Universiteit Leiden
The Netherlands

Opleiding Informatica

How to solve and generate

Marupeke puzzles

Casper Jol

Supervisors:
Jeannette de Graaf & Hendrik Jan Hoogeboom

BACHELOR THESIS

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

16/08/2023

Abstract

Marupeke is a logic puzzle game. We will look at different approaches to solving these puzzles and how to do it most effectively, as well as discuss ways on how to generate these puzzles from scratch, and when we can accept it as a "good" puzzle. Lastly we will also see if we can apply a Sat-solver to help solve these puzzles and look at the different ways we can use this.

Contents

1 Introduction **1**

2 The game: Marupeke **1**

3 Solving puzzles **3**
- 3.1 The environment . . . 3
- 3.2 Basic puzzles . . . 4
  - 3.2.1 Array search . . . 4
  - 3.2.2 Vector search . . . 5
- 3.3 Advanced puzzles . . . 6
  - 3.3.1 Method 1: Score-based guessing . . . 6
  - 3.3.2 Method 2: Backtracking . . . 7
- 3.4 Comparison . . . 7
- 3.5 Patterns . . . 8

4 Generating puzzles **10**
- 4.1 Random algorithm . . . 10
- 4.2 Cut-off algorithm . . . 11
- 4.3 Tests . . . 13

5 Sat-solver **16**
- 5.1 Logic formulas . . . 17
- 5.2 MiniSAT . . . 18
- 5.3 Finding the solution . . . 18
- 5.4 Sat-solver for unique solutions . . . 20

6 Conclusion **22**

References **23**

1 Introduction

Marupeke is a Japanese Tic Tac Toe-like game in which the player has to use logical reasoning to complete it. I stumbled upon this puzzle whilst looking for game to do research on for the bachelor project. After doing some research on this puzzle I found out that it is rather unknown and that little to no research had been done on it. It was, therefore, a good opportunity to take this game up as the subject for this thesis and find out what qualities it has.

After doing a few of the puzzles, my instructor and I had a few questions, such as what is the most efficient way to go about solving a puzzle, how could we efficiently generate Marupeke puzzles with unique solutions and how can we reduce this puzzle to a SAT problem? All these questions together resulted in the following research question: "How to solve and generate Marupeke puzzles?"

In this thesis I will go on to answer all of the above questions in detail. I will discuss what type of algorithm is used to solve the puzzles and how that has been constructed, why certain choices were made in the programming process, what constitutes an "advanced" puzzle and how we can generate them. Finally, I will also discuss the reduction of this puzzle to a satisfiabiliy problem and how we can use this to solve the puzzles. But first of all, I will explain the rules and aim of Marupeke.

2 The game: Marupeke

As mentioned in the introduction, Marupeke is a sort of Japanese adaptation of Tic Tac Toe. It was invented in 2009 by Naoki Inaba. There is a starting, square-formed, grid, usually varying from 6 by 6 squares to 10 by 10. The grid is filled up with some black squares, squares with an x, squares with an o and empty squares that are yet to be filled. An example of this starting grid can be seen in Figure 1. The goal is to fully fill the grid with x's and o's, with the constraint that there are never more than 2 consecutive symbols of the same type. This goes for all directions, including diagonally. This could be seen as a single-player version of Tic Tac Toe.

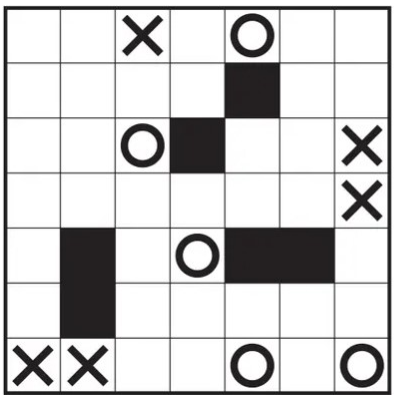


Figure 1: 7x7 starting grid

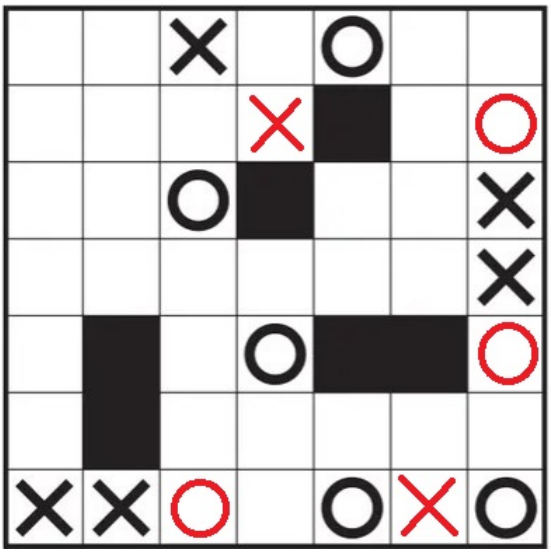


Figure 2: The grid of a Marupeke puzzle after 5 moves

If we take the starting grid from Figure 1 as an example, the first step would be to, place an 'o' to the right of the two x's and an 'x' between the two o's on the bottom line. This step is showcased in Figure 2. In this figure we also see that two other starting moves can be made, the o's on either side of the two x's on the right, and the 'x' between the two diagonal o's. After these have been placed we continue with this same strategy of finding either two adjacent symbols, or 2 symbols with a single space between them, and then placing the opposite symbol to stop the sequence. Marupeke puzzles can vary in difficulty, some being reasonably simple to fill in by hand, others requiring the player to spot patterns and think multiple steps ahead.

3 Solving puzzles

Now that the rules and goal of the puzzle have been explained, we can start looking at the best way to solve one. We will first look at the environment created for the algorithm to approach the puzzle and interact with it. We will then look at ways to solve the simpler puzzles, where less logical thinking is required. Finally we will look at the best way to solve the more complicated puzzles and why these need an alternative approach.

3.1 The environment

Before creating an algorithm that can successfully solve all puzzles, we made sure it could first solve the basic puzzles, and then later move up to the more complicated ones. For the simpler puzzles, there is no need to do think ahead; the puzzle can simply be filled in by following the steps we used in the example. The part of the algorithm that fills in the grid according to these steps is the most integral part of the whole algorithm, because even with the more complicated puzzles, these rules are essential. That is why we chose to start with this part.
First of all the algorithm needs a way to interact with the puzzle. This can be achieved by creating a 2-dimensional array in which the puzzle can be stored. All that is needed is a text file with some information on what the size of the puzzle is and which entry points contain which elements. What such a file looks like can been seen in Figure 3.
The first line of the Figure contains two numbers separated by a space. These numbers coincide

```
1   10 10
2   w x w x z w w z w z
3   w w w w w z w w w w
4   w z o w w w w o w z
5   z x z x w z w w w x
6   w w w z w z z w w w
7   w w w w w w w w w w
8   w w z w w w w z w w
9   w w o w o z w w w w
10  w w z w o w z w w w
11  w z w z w w w w w w
```

Figure 3: Text file with information about puzzle for conversion into array

with the size of the puzzle. So in this instance, the puzzle has 10 rows and 10 columns. This way the algorithm can immediately create an array with the correct size. Below the numbers we see the puzzle. There are four different characters; w corresponds to a blank space, z to a black space and x and o with their own characters.

3.2 Basic puzzles

Now that the puzzle has been placed in an array we will develop a function that can interact with this array and fill it up according to the rules of the game. It is possible to use brute force, but with this puzzle that is far from the best option. Brute force would fill up the grid in every way possible until it has found the solution but this is inefficient for basic puzzles; the rules of the puzzle are simple, so it is not extremely hard to apply them to the solving algorithm. In the next two paragraphs we will discuss two solving methods that can successfully be applied.

3.2.1 Array search

To approach each square in the grid we use a basic array walk, starting in the upper left corner and then moving from left to right. For every square the algorithm will check for a sequence of 2 symbols in its perimeter, meaning every direct neighbour and its extension when forming a sequence. This has been visualised on a random square in Figure 4.
Here the square marked with the green edges is the point that is approached. The red squares are the squares directly adjacent to the square that was approached. These get checked first.
If the adjacent square has the same symbol, then the opposite symbol must be placed on either side of the square, unless these squares are black. The algorithm will then move on to the next red square. If the square is empty, the algorithm will continue to the blue square in that same direction. If this has the same symbol as the green square the algorithm places the opposite symbol in between them, so in the red square. In Figure 4 we can see this is the case when the algorithm checks the top-right blue square. If a black square is encountered nothing happens and the algorithm moves on to the next red square. In Figure 5 we can see exactly in which cases a symbol is placed.

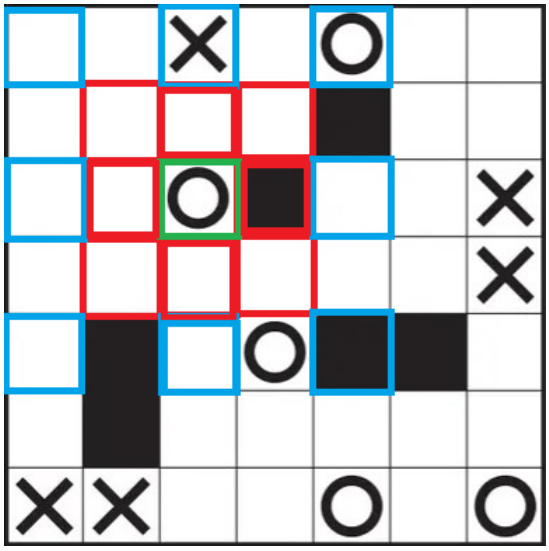


Figure 4: Visualisation of neighbour squares that are checked when a certain point is approached

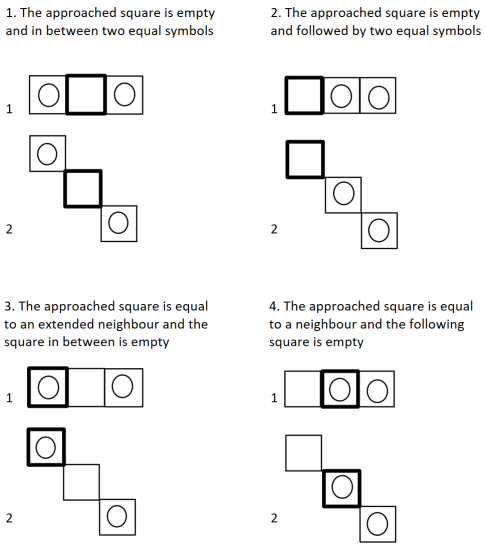


Figure 5: Cases a symbol is placed

This method is successful, however it is not extremely efficient, as it needs to check multiple squares way more often than required. This is because the algorithm will blindly go over the squares until it has found the solution. There is a way, however, for the algorithm to choose the square it approaches more closely.

3.2.2 Vector search

A more efficient way would be to keep track of which squares were most recently filled by the algorithm, and then approach that square to see whether any new moves have been made possible. This is because the next move will always come from a square in which the state has been altered, since this is essentially the new "clue". We can implement this by creating a vectorlist. This is then filled with all the squares, in the same order as they were approached in the previous section. These are then visited in order, removing the visited squares from the list and adding the squares in which we made a change to the end of the list. This way we essentially create a queue. After the algorithm has visited every square once, it will move back to the squares in which a change has taken place and work the queue. This way after the first initial round, the algorithm will not keep visiting squares in areas that are completely filled up, and thus are not of use anymore. It will merely visit the entries in which a state change has occurred and another move might be possible.

When the queue is empty, this means one of two things; the puzzle is finished, or there are no more moves possible with this algorithm and we must use another approach. This will often be the case with more advanced puzzles. This algorithm works slightly more efficiently on the basic puzzles than the one that uses the array search.

3.3 Advanced puzzles

As mentioned in the previous paragraph, it is not always possible to solve the puzzles by following the steps based on the rules. Sometimes, with more difficult puzzles, we see that only a few spaces can be filled in this way, before no longer being able to continue. In these cases, it is necessary to think ahead to be able to place a symbol. The puzzle shown in Figure 6 is an example of this. We can fill in the grid, until no more obvious moves are left and then have to use a different strategy. We will describe two different methods of dealing with these advanced puzzles.

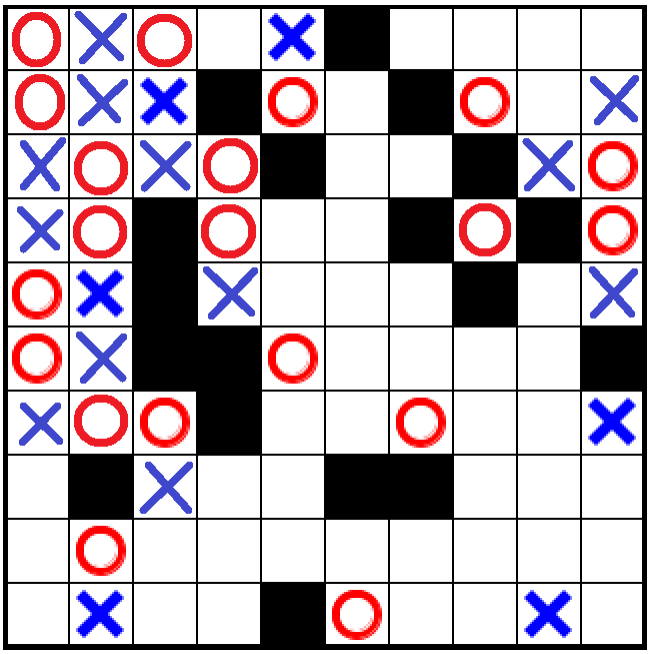


Figure 6: Example of puzzle that has been filled until there are no more obvious moves left

3.3.1 Method 1: Score-based guessing

One solution can be to create a copy of the playing board and first place an x in an empty square and then an o. For both symbols we rerun the original vector search algorithm; if one of the two leads to a contradiction we know this is the wrong symbol. This means we can now fill in the other symbol in that square in the original grid. We then rerun the original algorithm and go through this process again until the puzzle is completed. It is possible that in the copy of the board, one of the symbols instantly leads to the correct answer. In this case we consider the puzzle completed. This might ,however, not be the only possible solution, but this algorithm only finds one solution, even if others are possible.
A consideration here is that it is important which square is selected. There is little use in running the vector search on a tile which is not connected to any other symbols; in other words, we must consider the tile's 'busyness'. We do this by choosing a tile with the most symbols in its direct perimeter. By this we mean its eight direct neighbours and in turn all their neighbours. Essentially, this is all the red and blue squares from Figure 4, and including the unmarked squares between the blue ones. In this way, if one of the symbols leads to an error, there will be a good chance we can

still make further progress. We call this the 'busyness' score. The 'busyness' score of each square can be determined by visiting all the neighbouring squares and calculating a score based on the number of empty ones found. As a second factor we can also look at how central the square is. If a square is more central, there is a higher chance it can affect more squares on the grid further down the line. This is added as a decider, when multiple squares have the same 'busyness' score. It, therefore, does not immediately affect the initial score. When all the scores have been determined, the square with the highest score is visited. If this does not result in any progress, the square with the second highest score is visited, and so on until a square is found where progress can be made. Here is a quick recap of the steps for the score-based guessing algorithm:

1. **Apply vector search algorithm**
2. **Algorithm stalls → no more moves possible by following the rules**
3. **Calculate score for each square Place each square in ordered list**
4. **Test both symbols in square, then use the rule-based algorithm to continue the play through**
5. **If one of the symbols leads to a contradiction, the other symbol can be placed on the actual board → go back to step 1**
6. **If neither symbol leads to a contradiction, move to next square in ordered list**
7. **Continue until a symbol can be placed**

3.3.2 Method 2: Backtracking

Another way to approach solving the puzzle is to use a backtracking algorithm. This implementation will always lead to the solution, given that there is one. This algorithm does not use logic when filling up the grid but, it starts at the top-left corner and places an x in the first empty square it comes across. If this does not result in three equal symbols to be sequential, an x is placed in the next open square. The algorithm moves from left to right per row, moving to the next row when the end of the previous one is reached. If the x leads to a breach of the rules it is replaced with an o. If the o does not break the rules, the algorithm moves to the next square. If it does break the rules, the algorithm moves back until a square containing an x is found, replaces this with an o and tries again. This goes on this way until the grid has been filled up without any mistakes. Then a solution has been found.

3.4 Comparison

To figure out which of the two methods is better suited to solve puzzles, we had to run a test. We lined up 100 puzzles that were generated by the algorithm we will discuss later on in Section 4. We let both methods solve these all these puzzles consecutively and recorded the time it took. The results are shown in Table 1. In this Table we see that score-based guessing is at least four times as fast as backtracking. Because the time is measured in seconds we cannot see how long the algorithm took exactly. These results are in line with the expectations, since the backtracking algorithm has to do more work than the score-based guessing algorithm.

Backtracking is not as time-efficient as the score-based guess algorithm. It does,however, have one big advantage; it will always find all the possible solutions. So if a puzzle has more than one solution, this algorithm will find them. If we want to find out how many solutions a puzzle has, we must use backtracking. Score-based guessing gives us the first solution it finds, backtracking gives us all the solutions. This will be useful in the next phase, when trying to generate puzzles.

Method	Time in seconds
Score-based guessing	<1s
Backtracking	4s

Table 1: Time required by both solving algorithms to solve 100 puzzles

3.5 Patterns

In logic puzzles like Marupeke, it can be very useful to discover patterns, which might help solve the puzzle. This means that certain symbols could be placed in an empty square, without it being made apparent by following the rules of the puzzle. This could help in situations in which the vector search algorithm gets stuck after no more obvious moves can be made. To find such patterns we must analyse the puzzle more closely. In Figure 7 we can see the standard solution for a Marupeke puzzle for an empty grid.

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

Figure 7: Standard solution for empty grid

We can see here that the symbols show up in pairs, altering between each other. This suggests that placing a symbol in such a way to form an L shape with 3 equal symbols, this would cause a contradiction. This is only true under the condition that there is enough space around, so no black squares or sides. Figure 8 visualises this. To prove this, we will play out the scenario in Figure 9.

In the first grid we can see all the blue x's. These immediately follow from the three o's. In that same grid we can also see the small o's that must follow after the blue x's have been placed. Here we spot loads of contradictions. This proves that, provided there is enough space, we cannot have three o's together. The next questions is: what is enough space? In the next to grid we see all the x's and their consequent o's required to get a contradiction. This means there must be at least enough space for these x's and o's to be placed. This leads us to the patterns as seen in Figure 10. So if we spot two diagonal o's with the space around them matching either of these patterns, we can instantly place an x, because we know for sure that an o will lead to a contradiction. This also works when for rotated forms of these patterns.

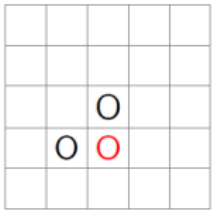


Figure 8: visualisation of the symbols

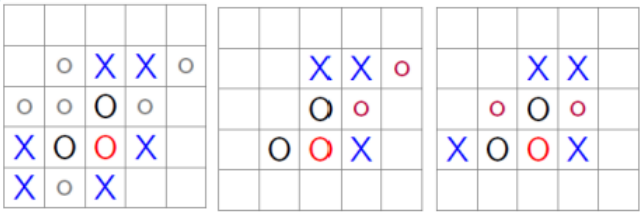


Figure 9: Playout of the scenario

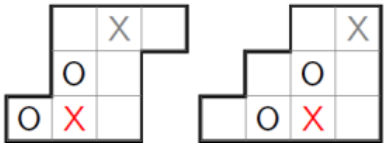


Figure 10: The two patterns found

Unfortunately, when it comes to implementing this in the solving algorithm, we found that these patterns occur quite rarely, so a constant search for them is not worth it. In the end we decided not to include these patterns in the algorithm.

4 Generating puzzles

When generating puzzles there is a constraint which needs to be considered, namely, that the puzzle must have a unique solution. If it does not, it cannot be considered a good puzzle. To generate a puzzle with a unique solution, an algorithm has been written in two parts. The first part generates a random starting grid, which might have loads of possible solutions, or none. The second part takes this grid and moderates it in such a way so that there is only one solution to the puzzle. For this study, we will generate puzzles in sizes ranging from 5x5 to 10x10, with the main focus being on 10x10 puzzles.

4.1 Random algorithm

To create the grid, the algorithm introduces a 10 by 10, empty array that will randomly be filled with x's, o's and black tiles. The question here is, how full must each grid be? If it is too full, it takes away a lot of the challenge and the puzzle becomes too easy to complete. If it is too empty, it may require the person trying to solve it to have to think ahead a lot of the time and make guesses, causing the puzzle to be too difficult. After researching existing Marupeke puzzles we found they we're filled to approximately 20%, with about 10% being black. We also found that the number of x's and o's was often quite similar, if not equal. With this in mind, we could set parameters on how to fill the grid. We filled this grid randomly, since we could spot no clear initial pattern or template with which to create the puzzles. We created bounds for the black tiles and symbols, and selected a random number between these bounds. We then filled the grid by randomly placing the black tiles and symbols on the board. Multiple bounds were tested and the results can be found in Table 2. On each board the test was run until 100 puzzles with at least 1 solution were generated.

Grid size	Bounds black tiles	Bounds symbols	Time	Unique solutions
5x5	5-7	4-6	1h 30m	7
6x6	3-7	3-9	1h 17m	1
6x6	2-11	4-14	2h	0
7x7	4-8	4-10	1h 10m	0
7x7	4-7	4-8	1h 18m	0

Table 2: Statistics of tests run on randomly generated boards

When we look at this Table, we see that on smaller sized boards we can generate a uniquely solvable puzzle. However, as the size of the grid increases this becomes less likely. With a 6x6 grid we see that this occurs once within the 2x100 puzzles generated and with the 7x7 grid it does not occur at all. Considering the goal is to generate puzzles with a size of 10x10, this method is not feasible. Furthermore, the run time is well over an hour, which is too long.

In order to generate a good puzzle, the algorithm needs to be smarter. A way to do this would be to implement patterns spotted in existing puzzles into the ones to be generated. After inspecting a book of 100 Marupeke puzzles we noticed that the black tiles often occurred in patterns of 3 tiles forming an L shape, six pairs attached diagonally, and six single tiles. This makes 21 black tiles

in total. We therefore filled the grid with the black tiles in line with these patterns. An example of what this could look like can be seen in Figure 11. After running the same test, there was a slight improvement in the time required to generate 100 solvable puzzles, about an hour, but the number that had a unique solution remained zero. This leads to the conclusion that simply relying on random generation is not enough.

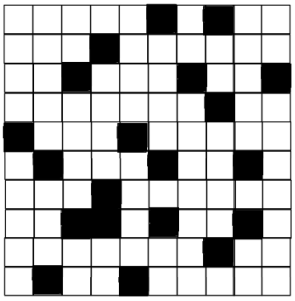


Figure 11: Example of how a grid could be filled with black tiles when using patterns

4.2 Cut-off algorithm

We needed a better algorithm than random generation. The algorithm had to be able to make decisions when creating the puzzle in order to assure its quality. We did, however, want to keep the random nature of the generation, to avoid the puzzles becoming repetitive. A way to do this is to generate a puzzle the same way as described above, but rather than using this as the finished product, allow the algorithm to alter or improve it such that it becomes a puzzle with one solution. We do this by calculating the number of solutions the puzzle has, and then forcing the puzzle to have only one of the found solutions by cutting off the other possibilities. This is why we call it the cut-off algorithm. The algorithm tests all the possibilities by adding a single symbol to an empty square and then checking how many solutions the puzzle would have if that were the starting grid. It does this for every empty square and the option that cuts off the most possible solutions is accepted and the relevant symbol is placed permanently. The algorithm repeats this process until a puzzle is found with a unique solution. The results of the tests run can be found in Table 3.

Bounds symbols	Time in minutes
3-4	25m
4-5	30m
5-6	60m
6-7	60m
7-8	110m
4-9	150m

Table 3: Time it took to generate 5 puzzles with a unique solution on a 10x10 grid tested on multiple bounds

This Table shows the time it took for multiple tested bounds of x's and o's to generate 100 uniquely solvable puzzles. This includes the time it took to generate the random starting grid. A range of 4-9 means that between 4 and 9 x's were generated and between 4 and 9 o's. So the number of x's and o's can differ. The black tiles were generated in the patterns as described in Section 4.1.

As can be observed in Table 3, a smaller range with fewer symbols to start with gives us the best results. This goes somewhat against expectations. Our expectation was that the fuller the grid is, the fewer possibilities there are for multiple solutions, since the board is already more restricted. However, when running the tests we observed that for fuller grids it took the algorithm a long time to generate a solvable puzzle. This was the bottleneck of the algorithm. When the generated starting grid is emptier to begin with, there is a bigger chance that a solvable puzzle is found, meaning the cut-off algorithm can be used sooner. If we compare the range 7-9, which has the highest lower-bound and 3-4, which has the lowest higher- and upper-bound, we see that these give us respectively the slowest and quickest results. the total time it takes to generate a uniquely solvable puzzle to decreases by about 75%.

This method enabled us to get the time down to about 30 minutes to generate five unique puzzles. This is already far better than before, but we can make the algorithm that improves the puzzle more efficient. At the moment the algorithm searches every option and chooses the best one. To try to further improve the time, we will experiment with different rates of improvement. We will call this the improvement rate. What we mean by this is that every symbol placed cuts off other solutions paths. Previously we checked every square and then placed the symbol that gave us the least remaining solutions, so the best improvement. By introducing the improvement rate we will also accept lesser symbols that are not necessarily the best option. For example, when placing a symbol achieves at least a 50% improvement, so that the number of possible solutions halves, that option can be accepted and the algorithm starts again. This way the algorithm will not have to search the whole grid every time, thus saving time.

However, at the later stages, when only a few solutions remain, we want to find the best option. This is because it will now take less time to find all the solutions for every square because fewer solutions remain. It is possible that the reward for searching for the best option is worth the slightly longer search time at this stage. We will refer to this number as the border. At the beginning, when the grid is relatively empty the number of solutions can be up to 100,000. To start the testing, we will set the border at 1000 solutions, meaning that when a grid has fewer than 1000 solutions left, the algorithm will revert back to the original cut-off algorithm, without the improvement rate and search for the best possible option again. We will refer to this new algorithm as the improvement-border algorithm.

Table 4 shows the results on the tests run with the improvement-border algorithm with a 50% improvement rate. We can see another clear improvement in the times found. This suggests that this is a correct approach. We ran the tests another time, this time seeing what happened if we bring back the random placement of the black tiles in comparison to when they were placed in patterns. We see that by implementing the improvement-border algorithm, the patterns in the black tiles no longer have much influence. We also see again that when the range for the number of symbols is set low, so in this case 3-4 or 4-5, it takes less time to generate five good puzzles.

Bounds symbols	Number of black tiles	Black tiles in pattern	Time
3-4	21	Yes	25m
4-5	21	Yes	15m
3-4	12	No	1h 45m
3-4	20	No	16m
4-5	20	No	17m
5-6	20	No	28m

Table 4: Time needed to generate 5 puzzles with a unique solution on a 10x10 grid, for starting grids with different bounds of symbols and black tiles using the improvement-border algorithm

4.3 Tests

The improvement-border algorithm can use a lot of fine tuning when it comes to the improvement-rate, the bound and the ranges. To figure out the best values for these, we will run tests and look at the results.
In the first test we will look at the best value for the improvement-rate.

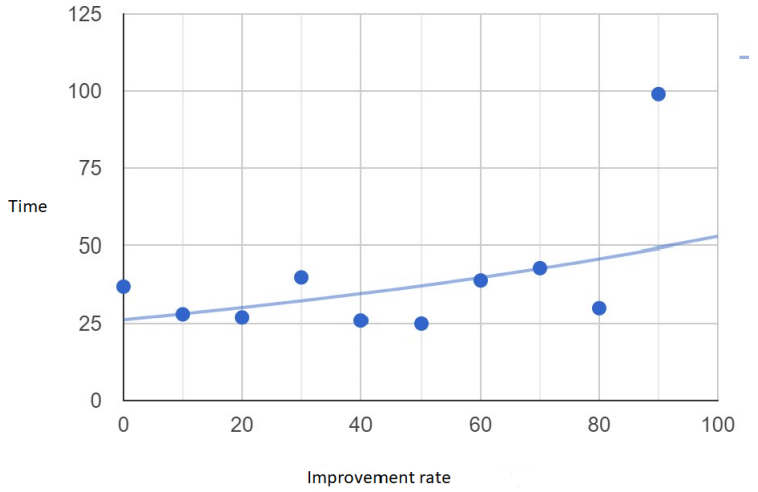


Figure 12: Duration for different improvement-rates to generate 10 (10x10)puzzles with a unique solution, with the time in minutes

As we can see in Figure 12 we could not detect a very clear trend line. We can see that the improvement-rate, from 60% upwards, the time required does start to increase, with a 90% improvement-rate taking much longer than the rest. This coincides with expectation, because looking for an improvement of 90% before is a very high bar. Such a high bar could sometimes be very useful, because the reward is also very high. In this case however, it turns out that this does not occur often enough for it to have a positive impact.

All these improvement-rates were tested with a border set at 1000 solutions, meaning that after the puzzle had fewer than 1000 solutions, the improvement-border algorithm would revert back to the cut-off algorithm. To test whether this border is in fact effective and, if so, whether 1000 is

the ideal border to set, we ran the same test as in Figure 12 but on multiple different borders to test the ideal value. We can see the results of these tests in Figure 13. Here we can see roughly the same trend as we did in Figure 12. The run times to reach 10 uniquely solvable puzzles stay similar until we ask for better improvement-rates. We can see that the increase in times start to happen from around 70%, except for when the border is set at 250. Here we see a huge spike in run times.

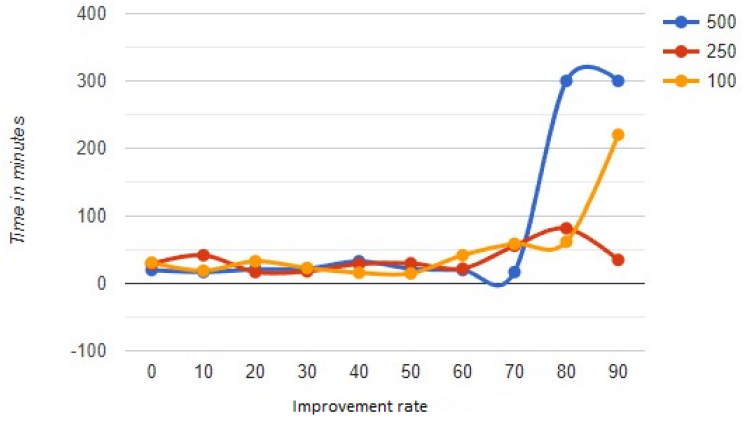


Figure 13: Duration for different improvement-rates to generate 10 puzzles with a unique solution, run for three different borders

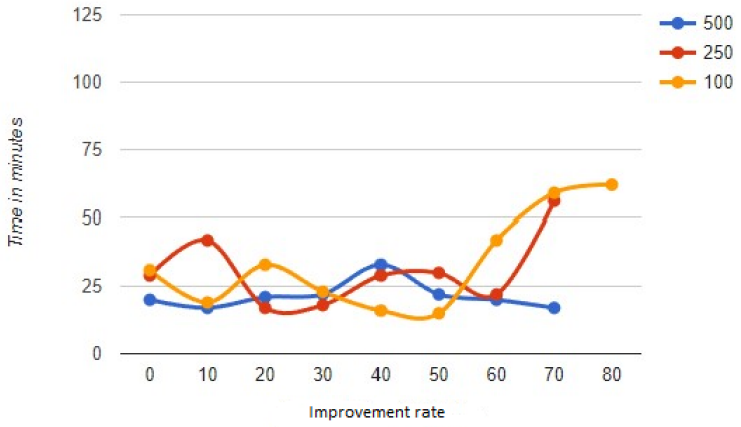


Figure 14: Duration for different improvement-rates to generate 10 puzzles with a unique solution, run for three different borders

In Figure 14 we can take a better look at the found results, with the huge spikes in run time being removed from the graph, namely the improvement-rate of 90%. Here we can see that the best found times are just below the 20-minute mark. While we can see that the two best run times are found with the border set at 100, there is not a huge difference, so we cannot state with 100% certainty which border is the most optimal.

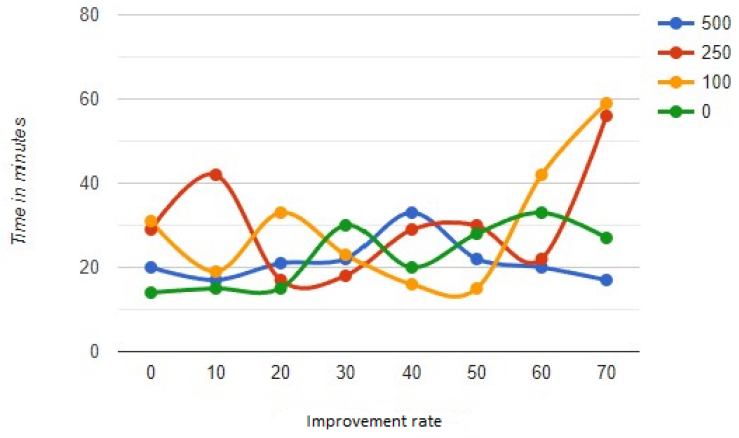


Figure 15: Duration for different improvement-rates to generate 10 puzzles with a unique solution, with the border removed

Figure 15 shows the graph indicating what happens when the border is completely removed. Here we see that with low improvement-rates, the search time for 10 unique puzzles is 14 minutes, the shortest found so far. The border was there to allow the algorithm to search for the best improvement, when a certain number of remaining solutions was achieved. The thought behind this is that the extra time needed to check all the results would be worth the reward of finding the best option. This turns out not to be the case. With the best time found with an improvement-rate of 0% and no border, we can conclude that the quickest way to generate puzzles with the improvement-bound algorithm is to place a symbol when any kind of improvement is found, hereby ignoring how big this improvement is. The border can be completely removed. In Figure 16 there are three puzzles that have been created using this final algorithm.

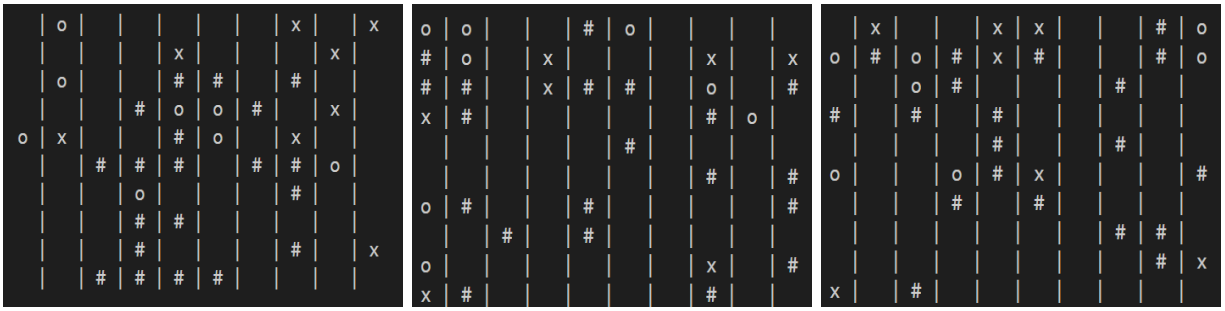


Figure 16: Three puzzles generated with the final algorithm

5 Sat-solver

A Sat-solver, or satisfiability solver, is an algorithm developed to determine whether a set of logical formulas can be satisfied. When given a list of Boolean functions, it will calculate whether a combination of variables exists, whilst still complying with the clauses. As output it will produce the list of variables that must be true. If no such combination of variables exists, the solver will report that the problem in unsatisfiable. So if a given set of logic formulas is satisfiable, it means that a combination of variables exists, all of which have been given boolean values, exists, such that all the clauses in the set are true.

A problem must be fed into the Sat-solver, in CNF(Conjunctive Normal Form). This means that the formulas are formulated in an "and over or" construction, also known as a product of sums. Statements can be negated by the "not" operator. Here are two examples of a CNF clause:

1. $(\neg X_1 \vee X_2 \vee X_3) \wedge (X_1 \vee \neg X_4)$

2. $(X_1 \vee \neg X_2) \wedge (X_1 \vee X_3) \wedge (X_2 \vee X_3)$

Here X_1.....X_4 are Boolean variables that can be either true or false. If these two problems are given to a Sat-solver, the output could be as follows:

1. $X_2 \wedge X_4$

2. $X_1 \wedge X_3$

This means that these variables are set to true, with the other variables set to false.

5.1 Logic formulas

We can apply a Sat-solver to Marupeke. To do this we must first reduce the puzzle to an instance of SAT. We can do this by constructing logical formulas that perfectly describe the puzzle and its constraints. We can then feed these formulas into a Sat-solver. The logical formulas for a Marupeke puzzle are described as follows:

m: Marupeke puzzle

z: size of one side of grid

c: character. Can be one of the three characters in a completed puzzle; black, x or o.

t: symbol. Can be one of the two symbols, x and o, that can still be placed in a not yet completed puzzle

So $m_{i,j,c}$ describes the character c on coordinate i,j in Marupeke puzzle m.

At least 1 character per entry
$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \vee_{c=1}^{3} m_{i,j,c}$
with c one of the characters x,o or black on coordinate (i,j)

Maximum 1 character per entry
$\wedge_{i=1}^{n} \wedge_{j=1}^{n} \wedge_{c=1}^{2} \wedge_{d=c+1}^{3} \neg m_{i,j,c} \vee \neg m_{i,j,d}$

$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \neg m_{i,j,t} \vee \neg m_{i+1,j,t} \vee \neg m_{i+2,j,t}$

$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \neg m_{i,j,t} \vee \neg m_{i,j+1,t} \vee \neg m_{i,j+2,t}$

$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \neg m_{i,j,t} \vee \neg m_{i+1,j+1,t} \vee \neg m_{i+2,j+2,t}$

$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \neg m_{i,j,t} \vee \neg m_{i+1,j-1,t} \vee \neg m_{i+2,j-2,t}$

The first two lines make sure that every entry is filled with exactly one character. Without these two lines an empty grid would also be deemed satisfied, when in reality this is obviously not the case. The bottom four lines state that either one of three consecutive entries in puzzle m, cannot all contain the same character t, with t being either an x or an o. Each one of the four lines checks a seperate direction.

5.2 MiniSAT

We used the MiniSAT Sat-solver on Marupeke. MiniSAT is an open-source Sat-solver[NE05] specifically developed for researchers to utilise[Whe08]. MiniSAT has developed it's own syntax based on CNF, called DIMACS CNF. Unfortunately, it is not possible to give the logical formulas that describe the whole set of CNF clauses to MiniSAT. We therefore had to convert these rules into CNF clauses. To do this we wrote a program to take care of the conversion. The clauses then had to be written as per the MiniSAT syntax. This looks slightly different to the clauses described in CNF-form. An example of what this DIMACS CNF format looks like can be found in below:

```
c Here is a comment
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

This example consists of 5 lines in total. The first two lines are introductory, the bottom three are clauses. In general, other than the first two lines, each line is a separate clause, so here we see three clauses. Each number in these clauses is a variable, with one indicating variable 1, 2 indicating variable 2, etc...The variables 5 and -5 both indicate variable X_5, with the - being the negation operator, so -5 is the equivalent of $\neg X_5$. The number 0 indicates that the end of the line has been reached, so it is also the end of a clause.
On the first line we see the letter c followed by the text "Here is a comment". Everything followed by the letter c on the same line is not considered by MiniSAT, so it is simply information for the programmer/user. Every file loaded into MiniSAT, which is simply a text file, must start with the letter p, followed by cnf, showing that the file is in CNF, then the number of variables used in the clauses, and finally the number of clauses in the file. If we view these clauses in normal CNF syntax, they would look like this:

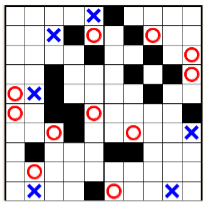
$$(X_1 \vee \neg X_5 \vee X_4) \wedge (\neg X_1 \vee X_5 \vee X_3 \vee X_4) \wedge (\neg X_3 \vee \neg X_4)$$

5.3 Finding the solution

To use MiniSAT as a solver for the Marupeke puzzles, we must therefore convert the logical formulas described in Section 5.1 into the DIMACS CNF syntax. To do this we must first create variables for each square and symbol. When we consider a filled-out grid, without knowing what it looks like every entry has three possible values: x, o or black. Since we are only focusing 10x10 puzzles, this means that there must be a total of 300 variables(100 entries, three possible values per entry). This means that three variables would be assigned for every entry. For entry 1, coordinate [0,0] in the grid, this would be variables 1, 2 and 3, for entry 2 with coordinate [0,1] this would be variables 4,5 and 6, and so on. Every first variable stands for the symbol x (variables 1,4,7,etc...), every second

variable stands for the symbol 0 (so variables 2,5,8,etc...) and every third variable stands for a black tile(3,6,9,etc...). We need a total of 1076 clauses to describe the problem. This consists of 100 clauses describing that every entry needs at least one value, so cannot be empty, another 400 clauses to describe that every entry can have at most 1 value, and the other 652 clauses making sure that in every direction there are no more than 2 consecutive x's or o's. We wrote a separate program that generates all of these clauses, to avoid having to write them out by hand.

To find the solution of a specific Marupeke puzzle, we must let the solver know what the puzzle looks like. This means we have to describe the puzzle with clauses and add these to the existing ones. To do this, we need 100 clauses, all with one variable, stating the value of each entry in the grid. We have 100 entries, so we need 100 clauses. In these clauses we force a variable to be either true or false, because a starting value cannot be changed. For example, for the first entry, where:
$m_{0,0,1} \vee m_{0,0,2} \vee m_{0,0,3}$
either variable 1, 2 or 3 would be set to true. Here 1, once again, corresponds to an x, 2 to an o and 3 to a black tile.
If the entry is empty, all we know is:
$\neg m_{0,0,3}$
In this case the third variable must be set to false, meaning either 1 or 2 can still be true, since
$m_{0,0,1} \vee m_{0,0,2}$
is still true. In Figure 17 we can see a possible starting grid for a puzzle and the resulting clauses describing this grid.



-3 0
-6 0
-9 0
-12 0
13 0
18 0
-21 0
-24 0
-27 0
-30 0
-33 0
-36 0
37 0
42 0
44 0
-48 0
51 0
53 0
-57 0
-60
-63 0
-66 0
-69 0
-72 0
75 0
-78 0
-81 0
84 0
-87 0
89 0
-93 0
-96 0
99 0
-102 0
-105 0
-108 0
111 0
-114 0
117 0
119 0
122 0
124 0
129 0
-132 0
-135 0
-138 0
-141 0
144 0
-147 0
-150 0
152 0
-156 0
159 0
162 0
164 0
-168 0
-171 0
-174 0
-177 0
180 0
-183 0
-186 0
188 0
192 0
-195 0
-198 0
200 0
-204 0
-207 0
208 0
-213 0
216 0
-219 0
-222 0
-225 0
228 0
231 0
-234 0
-237 0
-240 0
-243 0
245 0
-249 0
-252 0
-255 0
-258 0
-261 0
-264 0
-267 0
-270 0
-273 0
274 0
-279 0
-282 -
285 0
287 0
-291 0
-294 0
295 0
-300

Figure 17: Starting grid with matching formulas describing it

This input in combination with the already existing clauses talked about in Section 5.2 can now be fed to the Sat-solver. If the puzzle is satisfiable, it will return a similar type of list of variables as above, but now all the negative variables, indicating a white square, will be changed to a variable indicating either an x or an o. So in short, the clause "$\wedge_{i=1}^{z} \wedge_{j=1}^{z} \vee_{c=1}^{3} m_{i,j,c}$" will be satisfied.

This Sat-solver algorithm can be used as an alternative to the score-based guessing algorithm. On average it takes the Sat-solver around 0.006 seconds to solve a puzzle. For 100 puzzles this would be 0.6 seconds. Since all we know is that the score-based guessing algorithm takes less than 1 second, we cant conclude which is more efficient. What is certain is that they are both very fast and the margin between them will be small.

5.4 Sat-solver for unique solutions

A follow-up question with Sat-solvers is whether it can also determine whether a puzzle has a unique solution. This is possible, but it needs a little work-around.
A Sat-solver is a Boolean function, meaning that the input is either satisfiable or not. If we ask the question, "does this puzzle have a unique solution?", we must consider between three possibilities: the puzzle has zero solutions, the puzzle has one solution or the puzzle has more than one solution. When the question whether the puzzle has a unique solution is asked to the Sat-solver and it returns false, we still do not know whether it has more than one solution or zero solutions. This makes a great difference, because when generating puzzles, one with multiple solutions can be improved upon to have one solution, whereas a puzzle without a solution is discarded.

A way around this problem is to essentially feed the puzzle into the Sat-solver twice. The first time we do it in the way discussed in Section 5.3, where we find out if the puzzle has at least a single solution. In the event that this is satisfiable and the list of variables is returned which make up the solution, we can then exclude this solution from the possibilities and use the Sat-solver again. We do this by adding another clause to the already existing list of clauses. This clause is the exact solution that was just returned by the Sat-solver, but now with every variable negated. Since the DIMACS CNF syntax is in the CNF format, this clause states that at least one of the variables in the clause must be false. Since in the given solution all of the given variables are true, this method excludes that answer from possible solutions the Sat-solver is now permitted to find. In Figure 18 we can see an example of this method executed on the example of the DIMACS CNF shown in Section 5.2.

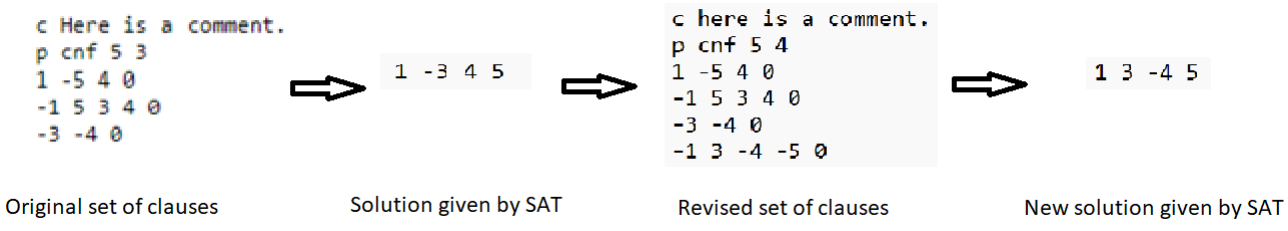


Figure 18: Example of strategy to exclude a solution from SAT

In short, the Sat-solver searches for a solution; if it finds one, we let it find another solution by excluding the solution it previously found, thus forcing it to find a new one. If the problem is still satisfiable, the puzzle has multiple solutions. If unsatisfiable, the puzzle has a unique solution.

An issue with this method is that the user must run the Sat-solver twice and update the file with clauses. To eliminate this problem we wrote a script that adds some extra code files to automate this procedure. This script does this by making use of a pipeline.

6 Conclusion

We took a relatively unknown puzzle and applied different strategies and methods to solve existing and generate new ones. The general strategies we used are known methods such as Sat-solvers and backtracking, but they had not yet been applied to Marupeke.
The vector search method created for Marupeke could be applied to other simple logic based puzzles. It is based upon the simple premise that the points where the state was most recently altered, will provide new information allowing us to progress. Furthermore the Sat-solver can be used for many other problems, as long as they can be reduced to an instance of Sat.

References

[NE05] Niklas Sörensson Niklas Eén. The minisat page, 2005.

[Whe08] David A. Wheeler. Minisat user guide: How to use the minisat sat solver, 2008.