TAKEGAKI

Ramon van Dam
Supervisors: W.A. Kosters & J.M. de Graaf

Table of Contents

1. Introduction

This paper is the result of my bachelor project about the game of *Takegaki* (also known as Loopy, Fences, Slitherlink and Dotty Dilemma), done in 2008 under supervision of W.A. Kosters and J.M. de Graaf at the Leiden Institute of Advanced Computer Science, Leiden University.

The goal of the project was to do a study on this kind of Japanese puzzles, in particular to find out in which way we can use the speed of computers to solve existing puzzles and generate new ones efficiently.

Chapter 2 will introduce the game of Takegaki and its rules, Chapter 3 shows a method for checking whether or not a puzzle is solved, which is used in Chapter 4 and Chapter 5, where methods are described for solving puzzles with logic rules and backtracking. Chapter 6 expands the logic rules with an extra strategy and Chapter 7 explains a method for generating new puzzles, while Chapter 8 gives information about the visual program that was created for the research. Chapter 9 lists some tests performed and in Chapter 10 one can find the conclusions of the research. Finally, there is the Bibliography and an Appendix A with the found (existing and new) logic rules.

2. Rules of Takegaki and terminology

A *Takegaki puzzle* is played on a rectangular grid with numbers on some (but not necessarily all) of the entries on the grid. Here are the necessary definitions:

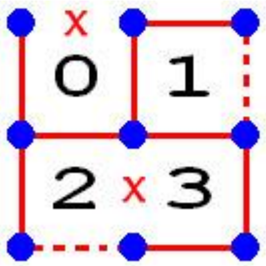


Figure 1

- An *entry* is a position (cell) on the grid, which may or may not contain a number. The places in ***Figure 1*** containing numbers are entries (here all entries have a number, but they can be empty).
- Each entry has four *edges* around it; to be precise above, below, to the left and to the right of it. In ***Figure 1*** these are indicated by red lines (normal and dotted) and crosses. Each edge is either a *line* (the normal red lines in ***Figure 1***) or a *cross* "x" (indicating that there can never be a line there), or it can be empty, in which case it is *unknown* whether a line or a cross belongs there. In ***Figure 1*** these unknown edges are indicated by dotted lines, but in a Takegaki puzzle these edges will not be drawn, so a line or a cross can be inserted later on.
- *Vertices* are the points where (at least) two lines come together, indicated as blue dots in ***Figure 1***.

The end goal of the game is to create a single loop of lines, such that the loop never crosses or touches itself. A number at an entry indicates how many of its four edges should eventually have a line in the loop, while for entries without a number it doesn't matter how many lines they have at their edges. A new puzzle contains only some numbers, every edge is unknown. ***Figure 2*** shows a new puzzle on the left, and its solution on the right. It is not necessary to mark the edges that don't have a line with an "x" (one can leave them unknown), but it is usually convenient to do so when solving a puzzle.

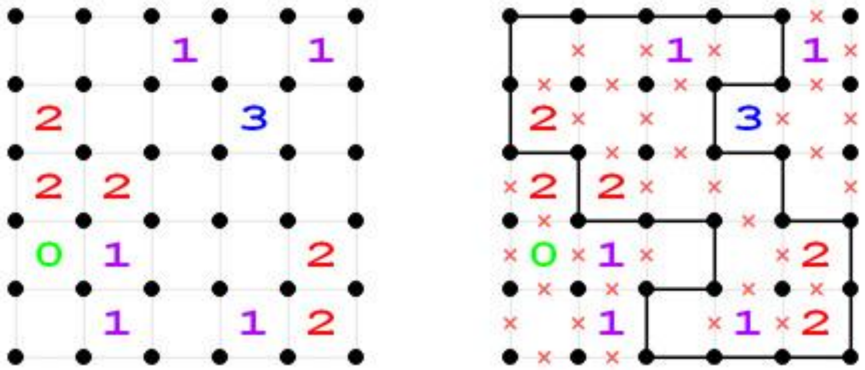


Figure 2

The numbers in an unsolved puzzle are chosen such that the puzzle has a unique solution, while removing any numbers results in a puzzle with multiple solutions. For example, if we remove the number 3 in the puzzle of ***Figure 2***, four additional situations are valid solutions. Takegaki puzzles always have only one solution. Takegaki is NP-complete, as proven by Takayuki Yato [3].

We use both relative and negative coordinates when referring to positions on the grid.

If we speak of *relative coordinates*, these are relative to a given place on the grid. These relative coordinates can indicate entries, lines and vertices (although we will never refer to vertices). ***Figure 3*** shows the way these positions are numbered relative to the entry in the middle of the grid and a couple of coordinates are also shown.

Absolute coordinates use the same system as relative coordinates, but here the coordinates are always relative to the vertex in the top left corner of the grid. In ***Figure 3*** the entry in the middle (marked there as (0,0)) has an absolute coordinate of (3,3).

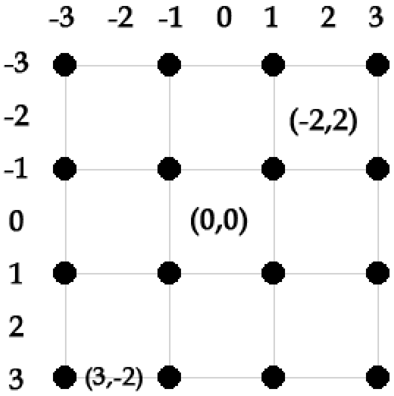


Figure 3

See [1] and [2] for more information about this type of puzzle.

3. Checking for a solved puzzle

An important task of any program that analyses Takegaki is to check whether or not the current situation is a solved puzzle or not. Since this check will have to be performed many times in both solving and generating puzzles, it is wise to spend some time to make this check as efficient as possible. The following sequence of steps has been found and used during the research:

1. Check if every number has the correct amount of lines, this is trivial to verify.
2. Check if every vertex has 2 or 0 lines touching it, this is also trivial to check.
3. Check if there is only one loop; we do this by picking a line from some loop and then following that around, until we are back at that line again. If there are unvisited lines, there are multiple loops.

An advantage of this approach is that the first step quickly sees that a puzzle is not finished; this step can determine for most situations that they are not solved puzzles. Another big advantage is that step 2 ensures that every line in the puzzle is part of a non-crossing and non–touching loop, because within a loop every vertex has either 2 or 0 lines adjacent to it; 1 line at a vertex would create a line that stops in the middle of nowhere and 3 or 4 lines would make the loop cross or touch itself, which is forbidden by the rules of Takegaki.

The order of the steps is important; any other order would probably lead to a less efficient algorithm, since the steps are now ordered according to how hard it is to execute them, from easy to hard.

Another task is to check whether or not a situation is a *promising* one, instead of a solved puzzle. A promising situation is a situation where a puzzle is not yet fully solved, but from where it is possible to get to a valid solution. This gives us the following slightly different sequence:

1. Check for every number if the amount of lines around it is less than or equal to its own value. If it is less, check if the correct amount can still be reached (this can for instance not be done if all the edges which are not lines already have a cross). We do this by counting the number of unknown edges around the number and checking if this is at least the number of lines that are needed for the entry to have the correct amount of lines.
2. Check if every vertex has at most 2 lines around it.

Here we cannot check if there is one loop, because the loop may not be finished yet. However, it is not necessary to know about this, the above steps are enough to see if a situation is promising. This information is used when solving puzzles with backtracking.

4. Solving with backtracking

The first approach used to solving puzzles was to use backtracking (see [4]), in which one step by step constructs solutions for problems while constantly checking whether or not the current direction can lead to a solution. If not, one heads back to the point where it went wrong and constructs further in a different direction than the one which lead to the non-promising situation. The following functions (in pseudo-code) have been used:

```
solveback(i, j)
  if(this edge has no line or cross)
    put a line at this edge;
    next(i, j);
    put a cross at this edge;
    next(i, j);
    mark this edge as unknown
  else
    next(i, j);

next(i, j)
  if(situationIsPromising())
    if(this is not the last edge)
      solveback(next edge);
    else
      if(isFinished())
        copy situation to solutions;
```

The variables `i` and `j` indicate the absolute coordinate (`i,j`) as described in Chapter 2. The algorithm starts at the edge above the entry in the top left corner of the grid, which has the absolute coordinate (`0,1`).

The function `situationIsPromising()` executes the steps for checking if a puzzle is *promising* described at the end of Chapter 3. This is where the pruning happens. Most of the situations are not promising at all, and this check skips most of these situations from the start.

In the created program there has also been inserted an option to set a maximum number of solutions to look for. This is particularly useful when generating new puzzles; we only want to see if a puzzle has multiple solutions, so we can stop when we have found two of them.

5. Solving with logic rules

Takegaki is a game where you start to see several patterns after you've played it a couple of times. The situation on the left in ***Figure 4*** will always end up in the situation on the right.

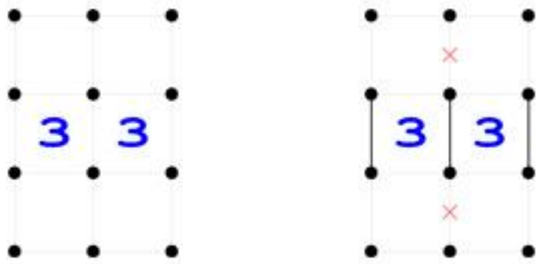


Figure 4

It is not hard to see why this is always true; for example, if we put a cross instead of a line at the edge to the left of the leftmost 3, we get the situation on the left in ***Figure 5***. The grid is now inconsistent, because the rightmost 3 can never get three lines around it anymore. The same reasoning can be applied for the rightmost line.

The edge between the two 3's has to be a line, because one would end up in the situation in the center of ***Figure 5***. This is a valid situation, but since we now have a very small closed loop this would allow no other edge on the grid to have a line. Chances are extremely small that a puzzle would consist of only this very small loop, so we make a line of the edge.

If we do insert the three lines at the correct places, but then put a line instead of the top cross in ***Figure 4***, we get the situation on the right in ***Figure 5*** (after applying some subsequent rules). This is also an invalid grid, because there is a vertex with three lines adjacent to it. So we should put a line instead of a cross at this edge, and the same goes for the other cross.

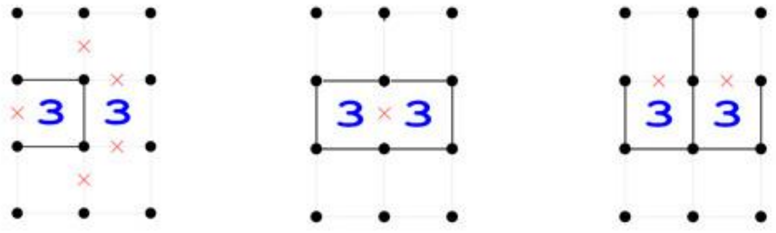


Figure 5

A human player will (perhaps unconsciously) use these patterns a lot to solve a puzzle, so I've simulated the use of these patterns when solving a puzzle. From now on these patterns will be called logic rules, and a logic rule will have to follow these guidelines:

1. A logic rule has some starting conditions; these are relative coordinates for an entry or a line on a grid with its value. For entries the value is simply the number on the entry (0, 1, 2, 3 or 4), and for lines the value is 0 for a cross and 1 for a line. In ***Figure 4*** these starting conditions are the two entries with 3 in them.
2. A logic rule has some ending conditions; these are the implied values for relative coordinates on the grid. These implied values must always be applicable if the starting conditions are met.
3. The starting and ending conditions can reference and edit both the values for numbers and lines on the grid.
4. The coordinates are always relative to an entry, but is doesn't matter which specific entry this is, as long as it is close to the relative coordinates.

I've created a very simple syntax to formalize these logic rules in files:

- A line starting with a percentage-sign "%" is a comment and will be ignored.
- A line starting with the character "s" indicates a starting condition, this line will be of the form "s *a*|*b*|*c*", where *a* and *b* are the x and y coordinates respectively on the grid and *c* indicates the value that that position should have.
- A line starting with the character "e" indicates an ending condition, this line will be of the form "e *a*|*b*|*c*", the characters *a* and *b* have the same meaning as above and *c* indicates the value that this position should have after applying the rule.
- A line starting with the character "q" indicates the end of the current logic rule. Any following starting and ending conditions belong to the next logic rule.

The order of these lines is not important, as long as each logic rule ends with a "q".

Here is an example of how to describe the rule in ***Figure 4*** with this syntax:

```
% rule number 14
s 0|0|3
s 0|2|3
e -2|1|0
e 0|-1|1
e 0|1|1
e 0|3|1
e 2|1|0
q
```

Appendix A lists the rules that were used during the research in this syntax. These logic rules have been found on the internet (like the above example, see [1]) and during the research. ***Figure 6*** shows a new rule found during the research.

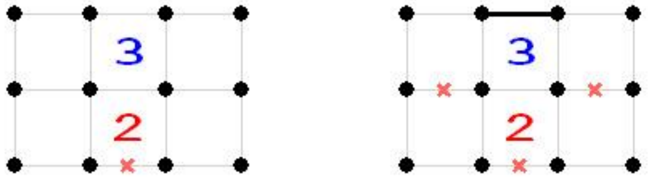


Figure 6

Whenever the program needs to apply these rules, it executes the following pseudo-code:

```
do {
  for(each available logic rule)
    for(each entry on the grid)
      for(each of the 8 symmetric variants of the rule)
        if(all the starting conditions are met)
          apply the ending conditions;
} while(changes have been made)
```

Every time the algorithm tests whether or not a certain logic rule is applied, the logic rules use the current entry (from the middle for-loop) as the center for the relative coordinates.

The eight mentioned symmetric variants of a rule are the rule itself, the 90°, 180° and 270° rotated versions and the mirrored versions of these four. These may not all be unique variations, a lot of rules are symmetrical in several ways (for example the rule depicted in ***Figure** 4*).

To check whether or not a starting condition holds, we need to keep something in mind: we can only check relative coordinates that are on the grid, except when a starting condition says something about a border that should be absent (marked with an "x"). Because a location outside of the grid is always absent, this condition is always valid if it references to a location outside the grid.

There is the question of what one should count as a logic rule; every puzzle together with its unique solution could be counted as a logic rule, since the starting situation will always end up in the one and only solution for that puzzle. And since puzzles can be of any size, to what size should we limit these logic rules? There may be countless rules for very large puzzles, but this would make the algorithm very slow, since it will try to apply every rule. There is a choice to be made between the number of logic rules to use within the algorithm; using only a couple of rules limits the number of solvable puzzles, but too many can make the algorithm too slow. The 36 rules used during the research have shown to be quite capable of solving hard puzzles, while still being fast to execute.

6. Avoiding loops

There are some puzzles which the logic rules cannot solve, but which are easy to solve by humans because of the fact that one can mark borders with a cross that would otherwise create new, small loops. An example can be seen in ***Figure** 7*, which shows the top half of a puzzle which has been partially solved by the logic rules.

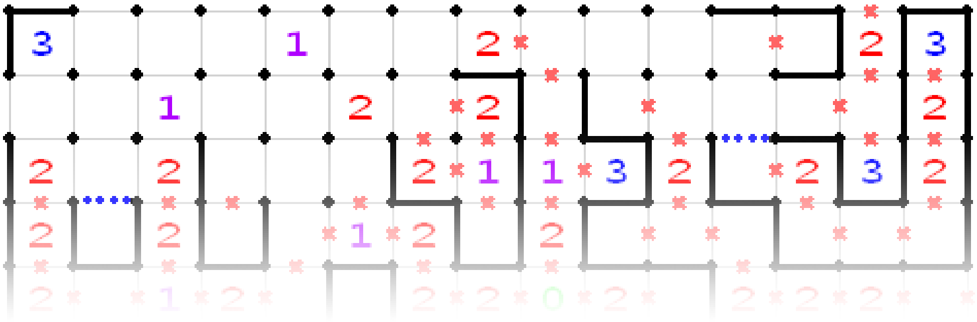


Figure 7

The logic rules are stuck here, but it is easy to see that the two edges indicated as blue dotted lines can never have lines there. If there were lines, there would be small loops besides the bigger loop and that is illegal, so we can safely mark these edges with a cross.

This strategy, from now on called *loop avoidance*, is to be performed every time the logic rules can't do anything and the puzzle is not solved yet. The strategy checks for every unknown edge whether or not it would create an illegal loop, and marks it with a cross if it does. If this strategy marks any edges with a cross, the logic rules can be applied yet again and this continues until the point where both the logic rules and this strategy can't find any new information for the puzzle. The following pseudo-code summarizes how one can combine logic rules and loop avoidance:

```
do {
  apply logic rules;
  if(applied no rules)
    loop avoidance;
} while(made changes)
```

The puzzle in ***Figure 7*** can be fully solved this way, while logic rules alone cannot solve it.

It is not necessary to choose whether or not to use this strategy, because the overhead is very little; up until puzzles of size 25x25 this strategy takes a maximum of about 0.150 seconds extra to execute. Since this overhead is very little and it expands the number of puzzles that can be solved to a large extent, from now on this strategy will always be used when solving puzzles with logic rules.

7. Generating new puzzles

Now that we have an efficient way of solving puzzles it is possible to generate new ones. The idea behind creating puzzles is to perform the following steps:

1. Create a loop.
2. Fill the grid with the correct numbers and set the edges to unknown.
3. Keep removing numbers at random until the last situation where the puzzle still has a unique solution. Any number removed after this leads to a puzzle with multiple solutions.

This sequence of steps can create every legal Takegaki puzzle; the following subchapters will elaborate on the finer details of these steps. An important thing to keep in mind is that a situation created by the first two steps can already have multiple solutions, in those cases one should start over again.

7.1 Creating loops

To create a loop, we start by creating a square at a random position and then growing this loop using some rules. The square is created by picking a random entry and putting lines at its four edges. I have found three rules that can generate every possible legal loop; conveniently enough these are logic rules like the ones used for solving puzzles, with the difference that for these rules we not only have to check whether the starting conditions hold, but the ending conditions also have to depict places that are on the grid. Another difference is that we do not simply apply every rule, but we apply them in a pseudo-random order for a fixed amount of time. ***Figure 8*** shows these three rules. Note that rule #2 and rule #3 are inverses of each other; this is to create more interesting loops. A cross means that in the current situation there is no line at that position.

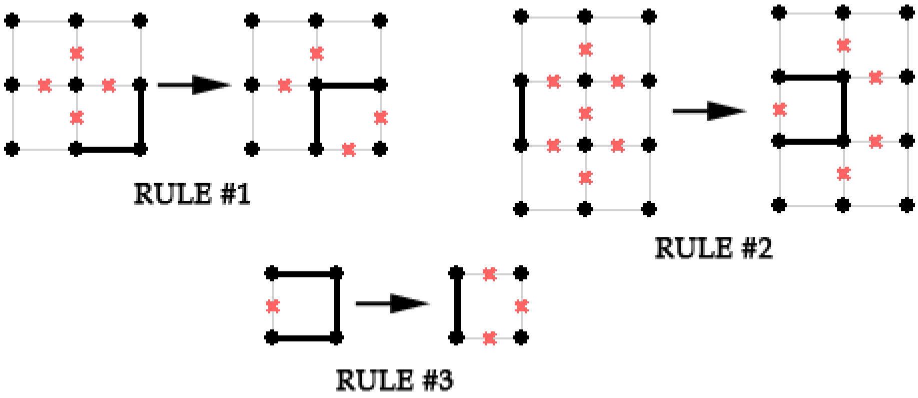


Figure 8

Figure 9 shows an example of how a loop can be created with these rules from an initial square.

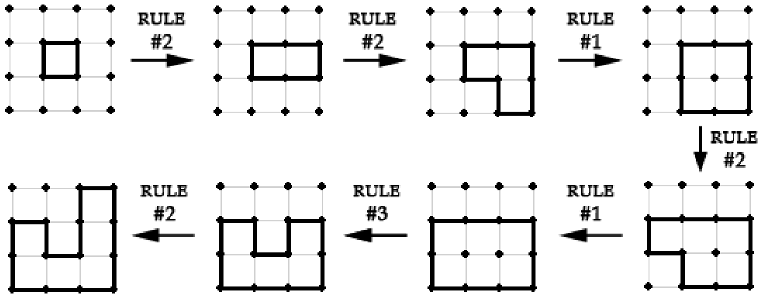


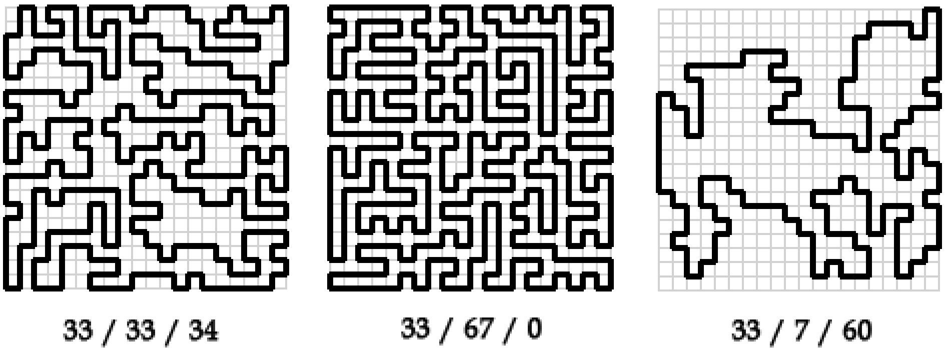
Figure 9

The mentioned pseudo-random order of applying the rules works as follows: each rule will be given a ratio between 0 and 100, the sum of the three ratios being 100. These ratios represent the chances for the individual rules to be used. For instance, a ratio of 90 for a rule will make sure it is used approximately 9 out of 10 times a rule is chosen at random, while a ratio of 5 will ensure that it will be very rarely used.

The standard division is simply to give each rule about 33% chance to be picked. This creates nice loops, but they are quite uniform in the sense that they will usually have relatively small corridors and the puzzle will span the entire grid. By changing the ratios for the rules a user can create loops that are more to his/her taste. These are some ratios and descriptions of the general loops generated by them:

- 33 / 33 / 34 : standard loops as described above
- 33 / 67 / 0 : extremely tight corridors, spanning the entire grid
- 33 / 7 / 60 : very broad loops, using a small portion of the grid

Figure 10 shows some loops created by these ratios.



33 / 33 / 34 33 / 67 / 0 33 / 7 / 60

Figure 10

The number of times these rules are applied also influence the final loop; I have found that you have to do this at least $n \times m$ times, with *n* and *m* being the width and height of the grid. During the research the algorithm executed these rules $n \times m \times 10$ times.

7.2 Preparing the grid

Now that we have a loop, the next step is to prepare the grid for the final step, which will delete numbers. To do this, we first fill every entry on the grid with the number that represents the number of lines at its four edges. After this we can remove every line on the grid by marking every edge with *unknown*. We now have a grid that contains only numbers, nothing else, and there are no empty entries on the grid. To avoid puzzles that have multiple solutions from the start, we check here if the grid completely filled with numbers has multiple solutions. If it does, we start over again.

7.3 Removing numbers

The final step in generating new puzzles is to remove as many numbers from the grid as possible, but only to the point where it is still uniquely solvable and after that point any number removed should create a puzzle with multiple solutions.

To remove as many numbers as possible, it is important not to simply pick numbers at random and stop when this leads to a non-uniquely solvable puzzle. This could result in very simple puzzles, because the algorithm could accidentally pick a couple of numbers in the same area and return a puzzle fully filled with numbers and only a small gap in that area. For this reason the random picking of the numbers is slightly more advanced: every time the algorithm is executed, the current available numbers are stored in a list. This list is then shuffled at random, and we start at the number that is now on top. If removing this number gives us a non-uniquely solvable puzzle, we put the number back and proceed to the next number in the shuffled list. If we do find a removable number, we actually do delete it from the grid and continue. If no removable number has been found at the end of the list, we stop, if we did delete numbers we reshuffle the remaining numbers and look for removable numbers yet again. The following pseudo-code is a simplified summary of this method:

```
do {
  shuffle remaining numbers;
  for(each number)
    delete number;
    if(!uniquely solvable)
      put number back;
} while(made a change)
```

There is an interesting choice to be made in this algorithm: since we now have two methods for solving puzzles (logic rules and a combination of logic rules and backtracking), we can use both options to check whether or not a puzzle is uniquely solvable. The combination can solve more puzzles, but it is way slower than using just logic rules. I have included both options in my program and the results are interesting: if we only use the logic rules to check if a puzzle is uniquely solvable, every generated puzzle can be solved with these rules since the generating algorithm stops when these logic rules can no longer solve the puzzle. If we also allow the algorithm to check the puzzle with backtracking, the puzzles become more challenging, because logic rules alone cannot solve them. From now on puzzles generated only with logic rules will be called *normal puzzles* and the ones generated with logic rules and backtracking *hard puzzles*.

8. Visual interface

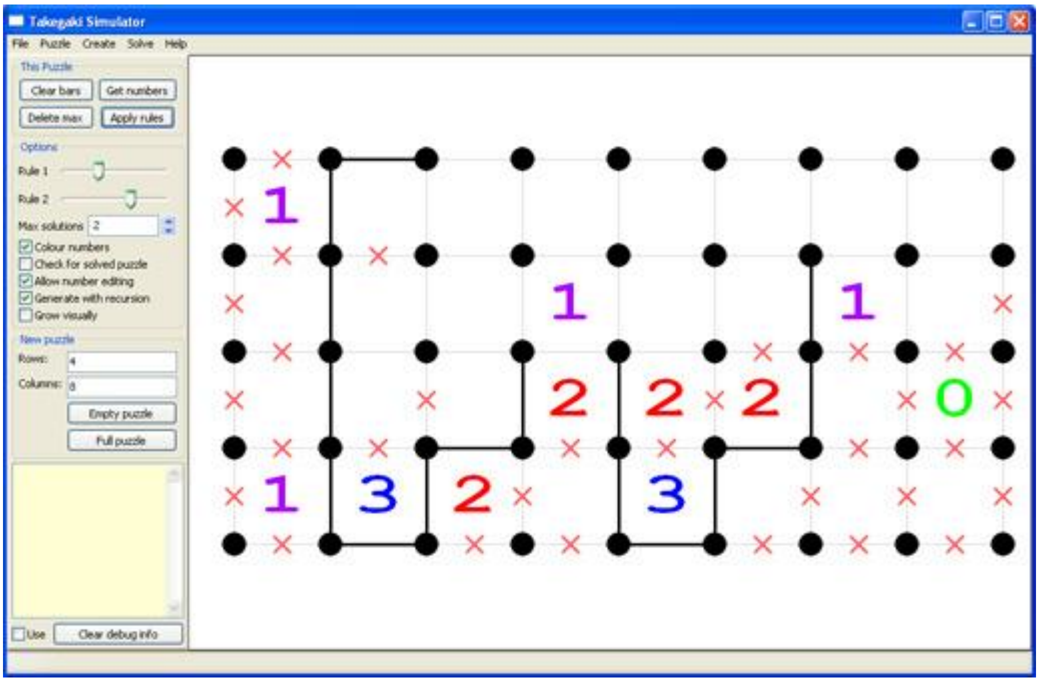
Because this type of puzzle and the research on it is quite visual, the decision soon was made to create a visual environment where all the different aspects of the research could be performed, instead of using a standard console application. This was all done with wxWidgets [5], which allowed me to quickly create the necessary windows without having to worry too much about the actual low-level creation of windows, so I could really focus on the research of the puzzle itself.

The end-product has the following general features:

- Play the actual game.
- Open and save specific puzzles.
- Automatically generate new puzzles, where one can choose the size of the puzzle and whether the program should generate normal or hard puzzles.
- Change the relative frequency of the three generation rules when creating new puzzles, so one can directly see what the influence is of each rule.

- Show the growth of a new puzzle in a visual way, so one can actually see the initial square expanding to form a bigger loop.
- A debug window (which can be switched off) that gives feedback about what exactly is done in the background.
- Load logic rules from a file and apply them at any time, whether a part of the puzzle is filled in or not.
- Completely solve a puzzle with backtracking, including the option to choose a certain solution if there have been found multiple solutions.
- Completely solve a puzzle with the logic rules and (if necessary) backtracking.
- Apply some specific funtions of the program, such as deleting as many numbers as possible and keeping its solution unique, clearing the edges and retrieving all the numbers on the grid when a puzzle is solved.
- Perform experiments.

The visual program was a big help during the research, because it provided a very intuïtive and fast way of looking at the way specific functions altered a puzzle. It was also a lot easier to recreate existing puzzles, without the program these puzzles would have to be inserted in text-files, which is a lot more work. ***Figure 11*** shows the program in action.



Max solutions: 2

☑ Colour numbers
□ Check for solved puzzle
☑ Allow number editing
☑ Generate with recursion
□ Grow visually

Rows: 4

Columns: 8

□ Use

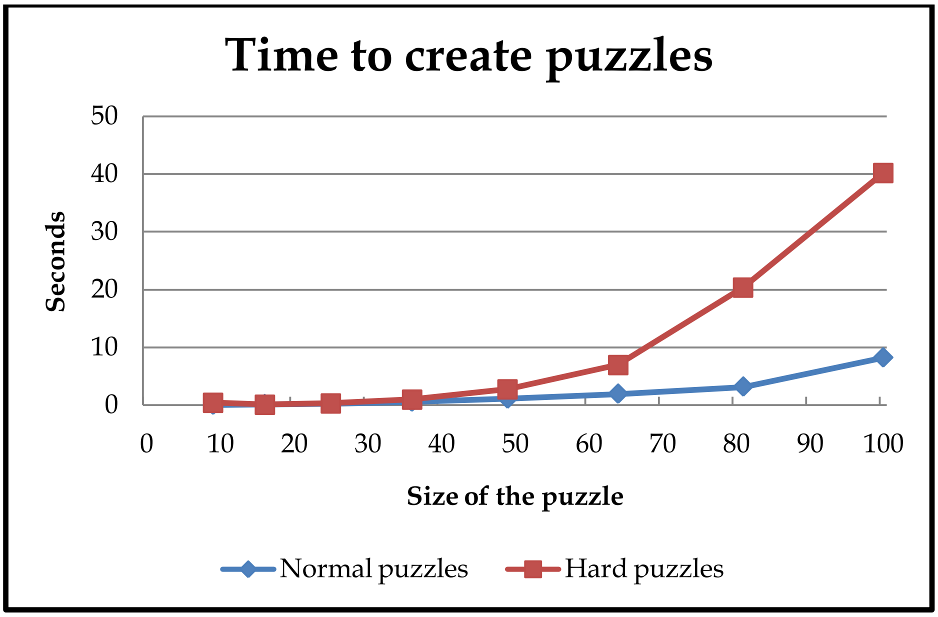
Figure 11

9. Experiments

To test the efficiency of the created algorithms, I have performed several experiments. These were all done in the visual program described in Chapter 8. All the graphs display the size of the puzzle on the x-axis, this means the numbers of entries on the grid. For instance, a 5x5 puzzle has a size of 25 and a 7x8 puzzle has a size of 56. The tests were performed 1000 times for each size and with square puzzles, since there is no significant difference with rectangular puzzles.

9.1 Time to create puzzles

The first experiment performed was to measure the time it took the algorithm to create new puzzles of size p^2, with $3 \leq p \leq 10$. Both the generation of normal and hard puzzles has been tested. ***Graph 1*** shows the results of the experiments.

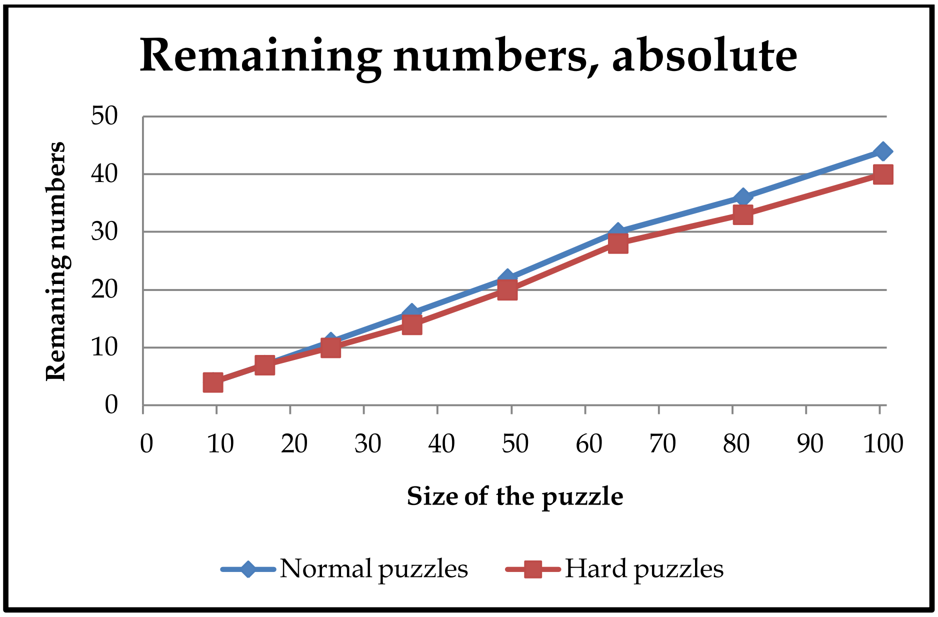


Graph 1

The graph makes clear that it is much harder to generate hard puzzles than it is to create normal puzzles. This is obvious, because the backtracking-method used for the hard puzzles is way less efficient than the logic rules. However, it may still be wise to generate hard puzzles, because these puzzles are more interesting. On the other hand the normal puzzles are challenging enough for beginning and intermediate players, so they can use the much faster generated normal puzzles.

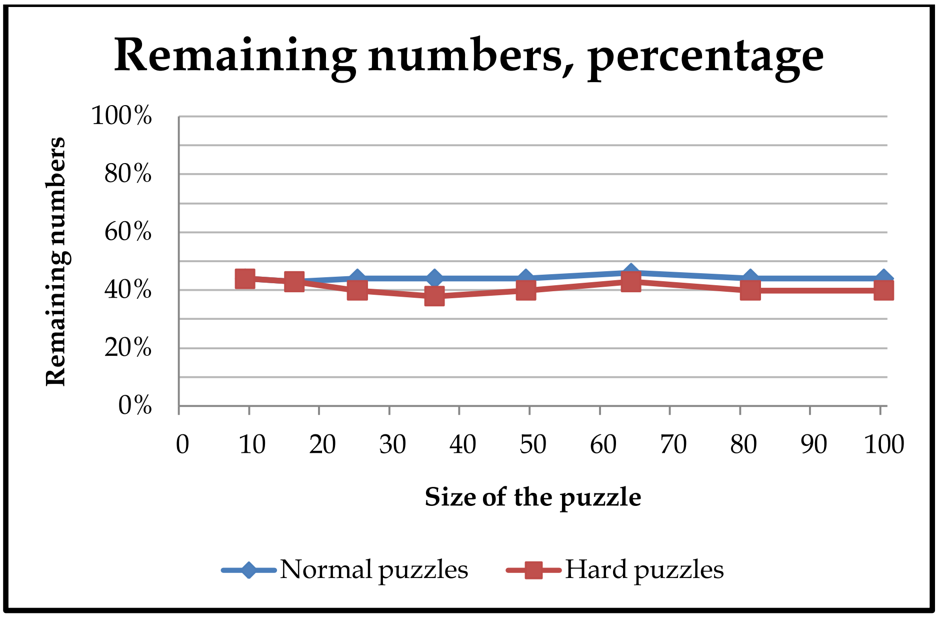
9.2 Average remaining numbers

Another interesting experiment was to see how many numbers remain on the grid after deleting as many as possible as described in Chapter 7.3. Again we did the experiment for both normal and hard puzzles; the results are shown in ***Graph 2***.



Graph 2

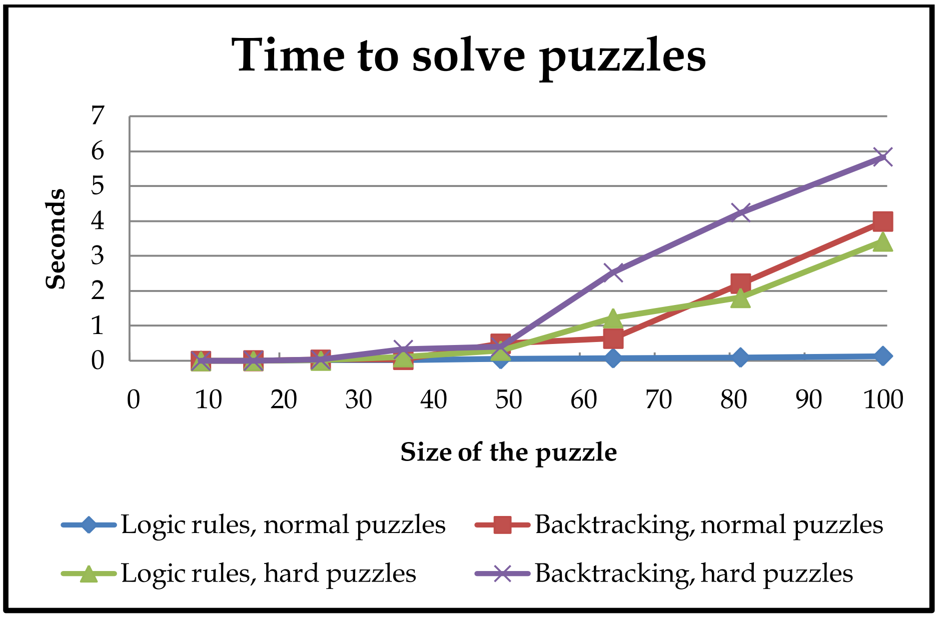
As we can see from the graph, there is not much difference between the normal and hard puzzles, but the hard ones always have slightly less numbers remaining. An interesting fact emerges when we view the remaining numbers relative to the size of the puzzle, as in ***Graph 3***. The normal puzzles end up with approximately 44% of the size of the grid in numbers, and for the hard puzzles this is 41%. It seems that there is a fixed ratio of empty and filled entries for unsolved puzzles, and the puzzles that can be played online at [6] also seem to follow this found ratio.



Graph 3

9.3 Time to solve puzzles

It is of course interesting to look at the difference in speed between solving puzzles with logic rules (and if necessary backtracking if the logic rules don't completely solve it) and backtracking alone. Another option for these tests is whether the tested puzzles are normal or hard puzzles. Therefore both solving methods have been tested on both normal and hard puzzles. ***Graph 4*** shows the results of these tests.



Graph 4

For both normal and hard puzzles the combination of logic rules and backtracking performs better than backtracking alone. This is to be expected; the logic rules perform some preparation for the backtracking, which then has a smaller tree of possible grid situations to explore.

In normal puzzles the difference between the combination and backtracking alone is big; the combination of logic rules and backtracking is much more efficient, because the puzzle can simply be solved by applying the logic rules, this is the way these puzzles were constructed in the first place. For hard puzzles the gap is much smaller, this is because these puzzles were generated using backtracking from the beginning, so the logic rules can't fully solve them most of the time.

10. Conclusions

This research has shown that it is indeed possible to efficiently solve Takegaki puzzles using logic rules. In every case applying the logic rules decreases the search tree, and the results in Chapter 9.3 show the efficiency of this method.

There has also been found an efficient way of generating new puzzles, and there are even some options to set when doing this; the ratios of the growth rules and whether or not to use backtracking influence the result a lot. The time it takes to create the puzzles is reasonable and much better than expected at the beginning of the research, especially the puzzles that are generated using only the logic rules, which are challenging and fun enough for most players of Takegaki.

There still are some things that are interesting enough to look at in the future: how many logic rules are there? This is not simply a matter of computing every possible configuration and checking what rules follow them, because that would make a new rule for every individual puzzle, which could lead to millions of rules. Another thing to look into is whether or not some machine learning algorithms such as Neural Networks and Genetic Algorithms (see [7]) can efficiently solve puzzles. One can also imagine that there might be totally different approaches to constructing loops when generating new puzzles, although the method suggested in this paper is already an efficient one.

Bibliography

1. Slitherlink. *Wikipedia.* [Online] [Accessed: June 10, 2008.]
`http://en.wikipedia.org/wiki/Slitherlink`

2. *Puzzles > Slitherlink [Nikoli].* [Online] [Accessed: June 30, 2008.]
`http://www.nikoli.co.jp/en/puzzles/slitherlink/`

3. Yato, Takayuki. *On the NP-completeness of the Slither Link Puzzle.* [Online] [Accessed: April 13, 2008.]
`http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL74-3.pdf`

4. Backtracking. *Wikipedia.* [Online] [Accessed: June 30, 2008.]
`http://en.wikipedia.org/wiki/Backtracking`

5. *wxWidgets.* [Online] [Accessed: June 9, 2008.]
`http://www.wxwidgets.org`

6. *Slither Link - online puzzle game.* [Online] [Accessed: June 10, 2008.]
`http://www.puzzle-loop.com`

7. Russell, Stuart and Norvig, Peter, *Artificial Intelligence: A Modern Approach (second edition)*, New Jersey: Pearson Education, 2003 - ISBN 0-13-080302-2

Appendix A

A listing of the found logic rules, formulated in the proposed syntax of Chapter 5.

```
% #1
% 0 -> 0x
s 0|0|0
e 1|0|0
q

% #2
%        x
% |1  -> |1x
%        x
s 0|0|1
s 1|0|1
e 0|-1|0
e -1|0|0
e 0|1|0
q

% #3
% x      x
% 1x  -> |1x
% x      x
s 0|0|1
s 0|-1|0
s 1|0|0
s 0|1|0
e -1|0|1
q

% #4
%  -      -
% |2   -> |2x
%         x
s 0|0|2
s -1|0|1
s 0|-1|1
e 1|0|0
e 0|1|0
q

% #5
%  x      x
% x2   -> x2|
%         -
s 1|0|0
s 2|-1|0
s 2|0|2
e 2|1|1
e 3|0|1
q
```

```
% #6
%          -
% x2x  -> x2x
%          -
s 1|0|0
s 2|0|2
s 3|0|0
e 2|-1|1
e 2|1|1
q

% #7
%          x
% |2|  -> |2|
%          x
s 1|0|1
s 2|0|2
s 3|0|1
e 2|-1|0
e 2|1|0
q

% #8
%  x      x
%  3   -> |3|
%         -
s 0|0|3
s 0|-1|0
e 0|1|1
e -1|0|1
e 1|0|1
q

% #9
% |     |
%   -> x x
% |     |
s 1|0|1
s 1|2|1
e 0|1|0
e 2|1|0
q

% #10
%         x
%  -   -> x -
% |       |
s 1|0|1
s 2|-1|1
e 0|-1|0
e 1|-2|0
q
```

```
% #11
% x        x
%  -   ->  - -
% x        x
s 1|0|0
s 1|2|0
s 2|1|1
e 0|1|1
q

% #12
%  x      x
%   x -> - x
%  |      |
s 1|0|0
s 1|2|1
s 2|1|0
e 0|1|1
q

% #13
%  x      x
%   x -> x x
%  x      x
s 1|0|0
s 1|2|0
s 2|1|0
e 0|1|0
q

% #14
s 0|0|3
s 0|2|3
e -2|1|0
e 0|-1|1
e 0|1|1
e 0|3|1
e 2|1|0
q

% #15
s 0|0|3
s 1|2|1
e -2|-1|0
e -1|-2|0
e -1|0|1
e 0|-1|1
e 2|1|0
q
```

```
% #16
s 0|0|3
s 2|2|3
e -2|-1|0
e -1|-2|0
e -1|0|1
e 0|-1|1
e 2|3|1
e 3|2|1
e 3|4|0
e 4|3|0
q

% #17
s 0|0|1
s 0|2|1
s 2|1|0
e 0|1|0
q

% #18
s 0|0|1
s 0|2|3
s 2|1|0
e -1|0|0
e 0|-1|0
e 1|2|1
q

% #19
s 0|0|1
s 0|2|3
s 2|2|1
e -1|0|0
e 0|-1|0
e 2|3|0
e 3|2|0
q

% #20
s 0|0|1
s 1|2|0
s 2|1|0
e 0|1|0
e 1|0|0
q

% #21
s 0|0|1
s 1|2|0
s 2|1|1
e -1|0|0
e 0|-1|0
q
```

```
% #22
s 0|0|3
s 0|2|2
s 0|3|0
e -2|1|0
e 0|-1|1
e 2|1|0
q

% #23
s 0|0|3
s 0|3|1
s 1|2|1
e -1|2|0
q

% #24
s 0|0|3
s 0|3|1
s 2|2|3
e -1|2|0
q

% #25
s 0|0|3
s 0|4|3
s 1|2|1
e -1|2|0
q

% #26
s 0|0|3
s 0|4|3
s 2|2|3
e -1|2|0
q

% #27
s 0|0|3
s 1|2|0
s 2|1|0
e 0|1|1
e 1|0|1
q

% #28
s 0|0|3
s 2|2|2
s 2|3|0
e -2|-1|0
e -1|-2|0
e -1|0|1
e 0|-1|1
e 3|2|1
q

% #29
s 0|0|3
s 2|2|2
s 3|4|1
e -2|-1|0
e -1|-2|0
e -1|0|1
e 0|-1|1
e 4|3|0
q

% #30
s 0|0|3
s 2|2|2
s 4|4|3
e -2|-1|0
e -1|-2|0
e -1|0|1
e 0|-1|1
e 4|5|1
e 5|4|1
e 5|6|0
e 6|5|0
q

% #31
s 0|1|0
s 0|2|2
s 1|4|1
e -1|2|1
e 2|3|0
q

% #32
s 0|1|0
s 0|2|2
s 2|3|1
e -1|2|1
e 1|4|0
q

% #33
s 0|1|1
s 0|3|1
s 1|2|1
e -1|2|0
q

% #34
s 0|1|1
s 0|3|1
s 2|2|3
e -1|2|0
q

% #35
s 0|1|1
s 2|2|2
s 3|4|1
e 1|0|0
e 4|3|0
q

% #36
s 0|1|1
s 2|2|2
s 4|3|1
e 1|0|0
e 3|4|0
q
```