Universiteit Leiden
The Netherlands

Opleiding Informatica

Solving Puzzles

using Cellular Automata

Stef van Dijk

Supervisors:
Hendrik Jan Hoogeboom & Walter Kosters

BACHELOR THESIS

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

08/08/2017

Abstract

This thesis is about cellular automata and whether or not they are able to solve different kinds of puzzles. In the first part it is explained what a cellular autamaton is and how it works. Furthermore is the connection between puzzles and cellular automata explained and some related work is given.

There are three types of puzzles which are investigated in this thesis and additionally it is explained how the cellular automata work which are created for this research to solve these puzzles. It concerns the puzzles: Maze, Nonogram and Binary puzzle. Afterwards, these cellular automata will be evaluated.

The knowledge gained in this research is used to speculate about other puzzles, concerning their capability to be solved by a cellular automaton. This is the case for: Sudoku, Tectonic and Kakuro.

In the conclusions all the research questions are answered. This will make clear that there are puzzles which are solvable by a cellular automaton. The puzzles that must be solved have some properties in common, while the cellular automata solving them can differ much in behavior. Instances of puzzles of a more difficult type are harder to solve for a cellular automaton and will make the automaton use more time to solve the puzzle.

Contents

Chapter 1

Introduction

Cellular automata are very smart, yet a bit rough, models with high computational powers which can simulate the most complex systems. They consist of a grid of cells, like some puzzles do. There are complex logical puzzles, that can be solved by pure calculation. Is there a way to connect such puzzles to cellular automata, by using the cellular automaton to simulate and thus solve the puzzle? Which (types of) puzzles can be solved by cellular automata? If it is even possible to solve puzzles using cellular automata, will these cellular automata look alike? Most logical puzzles have complexity levels. How do cellular automata deal with these levels? Since cellular automata are able to solve NP-complete problems [Gre87], these research questions are not contradicted in advance for NP-complete puzzles.

1.1 Cellular automata

A cellular automaton (CA) is a discrete model known in computability theory. CA's can be used to simulate complex systems which have a spatial structure in which every pixel/place acts as a cell in the CA. That is why CAs consist of a grid of cells, like in Figure 1.1. Every CA has a finite number of states, the one of Figure 1.1 has 2 states. Every cell of the CA is at any moment in exactly one of those states.

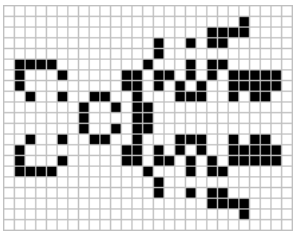


Figure 1.1: Grid with two states; black and white. Source: `http://conwaylife.com/w/index.php?title=Lightweight_spaceship`

CAs work iteratively, which means that there are multiple generations which follow each other. The formation of all the states of all the cells in a generation is called the configuration of that generation. The starting

configuration evolves every iteration of the CA. The fact that there are multiple different configurations, makes clear that a cell can change his state. This is exactly the point of CAs. Every iteration, all the cells in the grid simultaneously change state. In the new generation every cell is in a state that differs or not from its state in the previous generation. Figure 1.2 shows ten consecutive generations of a certain CA.

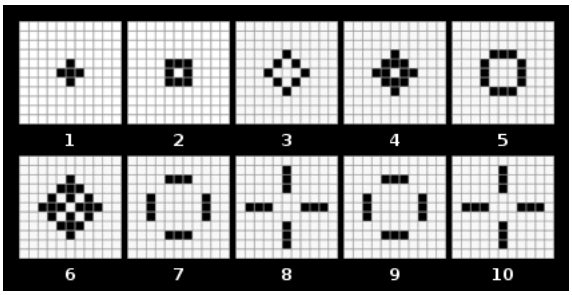
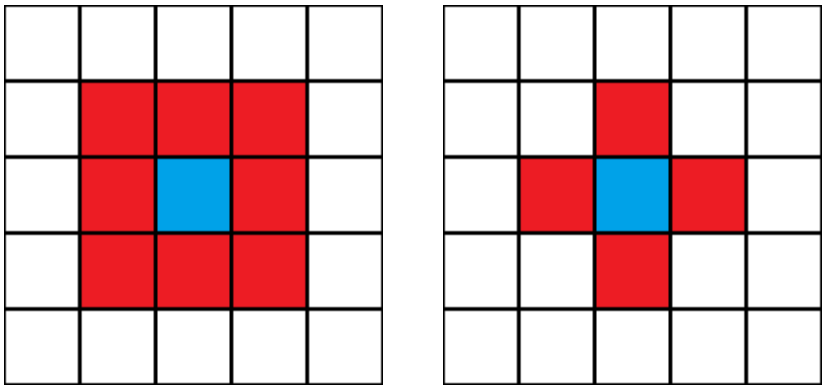


Figure 1.2: Ten consecutive generations of a certain CA. Source:
`http://wp.culturacolectiva.com/el-juego-de-la-vida-entre-el-caos-y-el-orden/`

The process of change of state is totally determined by the neighborhood of each cell and the set of rules of the particular CA. The neighborhood of a cell consists of all adjacent cells including the cell itself. The two most common types of neighborhoods are:

- Moore neighborhood
- Von Neumann neighborhood

The Moore neighborhood (Figure 1.3a) consists of all eight adjacent cells and the cell itself. The Von Neumann neighborhood (Figure 1.3b) consists of only four adjacent cells (the upper, the right, the left and the lower one) and the cell itself.



(a) Moore-neighboorhood (b) Von-Neumann-neighborhood

Figure 1.3: The two most common neighborhoods. Source: `https://en.wikipedia.org/wiki/Cellular_automaton`

However, there are more neighborhoods than only these two. These neighborhoods can be extended. For instance an extended Moore neighborhood consists of all the cells within two rings around the respective cell, this is called a radius of two. It is also possible to only have the corners of the Moore-neighborhood as neighborhood. The developer of the CA can use any neighborhood he needs.

Every iteration the algorithm makes each cell look to its whole neighborhood, to be more precisely, to the

states of all the cells of its neighborhood. The set of rules determines how every cell should change according to the states of that cell's neighborhood. This process of changing states happens every iteration for all cells simultaneously. A certain rule could for instance be:

"All the cells of state 1, with upper cell state 0 and lower cell state 0 become zero."

This is just only one rule, where every CA is said to have a complete ruleset. A ruleset must cover all the possibilities of configurations of the neighborhood. An example of a ruleset is:

"A white cell stays white in the next generation."

"A black cell with exactly one black neighbor stays black in the next generation."

"Any other black cell becomes white in the next generation."

Every possible configuration of a neighborhood of a cell is covered in this ruleset. In Figure 1.4 this ruleset is applied to a certain starting configuration for two iterations. All the following generations will be the same as the last one.

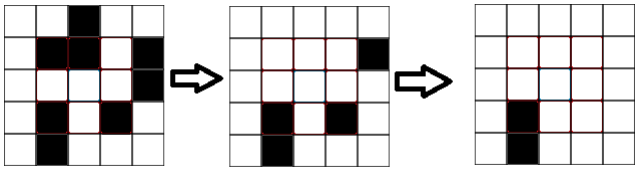


Figure 1.4: Three consecutive generations of a CA.

1.2 Elementary CA

For solving the puzzles, only 2-dimensional CAs will be used. But there are CAs that are much more simple, called elementary cellular automata. An elementary CA is a one-dimensional CA, in which the cells can only have one out of two states, zero or one. The neighborhood of each cell in an elementary CA consists of only the cell itself and its two adjacent cells.

Since every cell can only have state 0 or 1, there are $2 * 2 * 2 = 2^3 = 8$ possible configurations for the neighborhood of a cell. Each of these configurations must be specified as a rule in the CA, so it must have a resulting state, giving again two options: 0 or 1. This makes $2^8 = 256$ possible combinations of rules and thus 256 different CAs. Each of the eight configurations can be written as its states of the neighborhood forming a binary number. These are all of the sequence $111, 110, \ldots, 001, 000$. Stephen Wolfram proposed the Wolfram Code [Wol02] to give all the 256 possible rulesets a number from 0 to 255. These are logically created by putting all resulting states of the configuration sequence mentioned above in that same order next to each other. This will give a binary number from 0 to 255 that uniquely specifies all the elementary CAs, for instance the following ruleset:

111	110	101	100	011	010	001	000	current pattern
0	0	0	1	1	1	1	0	new state

This is rule 30, because the row before 'new state' is the binary number 30. A simple elementary CA which is only one-dimensional, has only three cells in its neighborhood and has two possible states, has 256 different possible rulesets. Imagine how enormously many rulesets a two-dimensional CA with nine cells in the neighborhood and ten possible states has. That is 10^9 possible configurations for the neighborhood of one cell. That makes 10^{10^9} possible rulesets and thus 10^{10^9} many different CAs.

1.3 Translating puzzles into CAs

As described before, cellular automata are models that can be used to simulate systems. This simulation can be used to predict the future of a certain system. This is usually the case when not all variables of the system are completely known or when the system is too inconsistent and thus too unpredictable to be totally sure of its outcome. These kinds of models are often used to describe or predict systems in nature. But when the beginning state and the behavior of a system are completely known, a CA can actually calculate its outcome. This outcome is then the exact answer to the issue. For applying such a model on a system, one could easily think of logical puzzles to be that system. That is what this thesis is all about: can cellular automata be used to solve puzzles? The term puzzle in this question is a wide-known term that must be specified further. It is a hard task to prove that some puzzles are not solvable by CAs, while it is much more easier to prove that some puzzles are solvable by a CA. In that case there must only be given a CA which solves the puzzle. So we assume that some puzzles are not solvable and we try to find the ones that are solvable by cellular automata. Since cellular automata cannot recognize natural language in any way, we state that all language puzzles, like crossword, are not solvable by cellular automata on their own. It could be that cellular automata can be used for solving them, but they will always need a dictionary or some other aid. The wordsearch puzzle, Figure 1.5, seems a natural language puzzle too.

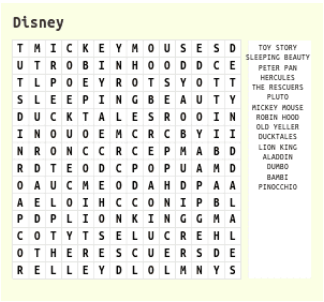
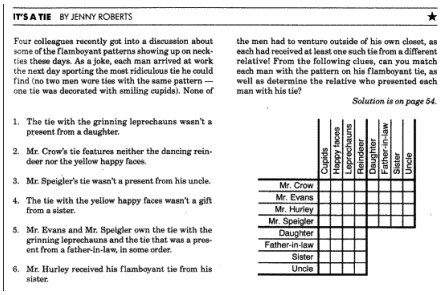


Figure 1.5: The language puzzle word search, for which is assumed that there is not a general CA which can solve all instances. Source: `http://www.kittybabylove.com/disney-word-search/`

Yet, it is kind of logical since the words are already given and only need to be found. Still, the words need to be loaded into the CA somehow and when we absorb this into the ruleset, a new CA must be built for every instance of this puzzle. This means there is not a general CA which can solve all instances of wordsearch. So that leaves us with the logical puzzles. Because cellular automata can only compute and do not have feeling

for natural language or insight, the puzzles need to be completely solvable by applying logical rules. Secondly, the puzzle needs to consist of a grid of cells, so that every cell of the puzzle can be linked to a cell of the CA and look to its neighboring cells. Thus, we assume that puzzles that do no consist of a grid are not solvable by any cellular automaton. For the puzzles that consist of a grid it is necessary that all the information that is needed to solve it is in the grid itself. The only way a CA can read information is by the states of its cells. These states must contain all the information. If there is necessary information outside the grid, a cellular automaton cannot solve the puzzle by itself. Figure 1.6 explains this.



IT'S A TIE BY JENNY ROBERTS ★

Four colleagues recently got into a discussion about some of the flamboyant patterns showing up on neckties these days. As a joke, each man arrived at work the next day sporting the most ridiculous tie he could find (no two men wore ties with the same pattern — one tie was decorated with smiling cupids). None of the men had to venture outside of his own closet, as each had received at least one such tie from a different relative! From the following clues, can you match each man with the pattern on his flamboyant tie, as well as determine the relative who presented each man with his tie?

Solution is on page 54.

1. The tie with the grinning leprechauns wasn't a present from a daughter.
2. Mr. Crow's tie features neither the dancing reindeer nor the yellow happy faces.
3. Mr. Speigler's tie wasn't a present from his uncle.
4. The tie with the yellow happy faces wasn't a gift from a sister.
5. Mr. Evans and Mr. Speigler own the tie with the grinning leprechauns and the tie that was a present from a father-in-law, in some order.
6. Mr. Hurley received his flamboyant tie from his sister.

	Cupids	Happy faces	Leprechauns	Reindeer	Daughter	Father-in-law	Sister	Uncle
Mr. Crow								
Mr. Evans								
Mr. Hurley								
Mr. Speigler								
Daughter								
Father-in-law								
Sister								
Uncle								

Figure 1.6: This logic puzzle consists of a grid, but on its own a cellular automaton cannot solve it, because it needs information outside the grid. Source: `https://imagemag.ru/img-ba_hardest-logic-problems.html`

In these cases that information must be translated somehow into the grid to be solvable by any cellular automaton.

There are some special cases, like the Rubik's cube, see Figure 1.7. This is a puzzle that is logical, consists of a grid with cells and has no information outside the grid. Yet, a cellular automaton is not very useful when solving a Rubik's cube. Cellular automata can compute answers to puzzles, but the problem with a Rubik's cube is not the answer. The answer is known; get the cube with all sides consisting of one color. The problem here is how to get that solution. A cellular automaton cannot communicate that roadmap to a user, if it is capable of solving the cube.

Figure 1.7: Rubik's cube. Source:
`https://www.legebyen.dk/shop/kategori/legetoj/indlaering/original-rubiks-terning-3x3-1-stk.html`

This leaves us much less puzzles to study, but still there are many puzzles and games that we can do research on. It is clear this cannot be done for all puzzles, but this thesis will pick a few and compare the cellular automata that are used to solve them.

1.4 Thesis overview

This thesis is about cellular automata and how to solve puzzles using them. This first chapter contains the introduction; Chapter 2 discusses related work; Chapter 3 discusses the CAs built for solving Mazes, Nonograms and Binary puzzles; Chapter 4 evaluates these CAs; Chapter 5 speculates about how other puzzles can be solved using the same techniques; Chapter 6 concludes.

This bachelor thesis has been developed in assignment of LIACS, an institute of University Leiden and is supervised by Hendrik Jan Hoogeboom & Walter Kosters.

Chapter 2

Related Work

Cellular automata are models that can be used in a wide area of different aspects of science. Therefore, there has been done research on cellular automata since they were discovered in the 1940's by Stanislaw Ulam and John von Neumann. Ulam and Von Neumann originally tried to provide insight into the logical requirements for machine self-replication

2.1 Conway's Game of Life

In 1970 Martin Gardner published an article about John Conway's Game of Life in 'Scientific American' [Gar70]. This game is just a cellular automaton with only two states; alive (black cell) and dead (white cell). Game of Life has a Moore-neighborhood and has the following rules:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

This particular cellular automaton became so popular because it was kind of recreational. Conway used his simulation model to do research on patterns. For instance he wanted to know if there are patterns which have a finite number of cells and can grow into patterns with an infinite number of cells. An example of such a pattern is the glidergun in Figure 2.1.

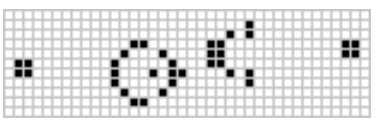


Figure 2.1: The glidergun is a repeating sequence of configurations that Game of Life can execute. Source: `http://www.conwaylife.com/w/index.php?title=Gosper_glider_gun`

Game of Life can be seen as a zero-player game that only needs an initial state together with its own ruleset. So, besides being a tool for solving puzzles, cellular automata are puzzles themselves.

2.2 Minesweeper

Minesweeper, see Figure 2.2, is a one-player game which is played on a computer. In this game there is a grid in which all cells are behind a hidden layer. Every cell is a mine or not, and the total amount of mines in the field is given. The cells that are not a mine show how many mines all their neighbors have together. The player's goal is to mark all the cells that are not mines by clicking them and find all those that are. The player can mark cells which he thinks are mines by putting a red flag on them. This all sounds like cellular automata

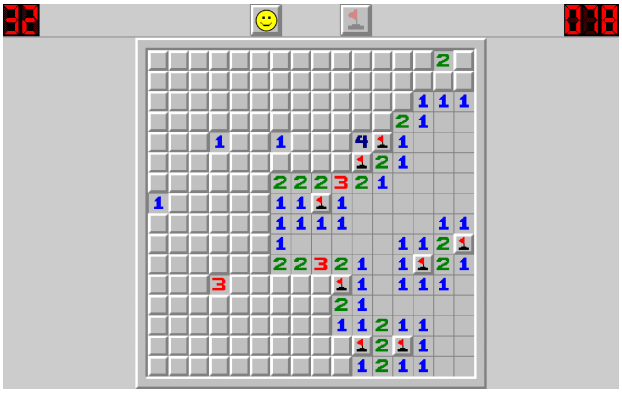


Figure 2.2: A game of Minesweeper that is not yet finished. Source: `http://www.moregameslike.com/minesweeper/`

and Minesweeper can be seen as a puzzle. It is a logical puzzle, which is a requirement of puzzles to be solved by a CA, explained in Section 1.3. It also consists of a grid of cells, that can have different states, namely hidden, flag, mine, number and some combinations of those. However, not all the instances of Minesweeper are totally computable. In some instances there occurs the situation in which none of the cells can be opened with certainty, so the user must take a guess. We will only look at those puzzles that are totally computable. Andrew Adamatzky wrote an article about how a cellular automaton plays Minesweeper [Ada97]. In this article is cellular automaton is presented that can solve every Minesweeper puzzle that is totally computable. However, Minesweeper begins in a state in which the user is forced to make one or more guesses. In this state, Minesweeper is not totally computable, although in Microsoft's version of Minesweeper the puzzle is generated after the first guess, therefore this first guess can never be a mine. So those guesses are already made before giving the puzzle to the cellular automaton. This way, the Minesweeper puzzle always starts in a totally computable state. The given cellular automaton has a neighborhood of 25 cells (extended Moore neighborhood with radius two). The state of a cell indicates if the cell is a mine, it is not a mine or it is still unknown. Therefore, the cell state will be a direct product $Q = \{\cdot, \#, \circ\} \times \{0, 1, \ldots, 8\}$ where "$x^t =' \#'$" means "it became known at time t that cell x contains a mine"; "$x^t =' \circ'$" means "at time t cell x becomes open"; "$x^t =' \cdot'$" means "cell x is closed at time t" and the number between 0 and 8 indicates the amount of mines directly around the cell x. This means that every cell could be at any time t in one out of $3 \times 9 = 27$ states, although a cell can only show its amount of neighboring mines if it is opened.

2.3 Maze

The classical game of the maze, in which a path must be found to get from the entrance to the exit, seemed likely to be solvable by a cellular automaton to us. It does indeed consist of a grid with cells and has only two states; wall and path. Although we could not find any articles published about this subject, Wolfram Demonstrations Project [Var12] shows a demonstration of how a cellular automaton could solve a maze, by shortening all the dead-ends with one cell every iteration, until only the main path is visible. This method is only applicable to Mazes without cycles, because a cycle has no dead-end. In Section 3.1 this method will be explained more widely together with another method.

Chapter 3

Algorithms

As described in Section 1.3 not all puzzles are suited to be solved by cellular automata. We tried to pick a few puzzles of which we thought they would be, which are the Maze, Nonogram and Binary puzzle. These puzzles we did research on and subsequently we tried to program them including the cellular automata that can solve it in C++. Since a CA is just a model and not something physical, it is good to know that we did not create a real cellular automaton for any of the puzzles. What we did do is make an algorithm that performs as a cellular automaton. If we can make this, then there must be a cellular automaton that represents this imitation and thus can solve that specific puzzle.

3.1 Maze

The Maze is a game in which there is a grid with paths (white cells) and walls (black cells), see Section 2.3. The outer border has two gaps; the entrance and the exit. The player's task is to find a path from entrance to exit but he is only able to walk over the paths and not through the walls. This right path is called the main path. There are many fake paths (which are not part of the main path) which lead to a dead-end, implemented to make the puzzle harder. Figure 3.1 shows an example of a Maze.

Since a Maze consists of a grid of cells that can only have two states (black or white) in the starting configuration, it seems to be solvable by a cellular automaton.

We decided to create two different cellular automata that can solve a Maze. One is like the approach of Wolfram Demonstrations Project [Var12], while the other cellular automaton works completely different. Although they work differently, they both are able to solve a Maze. The Mazes we focus on cannot have cycles in them.

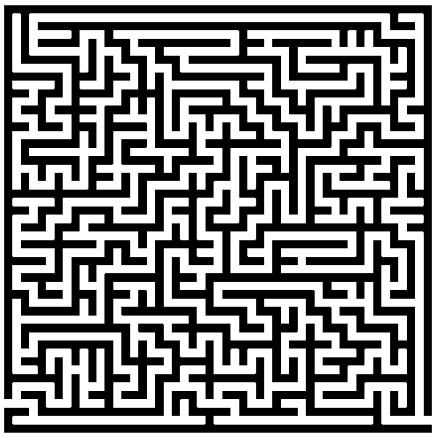


Figure 3.1: A Maze where the entrance is in the down-left corner and the exit in the down-right corner. Source: `https://nl.mathworks.com/matlabcentral/fileexchange/46072-a-solution-to-the-maze-problem-with-dijkstra?focused=3815895&tab=function`

3.1.1 First Maze solver

In order to create an algorithm that computes like a cellular automaton, we first must create the puzzle we want the cellular automaton to solve. Most of the Mazes are completely filled with paths, as in Figure 3.1, the paths reach all of the grid. In other words, besides the paths, also the walls cannot form squares of four cells. If a path can make such a square, that would mean that there are multiple (shortest) paths from entrance to exit, since one could 'walk' the square both clockwise and counterclockwise. We are not looking for the shortest path specifically, but we are looking for any path in our algorithms. So we create only mazes in which the squares described above can never occur. However, a square of walls can appear, since this only wastes some space in our grid and thus leaves less fake paths, it still holds the restriction of having only one path from entrance to exit. Unlike Figure 3.1, the Mazes in our algorithm can have wide black areas. Figure 3.2 shows an example of how a Maze may look like in our algorithm.

Figure 3.2: A random initialized Maze by the algorithm with entrance (yellow B), exit (yellow E) and path (red X's).

The key principle of this cellular automaton is that it can easily detect fake paths and shorten them cell by

cell. It relies on the fact that a fake path will eventually stop and thus find walls on each side, except the side where the path came from. This means that the last cell of a fake path, will always have three neighbors that are walls and one neighbor that is path. This is not the case with the main path, since every cell that is part of the main path must have a minimal of two neighboring cells that are also part of the main path, otherwise this would not be a path from entrance to exit. The entrance and exit cells themselves are out of the picture here. Therefore, this cellular automaton uses a Von-Neumann neighborhood. The only thing it does is search for the end of the fake paths every iteration, by looking for cells with exactly three neighboring walls and one neighboring path. Therefore, there is only one rule in the ruleset:

"If a cell is white and it has exactly one white neighbor, the cell becomes black"

The only other rules make the other cells remain in the same state. Every cell that holds to this restriction will turn into a wall the next iteration. This way, the cellular automaton shortens each fake path by one cell every iteration. Therefore, this algorithm will last exactly as many iterations as the length of the longest fake path. This fact makes this algorithm fast, but it also brings some trouble: the cellular automaton removes fake paths and in the end only the main path remains. So if the cellular automaton is done, the original Maze is gone and one can only see the main path. This problem is refuted in Section 4.1. After the cellular automaton solved the Maze of Figure 3.2, the Maze will look as in Figure 3.3.

Figure 3.3: The result of the Maze of Figure 3.2 after the cellular automaton solved it.

The cellular automaton did ten iterations, since the longest fake path has ten cells. This is the fake path that starts in the middle in Figure 3.2 and goes from there to the right followed by a path upwards. This cellular automaton starts with only two states and never uses any other states. The only states that occur are path and wall.

3.1.2 Second Maze solver

This second cellular automaton is built in order to keep the original Maze intact, unlike the cellular automaton described in Subsection 3.1.1. The key principle of this cellular automaton is that "it walks" the whole Maze

and marks which paths are fake and thus concludes which path is the main one. This cellular automaton has a Von-Neumann neighborhood. The Mazes are equally created as described in Subsection 3.1.1. The only difference is that the first cell that is path after the entrance is marked with a blue 9, because it is in state 9. This means that the second Maze solver uses more than two states in the starting configuration, contrarily to the first Maze solver. An example of such a Maze before it is given to a CA is in Figure 3.4

Figure 3.4: A random initialized Maze by the algorithm with entrance (yellow B), exit (yellow E) and path (red X's).

During the iterations the second Maze solver uses even more states than only the starting three. Wherein Subsection 3.1.1 there were only two states, this Maze solver uses eight possible states. From now on, we give each state a symbol. A wall means state 0 (imaged as black) and a path means state 'X' (imaged as red X). The first thing the cellular automaton does, is marking each cell that is a cross-road with another state. These are all cells that are in state 'X' (it must be a path) and have three or four neighbors that are also in state 'X'. That means it has more than two outgoing paths, so it has to be a cross-road. These cross-roads are marked as state 'T'. Notice that these cross-roads are searched for every iteration, but they will only be found all at once in the first iteration. After the first iteration the Maze of Figure 3.4 will look as in Figure 3.5.

Figure 3.5: A random initialized Maze by the algorithm with entrance (yellow B), exit (yellow E), path (red Xs) and cross-roads (Green Ts).

Besides all the Ts shown in Figure 3.5, also the first cell that is path after state '9' is marked with state '2' (imaged as blue 0). The automaton starts walking all paths from the entrance to exit. The associated rule that occurs here is: "If the current cell is in state 'X' and it has one neighbor in state '2' or in state '9', the current cell will be in state '2' in the next generation.". This will make the blue line of zeros and one nine walk over

the main path until it finds a cross-road, like in Figure 3.6, although this is not the first cross-road found.

Figure 3.6: In the middle of the algorithm. The blue path finds another cross-road.

When this happens, there is a cell which is in state '4', and has a neighbor in state '2', state '9' or state '5' (imaged as blue 5). This cell becomes state '5'. By doing this, the cellular automaton marks the cross-road also as path. In the next iteration, all the outgoing paths of this cross-road will be walked by the rule; "If the current cell is in state 1 and it has one neighbor in state '5', it will become in state '2'.". Now we know for sure that at least one of those paths after the cross-road is fake. The walker on this fake path will expand by one cell every iteration and can even find new cross-roads, until it becomes a dead-end. When this is the case the following rule will do the trick; "If the current cell is in state '2' and it has three neighbors in state '0', the current cell will become in state '-'.". State '-' (imaged as a purple -) will lead back to the last cross-road making all the cells that are in state '2' on the road also in state '-'. When it reaches the cross-road, nothing has to happen. The cross-road will turn into state '-' as well when all of its outgoing, minus one, paths are in state '-'. This means this cross-road does not belong to the main path, so it will pass the signal to its only path that is not yet in state '-'. When a cell in state '5' has a neighbor in state '-' but it will always keep two neighbors in state '2', this cell must be part of the main path. This process will go on until the blue path reaches its destination. The last cell, the one to the left of the exit, is manually put in state '8' (imaged as state 1). When this cell has a neighbor that is in state '2', the automaton has reached its destination, but it only needs to make sure that every fake path is marked with -'s. When these are both the case, the algorithm stops and the Maze of Figure 3.2 will look as Figure 3.7.

Figure 3.7: The cellular automaton has finished the Maze of Figure 3.4.

3.2 Nonogram

The Nonogram (also known as Japanese puzzle), see Figure 3.8, is a puzzle in which the right cells must be marked black. It is a logical puzzle that consists of a grid and therefore the connection is made to cellular automata.

3.2.1 Nonograms in general

Outside the grid, before each row and each column there is a certain amount of numbers, which is called the description. A single description always holds for only one row or column. The amount of numbers in that description determines the amount of black strings in that row or column, in which the length of each string of black consecutive cells matches the number itself. Multiple strings within a single row or column must always be separated by a minimum of one white cell. The order of multiple numbers within one description matters, the number that is mentioned first must also occur before the others in the puzzle. When the puzzle is finished mostly a drawing or simple shape will form.

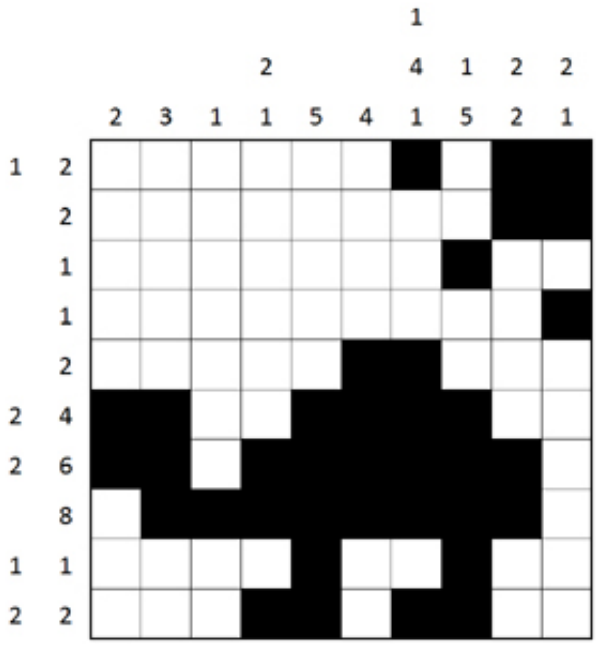


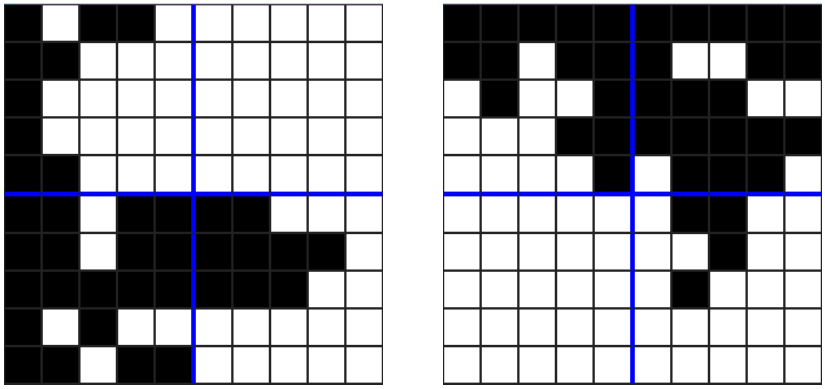
Figure 3.8: This Nonogram is finished and the black cells represent a camel in the sun. Source: `http://curiouscheetah.com/Museum/Puzzle/Nonograms`

For instance in Figure 3.8 above the first column is a single 2. In that column there is one black string of 2 cells, that is a match. However, the seventh column is different. Above that column are three numbers; respectively 1, 4 and 1. In that column there are three black strings with lengths in that same order, all divided by one or more white cells. If the white cells did not divide them, it would have been one big string of 6 black cells, which would not match the description above the column. A Nonogram starts with a completely empty grid, only the descriptions of the rows and columns are there at the start. The goal is to fill in all black cells so that the descriptions for both all the cells and all the columns hold. A uniquely solvable Nonogram means

that there is only one unique answer to the puzzle. Since cellular automata can only solve completely logical questions, we now only use uniquely solvable Nonograms to do research on solving Nonograms with cellular automata.

3.2.2 Global working of CA

A big problem for solving Nonograms by cellular automata is that at the start of the puzzle, the grid is completely empty. All the information that is needed to solve the puzzle, is actually outside the grid. With no information a cellular automaton cannot do anything. At this stage all the cells are in exactly the same state, which makes it impossible to make rules that will do something useful. If we still want to solve this puzzle, we need to make the restriction stated in Section 1.3 that a puzzle must always have the information inside the grid hold, by simply putting the information inside the grid. We have found a way to do that. In this way, each row and each column must be filled beforehand as the description says. For each row and column, this is possible in many ways, which is exactly the point of this puzzle. So they must be filled in a certain way that works for all Nonograms. Every black string, both horizontal and vertical, is pushed along to the front as much as possible. So the rows are filled as most to the left as possible and the columns are filled as most upwards as possible. This configuration will be in (almost) all the cases not the solution. There will always be some cells that are black according to their row and white according to their column or vice versa, like the cell on row 1 and column 2 in Figure 3.9. To solve this problem, we stop seeing this Nonogram as one 2-dimensional grid, but instead start looking at it as two grids; one vertical grid and one horizontal grid. There is still only one grid, but the CA will look at it with two different views. These two views can both be filled as described above simultaneously. Figure 3.9 shows how these grids will look.



(a) Horizontally pushed to the front (b) Vertically pushed to the front

Figure 3.9: The Nonogram of Figure 3.8 pushed to the front in two distinct grids.

For a certain cell, its information is in its own row and its own column. Therefore, a cell might want to look across its whole row and whole column. The ideal case would be, to have a Von-Neumann neighborhood wherein the radius is the length of the columns and rows. In that case, each cell could see precisely its whole row and whole column, and with all that information it will be almost always able to compute its destiny. To realize this, the neighborhood must be dynamic, because every cell will be somewhere else in the grid, but they all have to see their whole row and column. To do so, every cellular automaton that will be built, must

know its own size, so that it can anticipate to that size. If every CA that we will built must know its own size, the CA's that we will build for the different sizes of Nonograms will all be different. The thing we want to achieve is to build one cellular automaton that can solve all Nonograms, no matter their size. This makes the solution of using dynamic neighborhoods useless.

The key principle of the CA solving Nonograms has a lot to do with the two distinct grids from Figure 3.9. Since every cell is only dependent on its own row and column, we will treat each row and each column as a Turing machine which has its own operations and has nothing to do with the other rows and columns. The idea here is that each row and column acts as a single one-dimensional CA. We let each row and column compute all the cells that are definitively black or white. These rows and columns will help each other although they do still not work together to determine cells. When a certain row fills in a black cell because it is sure, the respective column may assume that this is correct and can sometimes compute other cells using this information. If we can repeat this process for all rows and columns we will eventually solve many Nonograms. When using only information concerning single row and columns, puzzles can often be solved partially, but not fully [KB09]. The instances of Nonogram which can be solved with this CA are called 'simple'.

The hard part is to make each row and column compute all cells that have only one option according to its description and its already determined cells in that row or column. From now on, we will call such an already determined cell a definite cell. There are many configurations in which a single row or column can determine definite cells in itself, for instance when a row has only a zero as description, Figure 3.10. All the cells in this row are definitely white.

Figure 3.10: Empty horizontal row

The same holds when a row has its length in the description, which makes it full. But this is not it. One row can detect all its cells as definite when there is only one possible configuration for it. Besides Figure 3.10 this is also the case when there are multiple numbers in the description which make the row full, see Figure 3.11.

Figure 3.11: Full horizontal row of length ten with multiple numbers

It could also be the case that not the whole row or column is definite, but just a few cells, like in Figure 3.12. The length of the row is ten and the description has one string of length nine. There are only two possible configurations for this row:

- The first nine cells are black and the last one is white.
- The first cell is white and the last nine cells are black.

In both cases, the eight middle cells are black. The only uncertainty is whether the leftmost or the rightmost cell must be black.

Figure 3.12: Horizontal row of length ten with description 9

In the previous examples there were no cells determined on beforehand when trying to determine cells. Already determined cells can say a lot about a row or column and therefore help determining other cells. In Figure 3.13 the description of the row is 4. The second cell is pre-determined as black, which gives us together with the description much information about the remaining part of the row. The third and fourth cell can also be determined black, while the last 5 cells can be determined white.

Figure 3.13: Row with description 4. The gray cell was pre-determined. With this information the black cells can be determined while the red cells can be determined to be white. The white cells remain unknown.

We want the cellular automaton to be able to detect all these kinds of configurations and find the definite cells in them. In order to find all the definite cells in a row or column by looking only at that row or column itself, we can find all possible configurations in that row or column. If in all the possible configurations a certain cell has always the same value, we are 100% sure that this cell will have this value in the solution of the Nonogram [KB09]. So, with this knowledge the only thing we need to do is pushing all our rows and columns of the two grids in such a way, that we construct all possible configurations. Then we can check for each configuration if it is valid with all the definite cells. If it is not, we push to the next configuration. If it is valid, we must save for each cell its value somewhere. If a cell gets both the values black and white during the pushing, this cell will not be definite afterwards. If the cell only gets one value during the pushing, the cell becomes definite with that value. This information will help the future iterations.

3.2.3 Implementation of the algorithm

As described in Subsection 3.2.2, we will use each row and column as a 1-dimensional CA that only looks at itself. Since a cellular automaton does not have the ability to save information in some memory, we must make good use of the abilities it does have. All the information we want to save, we must store in the states of the cells. In Subsection 3.2.2 we gave a solution to the problem that the starting configuration is empty. We can only implement the two grids, horizontal and vertical, by making the horizontal grid and vertical grid part of the state of the cells.

Since the states in our algorithm must keep all the information, we chose to make all the variables we use property of '*State*', so that the state of a cell contains all the information. Where in Section 3.1 the state was just a single number or color, it gets much complex with Nonograms. This way, *State* is not a number anymore, it defines the values of many attributes. The most important attribute of *State*, is the integer 'sure'. This integer can only have three values:

- 0; the automaton is absolutely sure this cell is white (imaged as black).

- 1; the automaton is absolutely sure this cell is black (imaged as white 1).
- 2; the automaton is not yet sure whether the cell must be white or black (imaged as blue 2).

Therefore, in the starting configuration of this automaton, all cells will have the value $sure = 2$, since none of the cells will be definite at this stage, see Figure 3.14.

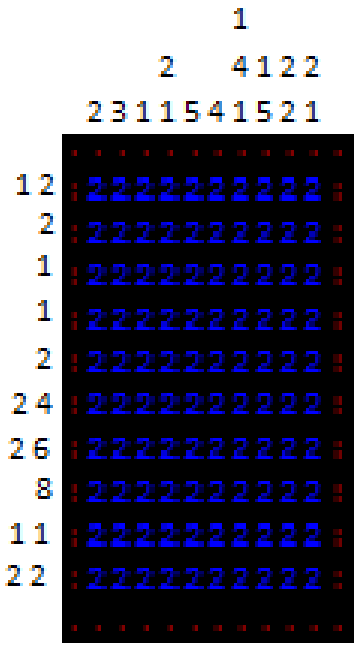


Figure 3.14: Beginning configuration of the cellular automaton showing only the attribute *sure* of its class *State*.

In Figure 3.14 the grid is outlined with red dots. These red dots are also cells of the grid, but their Boolean attribute '*border*' is set to true, where for all other cells this attribute is set to false. The algorithm makes sure that it will never happen that a cell's attribute *border* will be switched. The remaining part of this section will keep giving examples on the 10×10 cellular automaton of Figure 3.14.

The algorithm generating random Nonograms can only construct descriptions with a maximum of three numbers. It gives a random amount of random numbers, with the restriction that the line has a maximum of 10. Although every row and column is possible, there is no restriction on the combination of them. This means that most of the descriptions that are constructed can never lead to a solution. The only thing we need is a cellular automaton that gives a solution for those descriptions that are consistent. However, for testing single lines, this algorithm works pretty good. To actually solve Nonograms a consistent description should be implemented manually.

Before the automaton starts iterating, the values of all attributes of all cells must be set to an initial value. The initial value for *sure* is 2. To get to the starting configuration with the two grids, we introduce the Boolean attributes '*vgrid*' and '*hgrid*' of *State*. If a cell is black in Figure 3.9b, then its value of *vgrid* must be 1. Also, if a cell is white in Figure 3.9a, then its value of *hgrid* must be 0. The algorithms also puts the border in a state where the attribute *border* is true. All the other attributes get a logical initialization.

Since there are enormously many states (because of the many attributes in the class *State*), the ruleset which is

used every iteration is also big. The iteration is done in the function *iterate*(). In *iterate*(), the state of each cell is updated. A cell is specified by two integers i and j, in which i specifies the row and j specifies the column. The function *iterate*() does for all cells in the grid the following. Firstly, all the old attributes of the state of the cells currentGen[i][j] are copied to nextGen[i][j]. After this is done all the rules are applied to the cells, which changes some attributes of their state. After the rules are applied to every cell, all the attributes in nextGen[i][j]] are copied back into currentGen[i][j] for all i and j concurrently. The cells nextGen[i][j] are used in between so all the cells will be updated all at once.

What happens in applying the rules is what makes a cellular automaton unique. The first thing this cellular automaton does, is checking whether the cell is a border. If it is, it will stay that way and no other rules will be applied to this cell.

The rules of the automaton are going to take care of the rest that needs to be done. That is, in each row and column pushing the black strings in every possible position according to its description. For every order, check if it satisfies all its *sure* values. If the row or column is not in a valid configuration, push to the next line. If it is in a valid configuration, update the attributes *zeroseen* and *oneseen*, which keep track of the previous values in the *h/vgrid*. If *zeroseen* is true, it means that in at least one of its possible configurations this cell has been white. False means that this cell has not been white in any configuration. Note that this only keeps track of the *vgrid*, *hgrid* has its own *zeroseen* and *oneseen*. Algorithm 1 shows what happens for each row and column concurrently in pseudo-code.

```
while Nonogram not finished do
    while Not all configurations done do
        for each cell in row/column do
            if configuration is valid then
                Update zeroseen and oneseen
            end
        end
        Go to next configuration
    end
    if cell has zeroseen value 0 then
        Put cell's sure value to 1
    end
    else
        if cell has oneseen value 0 then
            Put cell's sure value to 0
        end
    end
    Clear zeroseen and oneseen;
    Go to first configuration
end
```

Algorithm 1: Algorithm for each cell and column

The hardest part in this algorithm is to create all the configurations for a row or column. This is done by pushing the black string in all possible ways, while holding the constraints of a Nonogram, such as having a minimum of one white cell between multiple black strings, for instance consider Figure 3.12. In the starting configuration the first nine cells are black and the last one is white. Of that first nine cells, *zeroseen* will be 0 and *oneseen* will be 1. Of the last cell it will be the opposite. The algorithm will now push the whole black

string one position to the right. The first cell will now also have $zeroseen = 1$, while it still has $oneseen = 1$. The same is valid for the last cell. The eight middle cells will only have $oneseen = 1$, while their *zeroseen* will be 0. Now, these middle eight cells will get $sure = 1$, while the outer two cells will stay in $sure = 2$. After this, values will be cleared and the string will be pushed back. This process will repeat until one of the two outer cells gets definite because of its column. Pushing rows with only one black string is easy; just keep pushing the string one cell to the right every iteration. When there are multiple numbers in the description, it gets harder to find all the configurations. Figure 3.15 shows ten times the same row, each time another configuration of the description $3 - 1 - 2$. The order of configurations shown here is the same order as how the automaton would do it.

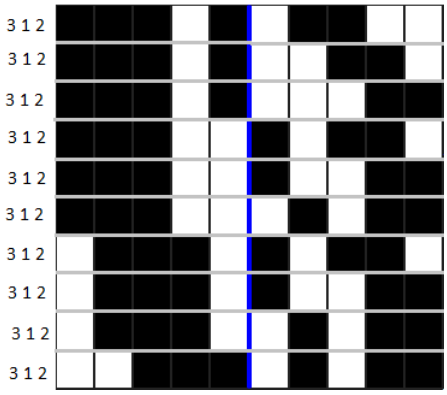


Figure 3.15: All 10 configurations of the description $3 - 1 - 2$.

All these configurations are found in lexicographical order by brute force. Every time the last possible black string is pushed one cell to the right, until it reaches the border. When this happens, the black string is pushed back until it reaches the previous black string. From here on, all the black strings where in between are not more than one white cell are pushed together one cell to the right. This process of finding all possible configurations is being repeated endlessly by the CA. Notice that this process is triggered for both the rows and column concurrently, as all of Algorithm 1. The rows and column do not show any signs of cooperation during this process. When a row or column determines a new cell as definite, there will always be a another row or column which receives extra information. Since this information is always right, it will not yield any problems for the other row or column, even if it receives the information in the middle of its own process.

The pushing of the rows, is partially executed by the integer *backtracker*, which is an attribute of *State*. This attribute can be seen as a walker, that is created at the bottom/left cell of a column/row. When this is created another attribute, *firedWalker* is set to true, which makes sure that in the next generation there will not exist a second walker. Notice that there is no such thing as a walker which can walk across a cellular automaton, but with the right rules this attribute can imitate one. For instance, when this *backtracker* is walking upwards in a column, the following rules must be set in the ruleset:

- If lower cell has *backtracker* value upwards (for instance 3), the cell itself gets *backtracker* value 3.
- If cell has *backtracker* value 3, cell gets *backtracker* value 0 (This is the initial *backtracker* value and means 'off').

The same way a walker can walk down, but with another value, for instance 2. The algorithm uses multiple walkers, all with multiple possible values. The same way there exists the attribute *walker*, which takes care of setting the seen values after each configuration. Also, *walker*2 sets the *sure* values according to the seen values after all configurations have been passed. Notice that for almost all attributes there are two, like there exist *vwalker*, *hwalker*, *vwalker*2 and *hwalker*2. When speaking of *walker* in general it is about both, because they do exactly the same, for vertical and horizontal views respectively.

3.3 Binary puzzle

The Binary puzzle (also known as Takuzu), see Figure 3.16, contains only zeros and ones when it is completed. The puzzle consists of a grid of cells and is completely logical, therefore the connection is made to cellular automata.

3.3.1 Binary puzzles in general

At the start, a certain amount of binary numbers is already filled in. The task is to fill in all the blanks, like in the better-known puzzle Sudoku. Contrarily, only zeros and ones can fill the blanks. While doing this, the following rules must be followed:

- There can never be three consecutive cells (horizontally and vertically) with the same number.
- Each row and each column must consist of $length/2$ zeros and $length/2$ ones. Here, *length* is the length of the row or column itself.
- Each row is unique and each column is unique.

These rules are now referred to as Rule 1, Rule 2 and Rule 3, respectively.

							1		
	0	0			0			1	
	0			1			0		0
		1				1			
1		1							1
							1		
	0			1				0	
				1	1				0
	0		0			1			0
0				0				1	

Figure 3.16: Starting configuration of a 10×10 binary puzzle. Source: `http://www.sudokunet.nl/binairepuzzels.php`

We propose five methods to calculate the values of cells. The first one, "Find pairs", comes from the first rule mentioned above, see Figure 3.17. In this row there are already two consecutive zeros, so to prevent the appearance of three consecutive zeros, there must be a one on both sides of these zeros. We refer to this method as "Method 1".

1	0	0	1		0			1	

Figure 3.17: Method 1 is applied in red to the second row of the puzzle of Figure 3.16.

"Prevent trios" is the second method to calculate cells. Find three consecutive cells, in which the outer ones are the same and the middle one is unknown yet. Fill the middle one with the value that differ from that of the outer ones to prevent a trio, see Figure 3.18. We refer to this method as "Method 2".

1	0	1							1

Figure 3.18: Method 2 is applied by filling in the red zero.

"Complete rows and columns" is the third method. Since every row and column must always contain an equal number of zeros and ones, cells can be determined when a row or column has reached its maximum amount of zeros or ones. This is the case when one of the two possible binary numbers is filled in half of the cells of a row or column. Figure 3.19 shows this process. The column has three ones and is of length six, so the other two numbers must be zero. We refer to this method as "Method 3".

		1			
		0			
1	0	1	0	0	1
		0			
		0			
		1			

Figure 3.19: Method 3 is applied by filling in the red numbers. Notice that the red one can be determined according to both Method 1 and Method 3.

"Eliminating non-unique rows and columns" focuses on the third rule mentioned above. Figure 3.20 shows how it is done. Since the first row cannot be the same as the second and we still need a one and a zero, this is the only valid way. We refer to this method as "Method 4".

0	1	1	0	0	1
0	1	1	0	1	0

Figure 3.20: Method 4 is applied by filling in the red numbers.

The last focus point for solving binary puzzles is "Remainder", which covers all remaining methods. This one is not as strict as the other ones. An example is shown in Figure 3.21, where there are three cells to fill; one 1 and two 0's. If we put the 1 at the end, the other two will be 0, creating a trio. This is not allowed, so the last one cannot be 1 and is therefore a 0. This is called a proof by contradiction. Together with some other method we refer to this method as "Remaining methods".

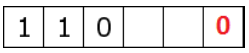


Figure 3.21: "Remaining methods" is applied by filling in the red zero.

3.3.2 CA solving binary puzzles

As for the other puzzles of this chapter, we tried to make a cellular automaton which can solve as many as possible instances of the binary puzzle. Although there could be a cellular automaton which can solve more instances of this puzzle, see Subsection 4.3, we chose to create a faster one. This subsection will explain its working.

The cellular automaton uses a Von-Neumann neighborhood with radius 2, so that every cell can see the next two cells in its four main directions. With this system the cellular automaton is able to execute the first three methods of calculating, explained in Subsection 3.3.1. The same as with the Nonograms of Section 3.2, instances of binary puzzles must be loaded manually into the algorithm. There is no function which creates random puzzles, as is the case in Section 3.1. The starting configuration of the puzzle we loaded is shown in Figure 3.22.

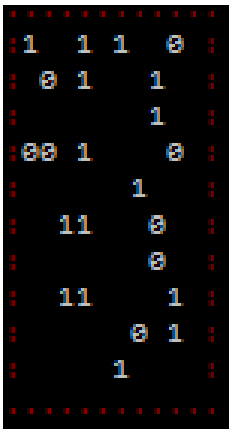


Figure 3.22: Output of algorithm showing starting configuration of a binary puzzle. The red dots are the border.

Similar to Section 3.2, the type *State* consists of multiple attributes. One of the attributes is *border*, which is implemented the same way as for the Nonogram solving cellular automaton. Also, the *sure* attribute is the same: $sure = 0$ means the cell is definitely a 0, $sure = 1$ means the cell is definitely a 1 and $sure = 2$ means that the cell is not definite yet.

Method 1 is very easy for a cellular automaton to execute. It is executed by the rule "If a cell has two consecutive neighbors with the same *sure* value that is not 2, this cell gets the *sure* value that is not 2 and not that of the consecutive neighbors.".

The single rule "If a cell has two opposite neighbors with the same *sure* value that is not 2, this cell gets the *sure* value that is not 2 and not that of the opposite neighbors." takes care of Method 2.

If the cellular automaton was only able to use the first two methods, only the easiest puzzles would be solvable for this cellular automaton. The example of Figure 3.22 could not be fully solved and the cellular automaton would end in the configuration of Figure 3.23 after only 6 iterations.

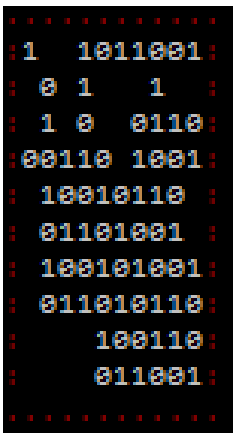


Figure 3.23: Ending configuration of Figure 3.22 if CA only uses Methods 1 and 2.

The last method that this cellular automaton can execute is Method 3. This is done by using a single walker in both views (horizontal and vertical) that is imitated by the attributes *vwalker* and *hwalker* of *State*. When this walker is activated, the last cell in the row or column will set its value of *firedWalker* to true to make sure each row and each column has only one walker. This walker walks upwards with value 1 of *vwalker* (for columns) and counts all zeros, ones and unknowns. This can be done by making them also attributes that walk with *vwalker*. When the walker reaches the top of its column (the upper neighboring cell is border), it calculates whether all the unknown cells can be set to the same value or not. This is the case when one of the following statements is true:

- The amount of ones minus the amount of zeros in the column is equal to the amount of unknowns in the column. This will set the *vwalker* to 3, which will make every unknown cell zero on its way down.
- The amount of zeros minus the amount of ones in the column is equal to the amount of unknowns in the column. This will set the *vwalker* to 4, which will make every unknown cell one on its way down.
- None of the above is true. This means that this column has not reached its limit for any of the two values zero or one. Thus, *vwalker* becomes 2 and walks down without changing unknown cells.

With all these abilities the cellular automaton can solve the example of Figure 3.22. Figure 3.24 shows its ending configuration. It takes the cellular automaton 63 iterations to solve this puzzle.

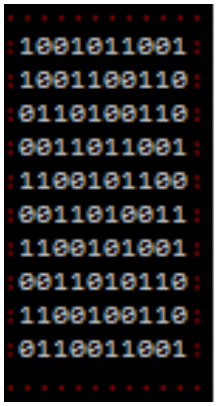


Figure 3.24: Ending configuration of Figure 3.22 after 63 iterations.

Chapter 4

Evaluation

In Chapter 3 there are three CAs for the puzzles Maze, Nonogram and Binary puzzles introduced. In this chapter these algorithms will be evaluated.

4.1 Maze-solving cellular automata

Note that the problem of the automaton of Subsection 3.1.1 (not being able to see the fake paths afterwards) could be solved much easier. For instance, this could be done by acting exactly as the algorithm does, only when changing paths into walls, changing them into another state 3 instead. In this case state 3 means fake path and this could be visualized by giving state 3 just another color. This CA would be much faster than the second Maze solver, see Subsection 3.1.2, since it is just as fast as the one described in Subsection 3.1.1. That cellular automaton will solve the puzzle in as many iterations as the length of the longest fake path.

The amount of iterations of the CA of Subsection 3.1.2 is equal to the length of the main path or the (waiting time + length) of the fake path that has longest (waiting time + length). The waiting time here is the amount of iterations before the cross-road of that fake-path is in blue. This is equal to the length of the main path from entrance to that cross-road.

Still, the CA of Subsection 3.1.2 is special because it walks across all the paths and fake paths in the grid. We chose to implement this CA instead of the one described above because this CA approaches the problem in a different way, while the other CA is kind of similar to the one described in Subsection 3.1.1. This shows us that some problems can be solved by many different kinds of cellular automata.

Both cellular automata described in Section 2.3 can solve the most complex mazes, according to the complexity model of mazes of [McC01]. The complexity of mazes differs for instance in the amount of fake paths and turns in both fake paths and main path. These differences in complexity are more applicable to humans than to computers. For computers it is not more difficult to solve the most complex mazes according to this scale, it only needs more computation time. If a computer program is able to solve a simple maze, it is almost always

able to solve the most complex mazes too, since there are not new rules or methods that it should apply, only more. Therefore we can state that both cellular automata we described are able to solve all mazes, except the ones which contain cycles. Since longer fake paths make both cellular automata use more iterations (as stated above), a higher complexity according to the model of [McC01] will make the automata use more iterations to solve the mazes.

4.2 Nonogram-solving cellular automaton

After the cellular autmaton is done with the Nonogram of Figure 3.14, the puzzle will look as in Figure 4.1.

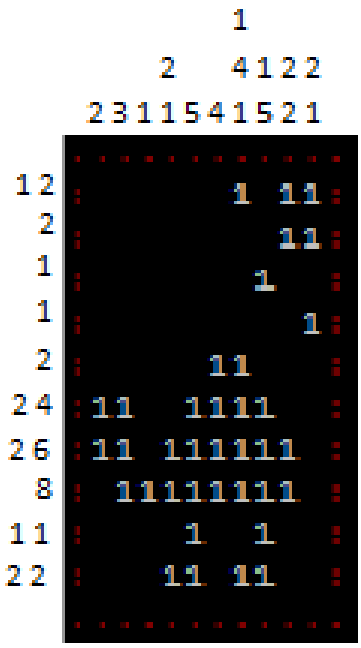


Figure 4.1: Ending configuration of Nonogram of Figure 3.14.

This configuration is reached after exactly 4489 iterations, which makes this the 4490th generation. We see that exactly the same cells are turned on and off is in Figure 3.8, therefore we can conclude that the cellular automaton solved this Nonogram correctly within 4489 iterations. Table 4.1 shows all the attributes of *State* with their possible values.

If we want an upper bound for the amount of possible states of this cellular automaton, we have to multiply or add for each attribute its amount of possible values. For *State*, this will leave us the following sum: $3 * 2 * 2 * 2 * 2 * 2 * 2 * 5 * 5 * 3 * 3 * 2 * 2 * 14 * 14 * 2 * 2 * 2 * 2 * 4 * 4 * 2 * 2 = 2^{15} * 3^3 * 4^2 * 5^2 * 14^2 = 69363302400 \approx 6.94 * 10^{10}$ possible states. Note that this is an upper bound. The biggest part of these states can never occur, for example a cell that is in state $border = true$. All its other attributes are set to an initial value at the beginning of the algorithm. A cell that is a border can never change state and a cell that is not a border can never become one. Therefore, the attribute gives us one extra state to the set of all other states. In

Attribute	Type	Possible values	Description
sure	integer	0–2	Shows definite value of cells
hgrid	Boolean	True/False	Current cell in horizontal view black or white
vgrid	Boolean	True/False	Current cell in vertical view black of white
honeseen	Boolean	True/False	Has there been a configuration wherein this cell is in hgrid black
voneseen	Boolean	True/False	Has there been a configuration wherein this cell is in vgrid black
hzeroseen	Boolean	True/False	Has there been a configuration wherein this cell is in hgrid white
vzeroseen	Boolean	True/False	Has there been a configuration wherein this cell is in vgrid white
border	Boolean	True/False	Is this cell a border or not
vwalker	integer	0–4	Updates seen values after each configuration
hwalker	integer	0–4	Updates seen values after each configuration
vwalker2	integer	0–2	Sets sure values after all configurations in row passed
hwalker2	integer	0–2	Sets sure values after all configurations in column passed
firedvWalker	Boolean	True/False	Indicates at last cell if vwalker is on in that row
firedhWalker	Boolean	True/False	Indicates at last cell if hwalker is on in that column
vbacktracker	integer	0–13	Manages all configurations of black strings in row
hbacktracker	integer	0–13	Manages all configurations of black string in column
vstarted	Boolean	True/False	Indicates at last cell if vbacktracker is on in that row
hstarted	Boolean	True/False	Indicates at last cell if hbacktracker is on in that column
vfirstblock	Boolean	True/False	Indicates if first black string is found at start vbacktracker
hfirstblock	Boolean	True/False	Indicates if first black string is found at start hbacktracker
vlegit	integer	0–3	Walker that checks if row configuration is valid
hlegit	integer	0–3	Walker that checks if column configuration is valid
vbringuptill	Boolean	True/False	Vbacktracker sets black strings back to cell with this marker
hbringuptill	Boolean	True/False	Hbacktracker sets black strings back to cell with this marker

Table 4.1: All 24 attributes of type *State* from Section 3.2

our upper bound calculation we took $x * 2$ instead of $x + 1$, where x is the rest of the calculation. The same is the case for many attributes which have values that can never hold for a cell at the same time. Another example is that there is a maximum of only one walker in a row or column per view. That means that for instance vertically seen, if *vwalker* has another value than its initial value 0, all the other vertical walkers like *vwalker*2, *vbacktracker* and *vlegit* can only have their initial value 0. The same holds horizontally seen. In our calculation we took for these four vertical walkers $x * 5 * 3 * 14 * 4$ respectively. We can lower this upper bound to $x(5 + 3 + 14 + 4)$. The upper bound will now become $(3 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * ((5 + 3 + 14 + 4) * (5 + 3 + 14 + 4))) + 1 = 3 * 2^{14} * (26 * 26) + 1 = 33226753$. This new upper bound of possible states is only $(33226753/69363302400) * 100\% = 0.0479\%$. This means our improvement on the upper bound is $100 - 0.0479 = 99.9521\%$. Perhaps there is even more to improve. This upper bound of 33226753 possible states means that there are at most that many states where a cell of this cellular automaton could get in while solving any Nonogram. For solving a particular Nonogram, a single cell will never reach this amount, not even close. Also, all the states of the automaton together will probably not pass all the possible states.

As stated in Subsection 3.2.2, when using only information concerning single row and columns, puzzles can often be solved partially, but not fully [KB09]. This has to do with the difficulty level of the puzzle. Puzzles that can be solved by only considering information from rows and columns separately, are stated to be of the simple type. However, the term simple is just relative, since nearly all Nonograms that appear in puzzle collections are simple. Figure 4.2 gives an example of a Nonogram that is not simple and thus cannot be solved by the cellular automaton of Section 3.2.

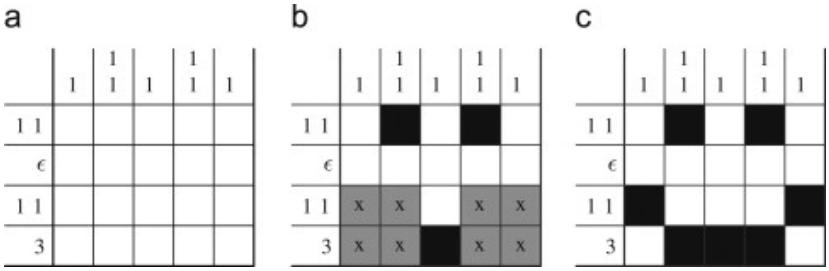


Figure 4.2: A 5×5 Nonogram, where (a) shows the initial puzzle; (b) is the partial solution; (c) is the final solution. The gray cells mean unknown and the ϵ means a single zero. Source: `https://www.semanticscholar.org/paper/Solving-Nonograms-by-combining-relaxations-Batenburg-Kosters/5be5e8f818e4e90b4a096e97fcb81e4ae12c105e`

In the partial solution of Figure 4.2(b) there will not be any row or column that can find a new definite cell by using only information from the line itself. Therefore, this Nonogram is not simple. It does however have one unique solution. To get to an answer, the solver could for instance guess one of the unknown cells. If he/she then notices that there is no correct solution anymore, the guess was wrong. If he/she does find a solution, it is a good one, although the solver is not sure yet if this is a unique solution.

4.3 Binary puzzle solving cellular automaton

In Subsection 3.3.1 we introduced five methods of calculating cells. We can make a difficulty model for instances of binary puzzles according to these methods. Each of those methods can be linked to one of the three rules at the start of that subsection. Every method tries to keep one of the rules valid. Method 1 and 2 together concern Rule 1. Method 3 concerns Rule 2 and Method 4 concerns Rule 3. The only method left, Method 5, concerns the application of all combinations of the three rules. According to this, we can set the following model of difficulty:

- Level 1; easy. These are all the instances of binary puzzle that can be solved using only Method 1 and 2.
- Level 2; medium. These are all the instances of binary puzzles that can be solved using Methods 1, 2 and 3.
- Level 3; hard. These are all the instances of binary puzzles that can be solved using all the Methods 1, 2, 3 and 4.
- Level 4; extreme. These are all instances of binary puzzles that do not adhere to the above levels. To solve such a binary puzzle one needs to use all the Methods 1, 2, 3 and 4 including Method 5, which is all the combinations of the other methods.

The cellular automaton of Subsection 3.3.2 can solve instances of binary puzzles of the levels 1 and 2. We tried to make the cellular automaton so that it could also solve puzzles of level 3 and 4. Therefore, it must be able to use Method 4 and 5, besides the Methods 1–3 it already uses. However, there is a big fundamental difference between the first three methods and the last two. The first three methods are applied to patterns which can be recognized. The first two can be recognized in one iteration after they appear, because all the necessary

information is in the neighborhood of the cell that can be set. The third rule needs information from the whole row or column, thus is implemented using a walker that gives all this information. The problem with the last two methods is that they are not applied by a pattern that can be recognized, but they only can be applied after a guess. Method 4 must first guess the remaining symbols, then check if that row or column is already used somewhere, and if so, change the last filled cells. If it is not used yet, it does not mean it is correct, because it can be used somewhere else later. Method 5 is also always applied after a guess. Generally, first a guess is applied to the last cells that can be filled. After the guess one of the other methods is applied. If it cannot be true, the guess was wrong and must be changed. If it still can be true, it again does not mean it is correct. These guesses can follow each other in deeper levels until a contradiction or a right solution is found. We have not proved yet that level 4 and 5 are not solvable by any cellular automaton, but we assume that it is true. If it is possible however, it would need many walkers that must communicate with each other in a complex way.

The instances of 10×10 binary puzzles of level 1 take between 7 and 9 iterations to be solved by the cellular automaton of Subsection 3.3.2. This amount is low because both Method 1 and 2 can be applied by looking only at the neighborhood of the changing cell. That means every iteration new sure values are set, because if not, it will never happen in the future. The puzzle is then either solved or not solvable by the CA.

The amount of iterations the cellular automaton takes to solve instances of binary puzzles of level 2 varies a lot. That is because Method 3 takes two times the size of the grid iterations. In our 10×10 example that are 20 iterations. However, this method starts over and over again and sometimes starts at a moment that is not optimal. For instance the following can happen. One iteration after Method 3 is started, the last cell in the row gets a sure value of 1 completing all the ones in that row. Now, Method 3 should actually start. But because it has just started, one must wait until it is done and started over, to apply Method 3 usefully. This means that it can take up to 39 iterations before Method 3 is applied correctly. When the puzzle can only be solved by applying Method 1 or 2 and Method 3 alternately, it is possible that it takes successively $39, 1, 39, 1 \ldots$ iterations to only apply a few rules. Therefore the maximum amount of iterations the cellular automaton needs to solve the 10x10 instance can get really big, while it could also be solved in 63 iterations, as the example of Figure 3.24 in Subsection 3.3.2. We think that it takes between 5.0×10^1 and 1.5×10^2 iterations on average to solve a 10×10 instance of level 2 by the cellular automaton.

Chapter 5

Other puzzles

In the previous chapters we have found algorithms for three different puzzles and we have evaluated them. Now, we want to use this knowledge to find other puzzles which are solvable by a cellular automaton and find out how that is possible. As we have seen in the previous chapters, cellular automata are suitable for applying elimination rules. Puzzles that can be solved by using only elimination rules are highly likely to be solvable by a cellular automaton. All the techniques we have used when solving the puzzles of the previous chapter rely on eliminating what is impossible, thus knowing what is correct. In Chapter 3 we saw the following methods of solving puzzles which rely on elimination:

- When solving mazes, the automaton searches for fake paths which we could eliminate, instead of looking for the main path as a human would do. A human would "guess" at a cross-road and eventually eliminate if the guess was wrong. The first cellular automaton we built starts with eliminating instead of guessing.
- When solving Nonograms, the automaton tries every configuration and eliminates those that are surely wrong, leaving some cells with only one possible value.
- Method 1 and 2 of solving binary puzzles both rely on eliminating by looking for possible trios and avoiding them by eliminating one option leaving one other option to fill in.
- Method 3 of solving binary puzzles relies on elimination as well because it eliminates values which make the amount of 0's and 1's in a row unequal.
- Method 4 of solving binary puzzles eliminates values which make to rows or two column equal.

We will now present some puzzles which we assume to be (partially) solvable by a cellular automaton and explain how this could be accomplished.

5.1 Sudoku

Sudoku, see Figure 5.1, is one of the best known puzzles across the world. Every row, every column and every section (inner 3×3 square) must be filled with exactly all the numbers 1–9. In all the three directions each number can only appear once. Now we only consider Sudoku's on a 9×9 grid, but there are also other sizes of Sudoku. To make a CA which solves Sudoku generally for all sizes will be even harder.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 5.1: Instance of a Sudoku. Source: `https://en.wikipedia.org/wiki/Sudoku`

When a human plays Sudoku, most of the methods the human applies rely on elimination. For instance, when a row contains a 5, all the other numbers in that row cannot be 5. A possible cellular automaton can also work this way. When a 5 is filled in a certain cell, there should be three walkers activated:

- A *hwalker* must walk across all the cells in that cell's row and eliminate the value 5 in all those cells.
- A *vwalker* must walk across all the cells in that cell's column and eliminate the value 5 in all those cells
- A *swalker* must walk across all the cells in that cell's section and eliminate the value 5 in all those cells.

There are multiple ways these walkers can work, since sometimes multiple walkers will be activated in the same row/column/section at the same time, it can be a good idea to make a walker for every number 1–9 in every row, column and section.

This way the automaton can eliminate many values, but still it is not able to apply simple rules. For instance in Figure 5.1 in the middle-left section, there is only one option to place the 3. A human would spot this place with the simple rules of row elimination and column elimination. The automaton will do something similar, but it will not yield the solution of where the 3 belongs since that cell still has multiple options of values. To solve this problem there should be more walkers which scan each section on the appearance of each number. That walker should detect that there is only one option for the 3 in that section. This could be a solution although it will be very hard to construct such an automaton, assuming this construction would not yield new problems.

The above shows that even if it is possible to make a cellular automaton which solves Sudoku's, it will be very hard and very complex with enormously many states, to only make it partially solvable (or a part of the instances fully solvable).

5.2 Tectonic

Tectonic, see Figure 5.2, is a puzzle with multiple grids inside one big grid. We will call these smaller grids sections. These sections vary in shape and amount of cells. A section can have a size of 1 to 5 cells. Each cell must be filled with a number from 1 to 5. Each section must be filled with unique numbers from 1 to its size. For instance a section consisting of four cells, must contain the numbers 1, 2, 3 and 4. This can be filled in any possible way. Also, neighboring cells can never have the same number.

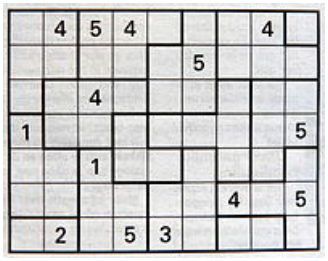


Figure 5.2: Instance of a Tectonic puzzle. Source: `https://nl.wikipedia.org/wiki/Tectonic`

This means there are two rules that a certain cellular automaton must apply to make it partially solvable: filling each section with all the numbers of its size and avoid neighboring cells with the same number. If the cellular automaton has 5 attributes of *State* which indicate for each number if it is possible for the cell, it could avoid neighboring cells with the same number by simply eliminating attributes as impossible if one of its neighbors has that value. Repeating this process could sometimes make values known for sure.

For the other rule, the cells of each section must know somehow the size of the section in which that cell is located. If the cellular automaton is initialized such that each cells is aware of the size of its section, it could eliminate all the numbers that are bigger than that size also at initialization. A section with only one cell can then determine in the first iteration that it must be a one because all the other values are eliminated.

Now the only rule that remains is when two cells are in the same section but they are not neighbors, they cannot have the same number. In that case, when one of the cells is already determined, the other cell must eliminate that value. This could be implemented in a cellular automaton by making a walker walk across all the cells of its section, each time a number is determined. During that walk it eliminates in every cell it passes that number from its possible values. We will not give further details on how this cellular automaton should work, only a global description. The cellular automaton described above, will not be able to solve all instances of Tectonic. Sometimes, like in the other puzzles, rules need to be combined. Figure 5.3 shows an example of a rule that this automaton could not apply.

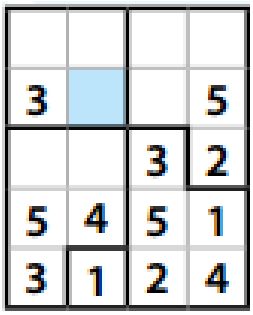


Figure 5.3: The blue cell must contain a 4, because the two cells beneath it must be 1 and 2. Although we do not know which of those cells must be 1 and which must be 2, they are both neighbors of the blue cell. In each case the blue cell has neighbors 1, 2 and 3, and since the size of its section is 4 it can only be 4. Source: `https://www.denksport.nl/media/wysiwyg/Info-paginas/tnn_UITLEG_WEB.pdf`

5.3 Kakuro

In a Kakuro puzzle, see Figure 5.4, the light-gray cells show a number and a horizontal or vertical direction. The following white cells from that light-gray cell to another non-white cell must sum together to the value in the light-gray cell using only the numbers 1–9. This is valid for all the light-gray cells and its consecutive white cells. There is one other rule: it is not possible to have the same number more than once in one sum.

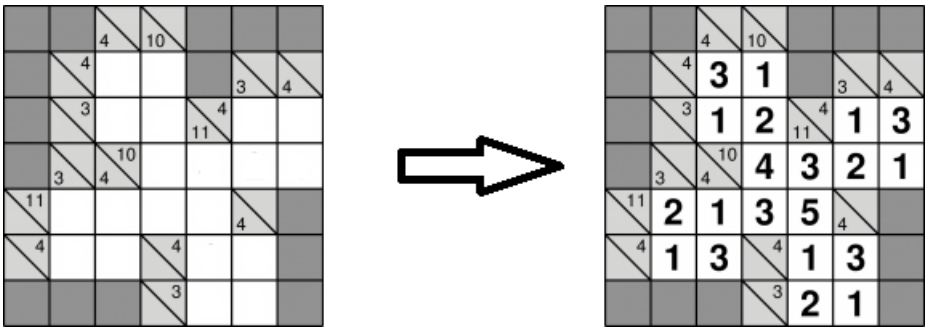


Figure 5.4: An instance of Kakuro. Source: `https://www.puzzlemix.com/Kakuro`

To find a cellular automaton that can solve instances of Kakuro, we will use some concepts that we used earlier in Section 3.2 for solving Nonograms. At start, we have to deal with the same problem as with Nonograms, that we have to take the information inside the (useful part of the) grid. This can be dealt with the same way as with Nonograms, by creating two views, horizontally and vertically. For each sum the first white cell will have the biggest possible value and the second white cell will have the biggest possible value remaining, and so on, taking into account the cells that will follow. The same way as with Nonograms we will create all configurations of these values. With Nonograms, the cellular automaton did this by pushing whole cells, this cellular automaton will push units of the values to next cells. Figure 5.5 shows the initialized configuration of a sum and the next configuration.

Figure 5.5: The first two configurations of a sum in the Kakuro-solving cellular automaton.

During this construction of all configurations, every configuration will be checked on validity, and if it is valid,

the cells will save that value as possible, like the cellular automaton solving Nonograms. If this process keeps repeating, we think this cellular automaton will be able to solve instances of Kakuro. Ruepp and Holzer [RH10] only show that Kakuro is NP-complete, but do not give a model for difficulty of different instances of Kakuro. Therefore, we cannot state if the cellular automaton described above can solve all instances of Kakuro.

Chapter 6

Conclusions

In this thesis we examined the appliance of CAs to solving puzzles. Very generally we can say that there are instances of puzzles that are solvable by any cellular automaton. We found different cellular automata which are able to solve instances of three types of puzzles:

- Maze puzzles
- Nonograms
- Binary puzzles

Also we speculated about the solvability of three other puzzles by cellular automata, but we are not able to prove that there are cellular automata which can solve these puzzles.

Although we found automata which are able to solve instances of the above puzzles, this does not mean these automata can solve all instances of those puzzles. The cellular automaton which solves mazes is able to solve all instances without cycles, while the automata concerning both Nonograms and Binary puzzles are only able to solve the simple instances. For most puzzles there are cellular automata which can apply their methods to solve parts of the puzzles. However, when methods need to be combined to make any progress in the puzzle, the cellular automata of both Nonograms and binary puzzles fail to apply this combination. As stated in Section 1.3 we will not prove that instances of puzzles are not solvable or methods are not applicable by any cellular automaton, but we will only prove those that are.

What is valid for people, the fact that instances of puzzles of higher complexity levels are harder to solve, is also valid for cellular automata. To solve instances of more complex mazes only takes more time, while cellular automata solving most other puzzles need to be significantly better in the way of being capable to apply new methods to solve the more complex instances. This is the case for both Nonograms and binary puzzles solving cellular automata. For these puzzles new methods must be used by the cellular automaton to solve more complex instances of the puzzle.

We saw with solving binary puzzles that incrementing complexity level can increase the number of iterations

it takes the cellular automaton to solve the puzzle exponentially. Where 10×10 instances of binary puzzles of complexity level 1 could be almost always solved within 10 iterations, 10×10 instances of complexity level 2 can take more than 100 iterations to be solved. We expect that same behavior when incrementing the complexity level to 3. Of course the number of iterations also depends on the fitness of the cellular automaton itself. Both cellular automata we built for solving mazes are able to solve all instances of mazes, while the first one always needs less iterations than the second one.

6.1 Answers to research questions

At the start of Chapter 1 some research questions were formulated. This section will try to answer those questions as good as possible with the new insights and information learned throughout this research.

Is there a way to connect puzzles to cellular automata, by using the cellular automaton to simulate and thus solve the puzzle?

Yes, there are puzzles which can be totally or partially solved by a cellular automaton. It depends on the puzzle and the complexity of that instance if it is solvable. There are restrictions that these puzzles must have to be solvable by a cellular automaton, such as consisting of a grid with cells. However, sometimes the lack of crucial properties of a puzzle can be compensated by help beforehand, like with Nonograms.

Which types of puzzles can be solved by cellular automata?

We know for sure that instances of mazes, Nonograms and binary puzzles are solvable by a cellular automaton. Puzzles using more elimination techniques to solve them are more likely to be solvable by any cellular automaton. Tectonic and Kakuro are likely to be solvable by a cellular automaton while Sudoku is likely to be at least partially solvable by a cellular automaton.

Will cellular automata that solve different puzzles look alike?

We used many "walkers" for different types of cellular automata, but that only says something about our own implementation. We did not recognize any other properties each cellular automaton solving puzzles must have.

Most logical puzzles have complexity levels. How do cellular automata deal with these levels?

Cellular automata solving puzzles of a higher complexity level need more and other methods for determining certain cells. The combination of multiple methods is hard for a cellular automaton. Higher complexity levels of puzzles also take the cellular automaton exponentially more iterations to solve them.

6.2 Future work

In the future we would like to see more puzzles getting solved by cellular automata. Especially, it would be great to see the puzzles we speculated about in Chapter 5 getting solved by cellular automata, in particular Sudoku. We are also curious if 3-dimensional CAs can be used to solve 3-dimensional puzzles, for instance a 3D Minesweeper. Besides solving more puzzles, we would like to see more of a pattern and similarities in the different CAs which can solve puzzles.

Bibliography

[Ada97] A. Adamatzky. How cellular automaton plays minesweeper. *Applied Mathematics and Computation*, 85:127–137, 1997.

[Gar70] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223:120–123, 1970.

[Gre87] F. Green. NP-complete problems in cellular automata. *Complex Systems*, 1:453–474, 1987.

[KB09] W.A. Kosters K.J. Batenburg. Solving Nonograms by combining relaxations. *Pattern Recognition*, 42:1672–1683, 2009.

[McC01] M.S. McClendon. The complexity and difficulty of a Maze. *Bridges: Mathematical Connections in Art, Music, and Science*, July:213–222, 2001.

[RH10] O. Ruepp and M. Holzer. The computational complexity of the kakuro puzzle, revisited. volume 6099, pages 319–330. Fun with Algorithms: 5th International Conference, FUN 2010, 2010.

[Var12] A. Varga. Maze solving with a 2d cellular automaton. `http://demonstrations.wolfram.com/MazeSolvingWithA2DCellularAutomaton/`, July 2012.

[Wol02] S. Wolfram. *A new kind of science*. Wolfram Media, 2002.