Universiteit Leiden
The Netherlands

Opleiding Informatica

Solving and Generating the Nurimeizu puzzle

Rob Mourits

Supervisors:
Jeannette de Graaf & Walter Kosters

BACHELOR THESIS

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

26/7/2022

Abstract

In this thesis the Nurimeizu puzzle is studied. The Nurimeizu is a logic puzzle where a maze has to be created by coloring cells black or white in order to solve the puzzle. The resulting maze should contain a path from S (the starting point) to G (the goal). This paper will focus on two questions regarding this puzzle. First different algorithms used for solving a Nurimeizu puzzle are discussed. The focus when comparing algorithms is on improving the efficiency. Secondly, randomly generating uniquely solvable Nurimeizu puzzles is discussed.

Contents

1 Introduction **1**
1.1 Related work 2

2 Rules **3**

3 Algorithms **3**
3.1 Checks 4
3.2 Backtracking 6
3.3 Contradiction algorithm 6

4 Generating puzzles **16**
4.1 Creating rooms 16
4.2 Assigning icons 17

5 Experiments **18**
5.1 Comparing algorithms 18
5.2 Effect of check functions 19
5.3 Effect of modifications 21
5.4 Generating puzzles 24

6 Conclusions and Further Research **29**

References **30**

1 Introduction

The Nurimeizu is a logic puzzle created by Nikoli [NC22]. Nurimeizu is a combination of the Japanese words ”nuri” and ”meizu”, which respectively translate to ”to paint” and ”maze”. The goal when solving a Nurimeizu is to paint a maze from a given starting point to a given goal. For this maze, a set of rules must hold, which make sure that the puzzle has exactly one solution.

Every Nurimeizu consists of a grid divided into rooms. Every room contains one or more cells and some cells contain a circle or triangle. There is always exactly one S (starting point) and one G (goal). The number of circles and triangles differs per puzzle. In this thesis, a gray cell means that the cell has not been colored yet, so that it is clear when a cell is colored white or black. The exact rules of the Nurimeizu are explained in Section 2. An example of a Nurimeizu puzzle and its solution can be found in Figure 1 [JJ22].

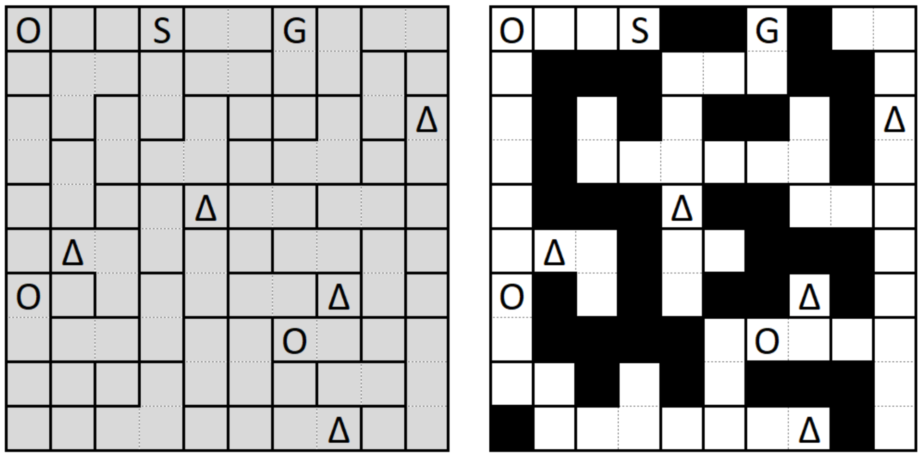

Figure 1: An example of a Nurimeizu puzzle and its solution [JJ22].

In this thesis different ways of solving this puzzle will be explained. First of all, in Section 3.2, a relatively slow backtracking algorithm is implemented. After this, in Section 3.3, a smarter algorithm using contradictions will be explained. This algorithm uses several methods to increase the efficiency. To analyze the effect of the different algorithms and methods, in Section 5, some experiments are conducted to compare the efficiency. Section 4 addresses how to generate different uniquely solvable Nurimeizu puzzles.

Since this project focuses on both solving and generating the Nurimeizu puzzle, we address two different research questions:

How successful are different algorithms to solve the Nurimeizu puzzle?
Can we generate uniquely solvable Nurimeizu puzzles in a reasonable amount of time?

This paper is part of a bachelor thesis at the Leiden Institute of Advanced Computer Science (LIACS), supervised by Jeannette de Graaf and Walter Kosters.

1.1 Related work

Not much research has been performed on the Nurimeizu puzzle, since it is not a very well-known puzzle, and it is created quite recently. There has been one paper published that studied the complexity of solving the Nurimeizu puzzle, together with two other logic puzzles. Iwamoto and Ide [II21] proved the NP-completeness of the puzzle by reducing the Hamilton path problem to the Nurimeizu. In the same research, two other puzzles are studies. In both puzzles a path has to be created, similar to the Nurimeizu. Both puzzles also proved to be NP-complete, by reduction from the Hamilton cycle problem.

There are however some other puzzles with similar aspects as the Nurimeizu puzzle. An algorithm to solve such a puzzle might contain techniques similar to the one studied in this thesis. Four examples of similar puzzles can be found in Figure 2.

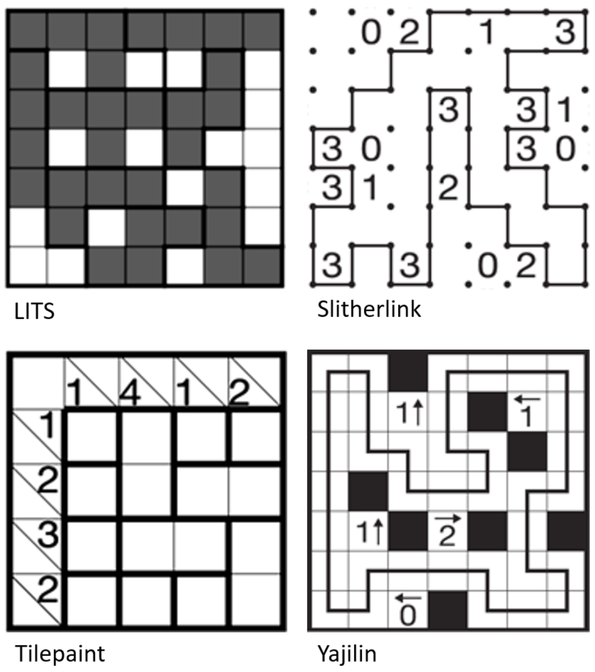

Figure 2: Four puzzles with similar aspects as the Nurimeizu.

The LITS and the Nurimeizu puzzle have one rule exactly in common. In both puzzles cells have to be colored black or white, and it is not allowed for a square of 2 by 2 cells to have the same color. In both the Slitherlink and the Yajilin puzzles a path has to be created through the provided grid. This is somewhat similar to the Nurimeizu, although in the Nurimeizu puzzle the path ends at a different cell than where it started, while in both the Slitherlink and the Yajilin puzzle the path forms a cycle. Finally the Tilepaint puzzle is one of the few puzzles containing rooms which have to be assigned a single color. All four puzzles are created by Nikoli [NC22], just like the Nurimeizu. There are several studies conducted which show that the LITS [BS17], Slitherlink [Kö12], Tilepaint [II22b] and Yajilin [II22a] puzzles are all NP-complete.

2 Rules

When solving the Nurimeizu puzzle, cells are colored black or white as long as the solution of the puzzle still satisfies a set of rules. Nikoli [NC22] lists five rules, but since most of these rules actually consist of multiple restrictions in the puzzle, they are split up into the following eight smaller rules. In the context of these rules, a path cannot visit the same cell twice and consists of only horizontally or vertically (not diagonally) connected cells. Note that these rules have to hold for the solution of the puzzle, when every cell is colored.

1. Every cell in the same room (area enclosed by bold lines) has the same color (either black or white).
2. Cells with an S, G, circle or triangle are colored white.
3. There must be a path from S to G consisting of only white cells (only horizontal or vertical, not diagonal).
4. Every cell with a circle must be on the path from S to G.
5. Every cell with a triangle cannot be on the path from S to G.
6. All white cells must be connected to each other.
7. White cells cannot form a cycle.
8. The cells in a square consisting of 2 by 2 cells cannot all have the same color.

Rules 1, 2, 7 and 8 are all rules that must not only hold at the end of solving the puzzle, but must also hold for every state of the puzzle when not every cell is yet colored. Rules 3, 4, 5 and 6, however, might not hold in every state while solving the puzzle. At the start for example, when no rooms are colored, there is not yet a white path from S to G. Furthermore, in an intermediate state it could be the case that two cells are colored white, without them being connected by other white cells. In a later state, however, these cells could become connected when uncolored cells between them are colored white.

Rules 3 and 7 together ensure that in a solution there will be exactly one path from S to G, since there are no cycles and cells cannot be visited twice. Therefore this will be the only and therefore the shortest path from S to G.

A valid Nurimeizu puzzle will have one single solution based on these eight rules. For solving puzzles we assume that the puzzles are uniquely solvable.

3 Algorithms

There are several ways to solve a logic puzzle. One of the most straightforward methods is a brute force approach, which means trying out different colorings to find a solution that does not break one of the rules. For the Nurimeizu a backtracking algorithm is implemented, which is only slightly more efficient than a brute force approach. Furthermore, as the main focus of this thesis, a faster algorithm is constructed, which tries to solve the Nurimeizu in a similar way a human would solve the puzzle. The algorithms for solving this puzzle are written in C++.

3.1 Checks

For all possible solving algorithms it is important to verify at any point if the current coloring of a puzzle is still valid or if it breaks any rules. If a puzzle state is "valid" this means that it does not currently break any rules and could therefore lead to a solution. If a puzzle state is "invalid" at least one of the rules is broken and the current coloring of the puzzle cannot possibly lead to a solution.

To verify the puzzle state, six different check functions are used to see if the current state is still valid. If not every cell is colored yet, it is important to note that these checks cannot ensure that the current state will result in a valid solution for the puzzle. It is possible that all the checks say the puzzle state is valid, but in reality it turns out the puzzle cannot be solved from this state. However, the checks can prove the opposite. If the checks return false the current state breaks one of the rules so it is impossible to reach a solved puzzle from this state. Because of this, these checks are only used to find contradictions to the rules.

For a puzzle that is completely filled (every room is either black or white), however, the checks do confirm that the puzzle state is valid, and that the current state of the puzzle is indeed the solution. This is important, since otherwise there would be no way to be certain that a valid solution is found.

During the backtracking and the basic contradiction algorithm, the checks are performed in the order in which they are explained. This order is based on two different measures. First of all we want to execute checks of lower complexity earlier, so that checks of higher complexity might not have to be executed. Furthermore, we want checks with a higher chance of a contradiction to be executed earlier, since then the chance is higher that other checks do not have to be executed.

color_room (rule 1): *Colors every cell in a room in the same color.*
The only rule that does not have a check function is the first rule. This rule states that every cell in a room must have the same color. Since the algorithms only color an entire room and never single cells, the resulting puzzle can never break this rule and therefore a check is not necessary.

check_icons (rule 2): *Checks if every cell with an icon is white or uncolored.*
One of the most simple checks is the check for the second rule. This function scans for every cell with an icon if it is white or uncolored. If it is black, this coloring is invalid and the function will return false.

check_squares (rule 8): *Inspects if there are squares of 2 by 2 cells with the same color.*
The function that checks rule 8 is also iterative over the puzzle. For every 2×2 square it is examined if all 4 cells have the same color, and if that is the case this function will return false.

check_circles (rule 4 and 5): *Checks if for every circle there is still a possible path to S and a possible path to G without triangles.*
This function performs a depth first search walk over all white and yet uncolored cells for every circle in the puzzle. This happens for every cell next to the cell with the circle. Only if one of these cells can still reach S and another cell next to the circle can still reach G, using a path of white and uncolored cells without a triangle, this function returns true. When there is no path from a circle to S or G in the current state, there will definitely be no path when all cells in the puzzle are colored. This is one of the checks that could return true when the current puzzle state is in fact not solvable. However, if the function returns false it is certain that this state can never lead to a solution. Since this function does not consider cells containing triangles during the depth first search from a circle to S or G, rule 5 is also automatically verified when this function is called.

Figure 3 shows an example of a state where the *check_circles* function will return true when, in fact, no solution can be reached from this state.

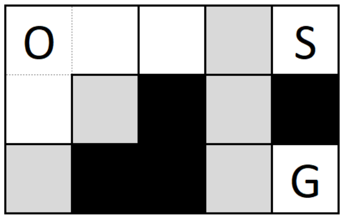

Figure 3: An invalid puzzle state where the circle check will return true.

In this situation the cell directly to the right of the circle can reach both S and G, but the cell directly under the circle can do the same. Therefore the circle check function will classify this as a valid puzzle state. It is however clear that a path from S to G cannot exist, since a path cannot visit the same cell twice.

check_s_g_path (rule 3): *Examines if there is a possible path from S to G for a puzzle without circles.*
When there is a path from every circle to both S and G, this means that there is per definition also a path from S to G. Therefore, when there are circles present, *check_circles* automatically verifies rule 3. For the case when there are no circles present, a separate function is needed to check if a path from S to G is still possible. This function performs depth first search using white and uncolored cells without a triangle from S and returns true if and only if G is reached.

check_cycle (rule 7): *Checks if there are cycles of white cells.*
For rule 7 a function is created which checks if there are white cells forming a cycle. This function iterates over the puzzle and keeps track of all visited cells during the process. For every white cell that is not yet visited a depth first search algorithm considering only white cells is used from this cell. If, during this search, a cell is visited that was already in the list of visited cells, a cycle has been detected. If not, the visited cells are added to the list and the iteration over the puzzle continues to find the next white cell that is not yet visited.

check_white_connected (rule 6): *Checks if all white cells can still be connected to each other.* Finally a function is created which checks if there are white cells that cannot reach all other white cells using only white and uncolored cells. If that is the case, it will never be possible for all white cells to be connected to each other. First this function performs depth first search from S (since S is always white because of rule 2) over all white and uncolored cells and keeps track of all visited cells. It is important to include the uncolored cells in this search. Two white cells that are only connected via uncolored cells can still be connected in later stages of the puzzle if these uncolored cells are also colored white. After this the function iterates over the entire puzzle and checks for every white cell if it is present in the list of visited cells. If all cells are indeed in the list, they can all reach S and are therefore in the same component of white cells that can still be connected. If this is not the case, there is a cell that is not connected and can never be connected to all other white cells, since, for example, the white cell with the S cannot be reached. In this case the function returns false.

3.2 Backtracking

The backtracking algorithm tries to solve the puzzle by coloring rooms until either the puzzle is solved completely or an invalid state is reached. When an invalid state occurs, backtracking tries to color the same room black instead of white. If that still results in an invalid state, previously taken steps will be reversed by uncoloring the last colored area to continue from a previous state. The coloring attempts are made based on the location of the rooms, from the top left to the bottom right. This means first all the rooms with a cell in the first row are colored, then all rooms with a cell in the second row, etc. To check if a state is invalid, the previously explained check functions are used. Note that this is a pure backtracking approach and that even the cells with an icon are not colored white at the start.

The process of backtracking can be visualized using a state action tree, where every action is assigning a color to a room and every state is a (partially) colored puzzle. The process of backtracking is depth first search through this tree, where invalid state nodes are pruned, so that the algorithm does not continue when the state is already invalid.

An example of such a state action tree can be found in Figure 4 for a small 3×3 puzzle example. In this example backtracking colors room 0 and 1 white. This is found to be an invalid puzzle state (rule 8 states that cells in a 2×2 square cannot have the same color), so this part of the tree is not explored further, but room 1 is colored black. Since this is also invalid (cells with an icon must always be colored white by rule 2), the backtracking algorithm uncolors room 1 and changes room 0 from white to black. After this, 3 rooms in a row are colored white since the current state continues to be valid. Since now every room is colored and the current state is valid, a solution is found so the tree does not have to be explored further.

3.3 Contradiction algorithm

The contradiction algorithm uses a different approach than backtracking. First of all, four actions are defined that a human would use to solve a Nurimeizu puzzle. These actions are used together

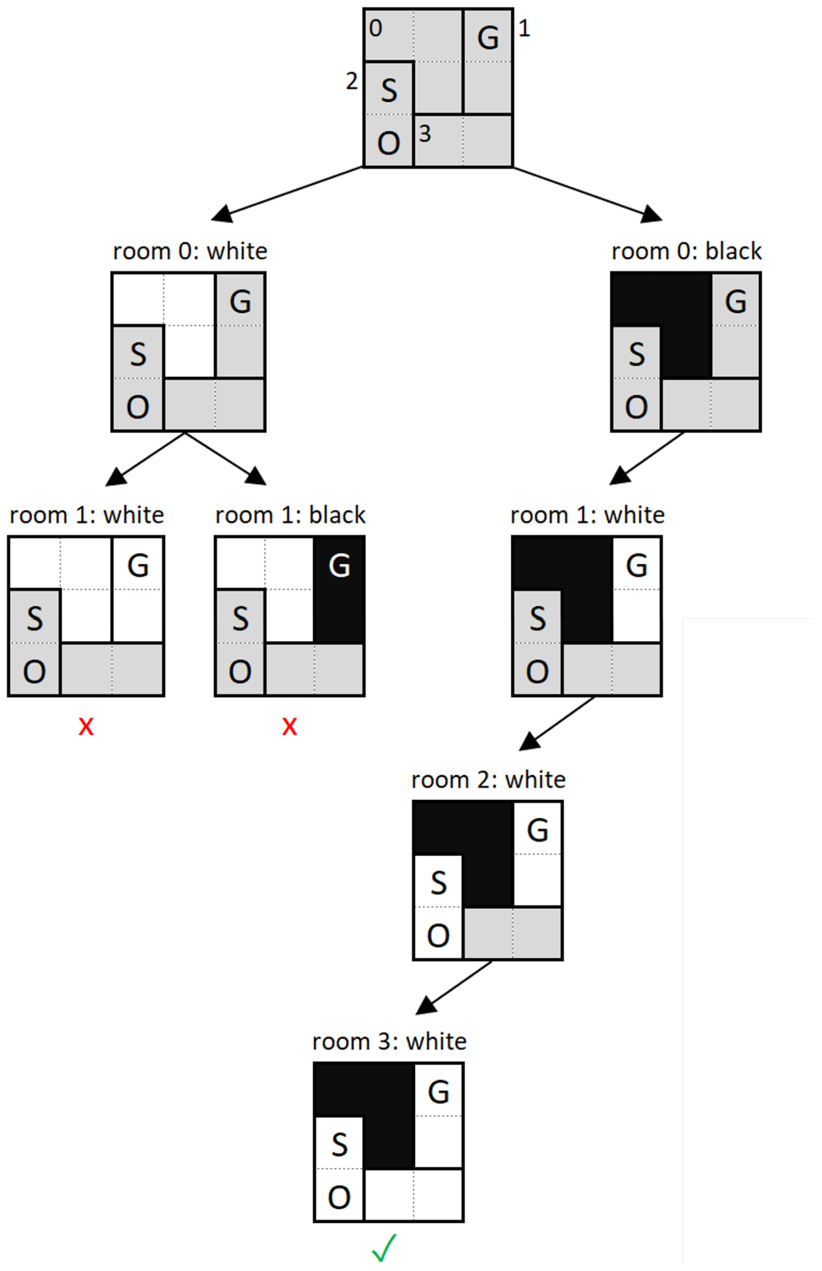

Figure 4: State action tree of the backtracking algorithm.

with an algorithm that makes coloring attempts to find a contradiction in a more efficient way than backtracking. Although these four actions are based on the same rules as the check functions, they serve a different purpose. Where the check functions only verify if the current state is valid for a certain rule the four actions in the contradiction algorithm perform colorings, which always result in a valid state (if the original state was also valid). After that some different methods to improve on the basic contradiction algorithm are explained.

3.3.1 Basic algorithm

In the basic version of the contradiction algorithm the following four actions are defined.

Icons: The first action is only used once at the start of solving the puzzle. For this the algorithm simply colors every room that contains an icon white, according to rule 2. This also ensures that the check for rule 2 does not have to be performed anymore, since it is not possible to color a room with an icon black anymore.

Squares: The second and most successful action uses rule 8. When there is a 2×2 square of cells where only one room is yet uncolored this rule can sometimes be applied. When all other cells in the 2×2 squares have the same color the uncolored cell must be the opposite color from the other cells in the square. Three situations where this is the case can be found in Figure 5. In the first and last example the uncolored room must be black and in the middle example the uncolored room must be white. This action is applied by iterating over all possible 2×2 squares and checking if this square fits the requirements to apply the rule.

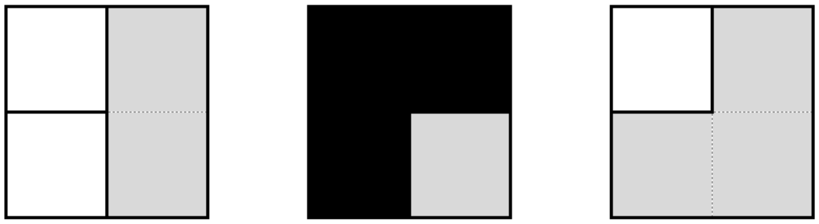

Figure 5: Three examples where the squares action can be applied.

Circles: Since every circle must be a cell on the path from S to G it must have at least 2 neighboring white cells without a triangle. So when 2 of the 4 cells next to a circle are black or contain a triangle, the other 2 cells must be colored white.

S/G: Something similar can be done for S and G. These cells must have at least one white cell without a triangle next to the cell itself. This means that if there is only one uncolored cell left for a S or G, while the other neighboring cells are black or contain a triangle, this cell must also be colored white.

At the start of the contradiction algorithm the icons action is applied to color all cells with an icon. After that the other three actions are executed, possibly multiple times since the changed state after executing all three actions once might make it possible for one of the actions to color rooms again after being executed a second time. This already colors a significant part of most puzzles.

After this the algorithm tries to find a room that would result in a contradiction when it is colored white or black. The basic contradiction algorithm uses the same order as backtracking, namely from the top left room it iterates over every row to the bottom right room. After the coloring of a room, but before the check if the state is valid, the three actions are applied again, since they are immediate results of the coloring. This way there is a larger chance to find a contradiction faster. After the actions the puzzle state is verified using the check functions from Section 3.1. If there is a contradiction when the room is colored either white or black, meaning that the check returns false and the puzzle state is invalid, the room is colored in the opposite color. Since the puzzle is uniquely solvable, and the previous state was valid, coloring a room white and black can never both lead to a contradiction. So if one of the two colorings leads to a contradiction, the other color must be the correct one for this room. After this coloring the three actions are executed again as a result of the coloring of the room.

Not every puzzle can be solved this way, because it can be the case that no contradiction can be found by coloring a single cell. If that is the case, the basic contradiction algorithm will use backtracking from that point on. This is of course inefficient, but since a large part of the puzzle is usually already colored, it is still significantly faster than performing backtracking from the start.

In Figure 6 the algorithm is applied to a small example puzzle. In the first step the rooms are numbered in the same way the algorithm would. In step 1 the icons action is applied, coloring all rooms with an icon white. In step 2 the squares action is applied action times. In step 3 the S/G action is applied, making sure that G has at least one white cell next to it. Since none of the actions can be applied anymore at this point the algorithm starts searching for a coloring of a cell resulting in a contradiction.

In step 4 room 1 is colored first white and then black. After both colorings the three actions are executed, but they have no consequences, meaning that they do no result in more rooms being colored. Since both resulting puzzle states are still valid according to the rules, no contradiction is found here. Room 1 is uncolored again and in step 5 the next uncolored room is used. First room 3 is colored white and then black, again with no consequences from the actions. This time, however, black results in a contradiction with rule 3 and 6, since S cannot reach G anymore and not all white cells are connected. Therefore room 3 must be colored white. Since still none of the three actions can be used, in step 6 the algorithm continues with the next room. Room 4 is colored first white and then black. The white coloring has consequences this time, since the squares action can be applied to color room 1 black. Since the black coloring creates an invalid puzzle state (again S cannot reach G anymore and the white cells can not all be connected to each other), room 4 must be colored white, followed by the squares action. Now the entire puzzle is colored and does not break any rules, so a solution is found.

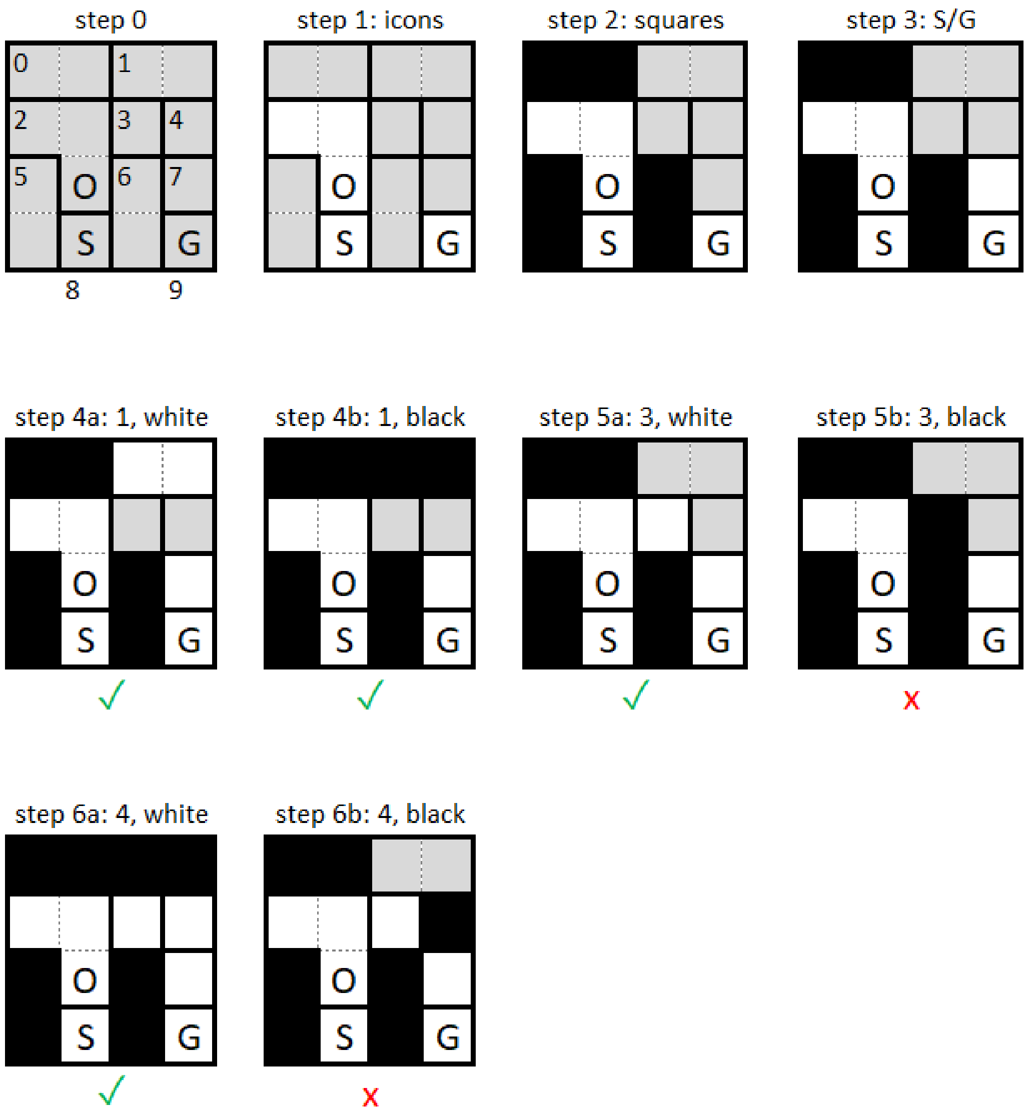

Figure 6: An example of how the contradiction algorithm solves a puzzle.

The contradiction algorithm is already faster than the backtracking approach for most puzzles, but there are still several ways to increase the efficiency. Below some methods that are used to increase the efficiency are discussed. The version of the contradiction algorithm using these four methods is called the modified contradiction algorithm.

3.3.2 Coloring two rooms

In the basic algorithm, backtracking is applied when none of the rooms result in a contradiction when colored black or white. Of course this is not very efficient, since backtracking might have exponential complexity. To postpone backtracking the technique of coloring two rooms is applied.

This works similar to the previously explained coloring of a single room in order to find a contradiction. The difference is that this time two rooms are colored. This method finds a room A for which coloring white or black does not result in a contradiction. The algorithm first colors room A white and applies the three actions. Then a second room B is chosen, which is colored first white and then black. If both white and black result in an invalid puzzle state, room A must be colored black. If this is not the case the same process is applied but with room A colored black. If this still does not result in a contradiction another room B is chosen, until every possible room B is used. If no room B can result in a contradiction a different room A is chosen.

When a contradiction is found this way and a room is colored, the algorithm will go back to attempting single room colorings, as explained in the basic algorithm. If no contradiction can be found by coloring two rooms, the algorithm performs backtracking on the uncolored cells.

Figure 7 shows an example of how this method works. Note that this is just an example and that the contradiction algorithm might not be able to reach this exact starting state. In the figure there are three uncolored rooms left, room A, B and C. When coloring one of those rooms neither coloring black nor white will result in a contradiction and therefore the normal coloring attempt has no effect. However, if we use the two room coloring method on room A and B, a contradiction is found and a room can be colored. In step 1a it is shown that after coloring room A white and applying the three actions (which have no effect) the puzzle state is still valid. When room B is colored white (1a i) or black (1a ii) both cases lead to a contradiction after applying the actions. In both cases room C can be colored by using the squares action, and both cases lead to a cycle of white cells, which is in contradiction with rule 7. Therefore room A cannot be white, and must be black as shown in step 1b. Since the two room coloring method resulted in a room being colored, the algorithm goes back to the single room coloring method, which is used on room B. Coloring room B white will result in a filled, valid puzzle state as a consequence of the squares action. When room B is colored black and the squares action is again applied, this leads to a contradiction, since the path from S to G does not contain all circles. Hence, we found a unique solution for this puzzle.

3.3.3 White path

In the basic algorithm it is tested if there is still a possible path using white cells without triangles from every circle to both S and G. However, when a human would solve a puzzle, it turns out that checking if there is already a white path from S to G is also useful. In a fully colored puzzle checking

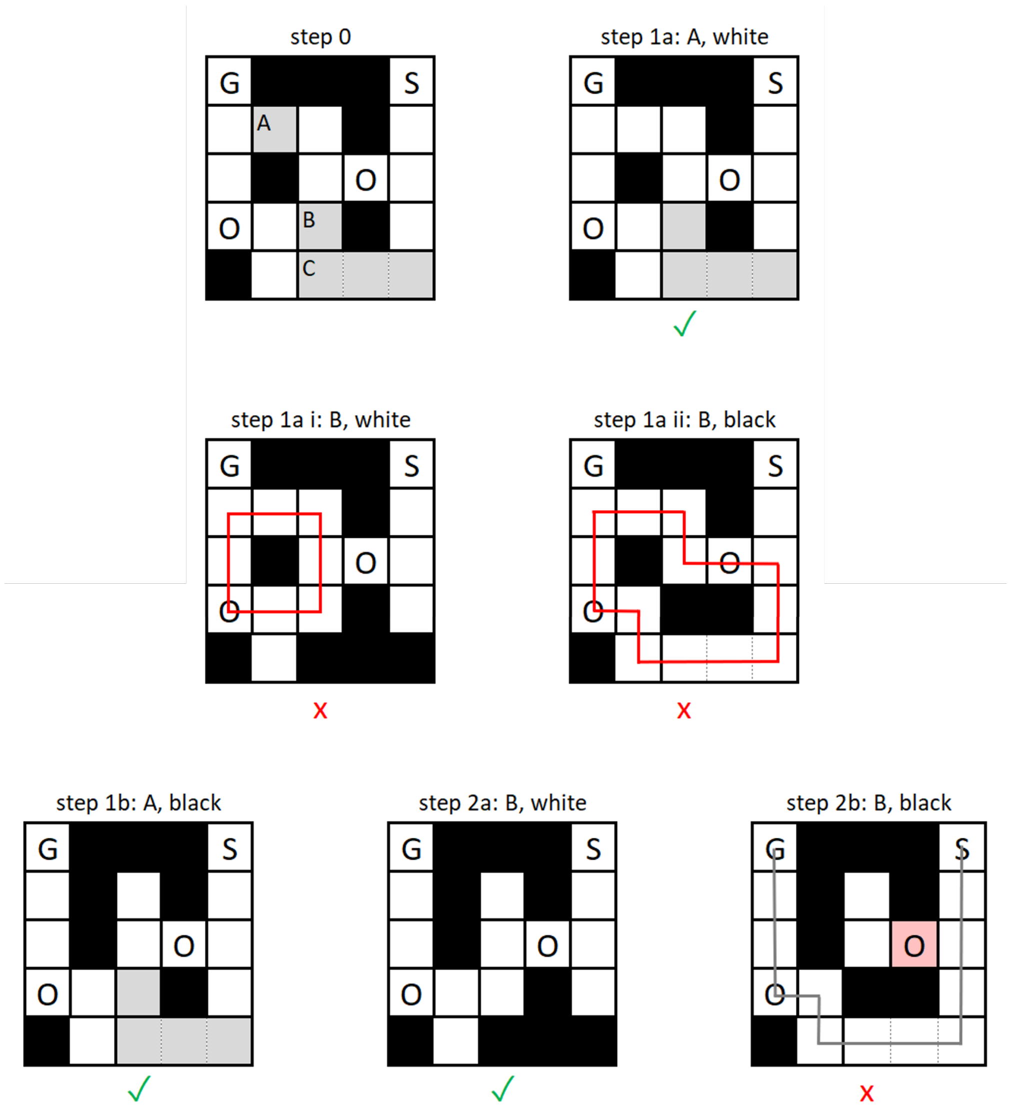

Figure 7: An example on how two room checking works.

this is not necessary, since the circle check already verifies if the path from S to G contains all circles. In a state where not every room is yet colored, however, searching if there is already a path from S to G and checking if all circles are on this path can turn out to be useful. To explain this, note that rule 3 states that there must be a path from S to G in the final state, and rule 4 states that all circles must be on this path. Since rule 7 states that white cells must not form a cycle, there will be only one path from S to G in the final state and this path must therefore contain all circles. This means that if there is a path from S to G in an intermediate state using only white cells that does not contain all circles, then the puzzle state is invalid, and a valid solution can never be reached from this state. This way a puzzle state can be found to be invalid while the basic algorithm would still classify the state as valid. This new check function which verifies this is part of the modified contradiction algorithm.

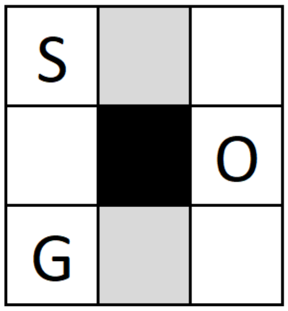

Figure 8: An example where the white path check can be used successfully.

For example, when in the puzzle in Figure 8 the middle left cell would be colored white, it would be a valid state according to the original checks, since it is still possible to create a path from the circle to both S and G, there is a path between S and G, and there are no cycles made out of white cells yet. It is clear however that this state cannot result in valid coloring, since the circle can never be on the white path. If the circle would be on the white path, a cycle of white cells would be created, breaking rule 7. So the white path function marks this puzzle state as invalid and the last coloring will be undone by the contradiction algorithm.

The new function uses depth first search to find a path of white cells from S to G, if it exists. On this path the function counts the number of circles. At the end of the function the number of circles on the path is compared with the total number of circles. If this is not equal the function returns false.

If the function does not find a path consisting of only white cells between S and G, this check cannot be used and returns true immediately.

3.3.4 Room based checks

When a room was colored black and white, the entire board was scanned. Another possible method might be applying the check functions based on the rooms that were colored in the previous step. In many situations it is sufficient to only examine a part of the puzzle. Therefore another modification is a check function which uses only the recently colored rooms to perform checks. Depending on whether the colored room is black or white some check functions are also not necessary.

One important thing to note is that the original check functions were called only once after both the room is colored and the three actions were performed. The room based checks, however, are called multiple times, namely once for the original colored room and every other room that has been colored by the three actions. Therefore this change might not always improve the efficiency but could also have a negative effect.

After coloring a cell, based on whether it is colored black or white, not all check functions need to be called, since some functions are only necessary for either a white or a black coloring. Some of these functions are changed, based on the room that has been colored. These changes are explained below for every check function.

check_icons (rule 2): Since this rule is only used in the backtracking algorithm, and not in the contradiction algorithm, this function remains the same.

check_squares (rule 8): For both white and black room colorings the square rule has to be checked. To improve this function, only the squares that contain a cell that is part of the room that was colored previously are checked, which means not all the squares in the entire puzzle have to be scanned. When the previous state of the puzzle did not break the squares rule anywhere in the puzzle, this state can only break the squares rule with a square containing newly colored cells. Therefore only checking squares containing these cells is enough to determine if this rule is broken after a room coloring.

check_circles (rule 4 and 5): The circles rule only has to be verified when the room is colored black, since a white cell cannot block a path from a cell with a circle to S or G. Improving the function itself is a bit harder. The normal circle check performs depth first search from every circle in the puzzle. To improve this based on the colored room the algorithm would have to check if this room blocks all possible paths between S, G or any of the circles. A function that does this specifically for the colored room would not be less complex than the original function. Therefore the regular circle check function is still used here, but now only when the newly colored room is black.

check_s_g_path (rule 3): For the same reason as *check_circles* this function is not changed further based on the previously colored room. It is more efficient to check whether a path from S to G is still possible than to see if the colored room might block a possible path.

check_cycle (rule 7): Only when the room is colored white the cycle rule can be broken, since a black coloring can never create a cycle that was not there before. The normal cycle check function had to iterate over the entire puzzle to find white cells that were not yet visited. The new function however only has to check if there is a cycle containing a cell from the newly colored room as a starting point. It checks this in the same way the original function did, by performing depth first search and only returning false when a cell is found twice and therefore a cycle is found.

check_white_connected (rule 6): Rule 6 has to be checked for both a white and a black coloring, since both room colorings can create a white cell that is not connected to the other white cells. A black coloring for example can cut off a white cell from all other white cells and therefore create an invalid puzzle state. A white coloring can also create an invalid puzzle state. If, for example, an uncolored cell cannot reach any white cells the puzzle state can still be valid, since the uncolored cell may still be colored black. When this cell is ultimately colored white however, the state will become invalid.
The checks for rule 6 can be split up for the different colorings. Both functions do significantly less work than the complete function which iterates over every cell in the grid. When a room has been colored black, this black room can be the reason one or more white cells are not connected to the rest of the white cells anymore. In this case, there must be a white cell adjacent to the room that was colored black that cannot reach S anymore. So when all white cells adjacent to the colored room can still reach S using a path of white cells, this means that the coloring does not break rule 6. Therefore the function is changed in the following way. When a room has been colored black the function first calls part of the original *check_white_connected* function which creates a list of all reachable white and uncolored cells from the starting point S. After that the new function only has to check if all white cells that are adjacent to the colored room are in this list. If these white cells are all in this list of connected cells, and the previous state was valid, then the new state after the room coloring must also be valid. When a room is colored white the function can be a bit shorter. In this case, the only thing that has to be checked is if a white cell can be reached from the previously colored room. If that is the case and the previous state was valid, then after the coloring all white cells are still connected, since all new white cells are connected to the previously established connected component of white cells.

white_path (rule 4): The extra check explained in Section 3.3.3 can also be changed slightly. This check only has to be performed when a room is colored white. The only way the previous state was valid and the new state is invalid is when a path from S to G has been created with the new room coloring. This can only be the case when this coloring is white. Therefore this function only has to be called when a room is colored white. Performing this function based on the previously colored room, however, would not decrease the complexity, for similar reasons *check_circles* and *check_s_g_path* are not improved.

squares action (rule 8): The only other function (that is not a check function) that could benefit from performing actions locally, based on the colored room is the squares action explained in Section 3.3.1. Instead of scanning the entire board, only the colored rooms have to be scanned. Since the squares action is called multiple times until no colorings can be made anymore, it can be the case that multiple rooms need to be scanned. When implementing this, the additional work the function had to perform to keep track of the colored rooms costed a lot of time, making this function run significantly longer than the original function. Therefore this method is not used in the modified contradiction algorithm.

3.3.5 Coloring order

The order in which the algorithm attempts to color rooms to find a contradiction can be changed. The basic algorithm performs the colorings from the top left to the bottom right. To change this order a room order vector is created which sorts the id's of the different rooms from "most likely to have a successful coloring" to "least likely to have a successful coloring". In this context, a coloring is succesful when it leads to a contradiction, meaning that a definitive coloring can be made.

We propose three different measures to indicate how likely it is for a room to have a successful coloring. How these different measures affect the time it takes to solve a puzzle will be discussed further in the experiments section.

- The first measure is the size of the room. Larger rooms have more impact on the puzzle if colored and might therefore have a bigger chance of a successful coloring. So larger rooms are visited earlier on in the algorithm than smaller rooms. This measure might be the most effective for puzzles with many different room sizes.
- Secondly the number of cells in the room that lay on the edge of the grid is taken into account. Since a path from S to G will have fewer options at the edges of the puzzle, rooms at the edges might have larger chance of a successful coloring. This measure will most likely be more effective for smaller puzzles, since the proportion of rooms on the edge of the puzzle will be larger than for large puzzles.
- Finally the number of colored neighbor cells is considered. The more cells next to the room are colored, the larger the chance that coloring this room might lead to a contradiction.

The first two measures are mostly effective for specific examples, namely small puzzles or puzzles with many different room sizes. Therefore we expect the last measure, based on the number of colored neighbors to be the most effective measure for most puzzles.

4 Generating puzzles

Using the puzzle solving algorithms, we can randomly generate Nurimeizu puzzles. It is important for a generated puzzle to have exactly one solution. This is checked by the adapted contradiction algorithm. The adapted contradiction algorithm only colors a room when it is sure that this coloring is part of the solution. Therefore, when a puzzle is solvable using only the contradiction algorithm, there must be one solution. If the contradiction algorithm can not solve the puzzle, a complete version of backtracking is applied. This version does not stop when a solution is found, but continues to search for a possible second solution. Only when there is exactly one solution the puzzle is found to be valid.

The generating algorithm keeps generating puzzles until a uniquely solvable puzzle is found. Every generated puzzle has to be solved using the adapted contradiction algorithm to find out if it is uniquely solvable. Therefore generating puzzles costs a lot of time, thus we only focus on relatively small puzzles consisting of 5 by 5 cells or smaller. The algorithm can also generate rectangular puzzles that are not a square, for example a 4×5 puzzle. When generating the puzzles there are several things that need to be determined. First of all the cells need to be assigned to the rooms.

4.1 Creating rooms

Since only smaller puzzles will be generated, the rooms in these puzzles will not be larger than 4 cells. For this a list of all possible room shapes is used. Since every rotation and mirroring of every room is listed as a different room, this list consists of 27 rooms. A list of all possible basic rooms (without rotating or mirroring) can be found in Figure 9. Note that a 2×2 room of size 4 is not present in this list since coloring a 2×2 room would immediately contradict rule 8.

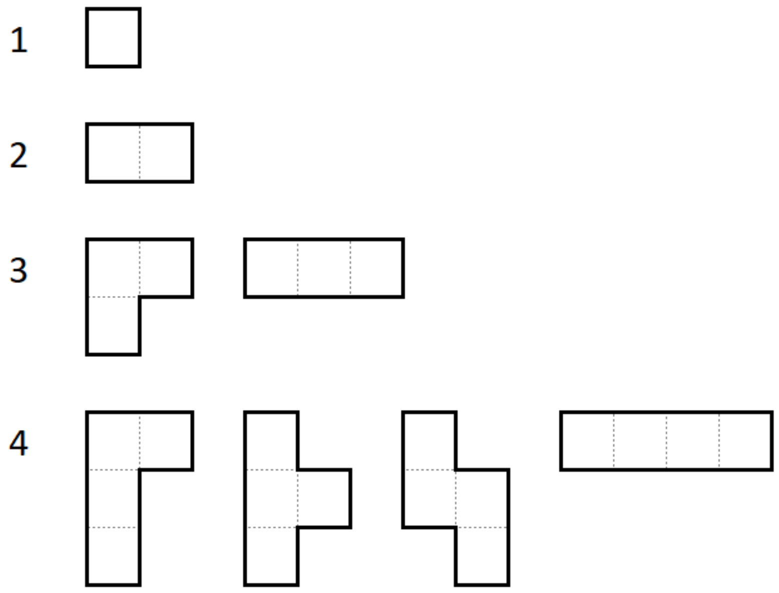

Figure 9: All possible rooms with 4 or fewer cells.

When generating the puzzle a random room shape is chosen from the list. Smaller rooms are given a larger chance of being chosen, since most puzzles consist mainly of rooms of size 1 and 2. More on this distribution will be explained in Section 5.4. The room that is chosen is then randomly placed in the grid, and the algorithm checks if this room fits without overlapping with other rooms. After that, the next room is randomly chosen. This is repeated until the entire grid is filled with rooms.

4.2 Assigning icons

After the rooms are assigned the icons have to be placed. First S and G are randomly placed, with as only constraint that they are not allowed to be next to each other (neither orthogonal nor diagonal).

After that the circles and triangles are placed. The number of circles and triangles is determined by a normal distribution. For this the average number of circles and triangles over 60 puzzles of size 10 by 10 was calculated. We used the first 60 puzzles of size 10 by 10 from the list of puzzles created by Angela and Otto Janko [JJ22]. A 10×10 puzzle contains 3.1 circles and 4.4 triangles on average, so we calculate the average number of circles and triangles for a puzzle of arbitrary size as the number of cells in the puzzle multiplied by respectively 0.031 and 0.044. This number is used as the average of the normal distribution. When, based on the normal distribution, a value lower than 0 is generated, this is changed to 0, and if the number of triangles and circles is larger than the number of free cells this number is changed to $\lfloor(\text{number of cells} - 2)/2\rfloor$ triangles and the same number of circles. This rarely happens, but it prevents the algorithm from entering an infinite loop trying to find an empty cell for a circle or triangle, when there are no empty cells left. The circles are then placed randomly with the only constraint that they should not be placed on a cell where another icon is already present.

5 Experiments

To compare the different algorithms and the effect of different modifications explained in Sections 3.3.2 to 3.3.5 to the basic algorithm, some experiments were conducted. For these experiments 10 puzzles of different sizes were used. These puzzles are all originally created by Angela and Otto Janko, and can be found on their website together with 160 other Nurimeizu puzzles [JJ22]. Table 1 shows the puzzles used, the number of rooms and the size in cells.

puzzle id	number of rooms	size
65	74	12×12
156	87	15×15
76	124	15×15
158	128	16×16
169	143	15×23
69	203	22×23
111	209	20×20
109	243	18×32
119	254	21×26
160	733	35×50

Table 1: The puzzles used in the experiments.

For the experiments based on comparing the different algorithms or methods, the runtime is used as a measure of efficiency. For these experiments the algorithm is performed 10 times to improve the accuracy. All experiments are performed on an Intel Core i7-8750H CPU. The average runtime of these 10 executions is listed in the table. The runtime is determined using `chrono::high_resolution_clock` from the C++ standard library. To decrease the deviation in runtime between the 10 runs, the program is executed on a single core using the linux command `taskset`. When the execution of an algorithm takes longer than 10 hours, the run is stopped and this execution is noted as >`36000.00` in the table.

5.1 Comparing algorithms

The first experiment is conducted to compare the efficiency between the backtracking algorithm, the contradiction algorithm and the modified contradiction algorithm. The differences between the modified algorithm and the contradiction algorithm are the four methods explained in Sections 3.3.2

to 3.3.5. This means that in the modified contradiction algorithm the two room coloring is applied, several checks are based on the previously colored room and the white path check is used. The coloring order is based on the number of colored neighbors, since this turned out to be the most effective measure, as will be explained in Section 5.3.3. The results of this experiment can be found in Table 2.

puzzle id	backtracking	contradiction	modified contradiction
65	0.27	0.04	0.04
156	1.67	0.20	0.22
76	225.71	150.48	17.92
158	0.49	0.32	0.82
169	10.02	0.57	0.25
69	>36000.00	1.48	0.87
111	831.38	121.64	1.75
119	30743.60	2831.88	117.92
109	287.09	1.04	0.82
160	>36000.00	5188.68	2080.66

Table 2: Runtime in seconds for the different algorithms.

One of the main things that stands out is the lack of consistency when it comes to the runtime based on the puzzle size. This shows that not just the puzzle size or the number of rooms, but also the configuration of the puzzles has a large effect on the runtime. For example, solving puzzle 76 with backtracking takes 460 times as long as puzzle 158, while puzzle 158 has 4 rooms more and is one row and one column larger than puzzle 76.

For almost every puzzle, the contradiction algorithm is much faster than the backtracking approach. The execution time of the basic contradiction algorithm compared to that of backtracking ranged from 1.5 times as fast for puzzle 76 to more than 24,300 times as fast for puzzle 69.

For most puzzles the modified contradiction algorithm is even faster than the basic contradiction algorithm. For some smaller puzzles however, such as 156 and 158, the basic contradiction algorithm is faster. For 158 even backtracking is faster than the modified contradiction algorithm. The reason for that is probably that all methods attempting to improve the efficiency require some extra calculations to be performed, which should in the end make the algorithm faster. These calculations cost some extra time, and for small puzzles the time spent on these different methods will have a bigger influence on the total execution time, especially when they are not very effective. In most cases, however, the modifications do have a positive effect, ranging from no improvement to 24 times faster than the basic contradiction algorithm.

5.2 Effect of check functions

The modified contradiction algorithm uses five different check functions to determine if a puzzle state is valid. Not every function will result in a contradiction as many times as some others.

This information is used to determine the order in which to call the check functions. To this end the modified contradiction algorithm is executed with a small change. Where in the original algorithm the checks found a contradiction when a single check returned false, this version executes every single check. The number of times each check returns false is counted. Table 3 lists for each check how often this check returned false out of the total amount of times a check returned false.

puzzle	*squares*	*white_connected*	*cycle*	*circle*	*white_path*
65	3.4	24.1	29.3	39.7	3.4
156	0.0	29.8	21.3	48.9	0.0
76	1.1	4.4	23.1	70.3	1,1
158	0.0	23.6	11.2	65.2	0.0
169	3.4	41.4	17.2	24.1	13.8
69	4.8	16.1	29.8	44.4	4.8
111	3.5	22.4	29.9	35.8	8.5
119	1.7	17.8	45.0	28.1	7.5
109	5.7	35.4	26.9	30.9	1.1
160	5.4	8.3	45.8	39.3	1.3
average	**2.9**	**22.3**	**27.9**	**42.7**	**4.2**

Table 3: Percentages a check function returning false out of all checks that returned false.

This clearly shows that some check functions are much more useful in the modified contradiction algorithm than other. On average, 42.7% of the times an invalid puzzle state is found, *circle_check* is one of the checks returning false. After the *circle_check*, the functions for rule 6 and 7, *white_connected_check* and *cycle_check* have the most effect. Finally *squares_check* and *white_path_check* result in a contradiction the least.

This table also shows the effect of the white path check function which was added in the modified contradiction algorithm. While it does not find as many contradictions as the most useful functions, 4.2% of the times a contradiction is found, the white path rule is broken. This shows that although the effect might be small, this additional rule does improve the total efficiency of the algorithm by finding contradictions faster than without this check.

Note that *squares_check* has little effect. The main reason for this is because the squares action already applies the rule many times. Only in a few cases the rule is actually broken by attempting to color rooms.

The results of this experiment are used in the modified contradiction algorithm. The order in which the check functions are called is based on both the results of this table and the complexity of the check function. First the *squares_check* is called because of the low complexity. After that the checks are called in decreasing order of their average percentage.

5.3 Effect of modifications

Four modifications on the basic algorithm were suggested in Section 3.3. The effect of these four modifications together is shown in Table 2, and the effect of the white path check function in terms of how often it led to a contradiction is already shown in the previous table. For the other three individual modifications some extra experiments were conducted to see their effect on the efficiency.

5.3.1 Coloring two rooms

The first modification (suggested in Section 3.3.2) is the addition of coloring two rooms when coloring a single room does not result in a contradiction. To test the effect of this, the 10 puzzles are solved using the modified contradiction algorithm with the two room coloring function and the modified contradiction algorithm without the two room coloring function. In the latter algorithm backtracking is used when no contradiction can be found by coloring only one room. For the algorithm with two room coloring backtracking is only performed if the two room coloring cannot find a contradiction. The results can be found in Table 4.

puzzle	with two room coloring	without two room coloring
65	0.04	0.05
156	0.22	0.20
76	17.92	58.40
158	0.82	0.83
169	0.25	0.25
69	0.87	0.87
111	1.75	2.77
119	117.92	2628.41
109	0.82	0.82
160	2080.66	759.02

Table 4: Runtime in seconds with and without two room coloring.

For most puzzles adding the two room coloring does not have any effect on the runtime. This is the case for puzzles that can be solved without the two room coloring or backtracking, and are therefore easier. The reason for this is simply because only single room colorings were needed to find a contradiction and solve the puzzle. Therefore neither the two room coloring function nor backtracking is used. For some puzzles however, the two room coloring function had to be used, and in those cases the efficiency increases significantly, as one might expect. This is the case for puzzles 76, 111 and 119. This shows that using the two room coloring function rarely has a negative effect, and when the runtime does decrease, the improvement is significant. Therefore the two room coloring is a useful modification.

As always there are exceptions to the general rule. For this modification puzzle 160 seems to be the exception, with the most surprising result. For this puzzle the two room coloring function takes more time than performing backtracking immediately. The most likely explanation for this is that the backtracking algorithm is "lucky" and the solution appears on the far left of the state action tree (as explained in Section 3.2). In this case backtracking is relatively fast and the extra work that comes with the two room coloring function apparently takes more time.

5.3.2 Room based checks

Another modification (suggested in Section 3.3.4) was changing the check functions such that they used the room that was previously colored. To examine the effect of these room based checks, we consider three different categories. All categories use the modified contradiction algorithm, with some changes for the check functions. The first category is the one used in the modified contradiction algorithm, and consists of the checks that are based on the colored room and uses the color which was assigned to the room. The second category uses the color which was assigned to the room, but uses the check functions from the basic contradiction algorithm, so not based on the location of the colored room. Checking based on color (which is used in the first two categories) means that when a room is colored white the functions *check_squares*, *check_white_connected*, *check_cycle* and *check_white_path* are called, and when a room is colored black *check_squares*, *check_white_connected* and *check_circles* are called. The final category uses the modified contradiction algorithm without room or color based checks, which means the checks are performed in the same way as in the basic contradiction algorithm. The results of this experiment can be found in Table 5.

puzzle	room and color based checks	only color based checks	no room/color based checks
65	0.04	0.04	0.03
156	0.22	0.35	0.28
76	17.92	128.30	17.57
158	0.82	1.13	0.52
169	0.25	0.33	0.22
69	0.87	1.39	1.08
111	1.75	2.07	2.21
119	117.92	128.39	106.84
109	0.82	0.90	0.59
160	2080.66	2191.78	2118.38

Table 5: Runtime in seconds with and without room/color based checks.

This experiment shows that it depends on the configuration of the puzzle whether or not the modifications actually reduce the runtime of the algorithm. For only 4 out of the 10 puzzles the room based checks result in a slightly lower runtime. The 6 other puzzles are solved faster with the original checks for the entire grid. Only using color based checks without the room based checks is never the fastest option for these 10 puzzles.

The reason that the original checks as used in the basic contradiction algorithm are so effective is mainly because in general the room based checks are executed very often in the contradiction algorithm. The original check function is called only once after both the room is colored and the actions are performed, while the room based checks are called once each time a room is colored. This means that when a large number of rooms are colored as a result of the actions after one coloring attempt, the room based checks perform more computations than one simple check over the entire grid.

This modification is therefore less successful than others, since it is only effective for some puzzles, but works negative for others. Especially the category where only color based checks are performed is not effective, since there are no puzzles where this is the fastest option. When the effect of both room and color based checks is negative, it is only a small increase in runtime. Therefore this modification is still used in the modified contradiction algorithm.

5.3.3 Coloring order

The final modification that was suggested in Section 3.3.5 was changing the coloring order. We considered three different measures which were used to determine the order in which rooms were colored. These measures are the number of colored neighboring cells, the size of the room and the number of cells in the room that are on of the edges of the puzzle. The runtime of the modified contradiction algorithm using these measures can be found in Table 6.

puzzle	colored neighbors	size	edges
65	0.04	0.09	0.07
156	0.22	0.29	0.26
76	17.92	17.04	10.27
158	0.82	2.20	0.58
169	0.25	0.14	0.14
69	0.87	0.84	1.16
111	1.75	1.47	2.27
119	117.92	149.56	171.17
109	0.82	1.35	1.64
160	2080.66	4140.17	3488.78

Table 6: Runtime in seconds for different coloring orders.

For 5 of the 10 puzzles the number of colored neighbors turned out to be the best measure. This measure is therefore used to determine the coloring order in all previous experiments. Using the number of cells on the edge of the puzzle or the size of the rooms both turned out to be the most effective for 3 puzzles. As expected, sorting based on room size is the most effective for puzzles 169, 69 and 111, which all have many rooms of different sizes. Sorting on edges is more effective for smaller puzzles, since in larger puzzles there are not many rooms on the edge of the puzzle. Just like the room based checks, there is not one choice which always decreases the runtime. It depends on the puzzle whether or not a certain measure has a positive of a negative effect.

5.3.4 Implementation

One thing that this thesis does not focus on is improving the implementation. There are several ways to improve the efficiency of the way certain things are implemented in the C++ code. One example of that is the way visited cells are saved in the *check_white_connected* function, which is used in all algorithms. In the version used for all experiments all cells visited in the depth first search walk are stored in a vector. When the function checks if a cell is already visited, it has to iterate over the entire list to see if there is a match. This can be improved upon using a two dimensional array of booleans, representing the puzzle. Every cell that is visited in the depth first search walk is set to `true` and every other cell will remain `false`. This way checking if a cell has been visited can be performed in constant time, instead of linear. This modification has been implemented and used on puzzle 111, decreasing the total runtime with 12.6%.

There might be several other ways to improve upon the implementation, but this thesis only focuses on performing computational improvements instead of improving the implementation. Therefore no further experiments have been conducted on improving the implementation.

It is important to note that such improvements on the implementation will have little effect on the conducted experiments. The *check_white_connected* function might be faster than before, but the number of times this function is called will remain the same. Therefore the absolute runtime of any algorithm will be lower with this improvement, but the runtime relative to other algorithms and modifcations will not be much different.

5.4 Generating puzzles

When randomly generating puzzles, several choices were made concerning the number of icons and the distribution of different room sizes and shapes. In this section some experiments are conducted which measure the effect of different choices by looking at the percentage of generated puzzles that are uniquely solvable. Another interesting measure might be how interesting it is to solve a generated puzzle, but since it is hard to objectively measure that, we will only look at the percentage of uniquely solvable puzzles. For these experiments puzzles of size 4×4, 4×5 and 5×5 are used.

5.4.1 Distribution of icons

One of the variables in generating puzzles is the number of circles and triangles used in an attempt to generate a uniquely solvable puzzle. To look at the effect of different numbers of icons some experiments are conducted with different numbers of triangles and circles. For this only puzzles consisting of 4 by 4 cells are used. With the number of triangles and circles ranging from 0 to 4, 10,000 puzzles are generated per combination. In this context, a puzzle is a grid where rooms and icons are assigned, but the puzzle does not have to be uniquely solvable. The room sizes are distributed according to distribution a from Table 9. What this distribution means is explained in Section 5.4.2. Table 7 shows the percentage of these generated puzzles that are uniquely solvable for the different numbers of icons.

It is clear from the table that the percentage of uniquely solvable puzzles is larger for a lower number of circles and triangles. The main reason for this is that most uniquely solvable puzzles can be solved using mostly the squares action. This is especially the case for puzzles with larger rooms. An icon in such a puzzle often makes the puzzle unsolvable. The icons (especially the circles) make it harder to create the path from S to G, since every circle has to be on this path, and the triangles

circles \ triangles	0	1	2	3	4
0	3.50	4.80	3.71	2.52	1.66
1	4.47	2.58	1.53	0.80	0.37
2	2.50	1.17	0.64	0.25	0.07
3	1.19	0.46	0.11	0.03	0.04
4	0.52	0.18	0.04	0.02	0.00

Table 7: Percentage of uniquely solvable 4×4 puzzles for different numbers of triangles and circles.

cannot be on the path. On the other hand, while they might be harder to generate, it is likely that uniquely solvable puzzles with more circles and triangles are more interesting to solve, since different rules can be used in solving the puzzle. It seems less interesting for someone to solve the puzzle when only the squares action is performed repeatedly without actually using one of the other rules.

Since the puzzles should be generated randomly, it is better to not use a fixed number of circles and triangles, but a normal distribution, since that will create more different puzzles. The averages of this distribution are based on the first 60 puzzles of size 10×10 created by Angela and Otto Janko [JJ22]. The value of the standard deviation is still variable and also has an effect on the percentage of uniquely solvable puzzles created. Therefore some experiments are conducted with the average number of circles and triangles of respectively 0.031 and 0.044 times the number of cells, as explained in Section 4.2. The standard deviation ranges from 0 to 3. With these values again 10000 puzzles are generated for each standard deviation and size. Table 8 shows the percentage of uniquely solvable puzzles generated.

σ	4×4	4×5	5×5
0	3.49	1.73	1.16
1	3.63	1.83	0.84
2	3.05	1.74	0.71
3	2.42	1.44	0.64

Table 8: Percentage of uniquely solvable puzzles per standard deviation σ and size.

Just like the previous table this shows that having fewer icons often result in more uniquely solvable puzzles. With a lower standard deviation the range of generated numbers is smaller, and the chance of the number of triangles or circles being 0 or 1 is higher. This makes the generation of such a puzzle more successful, as can be seen in Table 8. Again, a larger standard deviation will probably result in puzzles that are more interesting to solve, since having more icons often results in a puzzle where more different rules have to be used.

5.4.2 Distribution of rooms

Another variable that is important when randomly generating puzzles is the distribution of the different room sizes and shapes. In Section 4.1, Figure 9 shows the four classes of rooms used for generating puzzles. When filling the puzzles with rooms the different room sizes have a different chance of being generated. Most existing puzzles, for example the ones created by Angela and Otto Janko [JJ22], consist mostly of rooms containing 1 or 2 cells, so these rooms have a higher chance of being generated.

Five different distributions are defined. The first distribution is based on the distributions in other Nurimeizu puzzles, with a large chance of rooms containing 2 cells. This is the distribution used in the previous generating experiments. The second distribution contains no rooms of size 4. Distribution c is a uniform distribution where every room size has an equal probability. For distribution d the chance increases linearly with the size of the rooms and distribution e is the complement of d. These distributions can be found in Table 9.

distribution	1	2	3	4
a	30	40	20	10
b	40	40	20	0
c	25	25	25	25
d	10	20	30	40
e	40	30	20	10

Table 9: Percentages of a room size being generated per distribution.

For every room distribution and size 10,000 puzzles are generated. The number of circles and triangles are determined using a normal distribution with respectively 0.031 and 0.044 times the puzzles size as averages, and a standard deviation of 1, since this was the most effective for 4×4 puzzles. The percentage of these 10000 puzzles that are uniquely solvable can be found in Table 10.

distribution	4×4	4×5	5×5
a	3.63	1.83	0.84
b	2.26	1.14	0.48
c	4.82	2.63	1.32
d	7.29	4.55	2.56
e	2.62	1.13	0.54

Table 10: Percentage of uniquely solvable puzzles per room size distribution.

The results for this experiment are comparable to the effect of different numbers of triangles and circles. For most generated puzzles that are uniquely solvable mostly the squares action is used to solve the puzzle. Generating a puzzle where mostly the squares action has to be used has the largest chance of happening when the room sizes are relatively large. Therefore distribution d has the largest percentage of uniquely solvable puzzles generated, since there is a 40% chance of blocks of size 4. However, these puzzles tend to be less interesting to solve by a human. Two examples of puzzles generated by using distribution a and d can be found in Figure 10.

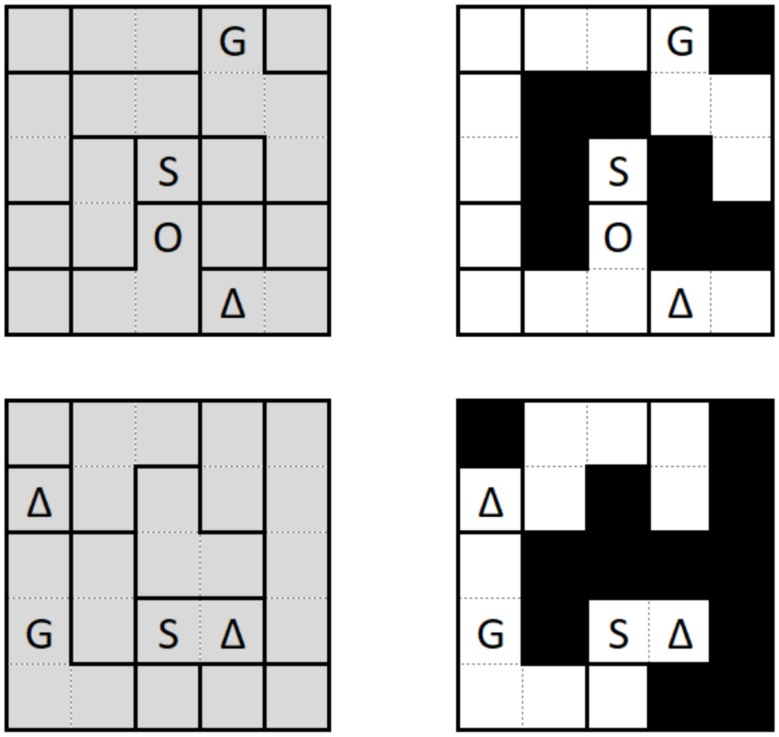

Figure 10: Two puzzles generated using distribution a and d and their solutions.

The top puzzle is generated using distribution a. There is only one room of size 4 and one of size 3. To solve this puzzle, different actions and checks have to be applied, while the bottom puzzle can be solved by using mostly the squares action, since it has many large rooms.
Puzzles with smaller rooms are probably more interesting to solve, since different rules have to be applied. Therefore the most realistic distribution for interesting puzzles is most likely distribution a, since this one is based on actual Nurimeizu puzzles, where rooms of size 2 occur the most.

5.4.3 Puzzles with rooms of size 1

Several room distributions are discussed, but there is still one unique case that turned out to be interesting. This case is puzzles with only rooms of size 1×1, which makes it very rare to be solved with only the squares action, which was the main reason puzzles with large rooms were not very interesting to solve. Since the most applied action cannot be used much anymore, it is rare for puzzles with only rooms of size 1 to be uniquely solvable, which makes generating such puzzles take more time than other puzzles. From 10,000 generated puzzles with rooms of size 1, only 0.01% turned out to be uniquely solvable. However, some interesting uniquely solvable puzzles were generated, such as the five puzzles in Figure 11.

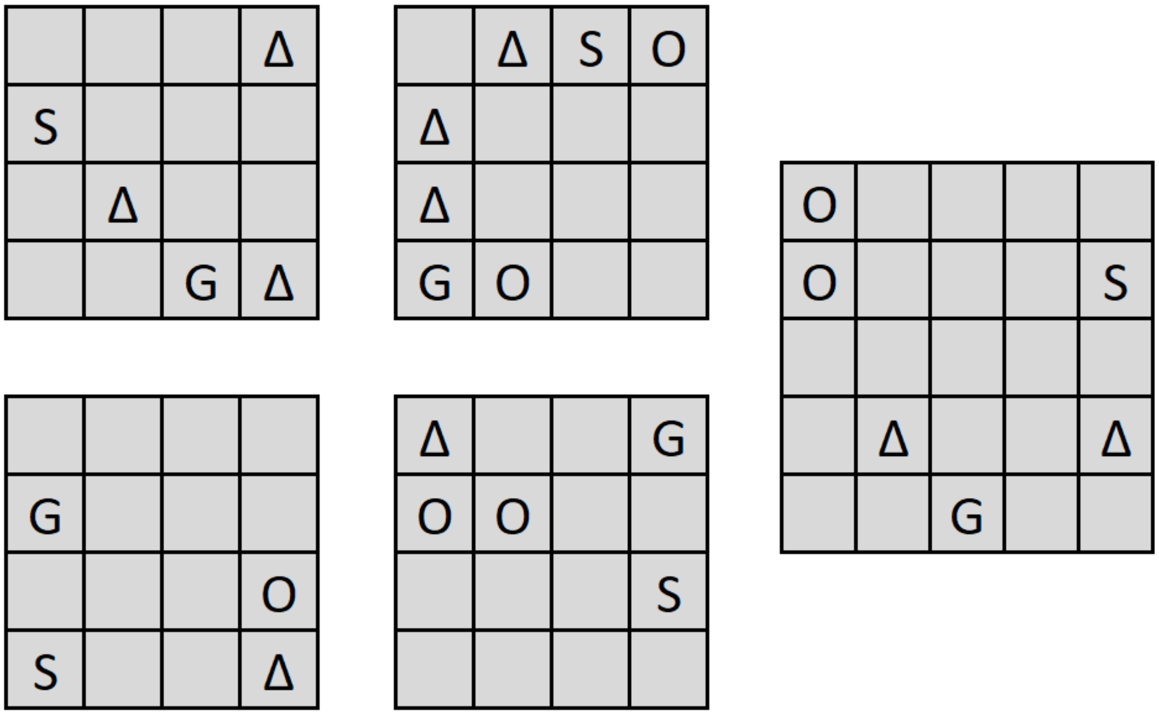

Figure 11: Examples of puzzles containing only rooms of size 1.

Although the success percentage of generating such puzzles is very low, they are very interesting to solve for a human. Several different rules and actions can be used to solve the puzzles, which makes it interesting to search for the solution. For example, when the modified contradiction algorithm solves the bottom left puzzle, four out of the five check functions are used to find a contradiction (only the squares check is not used). Next to that, all three actions are used for solving the puzzle.The top left puzzle will be hard to solve, since the modified contradiction algorithm has to use backtracking to solve it. The 5×5 room on the right can be solved without backtracking, but does need to use the two room coloring function. The other three puzzles can be solved using only single room colorings. The varied use of actions and checks make such puzzles interesting, and therefore a good alternative for larger puzzles, since large puzzles take even longer to generate than puzzles with small rooms. The solutions of the puzzles in Figure 11 can be found in Figure 12.

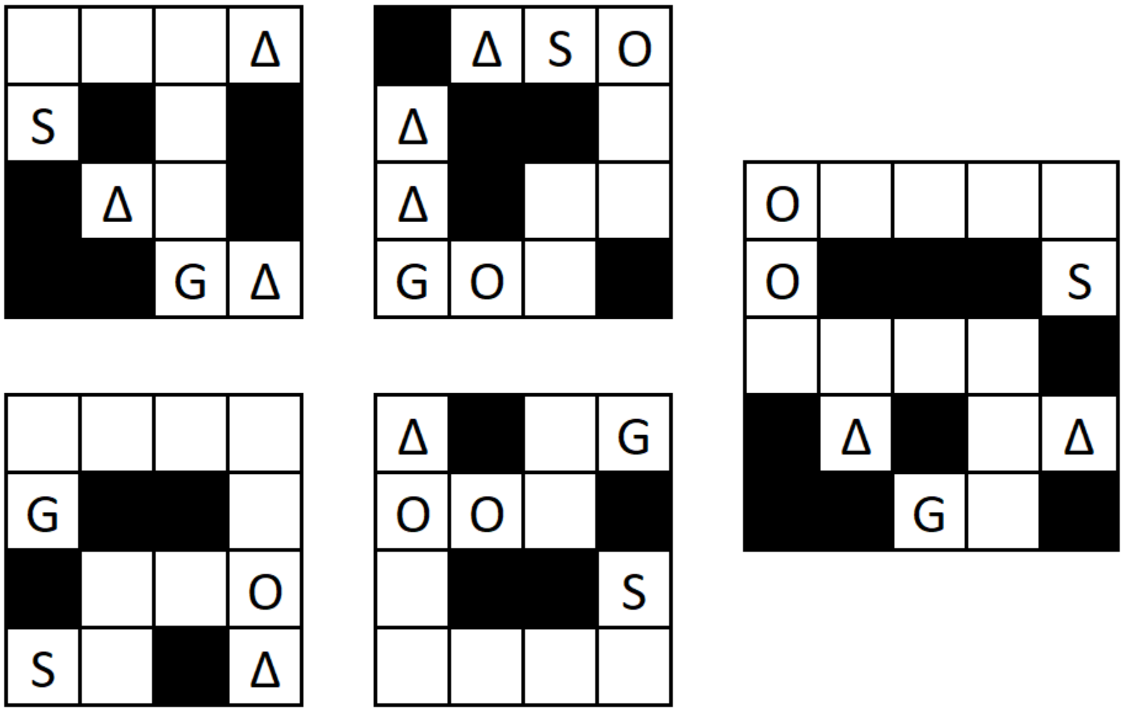

Figure 12: Solutions of generated puzzles containing rooms of size 1.

6 Conclusions and Further Research

In this thesis several ways of solving a Nurimeizu puzzle are discussed. The goal of solving a Nurimeizu puzzle is to color rooms to create a path from S to G. The backtracking algorithm is a simple, but slow way to solve the puzzle. A faster, contradiction based algorithm is presented, which can solve a puzzle faster in almost every case. When the contradiction algorithm cannot solve the puzzle, backtracking is applied, but this time only on the rooms that are not colored yet. Some modifications to the contradiction algorithm were considered as well. The modification that had a positive effect for almost every puzzle was trying to find a contradiction by coloring two rooms. Other modifications turned out to decrease the efficiency in some cases but have little effect or even a small negative effect for some other puzzles. In conclusion, the effect of algorithm choices depends considerably on the configuration of the puzzle.

Finally, uniquely solvable Nurimeizu puzzles were generated randomly. It was possible to create puzzles of size 5×5 and smaller in a reasonable amount of time. The distribution of the room sizes and the number of circles and triangles turned out to have an effect on the success rate of generating such puzzles. Most of these distributions that resulted in a larger success rate also resulted in puzzles that would be less interesting for a human to solve, since there would be little variation in solving techniques.

There are several aspects of solving and generating the Nurimeizu puzzle that further research could focus on. This thesis focused on an algorithmic approach of solving a puzzle, but it might also be possible to use a SAT solver to find a solution. Another thing that could be improved upon is the implementation of the algorithm. For example, changing the data structure in which visited cells were stored during the execution of the *white_connected* function turned out to improve the runtime. This thesis, however, focused on the computational side of improving the algorithm, but using different data structures or techniques could also decrease the runtime. Finally, it might be useful to execute parts of the algorithm in parallel. This way several coloring attempts can be performed at the same time, improving the efficiency of the algorithm.

References

[BS17] M. Biro and C. Schmidt. Computational Complexity and Bounds for Norinori and LITS. *33rd European Workshop on Computational Geometry*, pages 29–32, 2017.

[II21] C. Iwamoto and T. Ide. Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Article ID: 2021DMP0006, 2021.

[II22a] Y. Sato, A. Ishibashi and S. Iwata. NP-completeness of two pencil puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88:237–246, 2022.

[II22b] C. Iwamoto and T. Ide. Five Cells and Tilepaint are NP-Complete. *IEICE Transactions on Information and Systems*, Vol. E105-D, No. 3:508–516, 2022.

[JJ22] A. Janko and O. Janko. Nurimaze. *https://www.janko.at/Raetsel/Nurimaze/index.htm*, Accessed 10-3-2022.

[Kö12] J. Kölker. Selected Slither Link variants are NP-complete. *Journal of Information Processing*, Vol. 20, No. 3:709–712, 2012.

[NC22] Ltd. Nikoli Co. Nurimeizu. *https://www.nikoli.co.jp/en/puzzles/nurimeizu*, Accessed 18-1-2022.