universität innsbruck

Bachelor Thesis

Kurodoko

Johannes Koch
johannes.koch@student.uibk.ac.at

5 July 2020

Supervisor: Univ.-Prof. Dr. Aart Middeldorp

Eidesstattliche Erklärung

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

Datum	Unterschrift

Abstract

The primary focus of this thesis is the pen and paper puzzle *Kurodoko* and a variant called *Oh no*. After acquainting the reader with both puzzles and their rules, multiple SMT based solvers are presented for both puzzles together with an in-depth performance analysis. Moreover, generators for creating new *Kurodoko* and *Oh no* puzzles are showcased. In addition, a graphical user interface for playing the puzzles is introduced and it is shown that *Oh no* is NP-complete.

Contents

1 Introduction

Logic puzzles are a fun and challenging way of teasing the brain and honing one's logical and deductive skills. *Kurodoko* is a Japanese logic puzzle created and first published by Nikoli in 1991 [9]. In typical fashion, the rules for the puzzle are comprehensible and allow anyone to enjoy the game. So far only one official *Kurodoko* book was published by Nikoli. Since 2015 there also exists a popular variant of *Kurodoko* called *Oh no* [10]. *Oh no* has fewer and simpler rules than *Kurodoko*, making it even easier to get started.

Even though *Kurodoko* has been invented almost 30 years ago, the puzzle is still widely unknown. Nevertheless, some research on *Kurodoko* has been done. Jonas Kölker proved that *Kurodoko* is NP-complete in [11] and Tim van Meurs looked into the success rate of multiple solving approaches in [27]. On the other hand *Oh no* seems to be completely uncharted terrain. Hopefully this thesis will increase the awareness of these enticing puzzles.

1.1 Goals

The goals of the thesis are to devise and implement solvers for both *Kurodoko* and *Oh no*, to develop generators for the creation of new puzzles with varying difficulty and to develop a convenient graphical user interface for both puzzles. In addition to these goals, the NP-completeness of *Oh no* has also been proven.

1.2 Overview

In the next chapter both *Kurodoko* and *Oh no* will be introduced. In Chapters 3 and 4 respectively, the solvers and generators for both puzzles are presented. For some of the rules multiple different encodings are presented and the generators are able to adjust the difficulty of a generated puzzle. Thereafter the graphical user interface is showcased. The user interface allows users to play a collection of puzzles created by Nikoli and puzzles generated using the generators from Chapter 4. Finally the NP-completeness of *Oh no* is shown in Chapter 6.

2 Kurodoko and Oh No

In this chapter both puzzles and their rules are presented.

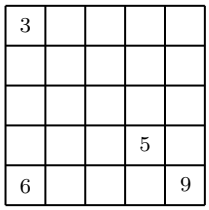
While the cells of both *Kurodoko* and *Oh no* are either white or black, it is common to distinguish between the notation of empty and white cells, since empty cells can be either black or white. In order to increase the readability and to emphasise this distinction, we refer to ⊡ as white cells and to □ as empty cells.

2.1 Kurodoko

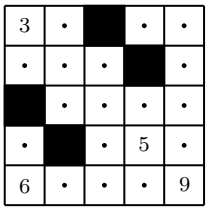
Initially Nikoli published this puzzle with the name *Kuromasu*, but later decided to rename the puzzle to *Kurodoko* [9]. *Kurodoko* is played on a rectangular grid, where initially some of the cells contain a number as a hint and all other cells are empty. The aim of the puzzle is to colour some of the cells black, such that all rules of the puzzle are fulfilled. All non-black cells are coloured white. These rules are [17]:

1. A white cell can see the other cells in the same row and column. Black cells block the view of white cells.
2. The number in a cell denotes the number of white cells that cell can see (**including** itself). Numbered cells cannot be black.
3. No two black cells may be horizontally or vertically adjacent.
4. All white cells must be connected horizontally or vertically and need to form a contiguous area.

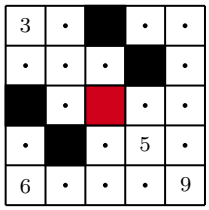
Example 2.1. Figure 2.1 shows an empty and a solved *Kurodoko* puzzle taken from [9]. Figure 2.1(c) shows a violation of the connectivity rule of the white cells. The cell coloured in red divides the white cells into two disconnected areas.



(a) The empty puzzle.



(b) The solved puzzle.



(c) A violation of rule 4.

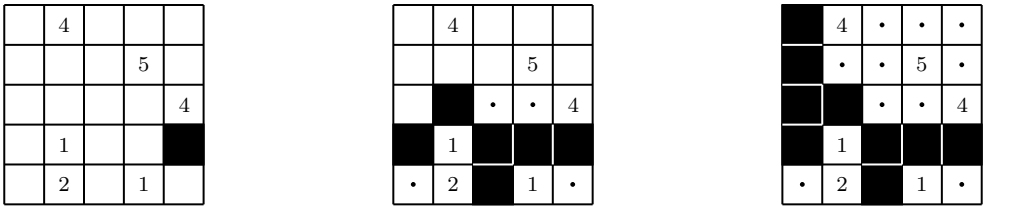
Figure 2.1: A *Kurodoko* puzzle.

2.2 Oh No

Oh no was created by Martin Kool in 2015 [20] and is a variant of *Kurodoko* [10]. In contrast to empty *Kurodoko* puzzles, empty *Oh no* puzzles may already contain black cells in addition to hints and empty cells. Other than that the aims of the puzzle are similar, but *Oh no* has fewer and arguably simpler rules [10]:

1. A white cell can see the other cells in the same row and column. Black cells block the view of white cells.
2. The number in a cell denotes the number of white cells that cell can see (**excluding** itself). Numbered cells cannot be black.
3. A white cell can see at least one other white cell.

Example 2.2. Figure 2.2 depicts the process of solving an *Oh no* puzzle.



(a) An empty *Oh no* puzzle. (b) A partially solved puzzle. (c) The solved puzzle.

Figure 2.2: Multiple steps of solving an *Oh no* puzzle.

The main differences between *Kurodoko* and *Oh no* are

- the removal of the rule that black cells cannot be neighbours,
- and the relaxation of the constraint that all white cells have to be connected, to the rule that a white cell can see at least one other white cell.

Another notable difference is, that in *Oh no* the number of cells a cell can see is counted **excluding** the cell itself as opposed to *Kurodoko*. Other than the difference in counting the visible cells, all the rules of *Oh no* are identical or less strict than the rules of *Kurodoko*.

Because of this, every *Kurodoko* puzzle can be turned into a *Oh no* puzzle if the hints are adjusted by decreasing all the numbers by one. If the *Kurodoko* puzzle does not contain a hint with the number 1, then any solution to the *Kurodoko* puzzle would also be a solution to the constructed *Oh no* puzzle. *Kurodoko* hints with the number 1 are an exception, because decreasing the number by one would lead to the number 0. An *Oh no* cell with the number 0 cannot be satisfied: The cell must be white since it has a number, therefore it must see at least one other white cell which contradicts the constraint that the cell sees exactly zero other cells.

3 Solvers

Solving a *Kurodoko* or an *Oh no* puzzle is a matter of colouring some of the initially empty (□) cells black (■) such that no rules are violated. While the rules are straightforward, finding a valid solution can be quite challenging. In this chapter solvers for both puzzles are presented. Finally the performance of the developed *Kurodoko* solvers is benchmarked. Furthermore, other freely available solvers are included in this benchmark as comparison.

3.1 Solver Outline

Since the puzzles have very similar rules, the overall solving procedure is roughly the same for both puzzles. Rather than trying to solve the puzzles directly, the puzzle is turned into constraints for an SMT solver:[1]

1. First the puzzle together with the rules are transformed into constraints for an SMT solver. In order to be able to do this, encodings for all of the rules are needed.

2. Then the solver is called and tries to satisfy all constraints. If there exists a solution, the solver will return a model where all constraints hold.

3. If the solver returns a model, the model is transformed into a solution for the puzzle. Otherwise, there is no solution.

For some of the rules, multiple different SMT encodings are presented. Those include the rule about the hints with numbers seeing other cells. Another rule where more than one encoding was devised is the *Kurodoko* rule that all white cells must be connected.

3.2 Oh No

The rules of *Oh no* can be summarised as

- every white cell can see at least one other white cell,

- and each numbered cell must be white and see exactly the same amount of other white cells as the number dictates.

[1] In the implementation Z3 [14], an open source SMT solver developed by Microsoft Research, was used through its C++ API.

If those two constraints are met for a puzzle instance, the puzzle is solved. Thus the formula which is handed over to the solver needs to ensure that these constraints are met. The solver then colours the cells such that the model returned by the solver is a solution to the puzzle.

3.2.1 Each White Cell Sees at least one Other

The first rule that each white cell can see at least one other white cell, is of course true for every white cell seen by a numbered cell, since each of those white cells can at least see the numbered cell. For white cells which cannot be seen by a single number this rule can still be of use since it demands that white cells cannot appear alone, disambiguating the colour of empty cells where all neighbours are black. Another consequence of this rule is that no cell can have the number 0, since that cell would not be allowed to see any other white cells.

The rule can be interpreted as if the cell is white, then (at least) one of its neighbouring cells must be white too. Figure 3.1 shows the constraint for a cell and its neighbours. In the figure, $\boxdot_c$ denotes that cell c is white.

This constraint needs to be added for every cell in the grid. Of course for border and corner cells, which do not have neighbours in all directions, the constraint needs to be adjusted to only include existing neighbours.

3.2.2 Number Constraints

A cell can see all the white cells in each direction up to the first black cell and cells containing a number must see exactly this number of other cells. In order to keep numbered cells from seeing too many other cells, black cells need to be placed to block the view.

Simple Enumeration

A naive encoding for the number constraint, which is presented as a baseline, is to just enumerate all possible ways such that a number can see the right amount of cells. Figure 3.2 shows all options for the number 2. This can be turned into a constraint for the solver by demanding that at least one of these options must be true, and thus the

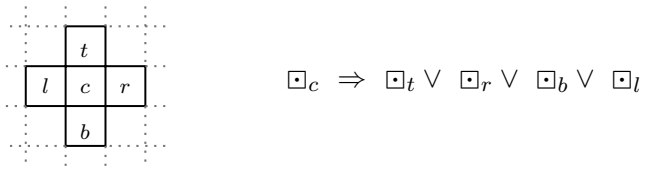


Figure 3.1: If cell c is white, this implies that at least one of the top (t), right (r), bottom (b) or left (l) neighbours must also be white.

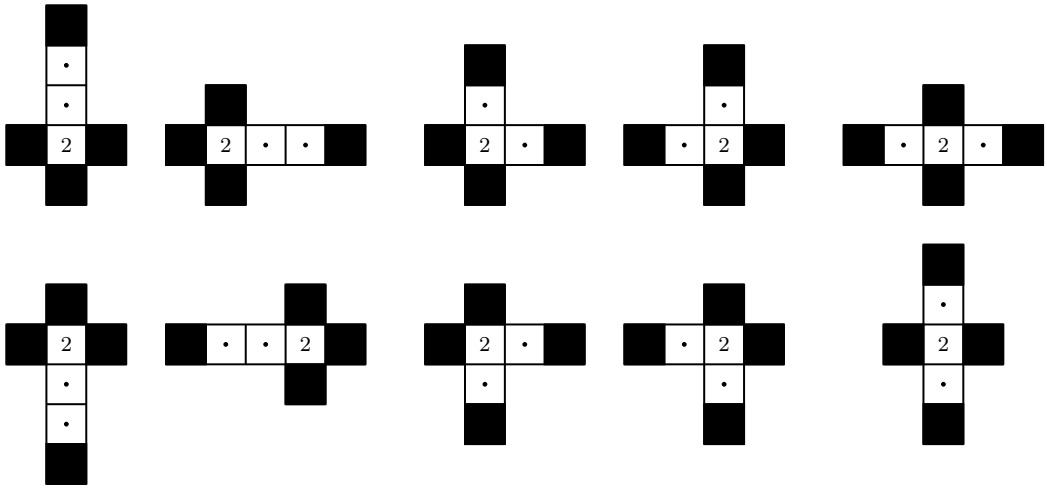


Figure 3.2: All the options for a cell with number 2 (3 for *Kurodoko*) to see exactly two other cells.

number constraint must also be satisfied.[2] Note that there is no need to demand that exactly one of the options must be true, since the options are disjunct. It is not possible for multiple of the options to be true at once, since some of the fields would need to be black and white (⊡) at the same time. Of course for cells near the border, where not all white cells are within the puzzle grid, some options do not need to be considered. Additionally some of the black cells which are needed to ensure that the cell cannot see any more cells may be replaced by the border of the grid.

Improved Enumeration

Enumerating all options in this way is rather inefficient as the method does not consider the hints of the puzzle instance to be solved. The puzzle instance may already eliminate some of the options.

Example 3.1. Consider the puzzle in Figure 3.3, where the enumerated options for the bottom left cell with the number 2 are shown.

[2]The representation of an option is the conjunction of the colours (⊡ or ■) of each cell in the option.

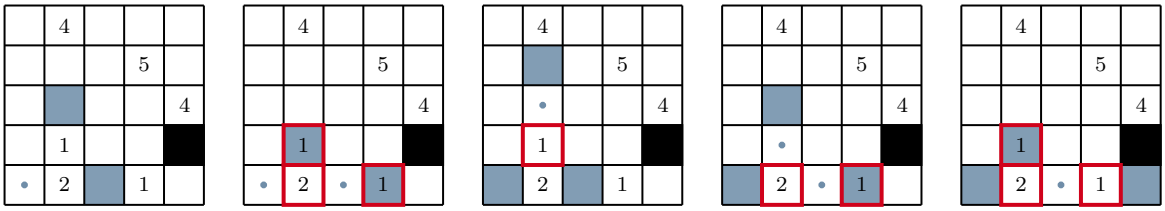


Figure 3.3: All the options for the bottom left cell with the number 2 which would be enumerated.

The cells filled in blue-gray denote the positions of the black cells for a given option. Cells highlighted in red violate the rules, as they either see too many cells or need to be both black and white which is not allowed. Only the leftmost option is a valid option, all the other options would directly violate a rule, by either making a cell see too many other cells or by colouring a numbered cell black.

One observation is that any adjacent cells which also have a number must be white and therefore the cell with the number 2 must see at least one cell to the top. But as the number in the adjacent cell is 1, it cannot see any more cells to the top or the number constraint of this 1 would be violated. To the right, the cell with the 2 cannot see any cells, because if it would see even one cell, the cell with the number 1 to the right would see too many other cells. Since the cell with the number 2 is in the last row, the only remaining direction is left and thus the cell needs to see exactly one cell to its left.

By considering the information given by the puzzle instance, the enumeration process can be improved, using the following insights:

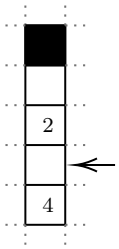
- The colours of some cells are already defined by the puzzle, which already removes all options conflicting with these colours. For example, a numbered cell cannot be black, thus all enumerated options where this cell would be black can never hold.
- If a cell with the number n_1 can see another numbered cell with the number n_2 where $n_2 < n_1$, then the first cell can see at most n_2 cells in this direction since the other cell would see too many if it would see more.
- If a cell can see at most k other cells (where $k < n$) in its row/column, but needs to see n other cells in total, then the cell must see the remaining $n - k$ cells in its column/row. This insight is often useful for human solvers to determine the colour of a cell.

Algorithm 1: The algorithm to collect the options for a given direction.

```
visible = 0 // number of cells visible in direction
smallest_number = number of the cell itself // smallest visible number
options = { }
current = first cell in direction
do
    // if the cell is a number, then no black cell can be placed at
       the current cell
    if current can be black (or the border of the grid has been reached) then
        options = options ∪ {(visible, smallest_number)}
    if the border has been reached or current is already black (■) then
        return options
    if current has a number which is smaller than smallest_number then
        smallest_number = number of the current cell
    current = next cell in direction
    visible = visible + 1
while smallest_number ≤ visible
return options
```

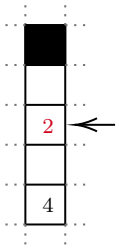
In order to incorporate these insights to reduce the number of options, the other potentially visible hints have to be taken into account for each numbered cell. This is done by adding a new step before the possible number layouts are enumerated. The restrictions a puzzle instance imposes on a numbered cell can be collected by looking at the potentially visible hints in each direction, as shown in Algorithm 1.

In order to illustrate this procedure, an example for the top direction is presented in Figure 3.4. Note that the smallest visible number is also collected. This is done, because the smallest number to the top can also see the cells seen to the bottom since they are in the same column and vice versa. Of course the same holds for the numbers to the left and to the right, because those are in the same row.



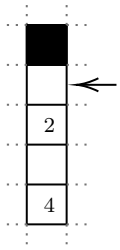
visible : 0
smallest_number : 4
options : $\{(0, 4)\}$

The first option for the cell with the number 4 is to see no other cells to the top. If this were the case, then the top neighbour of the cell would have to be black. Since the cell is empty, this is possible and thus seeing 0 other cells to the top is an option. The largest seen number in this case is the number 4 itself, which does not impose any additional restrictions.



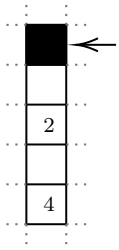
visible : 1
smallest_number : 2
options : $\{(1, 4)\}$

The second option would be that the cell could see exactly one cell to the top. In that case, the cell with the number 2 would have to be black. Since numbered cells must be white, this is not an option. In addition, the number 2 is lower than 4, and thus the smallest encountered number is 2. This decreases the upper limit on how many cells can be seen in the column to 2.



visible : 2
smallest_number : 2
options : $\{(1, 4), (2, 2)\}$

After the option to see one cell was considered, the next step is to consider seeing two cells to the top. Since the third cell to the top is empty, a black cell could be placed there. Furthermore, seeing two cells also does not exceed the lowest encountered number. Thus, seeing two cells to the top is an option. If this is the case, at most two cells could be seen to the top because otherwise the cell with the number two would see too many other cells.



visible : 3
smallest_number : 2
options : $\{(1, 4), (2, 2)\}$

Finally the black cell is considered. While this cell could be black, seeing three cells to the top would violate the number constraint of the lowest encountered number. Therefore seeing more than two cells is not an option and all options for the direction have been collected. Note that even if the smallest number would not lead to a conflict, there would not be a need to look further in this direction, because the black cell would block the view.

Figure 3.4: By taking other hints into account, a lot of options can be eliminated. Note that the current cell is marked with an arrow.

Once the options for all four directions have been collected, the possible layouts for a given number can be enumerated. The combination of the pairs $(v_t, sn_t) \in options_{top}$, $(v_r, sn_r) \in options_{right}$, $(v_b, sn_b) \in options_{bottom}$ and $(v_l, sn_l) \in options_{left}$ is only a valid layout for a given number, if these restrictions hold:

$$\begin{aligned} v_t + v_b &\leq min(sn_t, sn_b) \\ v_l + v_r &\leq min(sn_l, sn_r) \\ v_t + v_b + v_t + v_b &= number \end{aligned}$$

In order to capture all possible options, all combinations for the options of each direction have to be considered. The resulting constraint for the solver then is, that one of the remaining layouts must hold. Additionally, the cells which are white in every possible layout must also be white in the solution.

While this still enumerates all seemingly valid options, the incorporated improvements do improve both the work needed to compute the constraints for the solver and the search space the solver has to comb through. This method also does not suffer from more hints being in the puzzle. The simpler version of the enumeration has to do a lot more work if more numbered hints are present, whereas the improved version becomes more efficient and can eliminate even more options.

Encoding Visibility

Another and completely different approach to encoding the number constraints is to encode the notion of *seeing* other cells into the encoding. This can be done by looking at each one of the directions *top*, *right*, *left* and *bottom* separately.

If the top neighbour of a white cell is black, it can see 0 cells to the top. On the other hand, if the top neighbour of a white cell is also white, then it can see exactly one more cell to the top than the cells top neighbour can see to the top. Additionally, the white cells in the first row have no top neighbour and thus see no other white cells to the top. This can be expressed with two constraints, one for the first row and one for every remaining row. White cells in the first row cannot see other cells to the *top* since there are none:

$$\text{if } \boxdot_c \text{ then } sees\_top_c = 0 \text{ else } sees\_top_c = -1$$

The remaining cells can see their top neighbour (hence +1) plus whatever their *top* neighbour can see:

$$\text{if } \boxdot_c \text{ then } sees\_top_c = sees\_top_t + 1 \text{ else } sees\_top_c = -1$$

Note that by choosing the value -1 for black cells, the *sees_top* variable for the bottom neighbour of a black cell becomes 0 as intended.

The encoding for the other directions is analogous, but of course another row/column needs to be treated differently since the cells in the row/column have no neighbours in the respective direction. All directions together allow for a direct encoding of *seeing*,

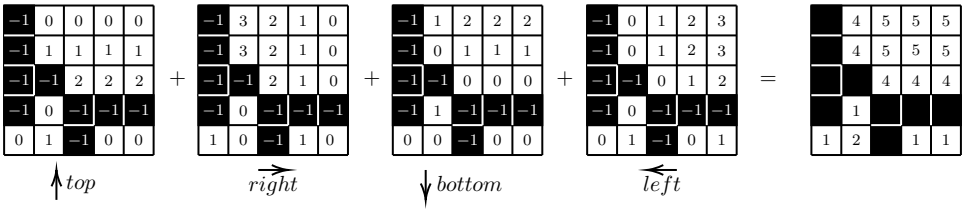


Figure 3.5: Using an additional integer variable for each direction allows for a direct encoding of *seeing* other cells.

since the number of seen cells simply is the sum of seen cells for each direction. Using these additional variables for each direction, a cell c with a number n can be encoded as follows:

$$sees\_top_c + sees\_right_c + sees\_bottom_c + sees\_left_c = n.$$

For the previously in Figure 2.2(a) presented *Oh no* puzzle, the directional variables and the computed number of seen cells is shown for every cell in Figure 3.5. Note that in the last grid showing the number of seen cells for each white cell, the black cells are empty. This is because while the encoding would suggest a black cell sees -4 other cells, the rules define the notion of *seeing* only for white cells.

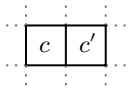
In order to solve an *Oh no* puzzle it is not necessary to know how many other white cells are seen by every cell in the puzzle, but to demand that the numbered cells can see exactly that number of other cells. Hence it suffices to add the constraints only for the rows and columns which contain a numbered cell.

3.3 Kurodoko

The general approach for solving *Kurodoko* puzzles is the same as for *Oh no*. However, as the rules of *Kurodoko* have differences, the encodings have to be adjusted to reflect the rules of *Kurodoko*. While *Oh no* requires white cells to see at least one other white cell, *Kurodoko* requires all white cells to be connected. Additionally, in *Kurodoko* black cells may not be next to each other and a white cell see itself too.

3.3.1 Black Cells Cannot be Adjacent

This rule demands that black cells cannot be neighbours of each other. For any two adjacent cells this can be directly expressed as a Boolean formula as shown in Figure 3.6. Using De Morgan's law this can also be expressed as one of two neighbouring cells must be white. Note that the colour black (■) is the opposite of white (⊡) and thus $\neg$■ $\equiv$ ⊡. This constraint of course needs to be added for every two adjacent cells, both horizontally and vertically.



$$\neg(\blacksquare_c \wedge \blacksquare_{c'}) \equiv \boxdot_c \vee \boxdot_{c'}$$

Figure 3.6: For two neighbouring cells c and c' it must hold that not both of them are black.

3.3.2 Number Constraints

This constraint is almost the same as its counterpart for *Oh no* with the minor difference, that in *Kurodoko* a cell also sees itself. Hence, decreasing every number by one, i.e. not counting the numbered cell itself, eliminates this difference. Therefore the presented encodings for *Oh no* can simply be adjusted to use $top+right+bottom+left = number-1$ instead of $top + right + bottom + left = number$. This change allows for the enumeration and the encoding of *seeing* to be used for *Kurodoko* too. Note that in a *Kurodoko* puzzle only cells with numbers can be hints and thus black (■) cells do not need to be considered in the improved enumeration.

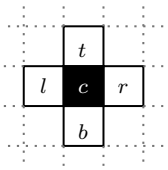
3.3.3 All White Cells are Connected

This rule demands that all of the white cells are connected to each other horizontally or vertically and thus all white cells form a continuous area. As for the number constraint, different approaches to encode this rule are presented.

Connectivity through Reachability

One way to ensure that all white cells are connected, is to ensure that every white cell is connected to every other white cell.

We refer to two cells c and c' being connected as c can reach c' and write this as $c \leftrightarrow c'$. The relation $\leftrightarrow$ is both symmetric ($x \leftrightarrow y \iff y \leftrightarrow x$) and transitive ($x \leftrightarrow y \wedge y \leftrightarrow z \Rightarrow x \leftrightarrow z$). Since reachability is transitive, it is not necessary to check this for every two white cells in order to demand that all white cells can reach each other. White cells can only be disconnected by black cells. Thus, it suffices to demand that a black cell does not disconnect any white cells. Since all neighbours of a black cell must be white, this is satisfied if all neighbours of the black cell can still reach each other as shown in Figure 3.7.



$$\blacksquare_c \Rightarrow t \leftrightarrow r \wedge t \leftrightarrow l \wedge r \leftrightarrow b$$

Figure 3.7: If a cell c is black, then all its neighbours must still be reachable from each other. Note that transitivity implies $l \leftrightarrow b$, $l \leftrightarrow r$ and $t \leftrightarrow b$.

To complete the encoding, the reachability relation $\leftrightarrow$ for white cells also needs to be encoded for the solver. The following encoding, proposed by my supervisor, encodes the relation over the existence of paths between two cells. Cells are connected if and only if there exists a path consisting of only white cells between them. Two cells c_1 and c_2 are reachable with a path of length 2^0, if and only if they are both white and neighbours:

$$c_1 \leftrightarrow_{2^0} c_2 \iff \begin{cases} \boxdot_{c_1} \wedge \boxdot_{c_2} & c_1 \text{ and } c_2 \text{ are neighbours} \\ \bot & \text{otherwise.} \end{cases}$$

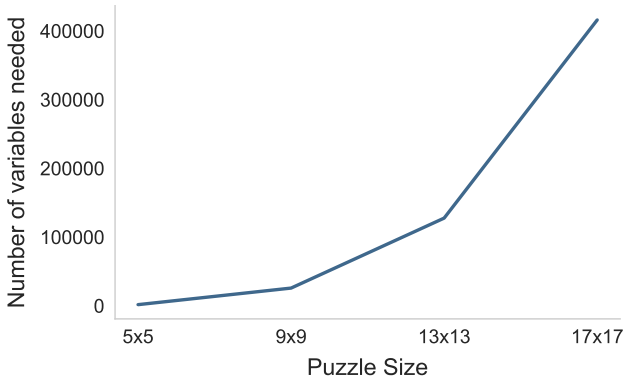
For paths which are longer, the transitivity of reachability is used. Two cells c_1 and c_2 are reachable with a path of length 2^l, if and only if there exists a third cell connecting them which is reachable by a shorter path of length 2^{l-1} by both c_1 and c_2:

$$c_1 \leftrightarrow_{2^l} c_2 \iff \exists c. c_1 \leftrightarrow_{2^{l-1}} c \wedge c \leftrightarrow_{2^{l-1}} c_2$$

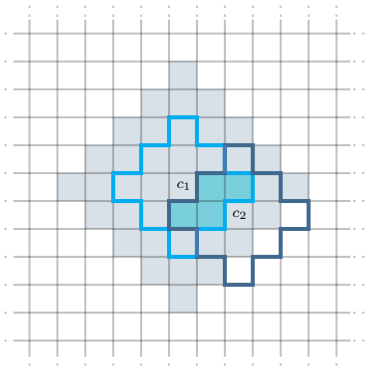
Here the cell c must be reachable by both c_1 and c_2 within 2^{l-1} steps. Figure 3.8(b) shows an example of all candidates for the cell c where $l = 2$.

By repeatedly adding these reachability constraints for increasing lengths, reachability for arbitrarily long finite paths can be encoded. The longest possible path of white cells in a given puzzle instance with R rows and C columns has a length of at most $R \times C$ cells. While the longest path is a lot shorter on average, the longest path in some puzzles is very close to this upper limit. Therefore the encoding needs to be added to the solver for paths with a length of up to $2^{l_{max}}$, where $l_{max} = \lceil \log_2(R \times C) \rceil$. The needed reachability relation $\leftrightarrow$ then just is the relation for the longest needed paths:

$$c_1 \leftrightarrow c_2 \equiv c_1 \leftrightarrow_{2^{l_{max}}} c_2$$



(a) The number of variables to encode reachability does not scale well.



(b) For each path between the cells c_1 and c_2 with a length of at most 4, there are multiple cells which could be on the path.

Figure 3.8: Limitations of this approach.

This encoding comes with the drawback that the number of additional variables and clauses needed to encode the reachability constraints does not scale well. In a puzzle with R rows and C columns, there are $\binom{R\times C}{2}$ ways to choose two different cells. This is the number of paths which need to be encoded $\lceil \log_2(R \times C) \rceil + 1$ times (one additional time for paths of length 2^0). Thus $\binom{R\times C}{2} \times (\lceil \log_2(R \times C) \rceil + 1)$ additional variables are needed. Figure 3.8(a) shows the increasing number of required variables for different puzzle sizes. For a 5×5 grid already 1800 additional variables are needed.

The number of clauses required scales even worse, since multiple clauses are needed to cover every potential cell c in the formula $c_1 \leftrightarrow_{2^l} c_2 \iff \exists c. c_1 \leftrightarrow_{2^{l-1}} c \wedge c \leftrightarrow_{2^{l-1}} c_2$. Cell c must be in the intersection of the cells reachable within 2^{l-1} steps from both c_1 and c_2. The relation for the cells which are definitely out of reach can simply be defined as false. While there are only four candidates (coloured in turquoise) in the example shown in Figure 3.8(b), this needs to be repeated for all cells potentially reachable within four steps which are coloured in blue-gray.

Simple Testing

Another straightforward way to add this constraint is to first have the solver find a model for the puzzle without any constraint regarding the connectivity of the white cells. Then it is checked if all white cells are connected to each other using depth first search.

If all white cells have been visited during the search, then all white cells are connected and a valid solution was found.

Otherwise, there must be some black cells which keep the white cells from being connected. Thus, in order for the white cells to be connected, not all cells which are black in the model can be black in the actual solution. Therefore it must be true, that in the solution, at least one of the black cells in the model must be white. This can then be added as an additional constraint to the solver, which in turn is restarted as shown in Figure 3.9. This procedure is repeated until the solver returns that there is no solution, or until all white cells are connected and a solution is found.

However, if there are many combinations where all rules but the white connectivity constraint are fulfilled, this approach is less effective. For example for puzzle Nr. 53 in [1] there exist more than 300000 combinations where all but the connectivity constraint are satisfied, yet only in one of those all white cells are connected. Two of the solution candidates, which violate the connectivity constraint, and the unique solution are shown in Figure 3.10. Note that the second solution candidate has a single black cell less, but

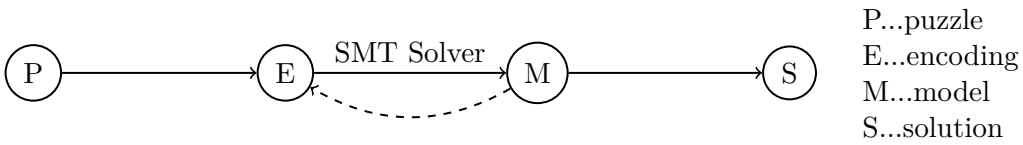
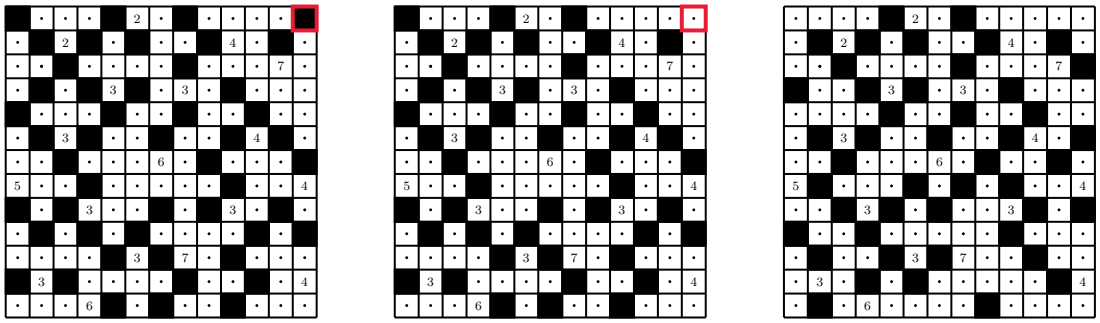


Figure 3.9: By repeatedly calling the solver with updated constraints, there is no need for a direct encoding of the rule.



(a) A solution candidate. (b) Another candidate. (c) The unique solution.

Figure 3.10: Some of the over 300000 solution candidates for the 53rd Nikoli puzzle.

the same violations of the connectivity rule.

A better Representation

While the previously introduced approaches work well enough for small puzzles, they do not scale well with increasing puzzle size and for puzzles where the connectivity rule is important to the solving process. Looking at the Figures 3.10(a) and 3.10(b), it is clear that just demanding some of the black cells to be white is insufficient. This does not necessarily prevent solution candidates where the white cells are disconnected.

An observation is that the same violations of the connectivity occur in multiple of these solution candidates. Therefore an improvement would be to first find the violations and then to ensure that these violations cannot occur in the next solution candidate. This way each iteration of the solving process would make the white cells more connected, converging faster towards a connected solution than the previous approach. In order for this to work, there needs to be a way to find all different ways in which the connectivity constraint can be violated. Another important observation is, that all the violations are either cycles of black cells or border to border paths of diagonally adjacent black cells.

Lemma 3.2. *The absence of cycles and border to border connections of diagonally adjacent black cells implies that all white cells are connected.*

Proof. Assume that not all white cells are connected, but there are no border to border paths and no cycles of diagonally adjacent black cells. Then there exist two cells c_1 and c_2 such that there is no path of white cells between them. This means that black cells must block the path at some point.

If one follows the blocking black cells in one direction, one either ends up at the same black cell or at the border. When the same cell is revisited, a cycle of diagonally adjacent black cells has been found, which contradicts the assumption that there are no black cycles.

In the other case where the border was reached, there cannot be a path between c_1 and c_2 on this side of the first encountered black cell, as all cells are blocked by black cells

until the border was reached. The black cells must form a diagonally connected path, as there would be a path between c_1 and c_2 otherwise. Thus, if there cannot be a path on this side of the first encountered black cell, there may still be one on the other side. If one now follows the black cells in the other direction from this black cell, one must again end up at the border of the grid. If the black cells are not continuous and one does not end up at the border, there is a hole and thus the black cells can be circumnavigated, indicating a path between c_1 and c_2. There also cannot be a cycle, since then the border would have never been reached in the first place. Both of these cases contradict the assumption and therefore one must end up at the border. Hence there exists a border to border path of black cells, which also contradicts the assumptions.

Hence the absence of cycles and border to border connections of diagonally adjacent black cells is sufficient and implies that all white cells are connected. □

Using this knowledge, one can improve upon the previous approach using depth first search to check if the white cells are connected. Once a solution candidate has been found, it is possible to find violations to the connectivity rule by using depth first search on the black cells. For each connected component of the black cells depth first search is called. If there is a cycle of adjacent black cells, then the depth first search tree contains a back edge. If two different black border cells are in the search tree, then there must be a border to border path of black cells.

Note that by keeping track of the parent of a node in the depth first search tree, both the black cells forming a cycle and the black cells on the border to border path can be collected. Since the path between the border cells is not necessarily the shortest path and dependent on the search order, another option is to use the A* search algorithm to find a shorter path between the black border cells. Note that both approaches find a single and thus not necessarily all paths between two black border cells. For each violation, then a clause can be added, demanding that at least one of the black cells in the violation must be white. This is shown in Figure 3.11. Note that the border cells have a blue outline in the depth first search trees.

The depth first search trees also gives rise to an SMT encoding, without the need of checking the solution candidate and then possibly restarting the solver with additional constraints. If there is no border to border connection of black cells, then the search tree

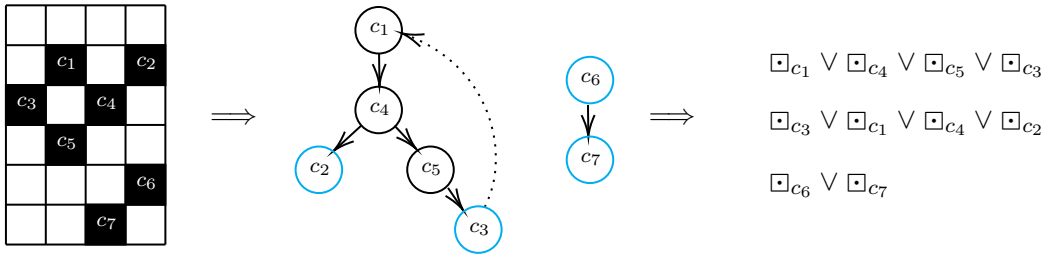
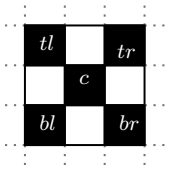


Figure 3.11: A solution candidate where the white cells are not connected, the depth first search trees and clauses preventing these violations.



$$at\_most\_one(\{tl \succ c,\ tr \succ c,\ bl \succ c,\ br \succ c\})$$

Figure 3.12: Every cell may only have one parent.

has at most one border cell. Additionally, if there are no back edges in the tree, then there are no cycles of black cells.

This can be encoded for the SMT solver using a parent-child relation $\succ$. For two diagonally adjacent cells c_1 and c_2, $c_1 \succ c_2$ denotes that c_1 is the parent of c_2. Each cell has at most one parent and if two diagonally adjacent cells are black, there must exist a parent-child relation between them. Additionally, between two nodes there can only be one parent-child relation, i.e. only either $c_1 \succ c_2$ or $c_1 \prec c_2$ can be true.

Figure 3.12 shows the constraint which ensures that each cell only has at most one parent. For a set V of boolean variables, the *at_most_one* constraint can be encoded as follows:

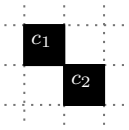
$$at\_most\_one(V) := \bigwedge_{\substack{x,y \in V \\ x \neq y}} \neg x \vee \neg y$$

The clauses needed to ensure that there is a parent-child relation between diagonally adjacent black cells are presented in Figure 3.13.

Moreover black cells on the border of the puzzle grid are treated special. Those cells cannot have a parent, as shown in Figure 3.14. This constraint already prohibits border to border black paths, as those would have two root nodes and therefore one black cell along the path would have to have two parent nodes, as displayed in Figure 3.15(a). However, this violates the *at_most_one* constraint for the parent-child relation. But cycles would still be possible if the black cells are not connected to a border cell. This is illustrated by Figure 3.15(b).

This can be circumvented by also introducing an integer variable for the depth of a cell in the tree as shown in Figure 3.15(c).

Now additional rules can be added to ensure that the depth is strictly increasing, which also prevents cycles. For a border cell c, $depth_c = 0$ because border cells have no parent. The depth for all other cells c_1 and c_2 is determined by the parent relation. If $c_1 \succ c_2$ then $depth_{c_2} = depth_{c_1} + 1$.



$$\blacksquare_{c_1} \wedge \blacksquare_{c_2} \implies c_1 \succ c_2 \vee c_1 \prec c_2$$

$$at\_most\_one(\{c_1 \succ c_2,\ c_1 \prec c_2\})$$

Figure 3.13: If two diagonally adjacent cells are black, then there must be a parent-child relation between them.

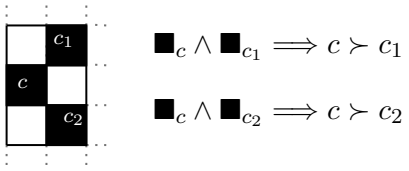
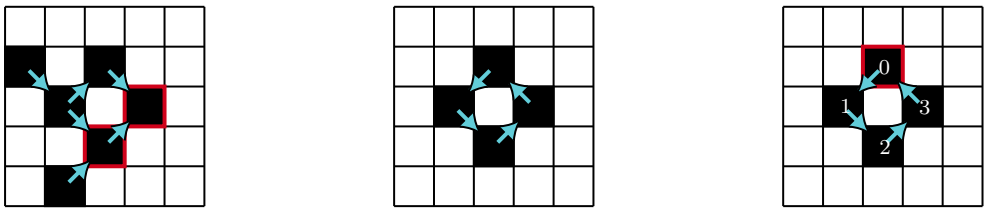


Figure 3.14: Black cells on the border cannot have a parent. Thus, any diagonally adjacent black cells are children of the black border cell.



(a) The black cells cause conflicts.

(b) Without depth, cycles can occur.

(c) Depth prevents cycles.

Figure 3.15: The connectivity rule can be encoded using a parent-child relation (blue arrows) together with depth.

Effectively, this encoding ensures that each connected component of black cells is a rooted tree with no two border cells. Since cycles and border to border paths of diagonally adjacent black cells are prevented, the rule that all white cells must be connected is satisfied.

For puzzles where the number of rows or columns is 1, both approaches using depth first search and rooted trees do not fully capture Lemma 3.2. If one dimension of the puzzle is 1, then every black cell is a border to border path of black cells and disconnects the white cells on both sides as shown in Figure 3.16(a). But both encodings only detect border to border connections between two distinct border cells.

Since each black cell is a border to border path, black cells cannot have white cells on both sides. This is only possible, if the black cell is either in the first or last cell as shown in Figure 3.16(b). All other cells cannot be black and thus must be white. Since only two cells would be undecided, resulting in only 4 different solution candidates, these puzzles would be rather boring for human solvers. Nevertheless a simple case distinction in the solver can be used to solve puzzles where one dimension is 1 efficiently, by iterating over the 4 options and checking the other rules.

(a) The black cell disconnects the white cells.

(b) None of the gray cells can be black.

Figure 3.16: If one dimension is 1, only the outermost cells of the puzzle can be black.

A simple Heuristic

The connectivity rule is the counterpart to the rule in *Oh no* that each white cell needs to see at least one other white cell. Therefore the encoding for the rule in *Oh no* could be used as a heuristic, to ensure that each white cell is connected to at least one other cell. While this is far from sufficient to demand that all white cells are connected, considering this as an additional constraint is not much extra work for the solver and does help to rule out at least some black cell positions without needing to look at the whole puzzle, which can be expensive.

This constraint would directly conflict with numbered cells with the value 1, since those cells cannot see any other cells. Luckily, because of the connectivity constraint the number 1 cannot appear in puzzles where both the number of rows and columns is greater than 1. If the number 1 would be in such a puzzle, then the puzzle has no solution because either the cell would see more than 1 cell or the connectivity rule is violated. Not considering rotation, there are only 3 rectangular puzzles that can contain the number 1. These are shown in Figure 3.17.

Figure 3.17: All valid puzzles with the number 1.

3.4 Performance Evaluation

In this section the performance of the previously introduced solvers is evaluated empirically. There are already many *Kurodoko* solvers and puzzles of different sizes freely available. On the other hand, at the time of writing this does not hold for *Oh no*.[3] Therefore this analysis focuses on *Kurodoko*, the previously presented encodings and other solvers found on the internet. Nevertheless, the solver performance for *Oh no* should be comparable when considering the solvers using the iterative connectivity approaches together with the connectivity heuristic. These solvers do not directly add constraints for the connectivity rule of *Kurodoko*, but incorporate the *Oh no* rule that each white cell can see at least one other white cell.

3.4.1 Solvers Included in the Measurements

In addition to the solving approaches presented in Chapter 3, other freely available solvers were included in the measurements. For both the number constraints and the connectivity rule of *Kurodoko* multiple encodings were presented. The list of included solvers is as follows:

- For the encodings presented in this thesis, multiple combinations are included in the benchmarks. The included encoding approaches are abbreviated as follows:
  - *IE*: The improved number enumeration.
  - *V*: The number constraint is enforced by encoding visibility.
  - *DFS/A**: The improved iterative connectivity approaches which add additional constraints for each violation. This approach was implemented using depth first search and utilising the A* search algorithm.
  - *RT*: The direct encoding of black cells as rooted trees.
  - *H*: The connectivity heuristic inspired by *Oh no*, which ensures that each white cell has at least one white neighbour.

  The following combinations are included:

  IE,RT *IE,DFS* *IE,H,A** *IE,H,DFS* *V,H,RT*

- *Bruteforce*:[4] A solver written in C++ which utilises a mix between backtracking and brute forcing. The solver was found on GitHub and was programmed by the GitHub user AvaLovelace1. For the black adjacency and the connectivity rule backtracking is used, whereas brute forcing is used for checking the number constraint. The solver recursively colours all cells, while making sure that no black cells are neighbouring and all white cells are connected. If one of those rules is violated, the solver backtracks and changes the colour of the last cell. By repeating

[3] While there are puzzles available at [10], these are limited to at most 9 by 9 grids.
[4] Found at [18].

this process the solver iterates over all coloured grids, where all rules but the number constraint are satisfied. For each such colouring, the solver then checks the number constraint, i.e., it is checked if every numbered cell can see the right amount of cells. If the number constraints are satisfied a solution is found, otherwise the solver continues until the next solution candidate is found.

- *LP*:[5] Another solver was written in Python by the GitHub user SaitoTsutomu. This solver first encodes the rules into constraints which can then be solved using a linear programming solver. For the connectivity constraint, the solver uses union find to check if a solution candidate satisfies the rule and then restarts the solver with additional constraints until a solution is found.

- *Prolog*:[6] Another solver which first encodes the rules and then utilises a solver written in Prolog. This solver encodes all rules and then uses the Prolog clp(fd) constraint programming library to find a model for the puzzle.

- *Backtrack*:[7] A backtracking solver written by Tarik Hoshan, which tries to assign four black cells around each numbered cell, such that no rule is violated, until the puzzle is solved. If a rule is violated, the solver backtracks.

- *Human-based*:[8] This solver was written by the GitHub user 3982myamya and implements some solving strategies a human solver would use too. These include colouring the adjacent cells of black cells white and eliminating some options for numbered cells which are limited in one direction by a black cell or the border. After each step the solver checks if all cells are still connected. If no more progress can be made using these rules, the solver begins to guess. If that too does not lead to progress, the solver recursively guesses again until a recursion depth of 3 is reached.

3.4.2 Method and Environment

Method

The main goal of the performance evaluation is to measure the runtime complexity of the solvers with regard to increasing input sizes on different puzzle types. Since measuring the performance of the solvers should complete within a reasonable amount of time, steps were taken to shorten the process while retaining acceptable accuracy.

First a given puzzle is solved once with a timeout of 15 minutes. If the solver manages to solve the puzzle within a cutoff time of 2 minutes, the solving process in repeated another 5 times.

Averaging these 6 results should avoid runtime effects from influencing the results. This approach should deliver precise results for fast solves where the influence of runtime effects

[5] Found at [23].
[6] Found at [24].
[7] Found at [8].
[8] Found at [26].

may still be noticeable. Another notable limitation of the measured data is that the solvers use different programming languages and thus the differences in speed of the languages itself may have a noticeable effect on the solving times. Because of this, the solving times for smaller and easier puzzles may be misleading. The faster algorithm implemented in a generally slower language may perform worse than an inefficient algorithm implemented in a faster language. However, for larger and more difficult puzzles this should not be a problem.

The puzzles included in the benchmark are

- all of the 99 Nikoli puzzles from the official book ([1]),
- the puzzles in the hardest three levels of difficulty which are available at the Janko web page [9],
- and empty puzzles of different sizes,

Environment

In order to make the environment reproducible, the measurements were executed in a Docker [13] container.[9] Since the solvers are implemented in different programming languages, a Python script was written to benchmark the solvers. This script first compiles all solvers, then transforms the puzzle into the needed representation for the solver, measures the solving time for the puzzle and finally stores the result in a database. Note that different approaches and interfaces were needed in order to communicate with other programming languages than Python, which may have lead to some minor overhead.

3.4.3 Results

First, let us consider only the solvers presented in this chapter. Figure 3.18 shows the average solving time for the different puzzle sizes contained in both the official *Kurodoko* book ([1]) and the Janko web page ([9]). In the figure, the average times are shown on a linear scale and the first minute is also shown with a logarithmic scale. Note that timeouts are not included in the average, i.e. the plots only take successfully solved puzzles into account. The figure shows that the direct encoding of visibility (V) performs and scales considerably worse than enumerating all possible number layouts (IE). Especially for the larger puzzles this becomes apparent. For the connectivity encoding there also is a visible performance difference between the iterative approaches (DFS and $A*$) and the encoding using rooted trees (RT). Moreover, the difference between *IE,H,DFS* and *IE,DFS* solvers indicates that the connectivity heuristic (H) does indeed improve the solving speed. Since the solvers are otherwise identical, the performance difference is caused by the rule that all white cells must be connected. The different implementations of the iterative

[9] The measurements were made in a Docker container with 2 CPUs, 4GB RAM, 2GB Swap and 64GB of disk space. This container ran on a Windows system with an Intel i5-4670K processor (4 physical cores and a base clock of 3.4Ghz) and 16GB RAM.

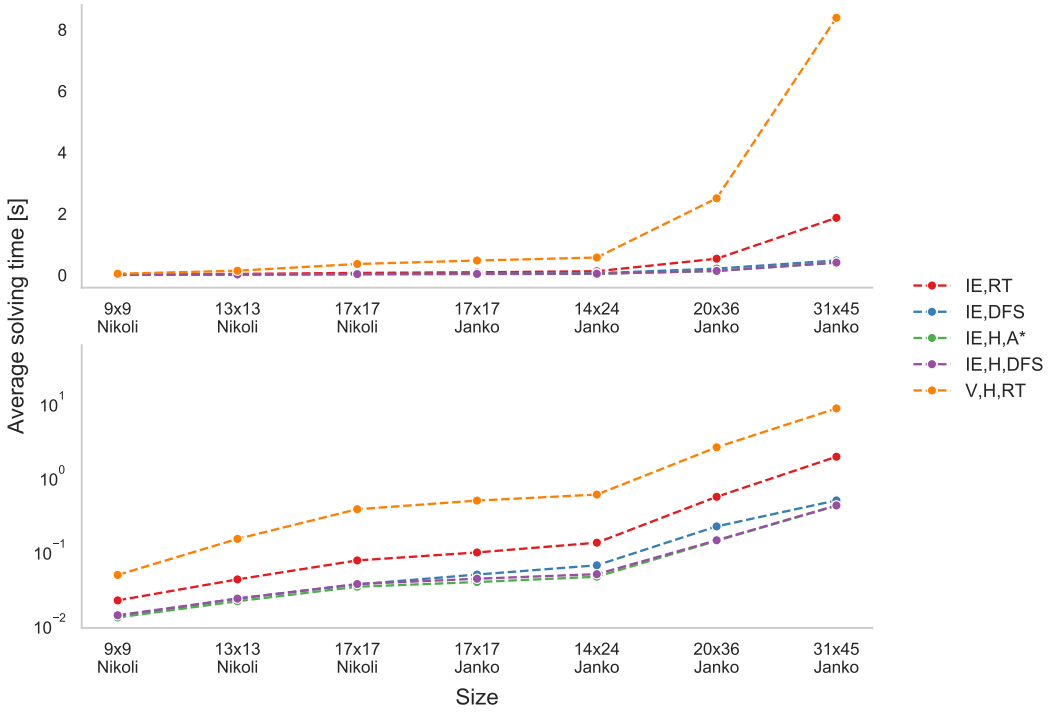


Figure 3.18: The average solving time per puzzle size for the 99 Nikoli and the 55 hardest Janko puzzles.

approach (*DFS* and *A**) performed nearly identically, indicating that the puzzle sizes were too small to observe noticeable differences between the variants. In other words, the connectivity rule plays an important role for some of the puzzles. Another notable observation is that the solvers using the iterative connectivity approaches consistently outperform the other solvers. Overall, the solvers handle the puzzles rather well and all of the included solvers managed to solve even the largest puzzles within 10 seconds.

The times for the other solvers together with the *IE,H,A** solver are shown in Figure 3.19. Again, the times are the average of all puzzles in the given size and without considering the timeouts. Thus, the absence of a marker or the line indicates that a solver was not able to solve any puzzle of this size within the time limit. Moreover, the second plot shows the first minute with a logarithmic scale. The figure clearly shows that larger puzzles are challenging for the solvers, as many of the solvers were not able to solve any fields with more than 17×17 cells. For example, the *Human-based* solver is very quick at solving the smaller puzzles, but fails to solve any of the larger puzzles. It can also be seen, that the *LP* solver is slower for the smaller puzzles but scales better than some other puzzles. The probable cause for this is that Python generally is slower than compiled languages such as C++ [6]. Moreover, there seems to be a discrepancy between

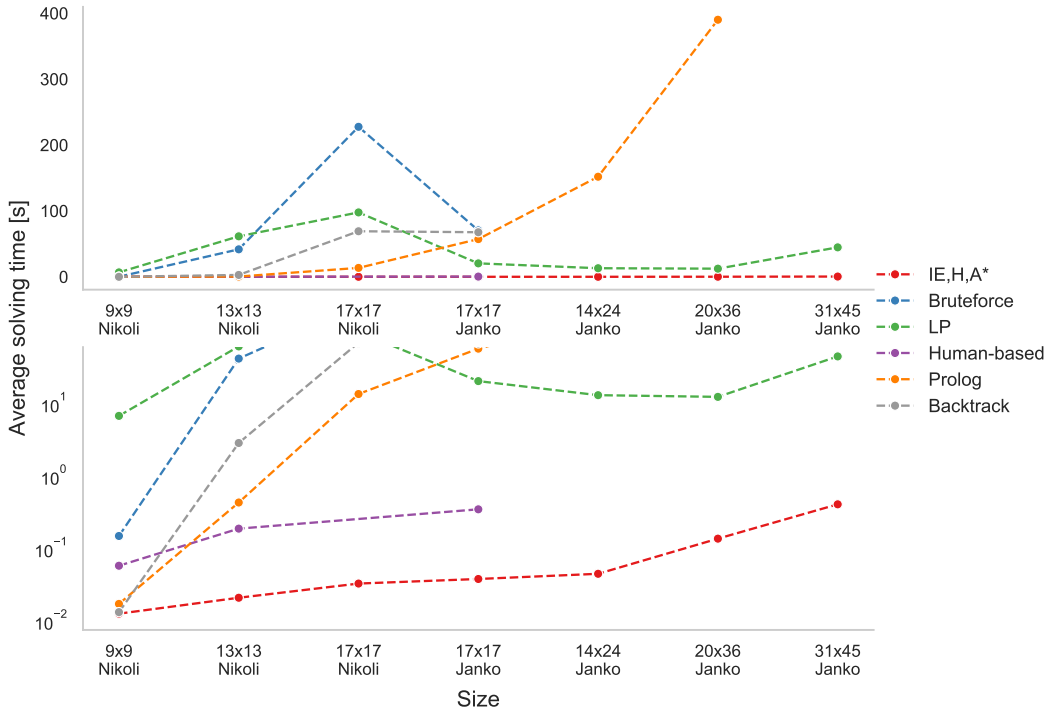


Figure 3.19: The average solving time per puzzle size, this time including the other solvers.

the difficulty levels of the Nikoli puzzles and the Janko puzzles. The *Bruteforce* and *LP* solvers need less time for the Janko puzzles of size 17×17 than for the Nikoli puzzles of the same size. While it looks like the *Prolog* solver performed better on the Nikoli puzzles of size 17×17, the plot is misleading because only successful solves are included.

Table 3.1 shows the individual times for some of the puzzles and how many puzzles could not be solved by each solver. The table shows that the solvers using the rooted tree encoding (RT) for the connectivity rule are not as fast as the iterative approaches (DFS and $A*$), especially when solving the larger puzzles. The performance difference between the direct encoding of visibility (V) and the improved enumeration (IE) is even bigger. Nevertheless, these solvers still managed to solve all of the Nikoli and Janko puzzles. Interestingly, the *Bruteforce* solver was able to solve some of the Nikoli puzzles where other solvers failed to find solutions. Although the *Human-based* solver solved the puzzles very fast, this particular solver only managed to solve less than half of the puzzles. Considering the number of timeouts, the LP solver did comparatively well since most of the other included solvers were not able to solve the larger puzzles efficiently. Nevertheless, the solvers presented in this chapter performed a lot better than any of the other solvers.

	Nikoli 24 EASY 13×13	Nikoli 53 MEDIUM 13×13	Nikoli 89 HARD 17×17	Janko 319 hard 20×36	Janko 320 hard 31×45	Timeouts Nikoli	Timeouts Janko
IE,RT	0.05s	0.05s	0.07s	0.57s	1.91s	-	-
IE,DFS	0.02s	0.03s	0.03s	0.25s	0.43s	-	-
IE,H,A*	0.02s	0.02s	0.03s	0.15s	0.38s	-	-
IE,H,DFS	0.02s	0.03s	0.03s	0.15s	0.38s	-	-
V,H,RT	0.18s	0.14s	0.32s	2.46s	8.01s	-	-
Bruteforce	13.64s	70.36s	67.54s	-	-	14	53
LP	-	-	1.50s	2.39s	40.35s	25	14
Human-based	-	-	-	-	-	53	53
Prolog	-	-	-	-	-	59	31
Backtrack	35.10s	-	10.59s	-	-	7	41

Table 3.1: A selection of puzzles where some solvers struggled and the number of times each solver failed to solve the puzzle within the time limit for both the 99 Nikoli puzzles ([1]) and the 55 puzzles from Janko ([9]).

Figure 3.20 shows the results for the empty puzzles of different sizes. Since the puzzles contain no numbers, these puzzles only require that no black cells are adjacent to each other and all white cells are connected. Hence, by simply colouring all cells white, a solution can be found instantly. However, as most of the solvers utilise constraint programming approaches, the constraint solver may explore other options, where black cells disconnect the white cells, first. While this benchmark is artificial and bears no resemblance to puzzles which are intended for humans, it does indicate how well the solvers handle the all white cells must be connected rule, which often caused performance outliers when measuring the solving performance of the Nikoli and Janko puzzles.

The *Human-based* solver is not included in the plot since it did not manage to solve any of the empty puzzles due to its branching depth limit. The second plot in the figure shows the first five seconds in more detail. Note that gaps indicate that a solver was not able to solve the puzzle within the time limit. Since the *Bruteforce* solver simply tries every combination of black cells and backtracks if a rule is violated, the solver is a good baseline in order to assess the performance of the other solvers. Additionally, only the *Backtrack* solver incorporates this insight and simply colours all cells white. Therefore, one would expect the performance of most solvers to be between those two.

However, many of the solvers do not scale well and perform worse than the *Bruteforce* solver. Especially the *LP* solver is only able to solve the smallest empty puzzles within the time limit. The *Prolog* solver also scales worse than the *Bruteforce* solver. The *IE,DFS* solver manages some larger puzzles, but also does not scale. Since the *IE,H,DFS* approach scaled a lot better, this indicates that the connectivity heuristic does improve the solving performance for the iterative connectivity approaches. However, the heuristic seems to have little impact on the connectivity encoding using rooted trees (*RT*). Another consequence of the absence of numbered hints is that the enumeration (*IE*) and visibility

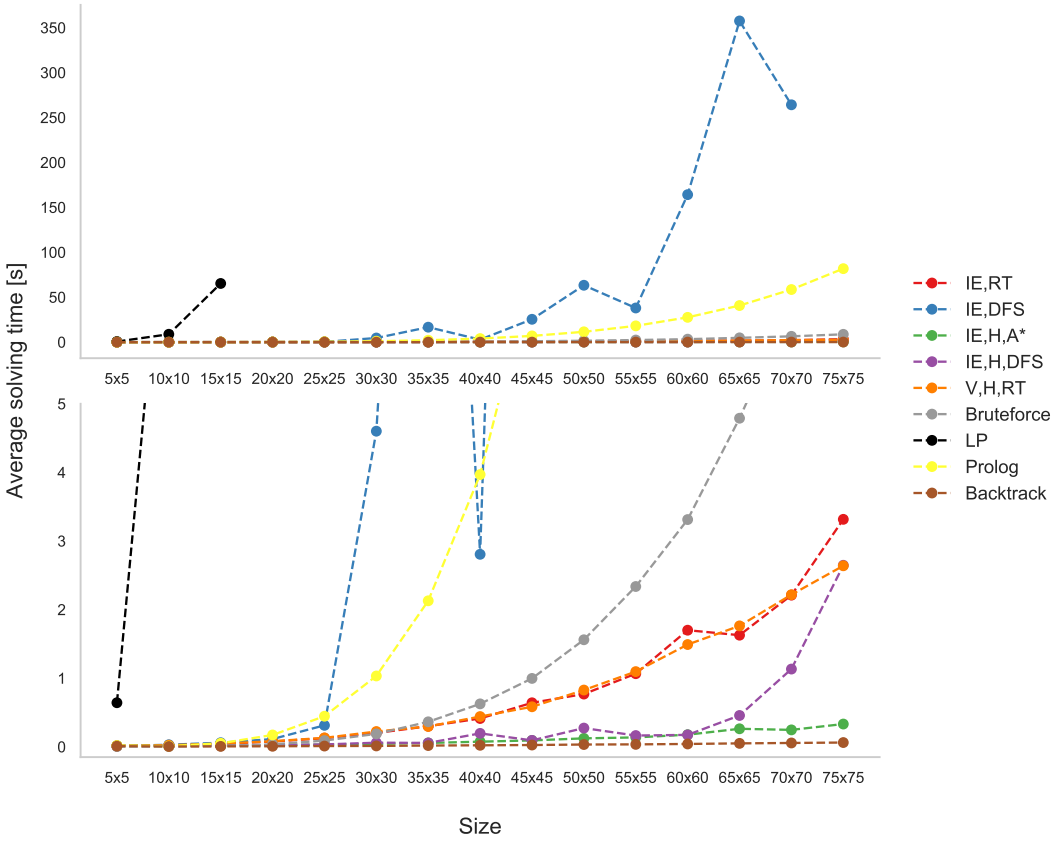


Figure 3.20: The solving times for empty puzzles of different sizes.

(V) approaches both add no constraints and effectively are the same. Moreover, the iterative connectivity approach utilising A* search in order to find black border to border connections does perform better and more consistently than the solvers using depth first search ($IE,H,A*$ vs IE,H,DFS).

4 Generators

In this section an approach to automatically generate both *Kurodoko* and *Oh no* puzzles is presented. Since both puzzles are very similar in their rule set, the same approach can be used for both puzzles. The generated puzzles should have a unique solution and there should be the possibility to adjust the perceived difficulty of the generated puzzles.

4.1 Generator Outline

The overall approach to generating a new puzzle is quite simple and can be applied to both puzzles:

> **Step 1:** First a solution is generated by randomly colouring some grid cells black, such that no rules are violated.
>
> **Step 2:** Then the solution is turned into a valid puzzle with a single unique solution. This is done by adding numbered hints to every empty cell.
>
> **Step 3:** Finally hints are removed while keeping the puzzle unique.

In the following subsections each step is explained in more detail and with examples.

4.1.1 Generating a Solution

In this step some of the cells in the grid are randomly coloured black (■), such that all rules are satisfied. Because of the differences in the rules of *Kurodoko* and *Oh no*, different approaches for generating such a solution are needed.

- For *Kurodoko* this means that no black cells may be neighbours and that all white cells must be connected. Because black cells cannot be adjacent, at most every second cell can be black. The connectivity constraint limits the number of black cells even further. Algorithm 2 shows the procedure to generate a valid colouring of a grid. Although the rules already restrict the maximum number of black cells a puzzle can have, the algorithm allows to limit the number of black cells. For checking if all white cells are connected depth first search over all white cells can be used, checking if the number of white cells is equal to the number of cells visited in the search.

Algorithm 2: Pseudocode for generating a solution for *Kurodoko*

```
choose a random number of black (■) cells
randomly shuffle the cells of the puzzle
for each cell of the shuffled cells do
    if the targeted number of black cells has been reached then
        break
    if the cell has no adjacent black (■) cell then
        make the cell black (■)
        if not all white cells are connected then
            make the cell empty again
```

- For *Oh no* the rules state that each white cell must see at least one other white cell. Additionally, *Oh no* puzzles where every cell is black would be boring, thus the number of black cells should be limited for *Oh no*.[1]

Algorithm 3: Pseudocode for generating a solution for *Oh no*

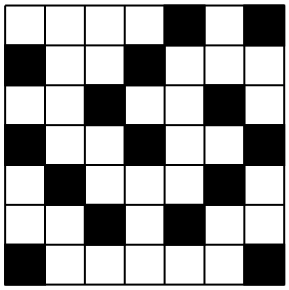
```
choose a random number of black (■) cells
randomly shuffle the cells of the puzzle
for each cell of the shuffled cells do
    if the targeted number of black cells has been reached then
        break
    make the cell black (■)
    if not all neighbouring white cells can see another white cell then
        make the cell empty again
```

Figure 4.1 shows solutions generated using this approach for both *Kurodoko* and *Oh no*. In these examples the differences between the positions where black cells are allowed become apparent.

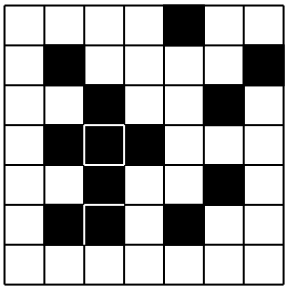
4.1.2 Turning the Solution into a Puzzle

The second step then is to calculate the number of seen cells for every white cell in the previously generated solution. These are then added to the grid as numbered hints. This would already be a valid *Oh no* puzzle since *Oh no* allows black cells to be hints. For

[1] The implementation uses normally distributed random numbers to limit the number of black cells for both *Kurodoko* and *Oh no*.



(a) A generated *Kurodoko* solution.



(b) A valid colouring for *Oh no*.

Figure 4.1: Example solutions which were generated for each of the puzzle types.

Kurodoko the black cells need to be replaced with empty cells. Hence after this step, already a valid puzzle was generated. However since almost every cell is already filled in this would be rather boring to play. Nevertheless this puzzle has a single unique solution. For *Oh no* all cells are already filled in. For *Kurodoko* all white cells are numbered, leaving no options for the positions of the black cells. Figure 4.2 shows the result of applying this step to the generated solutions shown in Figure 4.1.

4.1.3 Making the Puzzle Interesting

In the last step, already filled in cells are randomly replaced by empty (□) cells while making sure that the puzzle stays unique. This is done by first removing the value of a cell, checking if the puzzle is still unique using the solver and then only adding the value back to the puzzle if the puzzle was not unique. In Figure 4.3 the final puzzles of the previous examples are shown.

This simple approach works well for generating both *Kurodoko* and *Oh no* puzzles.

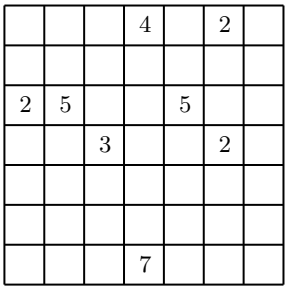
4	7	5	4		2	
	5	3		6	4	4
2	5		2	5		2
	5	3		5	2	
2		4	5	6		2
3	3		3		3	3
	6	5	7	5	6	

(a) The *Kurodoko* puzzle with all numbers.

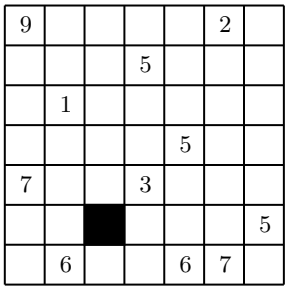
9	3	4	5	■	2	1
6	■	4	5	6	4	■
7	1	■	3	4	■	4
6	■	■	■	5	2	6
7	1	■	3	4	■	4
6	■	■	2	■	2	5
12	6	6	8	6	7	10

(b) The *Oh no* puzzle.

Figure 4.2: After this step, the puzzle is valid and has a single unique solution.



(a) The final *Kurodoko* puzzle.



(b) The final *Oh no* puzzle.

Figure 4.3: The finished puzzles, after the superfluous hints have been removed.

4.2 Approximating the Difficulty

Puzzles where trivial next steps are abundant or where the next step is very hard to deduce are less enjoyable for human solvers to play. However the perceived difficulty of finding the next logical step is very subjective, as some enthusiastic solvers may know lots of solving techniques while others will find some concealed reasonings hard to spot. Because of this, the generator should also be able to adjust the perceived difficulty. However, approximating the difficulty of a puzzle and adjusting the difficulty thereafter is non-trivial.

A simple way to adjust the difficulty of a puzzle is to add or remove hints. Adding more hints decreases the perceived difficulty, whereas puzzles with fewer hints are generally harder. Nonetheless, one drawback of this way of adjusting the difficulty is that there is no possibility of ending up with exactly the wanted difficulty. A puzzle may be too hard without a certain hint and too easy with the hint.

But before the hints can be adjusted, the difficulty first has to be estimated and compared to the target difficulty. While the solver does not directly return a difficulty for a puzzle, the SMT solver can return the statistics of the solving process. These statistics indicate how much work was needed to arrive at a solution for a given puzzle and can be used to estimate the difficulty of the puzzle. Some of the included characteristics are:

- The number of decisions the solver made. This number indicates how many times the solver had to effectively guess the colour of a cell, since there was no logical way to colour more cells.
- How many conflicts appeared, indicating how many times the solver needed to backtrack because of a wrong guess.
- Another attribute directly related to the solving process is the number of propagations the solver has made. A propagation indicates that the colour of a cell was deduced without the need to guess.

While many more characteristics are provided, most of them describe the solving process only indirectly and are not necessarily related to the difficulty of the puzzle. For

example the amount of memory used by the solver relates more to the size of the puzzle and the memory efficiency of the used encoding.

In addition to the statistics returned by the solver, for *Kurodoko* the number of solving iterations is also kept track of and available as a measure of difficulty.

In order to find out which of the statistics correlate to the difficulty, the solver statistics were collected for all of the 99 puzzles in the official Nikoli book.[2] These 99 puzzles are divided in three levels of difficulty, namely easy, medium and hard. Since the difficulty for all of those puzzles is known, the collected statistics can be used to compare the statistics for puzzles with different levels of difficulty. However, in the book, the more difficult puzzles also tend to be the larger puzzles. While larger puzzles generally take longer to solve than smaller ones, larger puzzles are not necessarily more difficult as finding the next move can still be easy. In order to reduce the effect of the puzzle size, the average per cell is used for each attribute.

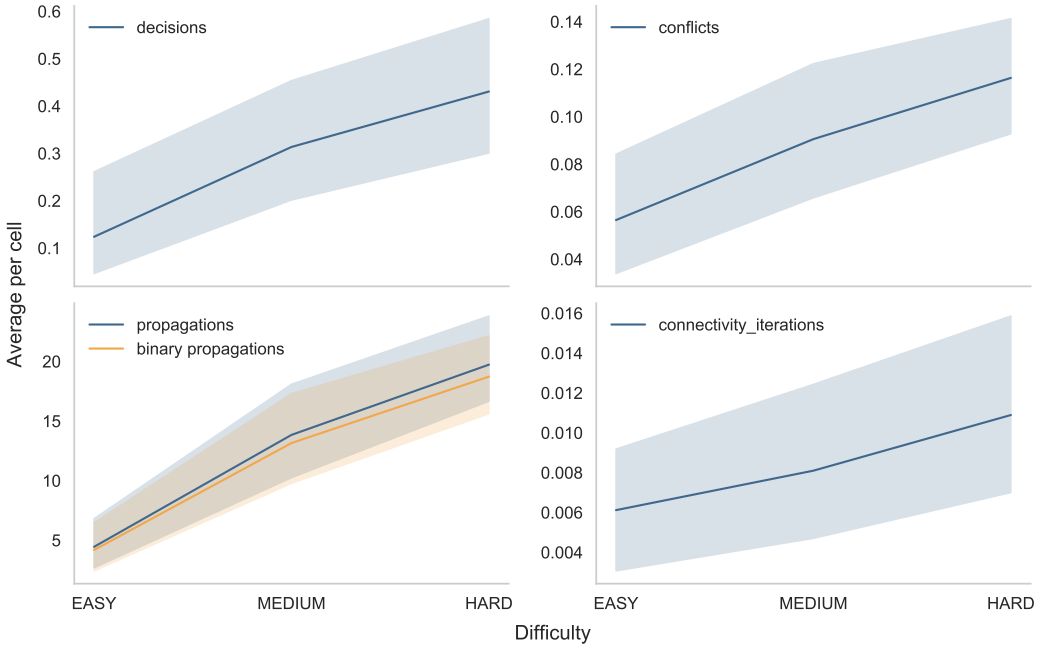


Figure 4.4: Many of the statistics returned by the solver correlate with the perceived difficulty of a puzzle. The thin line is the average for all puzzles of a given difficulty and the transparent area denotes the 95% confidence interval.

Figure 4.4 shows some of the statistics which are related to the difficulty of a puzzle.

[2]The presented statistics are for the *Kurodoko* solver using the improved enumeration for the numbers and the iterative approach for connectivity together with the heuristic inspired by *Oh no* (listed as IE, H, DFS in Section 3.4). In order to get accurate statistics for each puzzle, every puzzle was solved 10 times and the statistics were averaged for each puzzle and every difficulty.

These can then be used to adjust the difficulty during the third step of generating a puzzle. If the statistics without the hint indicate that the difficulty is too high, the hint is simply kept in the puzzle, even tough it would not be needed to ensure a unique solution.

Algorithm 4: Pseudocode for deciding whether a hint stays, if the puzzle would still have a unique solution without it.

```
for each of the correlating statistics do
    check if the average difficulty per cell is higher
    than the average for the difficulty
if the number of statistics indicating that the puzzle is too hard
  exceeds an arbitrary threshold then
    keep the hint in the puzzle
else
    remove the hint
```

Using this approach avoids that the generated puzzles are far too hard for the desired level of difficulty. Since the number of hints even in a hard puzzle tends to be higher than the number of hints required to keep the solution unique, there is no need to ensure that enough hints are removed. Nevertheless, as the perceived difficulty is subjective to the human solver, some puzzles will feel too hard or too easy regardless. Unfortunately there are no puzzles with already known levels of difficulty available for *Oh no*.[3] Because of this, the approach for adjusting the difficulty cannot be applied directly to *Oh no*. However, as the presented solvers for *Kurodoko* and *Oh no* are very similar, the collected statistics for *Kurodoko* can also be used for *Oh no*. For the *Kurodoko* solvers which use the iterative connectivity approach and the heuristic inspired by *Oh no*, the differences should be minimal. Using this approach for *Oh no* allows for adjusting the difficulty of the generated puzzles, although there may be a small shift in the perceived difficulty between *Kurodoko* and *Oh no*.

[3] While [10] provides puzzles, the available puzzles have no assigned level of difficulty.

5 GUI

One of the main goals of the project was to develop a graphical user interface for playing and solving both *Kurodoko* and *Oh no* puzzles. This section will give an overview of the architecture of the application and showcase the main features of the user interface.

5.1 Architecture

In order to make playing the game as convenient as possible, the GUI was implemented as a web application. This allows the application to be platform independent, as there is no need for a special operating system or runtime environment. Hence, only a web browser is needed to play the game.

The application is split into two loosely coupled parts, the user facing frontend and a web server providing access to the generators and solvers. The only interface connecting the components is an API, specifying how the data is exchanged using the Hypertext Transfer Protocol (HTTP) [4].

5.1.1 Frontend

Nowadays more than half of the traffic web pages register comes from mobile devices [25]. Because of this, the web page was made responsive and tries to utilise the aspect ratio of mobile devices. Using the Web Storage API,[1] the played puzzles and user preferences are stored without the need to log in, keeping the barrier to entry low. Furthermore, the application is implemented as a Progressive Web App,[2] which allows users to download and install the site, providing an experience similar to native apps. For the web application the JavaScript frameworks React[3] and Gatsby[4] were used.

5.1.2 Backend

The main purpose of the web server is to provide a GraphQL[5] based API which allows accessing both the solvers and generators. This API is then consumed by the user facing web application. To this end, Python[6] was used for creating the API, while the solvers are implemented in C++.

[1] See [12].
[2] See [21].
[3] Available at [3].
[4] Available at [5].
[5] See [22].
[6] Available at [19].

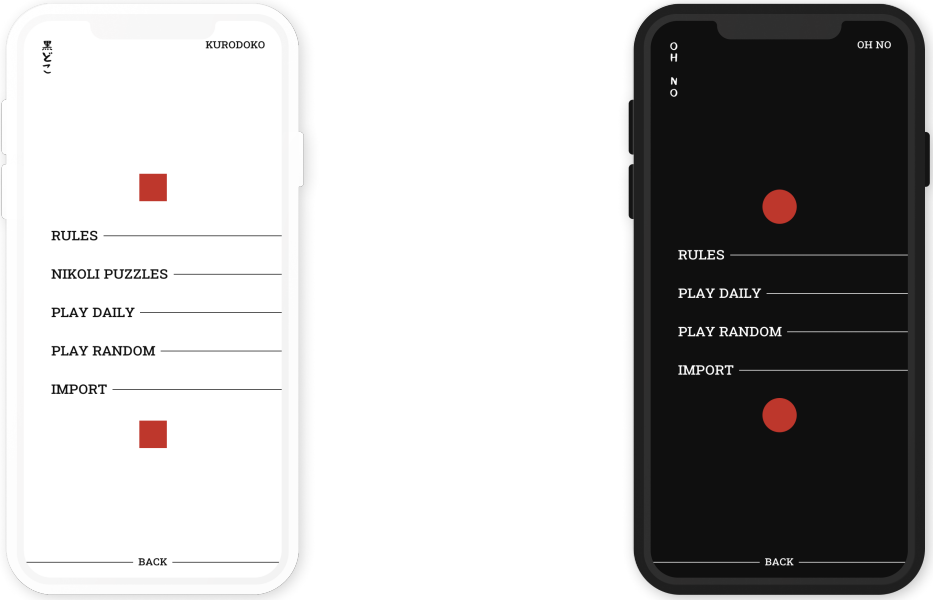
5.2 Features

In order to differentiate between *Kurodoko* and *Oh no*, the application is split into one part for each puzzle. Furthermore, different colour themes were used for each puzzle in the following screenshots. In the main menu a user can choose whether to play *Kurodoko* or *Oh no*. Additionally, the user can also import a puzzle or enter the settings on this page. Once a puzzle is chosen, the menu for the puzzle is shown. In this menu a user has the option to read the rules and to play a random or daily puzzle. For *Kurodoko* there is also the option to play one of the puzzles from [1].

In addition to the rules, there also is an interactive tutorial for each one of the puzzles, teaching the rules and some rudimentary solving techniques. If the user chooses to play a random or daily puzzle, then the size of the puzzle must be selected first. For the 99 Nikoli puzzles from [1], the user has to choose one of the puzzles. Puzzles which were already solved by the user are highlighted in red.

Once the game has started, the user can change the colour of the grid cells to solve the puzzle. To change the colour of a given cell, the user just needs to click (or touch on mobile devices) on the cell to cycle trough the colours. This way the user can mark the cell to be white (⊡), make the cell black (■) or make it empty (□) again. To better differentiate *Oh no* from *Kurodoko*, the grid cells for *Oh no* are round instead of square and have different colours.

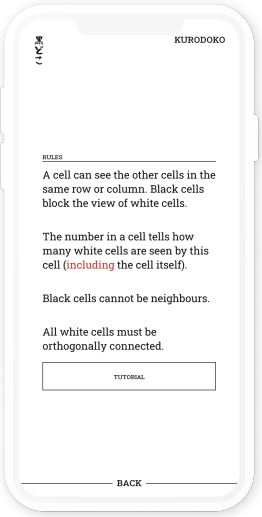
In addition to the puzzle grid, the user also sees a timer showing the elapsed time.



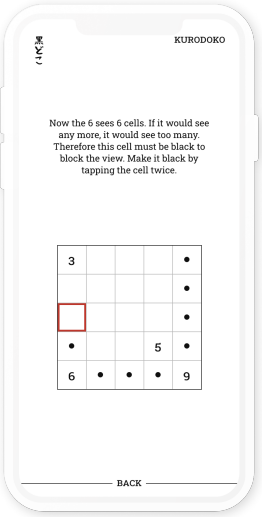
(a) The menu for *Kurodoko*

(b) The menu for *Oh no*.

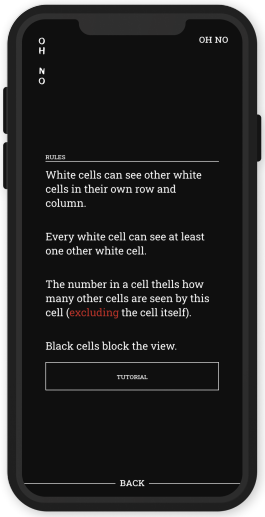
Figure 5.1: The website is split into one part for each puzzle.



(a) The rules of *Kurodoko*.

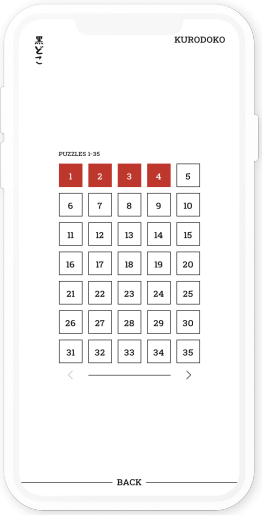


(b) In the tutorial one learns to play the game by solving a puzzle.

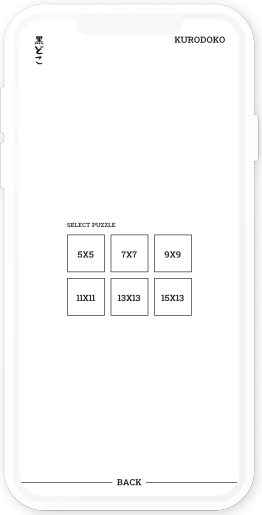


(c) The rules of *Oh no*.

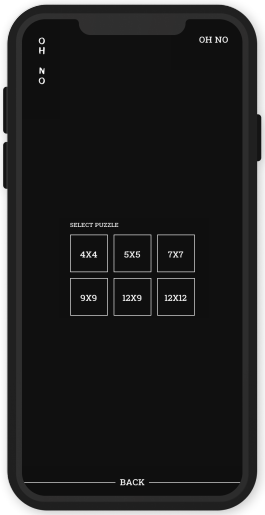
Figure 5.2: Both puzzles have an interactive tutorial.



(a) All 99 puzzles from [1] can be played.

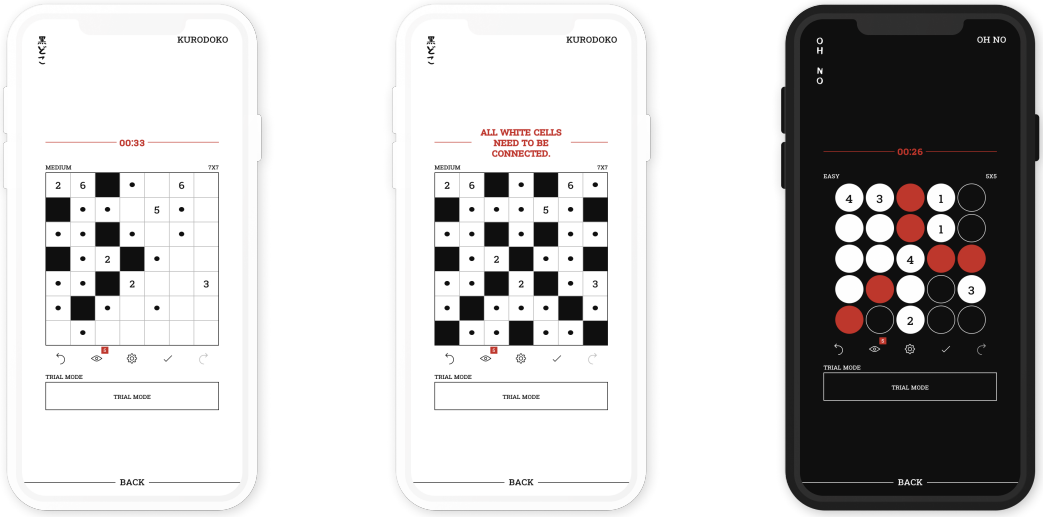


(b) In order to play a generated puzzle, the size has to be chosen first.



(c) For *Oh no* different sizes are available.

Figure 5.3: A user needs to select the size of the puzzle he or she wants to play.



(a) A *Kurodoko* puzzle.

(b) A user can check if any rules are violated.

(c) An *Oh no* puzzle.

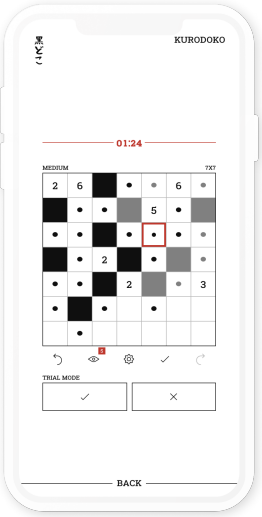
Figure 5.4: The *Kurodoko* and *Oh no* grids have a different design.

There are also extra features, which can help solve a puzzle. These include undoing and redoing moves, requesting additional hints, checking if any rules are violated and accessing the settings. A user can request at most 5 additional hints.

Another feature to help solving the puzzle is the trial mode. This mode allows a user to set a checkpoint when solving. Then the puzzle can be solved normally, where the first changed cell and all cells changed while in the trial mode are highlighted. This mode is active until the user either finalises the changes made in this mode or discards all changes made returning to the checkpoint.

In the settings, a user can view the rules of the puzzle, and enable or disable the different actions based on preference. Here the colour theme can also be changed between a light and a dark theme. Furthermore, the user has the options to export the current puzzle, continuing the solving process at a later time. This downloads the solving state as a JSON [2] file. In the settings the user can also reset his or her progress and use the solver to solve the puzzle.

By using the import page, which is accessible from all main menus, a user can manually enter a puzzle. Here the exported puzzle files can be uploaded too. Once a puzzle is successfully imported the user can choose to either have the solver solve the puzzle or to play the puzzle.

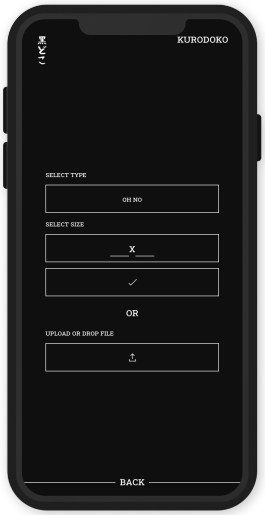


(a) The trial mode can be used to check if a cell can have a given colour.

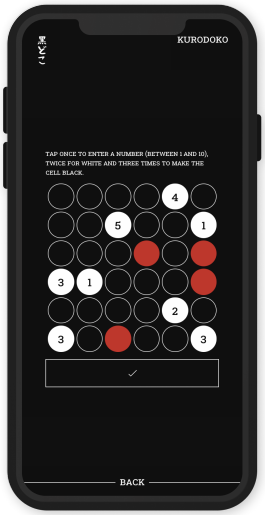
(b) The in-game settings.

(c) A solved *Oh no* puzzle.

Figure 5.5: The settings can also be accessed during a game.



(a) The import page.



(b) Puzzles can be uploaded or entered manually.

(c) When imported, the puzzle can be solved or played.

Figure 5.6: There are multiple ways to import puzzles.

6 Oh No is NP-Complete

In this chapter it is shown that *Oh no* is NP-complete. While the NP-completeness of *Kurodoko* has already been shown by Jonas Kölker in [11], this has not yet been done for *Oh no*. In order to show that *Oh no* is indeed NP-complete, it is necessary to show that *Oh no* is in NP and that the puzzle is NP-hard. Showing that the puzzle is in NP is easy, as it suffices to show that solution candidates can be verified in polynomial time.

To show the NP-hardness of *Oh no*, a reduction from 3SAT formulas to *Oh no* puzzles is presented. The Boolean satisfiability problem 3SAT, where every CNF clause has exactly 3 literals, was shown to be NP-complete in [7]. In the reduction, a circuit-board like *Oh no* puzzle is constructed for a given 3SAT formula. Multiple gadgets which behave similar to logic gates are used to break this construction down into smaller steps. First the needed gadgets are introduced, then the construction of the *Oh no* puzzle is presented. Finally the NP-completeness of *Oh no* is shown.

6.1 Gadgets

In this section the gadgets are introduced. In order to ease the understanding of the gadgets, empty cells which need to be white in all solutions are already marked as white.

The gadgets have inputs and outputs of size 3×3, which overlap with other gadgets in the reduction. The inputs are marked with a red line and the outputs, which are used as inputs of other gadgets, are marked with a dotted line. The inputs are always on the top and right side of a gadget, while the outputs are on the bottom and left sides.

As 3SAT is a Boolean satisfiability problem, a notion of *true* and *false* are needed in *Oh no*. Therefore we define that white cells (⊡) correspond to *true* and black cells (■) correspond to *false*.

Negation,Wire and Clause

The *Negation* and *Wire* gadgets are used to either negate or propagate a variable occurring in a given 3SAT clause. Figure 6.1 shows both gadgets. If the input of the *Negation* gadget is white, then the number constraint can only be fulfilled by making the output black and vice versa. Figure 6.2 shows all satisfying assignments for the *Negation* gadget. The *Wire* gadget is just a double negation, which returns the original value of the variable.

The *Clause* gadget has three inputs and represents a 3SAT clause without negation. If any literal in the clause evaluates to *true*, the 3SAT clause is satisfied, while it is unsatisifiable if none of the literals are *true*.

(a) *Negation* (b) *Wire*

Figure 6.1: The *Negation* and *Wire* gadgets.

(a) (b)

Figure 6.2: All possible ways to satisfy the number constraint of the *Negation* gadget.

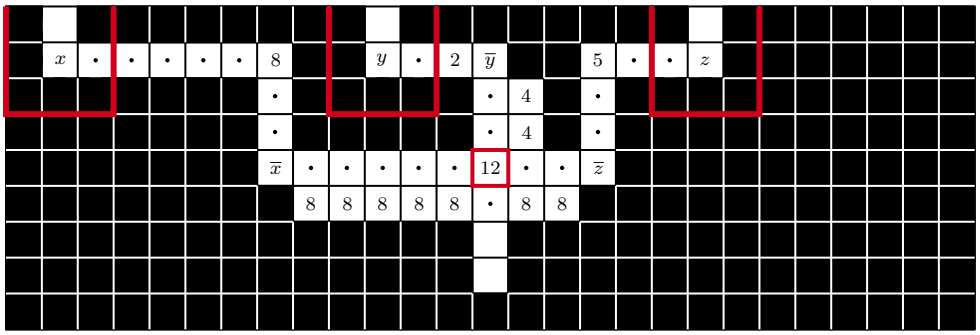
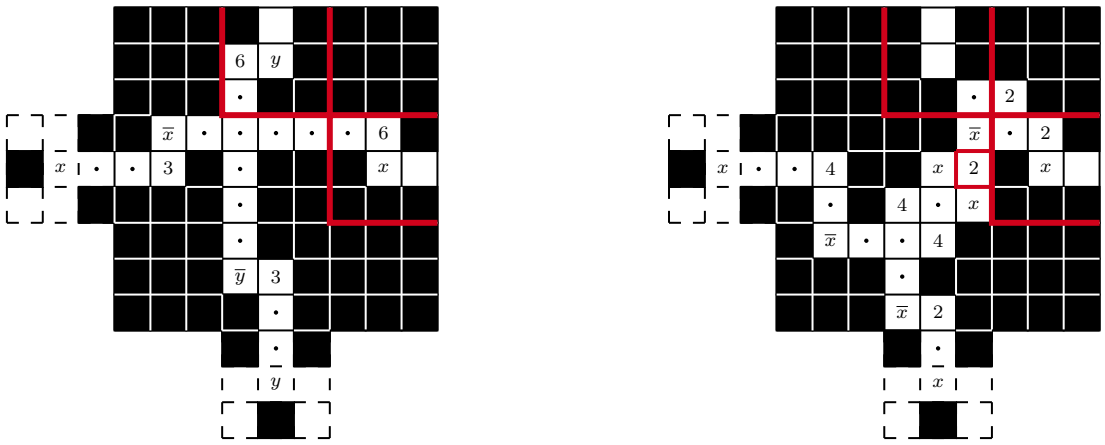


Figure 6.3: The *clause* gadget.

The cell marked red in Figure 6.3 is the centerpiece of the gadget. The cell needs to see exactly 12 other white cells. If all inputs are *false* (■) then the number constraint of this cell cannot be fulfilled since the cell sees more than 12 other white cells. If this is the case, the gadget causes the whole *Oh no* puzzle to be unsatisifiable. In all other cases where at least one input is white (*true*), there exists a solution to the gadget. Together the *Negation*, *Wire* and *Clause* gadgets can model a 3SAT clause with negated variables. If a variable occurs negated in the 3SAT clause, a *Negation* is used before the clause and a *Wire* otherwise.



(a) *Cross* (b) *Split*

Figure 6.4: The *Cross* and *Split* gadgets.

Cross and Split

The gadgets *Cross* and *Split* are needed to route the variables of the 3SAT formula to the *Oh no* equivalent to a 3SAT clause. Both gadgets are 9×9 squares and have two inputs and outputs each. The inputs are at the top and right of the gadgets and the outputs at the left and bottom sides. Figure 6.4 shows both gadgets. The *Cross* gadget routes the top input value to the bottom output and the right input to the left output, where the paths of the inputs cross. This is achieved by again propagating the variable values using a double negation, similar to the *Wire* gadget. The *Split* gadget throws away the top input and returns the value of the right input at both outputs. The gadget works in a similar way to the clause gadget. The number constraint of the cell marked red is responsible for splitting the input value, the other cells are just propagating the value by double negation.

By combining the *Cross* and *Split* gadgets the values of multiple variables can be propagated to the left and downwards. Figure 6.5 shows the interlacing of multiple *Cross* and *Split* gadgets.

Helper gadgets

Since an *Oh no* puzzle needs to be rectangular, a few helper gadgets are needed for the reduction. The helper gadgets shown in Figure 6.6 are all used as a sink for unneeded outputs and to provide the initial input values for the *Cross* and *Split* gadgets. The leftmost *Cross* or *Split* gadget will have an unused left output. Similarly the topmost and rightmost *Cross* and *Split* gadgets will have empty top and right inputs. Therefore the *Left End* gadget is used to void the leftmost outputs while the *Top End* and *Variable* gadgets are used as topmost and rightmost inputs respectively. Any further empty space can easily be filled using black cells, in order to make the constructed puzzle rectangular.

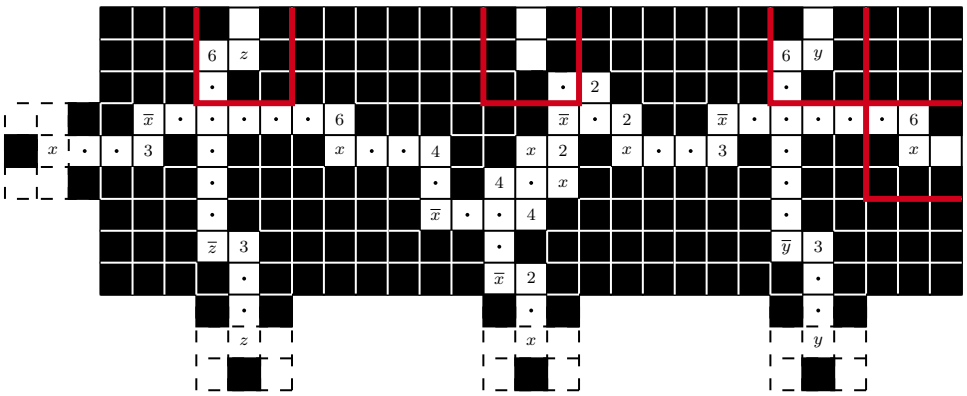
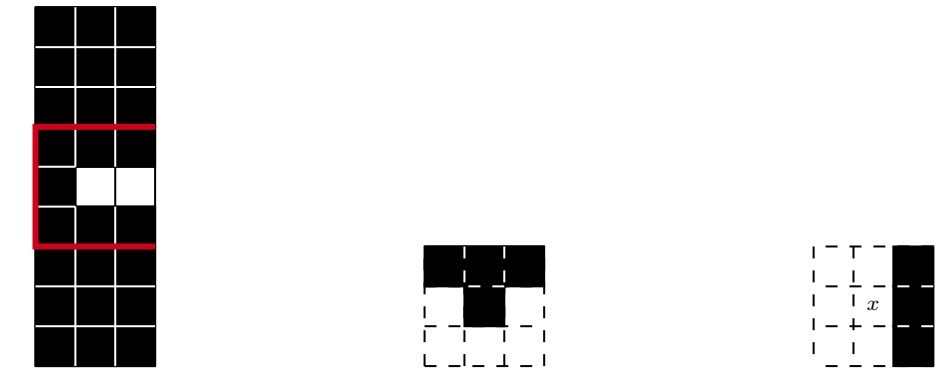


Figure 6.5: A combination of the *Cross* and *Split* gadgets.



(a) *Left End* (b) *Top End* (c) *Variable*

Figure 6.6: The helper gadgets.

6.2 Reduction

In this section the NP-hardness of *Oh no* is shown. This is achieved by constructing a *Oh no* puzzle using the previously introduced gadgets, which corresponds to a given 3SAT formula.

Theorem 6.1. Oh no *is NP-hard.*

Proof. Given a 3SAT formula F with n variables and m clauses, we can construct an *Oh no* puzzle representing this formula using the previously introduced gadgets. First the construction is presented and then it is shown that indeed every solution to the created puzzle also leads to a satisfying assignment for the 3SAT formula and that every satisfying assignment is a solution to the puzzle.

To construct an *Oh no* puzzle from F, first m *Clause* gadgets are placed next to each other. We define the first clause of the formula F to correspond to the rightmost clause gadget and the remaining clauses then are the *Clause* gadgets to the left in ascending order. The first variable occurring in a clause then will be mapped to the rightmost input

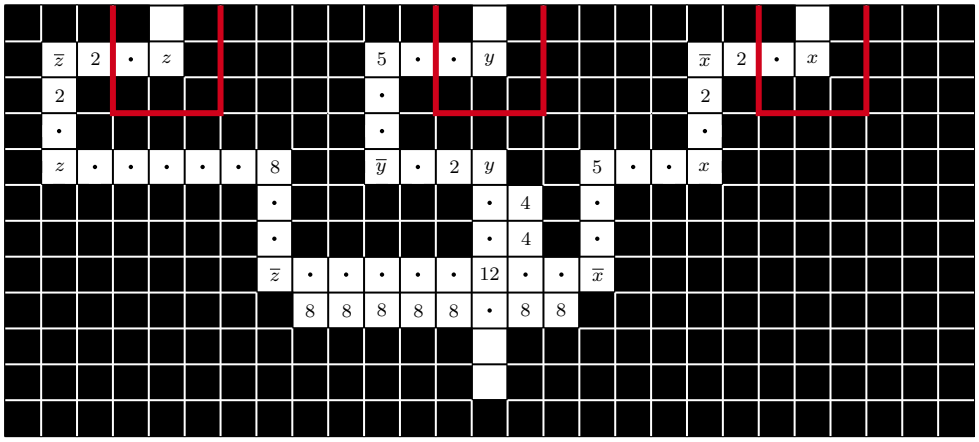


Figure 6.7: The gadget combination for the clause $x \vee \neg y \vee z$.

of the *Clause*, the second to the middle input and the third to the left input. Then the *Negation* and *Wire* gadgets are placed above the clause gadgets, such that literals which are negated in the clause pass through a *Negation* and the others pass through a *Wire*. Figure 6.7 gives an example for a single clause.

The next step is to route the variables to all clauses they appear in. This is done using the *Cross* and *Split* gadgets. Above each input of each clause, n *Cross* gadgets are placed on top of each other. The *Cross* gadgets now propagate the values from the top down to the clauses and from the right to the left. Then *Variable* gadgets are added as right input for all of the rightmost *Cross* gadgets.

In order to route the variables on the right side of the puzzle to the clauses, *Split* gadgets are used. For each clause and each input of a clause the *Cross* gadget which is in the same row as the *Variable* that should be propagated to the given input of the clause is replaced by a *Split* gadget. This causes the variable to be propagated down to the clause in addition to being propagated further to the left. This way the variables can be mapped to the right clauses.

Finally the helper gadgets are used to complete the puzzle. For the topmost *Cross* and *Split* gadgets *Top End* gadgets are added as top inputs and the leftmost outputs are collected by *Left End* gadgets. In order to make the puzzle rectangular, the empty space in the bottom left is simply filled with black cells.

Figure 6.8 shows the schema of the constructed *Oh no* puzzle for a 3SAT formula with n variables and m clauses. Constructing the puzzle for a given 3SAT formula can be done in polynomial time, since doing so is a matter of placing $3 \cdot n$ *Cross*/*Split* gadgets, three *Negation*/*Wire* gadgets and one *Clause* gadget for each of the m 3SAT clauses. Note that the *Top End*(*Variable*) gadgets are in a separate row(column) rather than merged with the topmost(rightmost) *Cross* or *Split* respectively in order to increase the legibility. This is also indicated by the red line. The example shown in Figure 6.9 depicts the full puzzle constructed from a 3SAT formula with two clauses.

If the 3SAT formula has a satisfying assignment, then this assignment can be turned

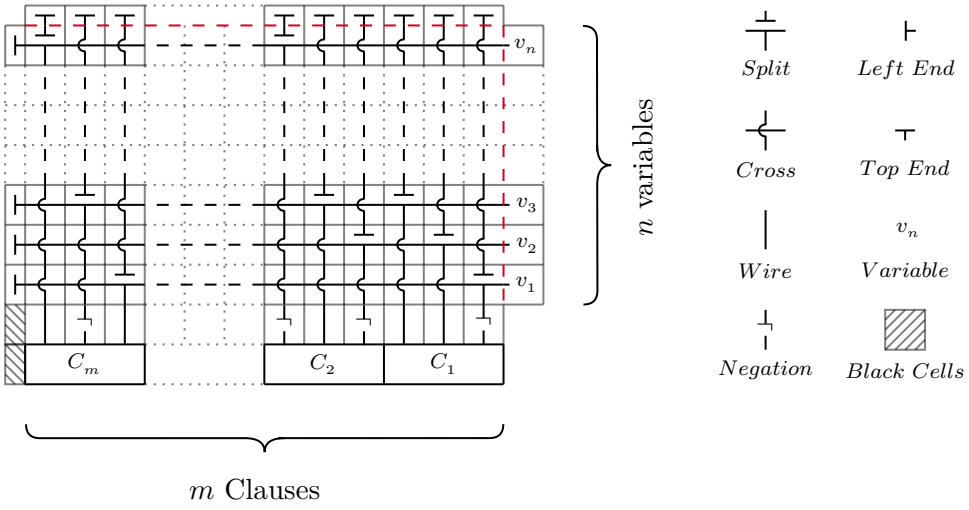


Figure 6.8: The outline of the constructed *Oh no* puzzle for a given 3SAT formula.

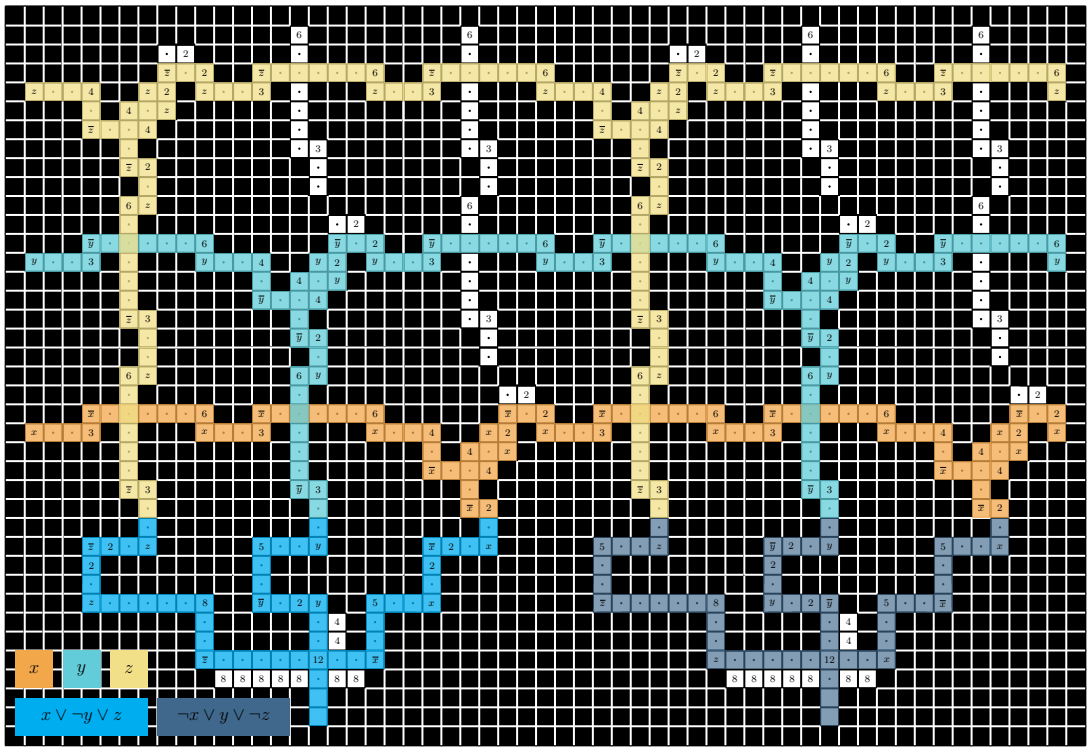


Figure 6.9: The *Oh no* puzzle constructed from the formula $F = (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$.

into a solution to the constructed puzzle. First the variable gadgets are colored in accordance to the assignment, i.e. if the valuation of a variable is *false*, then the cell is colored black (■), otherwise the cell stays white (⊡). Then the assigned values are

propagated through the *Split* and *Cross* gadgets of the constructed puzzle to the gadgets representing each clause. Since the assignment satisfies each clause, at least one of the propagated inputs to each *Clause* gadget must be *true* (⊡). Hence each *Clause* gadget is satisfied and all cells of the constructed puzzle can be colored without violating any of the rules. Therefore every solution to the 3SAT formula is a solution of the constructed *Oh no* puzzle.

Assume the puzzle constructed using the previously explained construction process has a solution. The solution of the *Oh no* puzzle satisfies all gadgets in the puzzle, which represent the clauses of the 3SAT formula F. Then all of the *Variable* gadgets on the right-hand side of the puzzles are either *true* (⊡) or *false*(■). Since the valuation of the variables given by the solution of the constructed puzzle satisfies all of the clauses in F, the same valuation of the variables in the 3SAT formula will satisfy all clauses in F and thus the whole formula F. Therefore any solution found for the constructed puzzle also provides a satisfying assignment for the 3SAT formula. Because of this, *Oh no* is NP-hard. □

Theorem 6.2. Oh no *is NP-complete.*

Proof. Since Theorem 6.1 already shows that *Oh no* is NP-hard, it suffices to show that solution candidates for a given puzzle can be checked in polynomial time.

In order to verify a solution candidate, it needs to be checked if the candidate violates any of the *Oh no* rules. If not, the candidate is a valid solution. The number constraint of a given cell can be checked by counting the number of other white cells the cell can see in all directions. If the sum of the visible cells equals the number of the cell, the constraint is satisfied. Since in a $r \times c$ *Oh no* puzzle at most all cells can have a number and each number can see at most $r + c - 2$ other cells, checking the number constraint for all numbered cells can be done in $O(r \cdot c \cdot (r + c - 2))$ which is polynomial. The rule that every white cell can see at least one other white cell can also be checked in polynomial time ($O(4 \cdot r \cdot c)$) by checking whether at least one of the up to four neighbouring cells is white for all white cells. Therefore *Oh no* is in NP.

Since *Oh no* is in NP and NP-hard, it also is NP-complete. □

7 Conclusion

In this thesis first the logic puzzles *Kurodoko* and *Oh no* were introduced. Since *Oh no* is a variant of *Kurodoko*, the rule set of the puzzles is quite similar. Then SMT encodings for the rules of both puzzles and all of their rules were presented. For some rules multiple different encodings and solving strategies were introduced. Additionally the performance of the developed solvers was evaluated together with some other solvers, which are freely available on the internet. Furthermore, generators for both puzzle types were introduced. Using the statistics of the introduced solvers, puzzles with chosen difficulty levels can be generated. Finally the graphical user interface which was developed to make playing the puzzles convenient was showcased. The web application can be used solve puzzles and to play custom and generated puzzles in addition to the puzzles in [1]. Moreover, the application has extra features which help solving puzzles, such as the trial mode.

7.1 Future Work

Improvements which could be addressed by further work include:

- First of all, by generating puzzles where the initial hints follow recognisable patterns such as symmetry, the puzzles could be made more enticing. This could be implemented by changing the strategy used to remove the hints, instead of removing the hints in a random order.
- Secondly, the approximation of a puzzle's difficulty could surely be improved by collecting solving data for more puzzles of known difficulty.
- While the graphical user interface works well on mobile devices, the available screen space is not utilised very efficiently on desktop screens. This could be improved to enhance the experience for desktop users.
- Another convenient improvement for the graphical user interface would be to add an URL to each puzzle, allowing users to replay and share puzzles more conveniently.

Bibliography

[1] *Where is black cells.* NIKOLI Co., Ltd., 1 2006. ISBN4-89072-065-0.

[2] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, 2017. https://doi.org/10.17487/RFC8259.

[3] Facebook Open Source. Reactjs. https://reactjs.org/. Accessed: 02-02-2020.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc2616: Hypertext transfer protocol – http/1.1, 1999. https://doi.org/10.17487/RFC2616.

[5] Gatsby, Inc. Gatsbyjs. https://www.gatsbyjs.org/. Accessed: 02-02-2020.

[6] I. Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: 18-02-2020.

[7] I. Holyer. The NP-Completeness of Some Edge-Partition Problems. *SIAM Journal on Computing*, 10:713–717, 1981. https://doi.org/10.1137/0210054.

[8] T. Hoshan. Kuromasu Solver using Java. http://hoshan.org/kuromasu-solver-using-java/. Accessed: 16-02-2020.

[9] A. Janko and O. Janko. Kuromasu. https://www.janko.at/Raetsel/Kuromasu/index.htm. Accessed: 03-02-2020.

[10] M. Kool. Oh no. https://0hn0.com/. Accessed: 02-02-2020.

[11] J. Kölker. Kurodoko is NP-Complete. *Journal of Information Processing*, 20(3):694–706, 2012. https://doi.org/10.2197/ipsjjip.20.694.

[12] MDN Contributors. Web Storage API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API. Accessed: 15-06-2020.

[13] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), 2014. https://doi.org/10.5555/2600239.2600241.

[14] Microsoft Research. Z3 Prover. https://github.com/Z3Prover/z3. Accessed: 02-02-2020.

[15] Monash University and Data61, CSIRO. Minizinc. https://www.minizinc.org/. Accessed: 18-02-2020.

[16] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In C. Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of LNCS, pages 529–543. Springer-Verlag, 2007. https://doi.org/10.5555/1771668.1771709.

[17] NIKOLI Co., Ltd. Official Kurodoko rules. http://www.nikoli.co.jp/en/puzzles/where_is_black_cells.html. Accessed: 02-02-2020.

[18] A. Pun. puzzle-solvers. https://github.com/AvaLovelace1/puzzle-solvers/blob/master/solvers/kurodoko-solver.cpp. GitHub Repository, Accessed: 16-02-2020.

[19] Python Software Foundation. Python. https://www.python.org/. Accessed: 02-02-2020.

[20] Q42.nl. Oh no. https://www.q42.nl/en/work/0h-n01. Accessed: 30-04-2020.

[21] S. Richard and P. LePage. What are Progressive Web Apps? https://web.dev/what-are-pwas/. Accessed: 15-06-2020.

[22] The GraphQL Foundation. Graphql. https://graphql.org/. Accessed: 02-02-2020.

[23] S. Tsutomu. opt4puzzle. https://github.com/SaitoTsutomu/opt4puzzle/blob/master/Kurodoko.ipynb. GitHub Repository, Accessed: 16-02-2020.

[24] unknown. Kuromasu: Resolution in Prolog by Constraints. https://jfoutelet.developpez.com/articles/kuromasu/. Accessed: 16-02-2020.

[25] unknown. Mobile web traffic statistics. https://www.thinkwithgoogle.com/data/mobile-web-traffic-statistics/. Accessed: 15-05-2020.

[26] unknown. sdvx. https://github.com/3892myamya/sdvx/blob/master/src/main/java/myamya/other/solver/kurodoko/KurodokoSolver.java. GitHub Repository, Accessed: 16-02-2020.

[27] T. van Meurs. Comparing Methods for Solving Kuromasu Puzzles. Bachelor's thesis, 2008.

A Puzzle File Format

The JSON files used for exporting and importing puzzles have the same structure for both *Kurodoko* and *Oh no*. A puzzle file must have one of the extensions `.json`, `.txt`, `.ohno` or `.kurodoko` to be accepted. As the file format is JSON, the puzzle is represented as a single object, having multiple key value pairs. While the exported puzzles contain additional information such as the current grid state, the elapsed time and the moves made so far, only a few keys are needed to successfully import a puzzle. The only required keys are:

- `type`: the puzzle type, either `"KURODOKO"` or `"OHNO"`
- `rows`: the number of rows as integer
- `cols`: the number of columns as integer
- `puzzle`: the puzzle, represented as an array or rows. Each row in turn is represented as a string containing the value of each cell in the row, separated by a single whitespace. The possible values for a single cell are `"."` for empty, `"B"` for black or a number for the numbered hints.

For an example puzzle, the representation is shown in Figure A.1.

```
{
  "type": "OHNO",
  "rows": 5,
  "cols": 5,
  "puzzle": [
    "4 3 . 1 .",
    ". . . 1 .",
    ". . 4 . B",
    ". . . . 3",
    ". . 2 . ."
  ]
}
```

Figure A.1: The representation of an example puzzle.

B Reference Solver Implementation

In order to disambiguate how to encode the rules as presented in Chapter 3, a reference implementation using the open source constraint modelling language MiniZinc[1] is provided here. MiniZinc allows for a high level encoding of constraint problems, which can then be used to solve problem instances [16]. The supported operators include the logical operators and (`/\`), or (`\/`), negation (`not`), implication (`->`) and the quantifiers forall and exists.

After MiniZinc has been installed, the problem encoding (`.mzn` file) can be called for a given problem instance (`.dzn` file) by either using the IDE or using the command line. For the command line, the command

```
minizinc <path_to_encoding>.mzn <path_to_instance>.dzn
```

can be used. By using the additional flag `-s` statistics are provided too and with `-a` all solutions are enumerated. Beware that the *Kurodoko* model sometimes allows for multiple occurrences of the same solution, since the depth assigned to black cells may be variable. For the number constraints the visibility encoding is used and for the connectivity rule of *Kurodoko* the rooted tree encoding is used.

B.0.1 Oh No

The *Oh no* encodings use a different representation, where -2 denotes an empty (□) cell and -1 denotes a black (■) cell. Every other number represents a white cell, such that hints with numbers are represented the number itself.

```
int: U = -2;                              % empty cell
int: B = -1;                              % black cell
int: W =  0;                              % white cell

int: rows; set of int: R = 1..rows;       % rows
int: cols; set of int: C = 1..cols;       % cols
array[R,C] of U..rows+cols-2: puzzle;     % puzzle
set of int: N = 1..rows+cols-2;           % numbered cell

% each variable represents a single cell. if the variable is true,
% then the cell is white in the solution else it is black
array[R,C] of var bool: model;

% every white cell can see at least one other
% i.e. "either the cell is not white, or there exists
% a neighbour (row/col difference of exactly 1) which is white"
constraint forall(r in R, c in C)(
```

[1] Available at [15].

```
  not model[r,c] \/
  exists([abs(r-r1)+abs(c-c1) = 1 /\
    model[r1,c1] |
    r1 in max(r-1, 1)..min(r+1, rows), c1 in max(c-1, 1)..min(c+1, cols)]));

% visibility arrays for number constraints
array[R,C] of var -1..rows-1: sees_top;
constraint forall(c in C)(
  if model[1,c] then sees_top[1,c] = 0 else sees_top[1,c] = -1 endif
);
constraint forall(r in 2..rows, c in C)(
  if model[r,c]
    then sees_top[r,c] = sees_top[r-1, c] + 1
    else sees_top[r,c] = -1
  endif
);

array[R,C] of var -1..cols-1: sees_right;
constraint forall(r in R)(
  if model[r,cols] then sees_right[r,cols] = 0 else sees_right[r,cols] = -1 endif
);
constraint forall(r in R, c in 1..cols-1)(
  if model[r,c]
    then sees_right[r,c] = sees_right[r, c+1] + 1
    else sees_right[r,c] = -1
  endif
);

array[R,C] of var -1..rows-1: sees_bottom;
constraint forall(c in C)(
  if model[rows,c] then sees_bottom[rows,c] = 0 else sees_bottom[rows,c] = -1 endif
);
constraint forall(r in 1..rows-1, c in C)(
  if model[r,c]
    then sees_bottom[r,c] = sees_bottom[r+1, c] + 1
    else sees_bottom[r,c] = -1
  endif
);

array[R,C] of var -1..cols-1: sees_left;
constraint forall(r in R)(
  if model[r,1] then sees_left[r,1] = 0 else sees_left[r,1] = -1 endif
);
constraint forall(r in R, c in 2..cols)(
  if model[r,c]
    then sees_left[r,c] = sees_left[r, c-1] + 1
    else sees_left[r,c] = -1
  endif
);

% for each hint, the already known color is added to the model
% black hints become false, numbered hints (white) become true
constraint forall(r in R, c in C where puzzle[r,c] > U)
                 (model[r,c] = (puzzle[r,c] >= W));

% the numbered hints are added - every numbered cell must see exactly the right
constraint forall(r in R, c in C where puzzle[r,c] in N)(
  puzzle[r,c] =
    sees_top[r,c] + sees_right[r,c] + sees_bottom[r,c] + sees_left[r,c]
);

solve satisfy;
```

```
output [ if puzzle[r,c] in N then show(puzzle[r,c])
  else if fix(model[r,c]) then "_" else "B" endif
  endif ++ if c == cols then "\n" else " " endif |
  r in R, c in C ];
```

Listing B.1: The file `ohno.mzn`.

```
rows = 5;
cols = 5;
puzzle = [| -2, +4, -2, -2, -2
          | -2, -2, -2, +5, -2
          | -2, -2, -2, -2, +4
          | -2, +1, -2, -2, -1
          | -2, +2, -2, +1, -2 |];
```

Figure B.1: A puzzle instance `ohno.dzn`.

This example can then be solved by executing `minizinc -s ohno.mzn ohno.dzn` in the command line.

B.0.2 Kurodoko

The *Kurodoko* encodings use a different representation, where -1 denotes an empty (□) cell, 0 denotes a black (■) cell and every other number represents white.

```
int: U = -1;                              % empty cell
set of int: N = 1..rows+cols-1;           % numbered cell

set of int: D = 1..4;                     % diagonal directions
array[D,1..2] of {-1,1}: DIRS = [| -1, -1 | -1, 1 | 1, 1 | 1, -1 |];

int: rows; set of int: R = 1..rows;       % rows
int: cols; set of int: C = 1..cols;       % cols
array[R,C] of -1..rows+cols-1: puzzle;    % puzzle

array[R,C] of var bool: model;            % is the cell white
% parent child relation, true if (r+DIR[i], c+DIR[i]) is parent of (r,c)
array[R,C,D] of var bool: parent;
array[R,C] of var 0..rows*cols div 2: depth; % depth of node in directed graph

% a helper function to encode the at most one constraint for arrays
predicate at_most_one(array[int] of var bool: x) =
  forall(i,j in index_set(x) where i < j)(
    (not x[i] \/ not x[j])
  );

% black cells are not adjacent (= one of two neighouring cells is white)
constraint forall(r in 2..rows, c in C)(model[r-1,c] \/ model[r,c]);
constraint forall(r in R, c in 2..cols)(model[r,c-1] \/ model[r,c]);

% connectivity heuristic: every white cell has at least one white neighbour
constraint forall(r in R, c in C where puzzle[r,c] != 1)(
  not model[r,c] \/
  exists([abs(r-r1)+abs(c-c1) = 1 /\
```

```
    model[r1,c1] |
    r1 in max(r-1, 1)..min(r+1, rows), c1 in max(c-1, 1)..min(c+1, cols)]));

% visibility arrays for number constraints
array[R,C] of var -1..rows-1: sees_top;
constraint forall(c in C)(
  if model[1,c] then sees_top[1,c] = 0 else sees_top[1,c] = -1 endif
);
constraint forall(r in 2..rows, c in C)(
  if model[r,c]
    then sees_top[r,c] = sees_top[r-1, c] + 1
    else sees_top[r,c] = -1
  endif
);

array[R,C] of var -1..cols-1: sees_right;
constraint forall(r in R)(
  if model[r,cols] then sees_right[r,cols] = 0 else sees_right[r,cols] = -1 endif
);
constraint forall(r in R, c in 1..cols-1)(
  if model[r,c]
    then sees_right[r,c] = sees_right[r, c+1] + 1
    else sees_right[r,c] = -1
  endif
);

array[R,C] of var -1..rows-1: sees_bottom;
constraint forall(c in C)(
  if model[rows,c] then sees_bottom[rows,c] = 0 else sees_bottom[rows,c] = -1 endif
);
constraint forall(r in 1..rows-1, c in C)(
  if model[r,c]
    then sees_bottom[r,c] = sees_bottom[r+1, c] + 1
    else sees_bottom[r,c] = -1
  endif
);

array[R,C] of var -1..cols-1: sees_left;
constraint forall(r in R)(
  if model[r,1] then sees_left[r,1] = 0 else sees_left[r,1] = -1 endif
);
constraint forall(r in R, c in 2..cols)(
  if model[r,c]
    then sees_left[r,c] = sees_left[r, c-1] + 1
    else sees_left[r,c] = -1
  endif
);

% add known white fields
constraint forall(r in R, c in C where puzzle[r,c] > U)
  (model[r,c] = puzzle[r,c] in N);
% add the number constraints
constraint forall(r in R, c in C where puzzle[r,c] in N)(
  puzzle[r,c] - 1 =
    sees_top[r,c] + sees_right[r,c] + sees_bottom[r,c] + sees_left[r,c]
);


% connectivity by forcing diagonally connected components of black cells to be a
% rooted tree (directed acyclic graph with root node)

% each cell has at most one parent node
```

```
constraint forall(r in R, c in C)(
  at_most_one([parent[r,c,i] | i in D])
);

% the depth of a node is 1 + the depth of the parent
constraint forall(r in R, c in C, i in D where r+DIRS[i, 1] in R /\ c+DIRS[i, 2] in C)(
    parent[r,c,i] -> depth[r,c] == depth[r+DIRS[i, 1],c+DIRS[i, 2]] + 1
);

% if two neighbouring cells are black, there exists a parent child relation
constraint
  forall(r in R, c in C, i in D where r+DIRS[i, 1] in R /\ c+DIRS[i, 2] in C)(
  not model[r,c] /\ not model[r+DIRS[i, 1], c+DIRS[i, 2]] -> (
    (parent[r,c,i] /\
     not parent[r+DIRS[i, 1], c+DIRS[i, 2], if i <= 2 then i + 2 else i - 2 endif]) \/
    (not parent[r,c,i] /\
     parent[r+DIRS[i, 1], c+DIRS[i, 2], if i <= 2 then i + 2 else i - 2 endif])
  )
);

% border nodes are root nodes and therefore have no parents and depth 0
constraint forall(r in {1,rows}, c in C)
  (depth[r,c] == 0 /\ forall(i in D)(not parent[r,c,i]));
% border nodes are root nodes and therefore have no parents and depth 0
constraint forall(r in R, c in {1,cols})
  (depth[r,c] == 0 /\ forall(i in D)(not parent[r,c,i]));

% these constraints are not necessary to find solutions
% they just reduce the number of times a single solution can occur
% semantically these ensure that
%   - white cells have no parent-child relation and no depth
%   - black cells with no diagonally adjacent black cells have depth 0

constraint forall(r in R, c in C, i in D where r+DIRS[i, 1] in R /\ c+DIRS[i, 2] in C)(
  (model[r,c] \/ model[r+DIRS[i, 1], c+DIRS[i, 2]]) ->
    (not parent[r,c,i] /\
    not parent[r+DIRS[i, 1], c+DIRS[i, 2], if i <= 2 then i + 2 else i - 2 endif])
);
constraint forall(r in R, c in C)(model[r,c] -> depth[r,c] = 0);
constraint forall(r in R, c in C)(
  not model[r,c] /\
  forall(i in D where r+DIRS[i, 1] in R /\ c+DIRS[i, 2] in C)
    (model[r+DIRS[i, 1], c+DIRS[i, 2]]) -> depth[r,c] = 0
);

solve satisfy;

output [ if puzzle[r,c] in N then show(puzzle[r,c])
  else if fix(model[r,c]) then "_" else "B" endif
  endif ++ if c == cols then "\n" else " " endif |
  r in R, c in C ];
```

Listing B.2: The file `kurodoko.mzn`.

```
rows = 5;
cols = 5;
puzzle = [|  3, -1, -1, -1, -1
         | -1, -1, -1, -1, -1
         | -1, -1, -1, -1, -1
         | -1, -1, -1,  5, -1
         |  6, -1, -1, -1,  9 |];
```

Figure B.2: A puzzle instance `kurodoko.dzn`.

This example can then be solved by executing `minizinc -s kurodoko.mzn kurodoko.dzn` in the command line.

C Benchmark Data

In this chapter, additional plots of the results are provided. First of all, the solving times of some combinations which were not included in the performance evaluation in Section 3.4 are shown in Figure C.1. The second plot shows the first 30 seconds with a logarithmic time scale. As a reference, the *IE,H,A** solver which performed best is also included. Note that the encodings are abbreviated as before, where *SE*,*R* and *SDFS* respectively denote the simple number enumeration, the connectivity encoding using reachability and the simple iterative connectivity approach. As expected, the reachability encoding (*R*) quickly reaches its limits. Furthermore, the simple number and connectivity approaches (*SE* and *SDFS*) perform quite well in most cases, although the improved versions are more consistent and faster. Figure C.2 displays the average time of all measurements for each one of the 99 Nikoli puzzles from [1] and the 55 hardest (difficulties 7, 8 and *hard*) puzzles available at the Janko web page ([9]). Note that the average solving time for each individual puzzle is presented both with a linear and a logarithmic time scale. Moreover, the puzzles are sorted by their size and thus the Janko puzzles appear out of order. Missing dots indicate that the solver failed to solve a given puzzle within the 15 minute period. The results for the other included solvers are presented in Figure C.3. Again the times are plotted with a linear and a logarithmic scale, but in this plot the logarithmic scale includes the first minute.

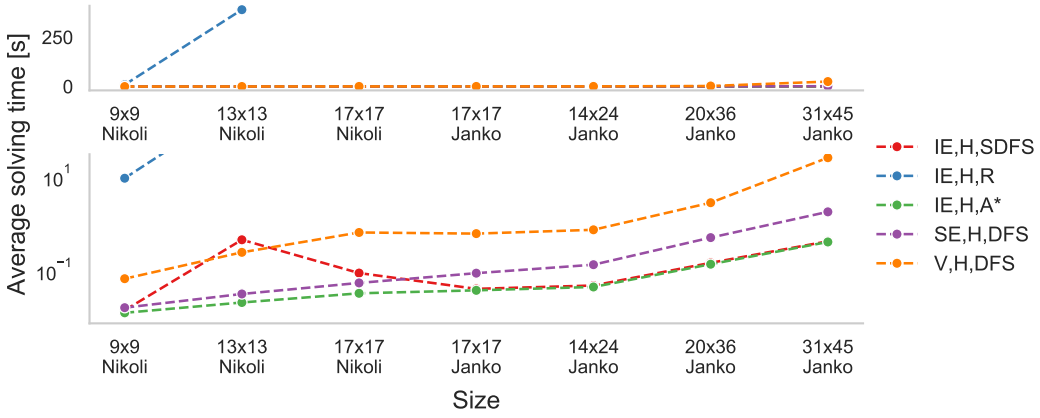


Figure C.1: The solving times for a few combinations not included in in Section 3.4.

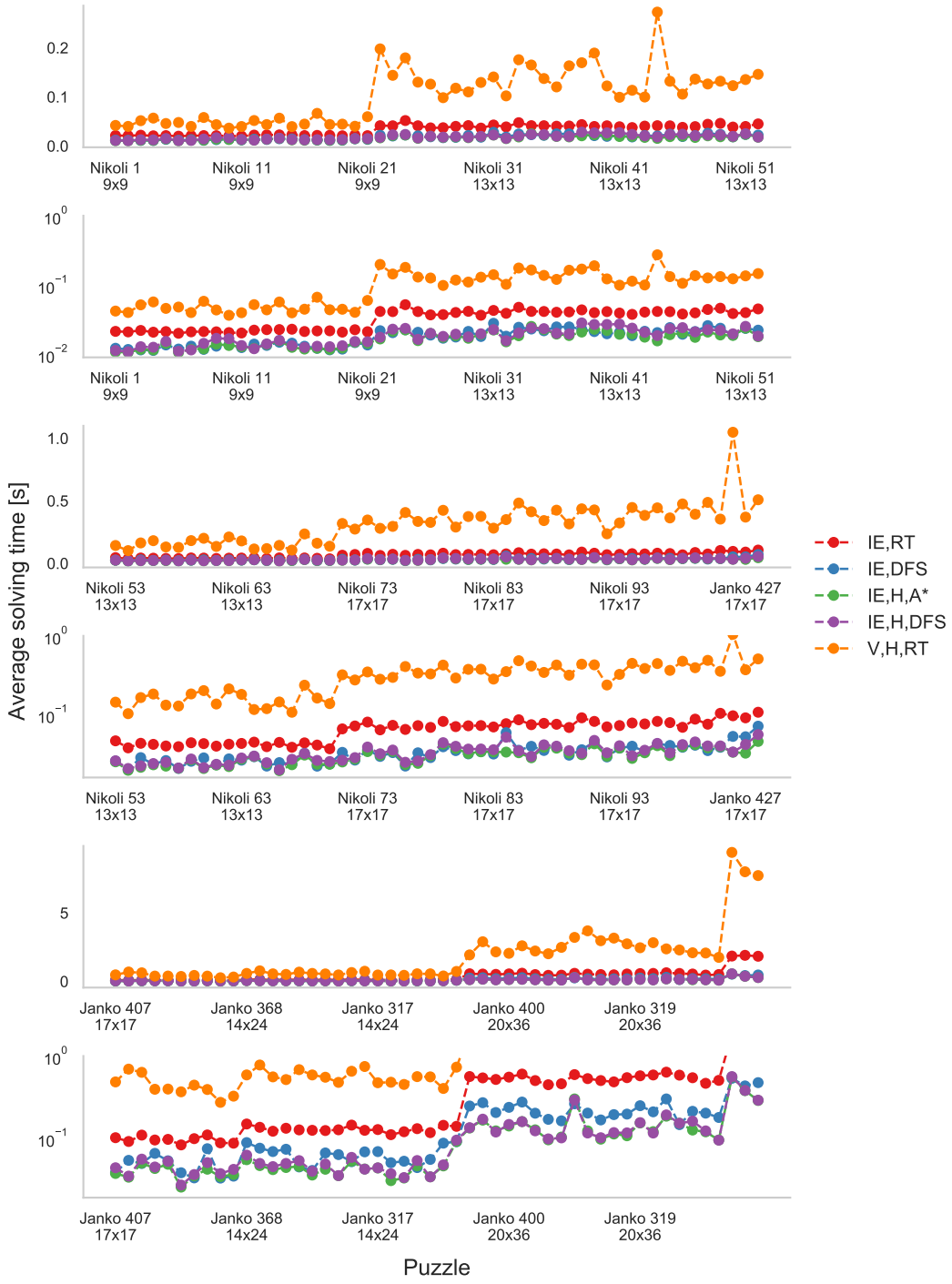


Figure C.2: The solving times for the solving approaches presented in Chapter 3.

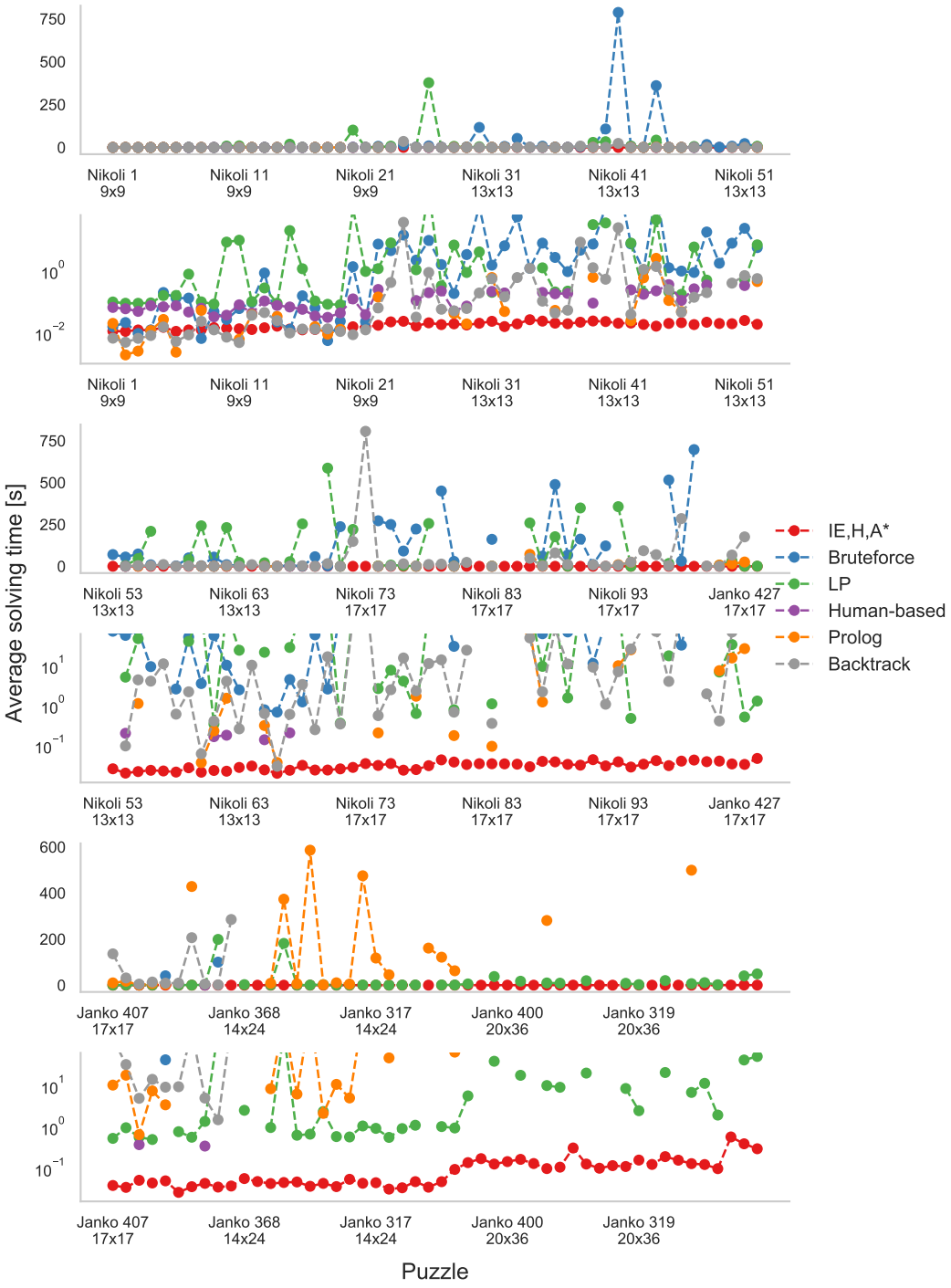


Figure C.3: The solving times for the other solvers included in Section 3.4.

D Difficulty Data

The data collected for all of the 99 *Kurodoko* puzzles taken from [1] is presented in Table D.1. The solver used to collect this data uses the improved enumeration approach for the number constraints. For the rule that all white cells must be connected, the iterative connectivity approach using depth first search and the heuristic were used. In Section 3.4 this solver is referred to as H, IE, DFS. Note that the statistics presented are the average per cell where each puzzle was repeatedly solved 10 times and then the statistics were averaged for each level of difficulty. The solver only needed to restart for puzzle 79 once, indicating that this puzzle was exceptionally hard for the solver.

Feature	EASY	MEDIUM	HARD
eliminated vars	0.59	0.27	0.22
rlimit count	95.89	152.06	184.48
max memory	0.03	0.02	0.02
memory	0.02	0.01	0.01
num allocs	1.11e8	9.91e8	2.46e9
connectivity iterations	0.01	0.01	0.01
connectivity violations	0.01	0.04	0.04
conflicts	0.06	0.09	0.12
decisions	0.12	0.31	0.43
propagations	4.45	13.84	19.73
binary propagations	4.18	13.16	18.73
final checks	0.01	0.01	0.01
mk clause	1.82	3.01	3.71
del clause	1.01	1.00	0.84
num checks	0.02	0.01	0.01
minimized lits	0.18	0.17	0.18
restarts	-	0.00	-

Table D.1: The average statistics per cell for the three difficulty levels in [1].

E NP-Completeness of Oh No

In order to illustrate the construction of an *Oh no* puzzle for a given 3SAT formula, an implementation in Python 3 is provided. This implementation contains a parser for 3SAT formulas in the DIMACS CNF format, a function to construct an *Oh no* puzzle for a given 3SAT CNF and a human readable encoding of the gadgets. In the main method, the puzzle for an example CNF is constructed and printed.

```
#!/usr/bin/env python3
"""
An implementation of the proposed construction of an Oh No puzzle
for a given 3SAT formula.

Note that the gadgets have been modified slightly, since some gadgets only
appear at the border of the puzzle which makes some rows of black cells unnecessary.
For example, the clause gadget doesn't need its bottom row.
"""

CROSS = [
    "B B B B . B B B B",
    "B B B 6 . B B B B",
    "B B B . B B B B B",
    "B . . . . . 6 B",
    ". 3 B . B B B . .",
    "B B B . B B B B B",
    "B B B . B B B B B",
    "B B B . 3 B B B B",
    "B B B B . B B B B"
]

SPLIT = [
    "B B B B . B B B B",
    "B B B B . B B B B",
    "B B B B B . 2 B B",
    "B B B B B . . 2 B",
    ". 4 B B . 2 B . .",
    "B . B 4 . . B B B",
    "B . . . 4 B B B B",
    "B B B . B B B B B",
    "B B B . 2 B B B B"
]

NEGATION = [
    "B B B B . B B B B",
    "B 5 . . . B B B B",
    "B . B B B B B B B"
]

WIRE = [
    "B B B B . B B B B",
    "B . 2 . . B B B B",
    "B 2 B B B B B B B"
```

```
]

# Note that the last row is not needed,
# since the border of the puzzle is a sufficient delimiter
CLAUSE = [
    "B . B B B B B B B B . B B B B B B B B . B B B B B B B",
    "B . . . . . . 8 B B . . 2 . B B 5 . . . B B B B B B B",
    "B B B B B B B . B B B B B . 4 B . B B B B B B B B B B",
    "B B B B B B B . B B B B B . 4 B . B B B B B B B B B B",
    "B B B B B B B . . . . . . . 12 . . . B B B B B B B B B B",
    "B B B B B B B B 8 8 8 8 8 . 10 8 B B B B B B B B B B B",
    "B B B B B B B B B B B B B . B B B B B B B B B B B B B",
    "B B B B B B B B B B B B B . B B B B B B B B B B B B B",
    # "B B B B B B B B B B B B B B B B B B B B B B B B B B B"
]

# helper gadgets

TOP_END = [
    ". . . B B B . . .",
    ". . . . B . . . ."
]

RIGHT_END = [
    ".",
    ".",
    ".",
    "B",
    "B",
    "B",
    ".",
    ".",
    "."
]

LEFT_END = [
    "B B",
    "B B",
    "B B",
    "B B",
    ". .",
    "B B",
    "B B",
    "B B",
    "B B"
]


def parse(lines):
    """
    Parses the lines of a 3SAT CNF formula,
    the result is a triple (nvars, nclauses, clauses)
    """
    nvars = 0
    nclauses = 0
    clauses = []

    for line in lines:
        literals = line.split()
        if len(literals) == 0 or literals[0] in "pc":
            continue
        clause = []
```

```
        for literal in literals:
            val = int(literal)
            nvars = max(abs(val), nvars)
            if val == 0:
                # make sure the clause has exactly 3 literals
                assert len(clause) == 3
                clauses.append(clause[:])
                clause = []
            else:
                clause.append(val)
    nclauses = len(clauses)
    return (nvars, nclauses, clauses)


def construct_ohno(cnf):
    """
    This function constructs an equisatisfiable ohno puzzle for a given
    3SAT CNF.
    """
    nvars, nclauses, clauses = cnf
    assert nvars > 0
    assert nclauses > 0
    # each variable needs 9 rows, the negation layer needs 3 and the clause 8
    rows = nvars * 9 + 3 + 8
    # each clause needs 9 rows for each of the 3 variables, on the left end two rows
    # for the end delimiters are needed
    cols = nclauses * 3 * 9 + 2

    grid = [['B' for col in range(cols)] for row in range(rows)]

    # the clauses are written in reverse order, i.e.
    # the first clause is on the right end of the puzzle
    # and the first variable is in the bottom row
    for i in range(nclauses):
        start_col = cols - (i + 1) * 3 * 9
        for j in range(3):
            col = start_col + j * 9
            var = clauses[i][2 - j]
            for k in range(nvars):
                row = k * 9
                if abs(var) == nvars - k:
                    __place_gadget(SPLIT, row, col, grid)
                else:
                    __place_gadget(CROSS, row, col, grid)
            row = nvars * 9
            __place_gadget(NEGATION if var < 0 else WIRE, row, col, grid)

    # finally the non variable parts of the grid are placed
    # placing the top and right ends overwrites the previously laid out network
    for row in range(0, rows - len(CLAUSE) - len(NEGATION), len(LEFT_END)):
        __place_gadget(LEFT_END, row, 0, grid)
        __place_gadget(RIGHT_END, row, cols -
                       len(RIGHT_END[0].split()), grid, overwrite_empty_only=True)

    for col in range(len(LEFT_END[0].split()), cols, 3 * 9):
        __place_gadget(CLAUSE, rows - len(CLAUSE), col, grid)
        for i in range(3):
            __place_gadget(TOP_END, 0, col+i*9, grid,
                           overwrite_empty_only=True)

    return {
        'type': 'OHNO',
```

```
        'rows': rows,
        'cols': cols,
        'puzzle': list(map(lambda row: ' '.join(row), grid))
    }


def __place_gadget(gadget, start_row, start_col, grid, overwrite_empty_only=False):
    """
    Places a gadget in a given position in the grid
    """
    for i in range(len(gadget)):
        row = gadget[i].split()
        for j in range(len(row)):
            if not overwrite_empty_only or grid[start_row + i][start_col + j] == '.':
                grid[start_row + i][start_col + j] = row[j]


if __name__ == '__main__':
    lines = [  # as read by file.readlines
        "p cnf 3 2",
        "-1  2 -3 0",
        " 1 -2  3 0"
    ]

    cnf = parse(lines)
    puzzle = construct_ohno(cnf)

    for row in puzzle['puzzle']:
        print(row)
```

Listing E.1: A Python implementation of the construction.