j,0} \vee t_{i-1,j}) \wedge (\neg t_{i,j} \vee \neg p_{i,j,1} \vee t_{i,j-1}) \wedge (\neg t_{i,j} \vee \neg p_{i,j,2} \vee t_{i+1,j}) \wedge (\neg t_{i,j} \vee \neg p_{i,j,3} \vee t_{i,j+1}) \tag{9}$$

$$(\neg p_{i,j,2} \vee \neg p_{i+1,j,0}) \wedge (\neg p_{i,j,3} \vee \neg p_{i,j+1,1}) \tag{10}$$

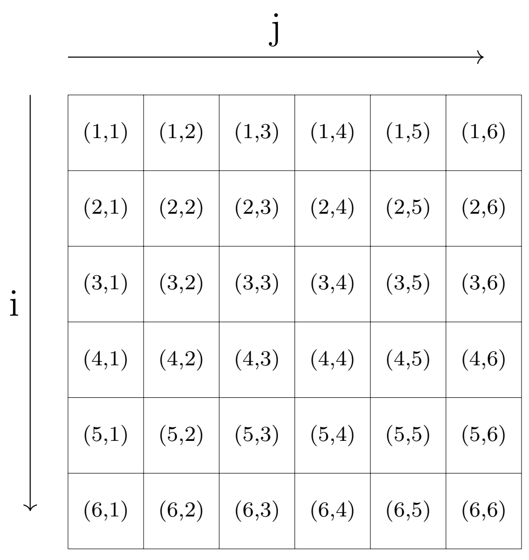
$$(\neg q_{i,j,2} \vee \neg q_{i+1,j,0}) \wedge (\neg q_{i,j,3} \vee \neg q_{i,j+1,1}) \tag{11}$$

Finally, we have to convert formula 8 into a CNF formula. We first apply De Morgan's law for the negation of a disjunction which means that $\neg(a \vee b)$ equals to $\neg a \wedge \neg b$. This gives us a formula consisting of four parts and each part is of the form: $c \longrightarrow (\neg a \wedge \neg b)$. We can split each part in two clauses $c \longrightarrow \neg a$ and $c \longrightarrow \neg b$ joined by a conjunction. This gives us in total eight clauses and for each clause we remove the implication which results in formula 12.

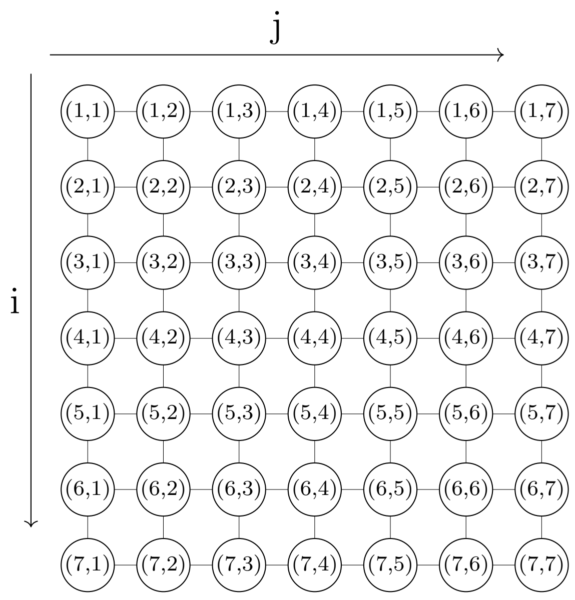
$$\begin{gathered}(\neg p_{i,j,0} \vee \neg q_{i,j,3}) \wedge (\neg p_{i,j,0} \vee \neg q_{i,j+1,1}) \wedge (\neg p_{i,j,1} \vee \neg q_{i,j,2}) \wedge (\neg p_{i,j,1} \vee \neg q_{i+1,j,0}) \wedge \\ (\neg p_{i,j,2} \vee \neg q_{i+1,j,3}) \wedge (\neg p_{i,j,2} \vee \neg q_{i+1,j+1,1}) \wedge (\neg p_{i,j,3} \vee \neg q_{i,j+1,2}) \wedge (\neg p_{i,j,3} \vee \neg q_{i+1,j+1,0})\end{gathered} \tag{12}$$

4.4 Assigning variables

To use the formulas mentioned before we need to map the used variables to variables recognised by the SAT solver. MiniSAT uses positive integers to represent variables. Therefore, we need numbered variables for every tile and every possible arrow. We will use a board of size $n \times n$. First, we need to show the numbering of tiles and vertices on the board before we can map the variables.



(a) Tile coordinates



(b) Vertex coordinates

Figure 16: Tile and vertex numbering on 6×6 board

We choose to number the tiles and vertices starting from the uppermost corner. Figure 16 shows the i-axis and j-axis of the board and the coordinates of each tile and vertex. To denote the tile variables for the SAT solver for every tile at position (i, j) we use formula 13 and in total we will need $n \times n$ variables for all the tiles.

$$var(t_{i,j}) = n \times i + j \tag{13}$$

Every tile and every vertex has four possible directions for an arrow and thus we need four variables for every tile and vertex to notate the arrows. In Section 4.2.1 we mentioned that d represents numbers 0, 1, 2 and 3 from the set D' which are the directions up, left, down and right, respectively. Since the first $n \times n$ variables are used for the tiles, we will use the next $n \times n \times 4$ variables for the tile tree arrows. Formula 14 shows how the variables for every tile arrow $p_{i,j,d}$ can be derived and $n \times n + 1$ is added to not overlap with the tile variables.

$$var(p_{i,j,d}) = (n \times n + 1) + n \times 4 \times i + 4 \times j + d \tag{14}$$

Finally, for the vertex arrows we use formula 15 and we add $n \times n + n \times n \times 4 + 1$ to prevent any overlap with other variables. Since we discard vertex arrows from the edges of the board, we only need $n \times n \times 4 - n \times 4$ vertex arrow variables. In total the SAT solver requires $9 \times n \times n - 4 \times n$ variables and thus the number of variables necessary grows linear with the size of the puzzle. In other words, SAT requires a constant number of variables per tile, or $O(N)$ in total.

$$var(q_{i,j,d}) = (n \times n + n \times n \times 4 + 1) + n \times 4 \times i + 4 \times j + d \tag{15}$$

4.5 Example

As an example we will show the CNF file for the puzzle illustrated in Figure 17. We have chosen a puzzle of size 4×4 because the number of clauses for bigger puzzles is too large to be showing their CNF file in this paper. A 6×6 puzzle already averages around 870 clauses.

			2
		1	
3			
3	1		

Figure 17: Simple 4×4 puzzle

In each clause in the CNF file a disjunction is represented by a white space and each clause must end with a white space followed by a zero. The file must start with a line of the form `p cnf NUMBER_OF_VARIABLES NUMBER_OF_CLAUSES` [12]. In our example we have left comments for the reader to distinguish which clauses are used for each rule. Lines that start with a `c` are comments. The following lines will be contained in the CNF file of Figure 17:

```
p cnf 128 331
c 1 black tile
c per number
12 2 0
-12 -2 0
3 11 0
3 6 0
3 8 0
11 6 0
11 8 0
6 8 0
-3 -11 -6 -8 0
-12 0
1 16 0
-1 -16 0
10 13 0
10 15 0
13 15 0
-10 -13 -15 0
c numbered tiles
4 0
7 0
9 0
13 0
14 0
c no 2 black
c neighbours
1 2 0
1 5 0
2 3 0
2 6 0
3 4 0
3 7 0
4 8 0
5 6 0
5 9 0
6 7 0
6 10 0
7 8 0
7 11 0
8 12 0
9 10 0
9 13 0
10 11 0
10 14 0
11 12 0
11 15 0
12 16 0
13 14 0
14 15 0
15 16 0
c 1 arrow per tile
-19 -20 0
19 20 0
-22 -23 0
-22 -24 0
-23 -24 0
 22 23 24 0
-26 -27 0
-26 -28 0
-27 -28 0
 26 27 28 0
-33 -35 0
-33 -36 0
-35 -36 0
 33 35 36 0
-37 -38 0
-37 -39 0
-37 -40 0
-38 -39 0
-38 -40 0
-39 -40 0
 37 38 39 40 0
-41 -42 0
-41 -43 0
-41 -44 0
-42 -43 0
-42 -44 0
-43 -44 0
 41 42 43 44 0
-45 -46 0
-45 -47 0
-46 -47 0
 45 46 47 0
-49 -51 0
-49 -52 0
-51 -52 0
 49 51 52 0
-53 -54 0
-53 -55 0
-53 -56 0
-54 -55 0
-54 -56 0
-55 -56 0
 53 54 55 56 0
-57 -58 0
-57 -59 0
-57 -60 0
-58 -59 0
-58 -60 0
-59 -60 0
 57 58 59 60 0
-61 -62 0
-61 -63 0
-62 -63 0
 61 62 63 0
-65 -68 0
 65 68 0
```

```
-69 -70 0
-69 -72 0
-70 -72 0
 69 70 72 0
-73 -74 0
-73 -76 0
-74 -76 0
 73 74 76 0
-77 -78 0
 77 78 0
 c 1 arrow per vertex
-101 -102 0
-101 -103 0
-101 -104 0
-102 -103 0
-102 -104 0
-103 -104 0
 101 102 103 104 0
-105 -106 0
-105 -107 0
-105 -108 0
-106 -107 0
-106 -108 0
-107 -108 0
 105 106 107 108 0
-109 -110 0
-109 -111 0
-109 -112 0
-110 -111 0
-110 -112 0
-111 -112 0
 109 110 111 112 0
-117 -118 0
-117 -119 0
-117 -120 0
-118 -119 0
-118 -120 0
-119 -120 0
 117 118 119 120 0
-121 -122 0
-121 -123 0
-121 -124 0
-122 -123 0
-122 -124 0
-123 -124 0
 121 122 123 124 0
-125 -126 0
-125 -127 0
-125 -128 0
-126 -127 0
-126 -128 0
-127 -128 0
 125 126 127 128 0
-133 -134 0
-133 -135 0
-133 -136 0
-134 -135 0
-134 -136 0
-135 -136 0
 133 134 135 136 0
-137 -138 0
-137 -139 0
-137 -140 0
-138 -139 0
-138 -140 0
-139 -140 0
 137 138 139 140 0
-141 -142 0
-141 -143 0
-141 -144 0
-142 -143 0
-142 -144 0
-143 -144 0
 141 142 143 144 0
 c tile arrows can't
 c point backwards
-19 -33 0
-20 -22 0
-23 -37 0
-24 -26 0
-27 -41 0
-28 -30 0
-35 -49 0
-36 -38 0
-39 -53 0
-40 -42 0
-43 -57 0
-44 -46 0
-47 -61 0
-51 -65 0
-52 -54 0
-55 -69 0
-56 -58 0
-59 -73 0
-60 -62 0
-63 -77 0
-68 -70 0
-72 -74 0
-76 -78 0
 c vertex arrows can't
 c point backwards
-103 -117 0
-104 -106 0
-107 -121 0
-108 -110 0
-111 -125 0
-112 -114 0
-119 -133 0
-120 -122 0
-123 -137 0
-124 -126 0
-127 -141 0
-128 -130 0
-135 -149 0
-136 -138 0
-139 -153 0
-140 -142 0
-143 -157 0
-144 -146 0
 c arrows from white
 c tiles only point
 c to white tiles
-1 -19 5 0
-1 -20 2 0
-2 -22 1 0
-2 -23 6 0
-2 -24 3 0
-3 -26 2 0
-3 -27 7 0
-3 -28 4 0
-5 -33 1 0
-5 -35 9 0
-5 -36 6 0
-6 -37 2 0
-6 -38 5 0
-6 -39 10 0
-6 -40 7 0
-7 -41 3 0
-7 -42 6 0
-7 -43 11 0
-7 -44 8 0
-8 -45 4 0
-8 -46 7 0
-8 -47 12 0
-9 -49 5 0
-9 -51 13 0
-9 -52 10 0
-10 -53 6 0
-10 -54 9 0
-10 -55 14 0
-10 -56 11 0
-11 -57 7 0
-11 -58 10 0
-11 -59 15 0
-11 -60 12 0
-12 -61 8 0
-12 -62 11 0
-12 -63 16 0
-13 -65 9 0
-13 -68 14 0
-14 -69 10 0
-14 -70 13 0
-14 -72 15 0
-15 -73 11 0
-15 -74 14 0
-15 -76 16 0
-16 -77 12 0
-16 -78 15 0
 c tile and vertex
 c arrows can't cross
-19 -102 0
-20 -101 0
-22 -101 0
-23 -104 0
-23 -106 0
-24 -105 0
-26 -105 0
-27 -108 0
-27 -110 0
-28 -109 0
```

```
-33 -102 0
-35 -118 0
-36 -103 0
-36 -117 0
-37 -104 0
-37 -106 0
-38 -103 0
-38 -117 0
-39 -120 0
-39 -122 0
-40 -107 0
-40 -121 0
-41 -108 0
-41 -110 0
-42 -107 0
-42 -121 0
-43 -124 0
-43 -126 0
-44 -111 0
-44 -125 0
-45 -112 0
-46 -111 0
-46 -125 0
-47 -128 0
-49 -118 0
-51 -134 0
-52 -119 0
-52 -133 0
-53 -120 0
-53 -122 0
-54 -119 0
-54 -133 0
-55 -136 0
-55 -138 0
-56 -123 0
-56 -137 0
-57 -124 0
-57 -126 0
-58 -123 0
-58 -137 0
-59 -140 0
-59 -142 0
-60 -127 0
-60 -141 0
-61 -128 0
-62 -127 0
-62 -141 0
-63 -144 0
-65 -134 0
-68 -135 0
-69 -136 0
-69 -138 0
-70 -135 0
-72 -139 0
-73 -140 0
-73 -142 0
-74 -139 0
-76 -143 0
-77 -144 0
-78 -143 0
```

5 Generating puzzles

We want to be able to generate puzzles that can be used to test our two solving methods. We implemented a program in C++ utilising the framework that we created for the algorithmic solver, which we describe in Section 3.1. The program first generates a puzzle including its solution and then evaluates whether the solution is unique. Once we have found a puzzle with a unique solution, we clear all tiles except for the numbered ones, resulting in a solvable puzzle.

It is important that the solution to the generated puzzle is unique. This means that for every empty tile, the correct colour for that tile should be derivable using the rules and the colouring of other tiles. Otherwise, if there are multiple solutions to a puzzle, then for some tiles you could decide yourself what colour you assign to them. If we discover that the puzzle does not have a unique solution, we keep generating new puzzles until we find one that does.

5.1 Generation method

The program starts with iterating over an empty board and randomly generating numbered tiles with a corresponding black tile. To determine how many numbers we generate, we use the Janko website [3] as a guideline. Of the 210 Kuroshuto puzzles available on the website, we counted how many numbered tiles occur per puzzle using our program. On average 32,39% of their puzzles are filled with numbers, with the least filled puzzle containing 26.56% and the most filled puzzle containing 45.31%. Therefore, we will approximately generate a number every third tile in our program.

If the board is of size $n \times n$, then the number must be between 1 and $n - 1$ and we will generate it randomly. Next, we need to colour one of the tiles that the number reaches black. We will check if one of those tiles is already black, and there are three possible outcomes. First, if multiple are black, the number is not allowed at this position and is removed. Second, if exactly one is black, we can keep the number and proceed to the next tile. Third, if it does not reach a black tile yet then we must generate it.

To generate a corresponding black tile, we will first randomly choose which of the four tiles the number reaches will be coloured black. If that tile is an empty tile and eligible following the rules of the puzzle, then we colour it black. Consequently, we colour all affected empty tiles white similar to when we are solving a puzzle as illustrated in Figure 7c. If the chosen tile is not eligible then we randomly chose another tile that the number reaches either until we have found a tile that can be coloured left or until there are no options left. If there are no options left, then we cannot place the numbered tile and we remove it.

We want to ensure that when you solve the puzzle you have to use the rule that no white tiles can be cut off. This makes the generated puzzles more complex. To do so, we will look at a black tile's diagonal neighbours as soon as it is generated, and check which are empty. For every empty diagonal neighbour we will assess if making one of those black would cut off some of the white tiles and if so, we will make that tile white. An example is shown in Figure 18 where the tile highlighted with red is a diagonal neighbour of the black tile. If the highlighted tile would be black then the top-left tile of the board would be cut off. We have decided to limit our attempt to enforcing this rule to only a local search, otherwise it becomes quickly very complex.

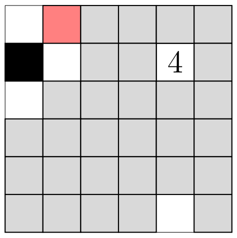


Figure 18: If tile marked with red would be black it would cut off the white tile

Even though we try to enforce that connectivity rule must be used for every generated puzzle, it is not always applied when solving if other rules also lead to that tile being coloured white. However, it does give puzzles a higher difficulty level if you do need to apply the rule.

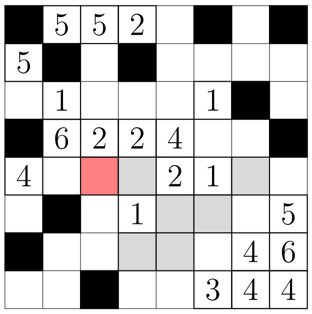
If we have not been able to generate a number for a few iterations of the board, then we will first colour an empty tile black and then find a position for a numbered tile. We choose both randomly the distance and direction from the black tile and then evaluate whether the numbered tile is allowed there. It is allowed if the tile is currently empty or white and if placing the numbered tile there does not result in the numbered tile reaching more than one black tile. We keep randomly selecting distances and options until we find one that is permitted. If there does not exist a possible combination, then we cannot make the tile black and we try to colour another black and repeat this process.

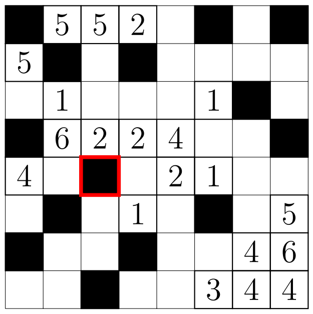
If we cannot find a tile that we can colour black after repeatedly trying, then we will not be able to generate a complete puzzle from the current state of the board. In that case we will remove all coloured tiles and start over with an empty board. It is more likely for this to happen with smaller sized puzzles.

5.2 Checking for uniqueness

We want to assess if the puzzle that we have generated has a unique solution. To do so, we solve the puzzle and while solving we check if we can find at least two solutions. As soon as we find more than one solution, we stop our search and return that it is not a unique puzzle. It is not necessary to try and find all solutions after we have found the second and this saves us time.

First, we use the same function as in Section 3.2, except for that we do not colour a tile white if no numbered tile reaches it. This function only colours a tile whenever that tile must be that colour following the rules of the puzzle which makes it deterministic and therefore the (partial) solution this function gives us must be unique. If this function can solve the complete puzzle than we know the puzzle has a unique solution. Alternatively, if the function from Section 3.2 cannot solve the complete puzzle, then there are either multiple solutions or there exists only one but we need a backtracking algorithm to find it. We use a recursive algorithm to determine which condition applies to the puzzle.


(a) Empty tile marked red can be both black and white


(b) Red outlined tile being black leads to a solution

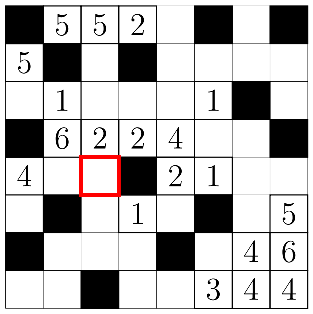

(c) Red outlined tile being white leads to a solution

Figure 19: Tile highlighted with red could be coloured both black and white

In the new recursive algorithm we try to solve the puzzle using the same strategy as in 3.3. However, when we have found that colouring a tile black gives us a possible solution, we will also check if colouring it white leads to a solution. For example, if we colour the empty tile that is marked red in Figure 19a, then that results in the board shown in Figure 19b. Next, we will backtrack to the tile marked red while erasing the colours of tiles we had just set. Now we will colour the marked tile white and this also ends in an allowed solution illustrated in Figure 19c.

We use variable *solutionFound* to keep track of if we have found a correct solution. Whenever the complete board is coloured and it adheres to all the rules, we check if *solutionFound* is set to true or false. If it is set to false, then this is the first solution we have discovered and we will change it to true. Otherwise, we have found a second option and the puzzle is not unique. We have variable *isUnique* to store whether we have already found multiple solutions and this will be set to false. As soon as this variable is set to false, we return in every recursive call to prevent the program from looking for more possibilities.

6 Experiments

In this section we will compare the two solving methods. First, we will look at the runtime of each method solving a generated puzzle. For the algorithmic solver we make a distinction between when backtracking is or is not applied. We assume that the algorithmic solver takes more time solving puzzles that require backtracking compared to puzzles that do not. It will be interesting to see if the SAT solver also spends more time on puzzles that the other solving algorithm requires backtracking for.

Furthermore, the 210 Kuroshuto puzzles on the Janko website are divided into eight different classes indicating their difficulty level [3]. In this chapter we will also evaluate if our two solving methods would divide the available puzzles approximately into similar difficulty levels.

6.1 Code

Both the algorithmic solver as well as the generator are developed in C++. We use MiniSAT as our SAT Solver [11]. MiniSAT processes a CNF file and determines whether it is satisfiable. We have developed a C++ program that converts each puzzle into the required CNF formulas and saves them to an output file. To automate the process, we have created a Bash script that first runs the C++ program to generate the CNF file and then passes it to the MiniSAT solver. Alterations to the bash script can be made to run experiments on the SAT solver. To measure how long it takes to solve a puzzle with the SAT solver we combine the time it takes to generate the CNF file and the time the MiniSAT takes.

With the algorithmic solver we log each time the function is called recursively or when other techniques were used. These other techniques will be mentioned in Section 6.3. This logging allows us to record for each puzzle which techniques were used. To use these records, we have written a C++ program that sorts all statistics such that they can be easily copied into a table.

6.2 Using generated puzzles

We will first evaluate the performance of the generating algorithm by measuring how much time it takes on average to generate unique puzzles. Table 2 shows the average time required to generate a puzzle per board size across 100 puzzles. It is immediately evident that the difference in time generating puzzles of size 8×8, 10×10 and 11×11 is very small. Solving 12×12 takes much more time relatively compared to the other puzzle sizes. This can be explained because the puzzle size difference in the table grows exponentially and therefore larger puzzles will take more time. An outlier is the average generating time for 9×9 puzzles. We noticed that sometimes generating a puzzle takes much longer compared to generating other puzzles of that size. This could occur for example if it takes much longer to find a unique puzzle and could have happened while generating the 9×9 puzzles.

Size	**Average time**
8×8	0.1034
9×9	1.2767
10×10	0.3268
11×11	0.4589
12×12	2.3676

Table 2: Average time in seconds generating a puzzle per board size across 100 puzzles

We can utilise the generated puzzles to compare the two solving methods by analysing the time each method takes, distinguishing between puzzles that do and do not use backtracking to clearly observe the difference in speed. We generate 100 puzzles per board size and measure how long both solving methods take. The results are entered in Table 3. If the algorithmic solver used backtracking, then the results are placed in the third column, otherwise in the second.

Size	Algorithmic solver without backtracking	Algorithmic solver with backtracking	SAT solver
8×8	0.014478	0.013525	0.061488
9×9	0.013619	0.016785	0.063569
10×10	0.012668	0.012616	0.062334
11×11	0.014355	0.017142	0.072780
12×12	0.012073	0.013057	0.065542

Table 3: Average time in seconds solving a puzzle per board size across 100 puzzles

From Table 3 we can conclude that the algorithmic solver does not take longer solving puzzles that require backtracking. In some cases, it is even quicker than with puzzles without backtracking. This is surprising as we assumed that puzzles without backtracking would be quicker to solve. We also notice that the algorithmic solver is approximately five times quicker than the SAT solver. However, the solving time for both solvers is very short and on average does not change for these board sizes. This indicates that both solvers are very effective.

6.3 Comparing difficulty level

It will be interesting to see if our two solvers will give a similar classification as the one provided by the Janko website [3]. We will determine a classification that labels how difficult to solve each puzzle is for humans and compare this to the one the Janko site decided.

The Janko website contains 210 Kuroshuto puzzles that are divided into eight difficulty levels. The reasoning for this division is not explicitly mentioned, but a general explanation is provided that holds for every type of puzzle on the website [13]. Only average solving time has been used to measure the difficulty level and the scale is approximately exponential. For each puzzle on the website the source code is available which allows us to easily use them as input.

To determine our own classification, we will primarily consider whether certain techniques were applied by the algorithmic solver. Since this solver is most similar to a human approach, it shows us the steps a human would have to take to solve the puzzle. Therefore, it also tells us how difficult it is for a human. The first and most important complicated technique is backtracking. For each puzzle we measure if backtracking is used and if so, how often. We will then divide all 210 puzzles in sets based on the number of times backtracking is applied and these sets we will call *backtracking sets*.

Furthermore, we notice that many puzzles can be solved without using the rule that all white tiles must be connected. This rule forces a person to look at the board from a different perspective and to search for patterns of black tiles that almost cut off some white tiles. There is also a difference in difficulty if you only have to check for a diamond structure, which is immediately visible and takes less time, or if the complete board must be checked. Finally, we also check if the diagonal position that was mentioned in Section 3.1.3 is used.

To make a distinction between puzzles within each backtracking set we will use a point system. Checking for a diamond structure or for a diagonal position is considered equally hard. If one of these occur and are used, then the puzzle will receive for each of these techniques one point. If the solver uses flood fill to determine if some white tiles are cut off, then the puzzle receives two points. For each puzzle the points are added together and this number can be from zero up to and including four. Thus, we can sort all puzzles in each backtracking set into at most five groups. Within each group puzzles are sorted based on the solving time by the algorithmic solver.

All results are entered in Table 4. The first column indicates how many times backtracking was applied, and each row corresponds to a group within that backtracking set. The first row of each set contains the easiest puzzles and the last row the hardest. The puzzle numbers are in bold and behind each puzzle number the time the SAT solver took is added. We have chosen to disregard puzzle size as a factor in our own classification since this is already contained in the solving time. Larger puzzles inherently take longer to solve, due to the existence of more tiles that require examination.

We notice that most puzzles can be solved without using backtracking. This is beneficial because using backtracking takes more time. Most puzzles can even be solved without checking if some tiles are cut off or if there exists a diagonal position. Therefore, we can assume that most puzzles on the Janko website are easy to solve for our algorithm, and thus for humans too.

Our classification is different from the one the Janko website provides. They only have eight different levels and most of the puzzles are placed into level three or four. On the other hand, we have much more levels and most of our puzzles are sorted into the easiest level. Furthermore, there are some similarities in puzzles that both classifications label as difficult. For example, according to the Janko website puzzles 19 and 80 are the most difficult. We consider puzzle 80 also as one of the most difficult puzzles. Although, in our case puzzle 19 only requires using backtracking once and there are many puzzles that are more difficult.

It is interesting that the time the solving method using the SAT solver takes is very consistent over all puzzles. It does not seem impacted by the difficulty level of a puzzle. This is a similar conclusion to the one we found in Section 6.2.

0:	**47** (0.0492), **87** (0.0558), **36** (0.0556), **75** (0.057), **11** (0.0686), **43** (0.0597), **32** (0.0599), **71** (0.0512), **39** (0.0559), **148** (0.0541), **3** (0.0559), **204** (0.0707), **35** (0.0545), **73** (0.0568), **52** (0.0545), **77** (0.0494), **156** (0.0601), **206** (0.0595), **101** (0.0591), **29** (0.063), **115** (0.0625), **21** (0.0612), **194** (0.0548), **144** (0.0768), **74** (0.0554), **33** (0.0766), **112** (0.0572), **135** (0.0567), **139** (0.0591), **210** (0.0654), **171** (0.06), **160** (0.0565), **85** (0.0602), **181** (0.0539), **30** (0.0581), **82** (0.0615), **70** (0.055), **31** (0.0479), **78** (0.0637), **161** (0.0657), **203** (0.054), **188** (0.0553), **108** (0.0571), **179** (0.057), **124** (0.0547), **125** (0.0547), **174** (0.0638), **2** (0.0605), **59** (0.0571), **165** (0.0547), **166** (0.0699), **106** (0.0555), **100** (0.058), **24** (0.0598), **65** (0.0578), **136** (0.0565), **158** (0.0574), **5** (0.0585), **155** (0.0748), **208** (0.0551), **157** (0.055), **7** (0.057), **198** (0.0798), **15** (0.0554), **98** (0.0883), **178** (0.0643), **153** (0.0591), **193** (0.0636), **42** (0.0545), **72** (0.0605), **68** (0.0566), **95** (0.0611), **138** (0.0582), **48** (0.0548), **168** (0.0544), **93** (0.0626), **105** (0.0575), **88** (0.0688), **149** (0.059), **20** (0.0547), **143** (0.0824), **187** (0.0662), **169** (0.059), **22** (0.0616), **182** (0.0788), **34** (0.0673), **58** (0.063), **197** (0.0586), **142** (0.0755), **69** (0.0593), **119** (0.0575), **107** (0.0603), **126** (0.0629), **120** (0.0557), **17** (0.0708), **118** (0.0631), **41** (0.0579), **164** (0.0604), **13** (0.0614), **159** (0.0629), **191** (0.0652), **89** (0.0705), **53** (0.0555), **38** (0.0621), **27** (0.0626), **44** (0.0655), **102** (0.0543), **117** (0.0592), **64** (0.0586), **67** (0.0564), **61** (0.0615), **150** (0.0617), **40** (0.0494), **183** (0.0633), **113** (0.0597), **137** (0.0589), **46** (0.0566), **51** (0.0601), **90** (0.062), **127** (0.0623), **114** (0.0521), **134** (0.0605), **14** (0.0549), **132** (0.0764), **84** (0.0558), **62** (0.0567), **79** (0.0703), **45** (0.0628), **99** (0.0849), **12** (0.0589), **184** (0.0543), **57** (0.0571), **152** (0.0564), **111** (0.0626), **116** (0.0645), **91** (0.0565), **103** (0.0676), **8** (0.0605), **94** (0.0572), **128** (0.0644), **110** (0.066), **109** (0.0577), **177** (0.0571), **163** (0.0663), **154** (0.0542), **122** (0.0622), **133** (0.0645), **189** (0.0591), **28** (0.0633), **131** (0.0611), **172** (0.0594), **1** (0.0586), **121** (0.0722), **140** (0.0621), **18** (0.0579), **180** (0.0547), **130** (0.0573), **170** (0.059), **56** (0.0664)
	200 (0.0574), **202** (0.0551), **25** (0.0558), **37** (0.0568)
	195 (0.0558), **10** (0.0630), **104** (0.0500), **209** (0.0565), **92** (0.0639), **141** (0.0590), **123** (0.0569), **207** (0.0628), **173** (0.0501)
1:	**54** (0.0571), **9** (0.0549), **86** (0.0560), **145** (0.0683), **96** (0.0598), **23** (0.0562), **146** (0.0682)
	205 (0.0698), **185** (0.0610), **49** (0.0568), **4** (0.0557), **26** (0.0561), **201** (0.0543), **63** (0.0568), **186** (0.0591), **66** (0.0564), **151** (0.0568), **19** (0.0531)
	16 (0.0591)
	167 (0.0699)
2:	**97** (0.0643), **55** (0.0715)
	81 (0.0630), **83** (0.0630), **190** (0.0557), **176** (0.0687), **50** (0.0748)
	6 (0.0569), **192** (0.0658)
3:	**199** (0.0663), **175** (0.0621)
	162 (0.0650), **60** (0.0578)
4:	**76** (0.0554), **80** (0.0571), **196** (0.0534)
	129 (0.0575)
5:	**147** (0.0549)

Table 4: 210 Janko puzzles divided according to their difficulty using our own classification

7 Conclusions and Further Research

The Kuroshuto puzzle is a one-player combinatorial puzzle where little if any research has been carried out previously. First, we have developed an algorithmic solver to solve the Kuroshuto puzzle. Lists are used for the relation between numbered and empty tiles and patterns for efficiency. We try solving the puzzles using only the rules and patterns. If we are not completely able to, then we apply backtracking. With backtracking we guess the colour of a tile and then continue solving using the rules until we either found the solution or realise the guess was incorrect and backtrack.

The second solver is the SAT solver. We have created CNF formulas that describe the rules, which the SAT solver takes as input. For the rule that all white tiles must be connected we used Reinhardt's idea of having two trees [9], one for the tiles and one for the vertices. The two trees are constructed from the CNF formulas. This method only works for puzzles that have at least one numbered tile positioned at one of the edges of the board.

For a puzzle of size $n \times n$ the SAT solver requires $9 \times n \times n - 4 \times n$ variables, which is a constant number of variables per tile, or $O(N)$ in total with $N = n \times n$. As far as we know, this is the first time this technique, using two trees, is applied for connectivity in grid-like puzzles and therefore the first time that is has been encoded with a number of variables that is constant per tile. Other methods mentioned in Section 2 use at least $O(N^2)$ variables. Thus, our technique requiring only $O(N)$ variables is a great improvement in the number of variables to code connectivity.

Finally, we have developed a program that generates unique puzzles. Numbered tiles and black tiles are generated randomly and all affected tiles are coloured white. We tried to enforce that the connectivity rule must be used for each puzzle. To check if puzzles are unique, we used backtracking to find all possible solutions to a puzzle.

Our experiments showed that by increasing the puzzle size it takes longer to generate puzzles. Furthermore, the time the SAT solver takes to solve these puzzles stays very consistent and minimal with different size and difficulty level puzzles. The algorithmic solver is a bit quicker than the SAT solver, even when backtracking is applied. It can also be used to determine the difficulty level of puzzles. Most puzzles from the Janko website [3] can be solved without backtracking and our classification differs from the one the Janko site gives for their puzzles.

There are a few matters that could be considered for further research. First, when the algorithmic solver applies backtracking it iterates from the top-left corner of the board until it finds the first empty tile. Perhaps this can be refined by choosing a tile for backtracking that affects many other tiles. For example, one that is in an area that is very populated by numbered tiles and already coloured tiles. Also, the generating method could be improved because it takes a long time generating larger puzzles. This could possibly be achieved by transforming generated puzzles into puzzles that have a unique solution. Furthermore, the SAT solver does not verify whether puzzles have a unique solution and we have not examined whether the reduced number of variables indeed shortens the time for the SAT solver. Both could be investigated in further research.

References

[1] GridPuzzle. https://en.gridpuzzle.com/rule/kuroshuto (accessed 18-01-2025)

[2] Nikoli Puzzles. Nikoli. https://www.nikoli.co.jp/en/puzzles/ (accessed 18-01-2025)

[3] Janko, O. and Janko, A. Kuroshuto. Rätsel und Puzzles. https://www.janko.at/Raetsel/Kuroshuto/index.htm (accessed 09-12-2024)

[4] Heslenfeld, N. (2024). Solving and generating Fobidoshi puzzles. LIACS Thesis Repository. https://theses.liacs.nl/pdf/2023-2024-HeslenfeldNNiels.pdf

[5] Mourits, R. (2022). Solving and Generating the Nurimeizu puzzle. LIACS Thesis Repository. https://theses.liacs.nl/pdf/2021-2022-MouritsM.pdf

[6] Hegeman, H. (2022). Generating Pipes puzzles using maze-generating algorithms. LIACS Thesis Repository. https://theses.liacs.nl/pdf/2022-2023-HegemanM.pdf

[7] Filmus, Y. SAT algorithm for determining if a graph is disjoint. Computer Science Stack Exchange. https://cs.stackexchange.com/questions/111410/sat-algorithm-for-determining-if-a-graph-is-disjoint/#111411 (accessed 19-12-2024)

[8] Bofill, M. et al. (2023) On Grid Graph Reachability and Puzzle Games. arXiv preprint arXiv:2310.01378, 2023 https://arxiv.org/abs/2310.01378

[9] Reinhardt, K. (1998). On some recognizable picture-languages. Mathematical Foundations of Computer Science 1998. LNCS, vol 1450. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0055827

[10] Claessen, K. et al. (2008). SAT-solving in practice. 2008 9th International Workshop on Discrete Event Systems, Gothenburg, Sweden. https://doi.org/10.1109/WODES.2008.4605923

[11] Sorensson, N. (2013). Minisat. https://github.com/niklasso/minisat.git

[12] Wheeler, A. (2008). MiniSAT User Guide: How to use the MiniSAT SAT Solver https://dwheeler.com/essays/minisat-user-guide.html (accessed 20-01-2025)

[13] Janko, O and Janko, A. Schwierigkeitsgrade. Rätsel und Puzzles. https://www.janko.at/Raetsel/index.htm#schwierigkeit (accessed 20-01-2025)