Hashiwokakero

T. Morsink

August 31, 2009

Contents

1 Introduction

This thesis is the result of a Bachelor Project at the University of Leiden in the Netherlands. The project was supervised by Dr. W.A. Kosters. It is about Hashiwokakero, a Japanese puzzle. The object of the puzzle is to connect islands by means of bridges into a single connected group. With this thesis we will show how efficient our own solver is in contrast to another proven one.

2 What is Hashiwokakero?

2.1 The rules

Hashiwokakero (also known as Bridges) is an interesting Japanese puzzle from Nikoli [1], the publisher of puzzles like Sudoku and Nonograms. The rules are quite simple, yet within the rules one can make pretty difficult puzzles. Hashiwokakero is played on a rectangular grid. The size of the grid doesn't really matter. In the grid the player will find nodes, also known as "islands"(see the left picture of Figure 1). These nodes have a value in them ranging from 1 through 8. The goal is to connect all these islands with bridges. In the end this should be a single connected group. The player draws bridges between the islands in a straight line. These bridges cannot cross other bridges and/or islands, and each bridge connects exactly two islands. Every pair of islands has a maximum of 2 bridges connecting them. In the end you will have a single connected group with on each island the number of connected bridges matching the value in the island (see the right picture in Figure 1).

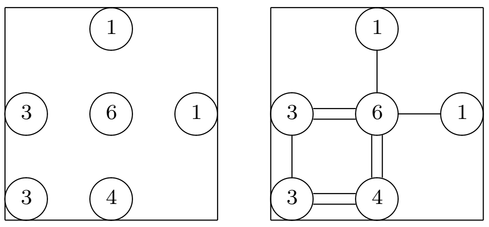


Figure 1: Example puzzle with solution

2.2 Easy solving statements

To solve these puzzles there are a number of easy things to remember. If you have, for example, an island with value 4 in the corner of your grid, you have only one way to build bridges. Since a pair of islands cannot be connected with more than 2 bridges, the island with value 4 will have 2 bridges to his first neighbouring island and 2 bridges to his second neighbour. This simple fact is easily translated into a bit more formal statement:

"If an island has value n, with n even, and precisely $n/2$ neighbours, you can draw two bridges to each of n's neighbours."

If an island has value $n-1$, with n even, with $n/2$ neighbours, you cannot put two bridges to each of n's neighbours, but you do know already something else. If an island has value $n-1$, you can draw *at least one bridge* to each of its neighbours.

With these two simple statements you can solve most puzzles of easy difficulty. However, the puzzles of hard difficulty will give you some problems. Hard puzzles will use the fact that it needs to be a single connected group of islands and give you the illusion there is more than one way to solve the puzzle.

3 Building an Own Solver

3.1 The easy solver

As a first part in this research project, a program in C++ was built to solve these puzzles, using basically the two simple rules of solving mentioned in Section 2. This is a pretty simple program, but as soon as puzzles needed the "single connected group rule" to get solved, it fails. Also the program had some difficulties solving certain puzzles, for example it connected two islands of value 1 to each other, or did not draw a bridge where there clearly should be one due to lack of checks or the fact that it needed so many checks that is was better to work on a new program.
This new program was designed to basically just solve the puzzles without the "single connected group rule". The approach was a bit different. If you have a node of value k, calculate the maximum of bridges (after this called max) you can attach to this node, ignoring the value of k, but not ignoring the values of the neighbours. If after this calculation $k = max$, draw all bridges to the neighbours. If $k = max - 1$ AND $k > m$, with m being the number of neighbours of the node, you can draw one extra bridge to all neighbours

(except to nodes with a value left of 1, these are only drawn with the first rule), i.e., if there is one bridge draw the second, and if there is no bridge then draw one. From now on if the last approach is used, we will refer to it as the easy solver.

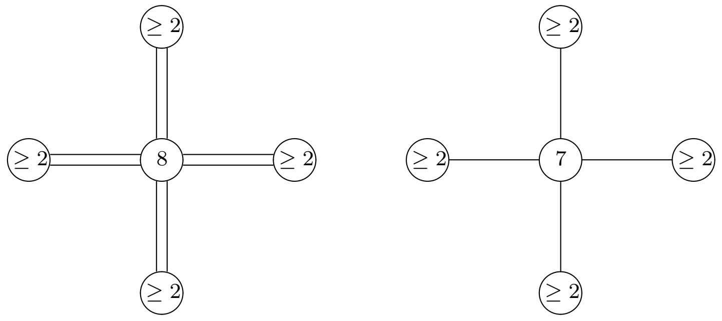


Figure 2: $n = 8$. Left: neighbours have value ≥ 2. Right: neighbours have value ≥ 2 and the middle node has value $n - 1$

Note in Figure 2 in the left picture with n being 8, that every neighbour of that node will need at least a value of 2 or higher to let the node with value 8 draw all of it's bridges. If any of the neighbours would have a value of 1 or even 0, the middle node would only be able to draw, respectively, seven or six bridges. In the right picture of Figure 2 you see what happens when a node has a value of 7. If every neighbouring node has at least a value of 2 and you have 7 bridges to distribute, there is no indication of which pair of nodes should only be connected with only one bridge. The only real fact we know is that every pair of nodes should be connected with at least one bridge, because you still need to distribute 7 bridges over four pairs of nodes.

3.2 The better solver

After seeing that the easy solver was not going to solve puzzles with the "single connected group rule" in them, we needed some extra solving capabilities in the program. The easiest way to get some extra solving power in the program was to let the program guess once and then use the same solving paramaters as in the Easy Solver. So the steps in the new program were going to be:

1. Solve the puzzle as best you can with the easy solver.

2. Let the program draw a single bridge (a double one if there already is one; this is referred to as an extension) and, after the one guess, solve the puzzle again with the easy solver.
3. Check if a solution is found and if the solution is a single connected group, if not make a note that on that spot no bridge extension can be drawn again.
4. Repeat step 2 and 3 for every possible bridge extension.

Note that in this case, if every possible bridge leads to an unsolvable puzzle, the program might also indicate that a solution is not possible. Note that, contradictory to normal research behavior, if after the one guess and solving it again the solver happens to find a solution, the program will end saying it has found a solution to the problem. This solution is probably a non-unique solution.

3.3 Single connected group

The check to see if the puzzle is a single connected group requires some extra explanation. The check will start at every node and will try to reach every other node in the puzzle. If it cannot reach a node it will check why the puzzle still is not a single connected group. There are a two distinct cases in which you will see the guess of the solver fail, causing it to ban all bridges between two islands, and one case in which options are still open, and the check solver cannot say if the puzzle is solvable, but it can say that at the moment it is not unsolvable. These three situations are shown in Figure 3. Note that the pictures are not complete puzzles, but small portions of a larger one. The three situations are:

1. There is a group of islands with at least a possibility to draw a bridge to another node completely cut off from possibilities around it. This will cause the solver to fail; see Figure 3.1.
2. There is a group of islands connected together, with no more possibilities going outward to another part of the puzzle. Just to be clear, this is not a complete puzzle. This will cause the solver to fail; see Figure 3.2.
3. There is a group of islands connected together, with one or more possibilities to draw a bridge to another part of the puzzle, connecting those two or more groups of islands; see Figure 3.3.

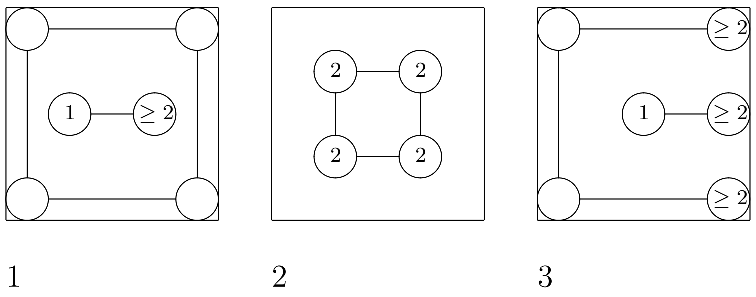


Figure 3: Examples of the options above. A part of the puzzle is depicted.

4 Solving the Puzzle with the Sugar CSP-Solver

While looking for ways to solve these puzzels, a solver called Sugar was found, made by Naoyuki Tamura [2]. Sugar is a SAT-based Constraint Solver. It is based on a new SAT-encoding method called "order encoding". We wanted to find out how well or how fast this solver could solve the puzzles in comparison with the solving methods from the programmed solver. Those methods use random functions a bit more, so they could take a lot of time.

4.1 Getting the input file format right

Sugar needs to get the problem handed in a special way. First a list of all variables with the appropriate constraints it is to be given. In our case the variables are potential bridges with a value between 0 and 2 (being the minimum and maximum number of bridges between two islands). In general the notation would be

```
(<variabletype> <variablename> <bottomconstraint> <topconstraint>).
```

So for example, `(int x 0 2)` would mean that the integer x would have a value $0 \leq x \leq 2$.And `(int x_00_06 0 2)` means that the integer variable `x_00_06` (the bridge between coordinates (0,0) and (0,6)) has to be between 0 and 2. After declaration of the variables you get a list of statements that sum the variables to a certain value. In this case this value was the value of the node and the variables summing to that value are the potential bridges attached to that node. So, for example, `(= (+ x_00_50 x_00_06) 2)` means that variable `x_00_50` summed with variable `x_00_06` should equal 2, meaning that the cross section of the lines going from (0,0) to (5,0) and from (0,0) to (0,6), so in this case the point (0,0), will need two bridges going from that point.

To accomplish this format we made a parser that would take a puzzle we created in the form we use in all our programs and write it to the form required for Sugar, as seen in Figure 4.

```
;bridges
(int x_00_50 0 2)
(int x_00_06 0 2)
(int x_06_66 0 2)
(= (+ x_00_50 x_00_06) 2)
(= (+ x_06_66 x_00_06 ) 2)
(= (+ x_00_50 ) 1)
(= (+ x_06_66 ) 1)
```

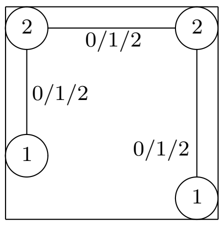


Figure 4: Example `inputfile.csp` format, with corresponding puzzle. Here 0/1/2 denotes the possible values for bridges. A line that starts with `;` contains a comment.
No attempt is made to guarantee connectedness.

The problem with the the `inputfile.csp` from Figure 4 is that there is no way to ensure that the solution is a single connected group. To that end you need a system to build a spanning tree. Naoyuki Tamura [2] made a script that, besides making the statements that our self built csp code generator also does, makes such a tree. The new `inpufile.csp` generated from this script is seen in Figure 5. As you can see in Figure 5 the first seven lines are basically (if noted somewhat different) the same commands as in Figure 4. The difference is in that the bridges have directions. The numbers in the variable names are still the coordinates only for this example `v_0_0` means the vertical bridge that is drawn from node `(0,0)` and `h_0_0` is the horizontal bridges drawn from node `(0,0)`. The values -1 and -2 mean in this case, that vertical bridges have a direction upwards and horizontal bridges have a direction to the left. The values 1 and 2 are for directions down and right. The sum of the absolute values of the bridges attached to a node (the horizontal and vertical ones) are again the value of the node. Now this is established we introduce a group of new variables, 'z'-variables and 'r'-variables, corresponding to the nodes in the puzzle. The 'z'-variables always have a value between 1 and the number of nodes in the puzzle, while the 'r'-variables always have a value of either 0 or 1. Now there is a group of iff-statements. For example, `(iff (= z_0_0 1) (= r_0_0 1))` means that the variable `z_0_0` is 1 if and only if `r_0_0` is 1.

The next important thing to note is the line:

```
(int v_0_0 -2 2)
(int h_0_0 -2 2)
(= (+ (abs v_0_0) (abs h_0_0)) 2)
(int v_0_6 -2 2)
(= (+ (abs v_0_6) (abs h_0_0)) 2)
(= (+ (abs v_0_0)) 1)
(= (+ (abs v_0_6)) 1)
(int z_0_0 1 4)
(int r_0_0 0 1)
(iff (= z_0_0 1) (= r_0_0 1))
(int z_0_6 1 4)
(int r_0_6 0 1)
(iff (= z_0_6 1) (= r_0_6 1))
(int z_5_0 1 4)
(int r_5_0 0 1)
(iff (= z_5_0 1) (= r_5_0 1))
(int z_6_6 1 4)
(int r_6_6 0 1)
(iff (= z_6_6 1) (= r_6_6 1))
(= (+ r_0_0 r_0_6 r_5_0 r_6_6) 1)
(=> (> v_0_0 0) (< z_0_0 z_5_0))
(=> (> h_0_0 0) (< z_0_0 z_0_6))
(=> (= r_0_0 0) (or (< v_0_0 0) (< h_0_0 0)))
(=> (> v_0_6 0) (< z_0_6 z_6_6))
(=> (< h_0_0 0) (< z_0_6 z_0_0))
(=> (= r_0_6 0) (or (< v_0_6 0) (> h_0_0 0)))
(=> (< v_0_0 0) (< z_5_0 z_0_0))
(=> (= r_5_0 0) (or (> v_0_0 0)))
(=> (< v_0_6 0) (< z_6_6 z_0_6))
(=> (= r_6_6 0) (or (> v_0_6 0)))
```

Figure 5: Example of a better `inputfile.csp` file for the puzzle in Figure 4 Here connectedness is taken into account.

```
(= (+ r_0_0 r_0_6 r_5_0 r_6_6) 1)
```

This line says that the sum of all ‘r’-variables is 1. This means that precisely one of these variables can have the value 1. The idea behind this is, that from all the nodes in the puzzle, one is chosen to be the root of the spanning tree, we are trying to make. In this example we will chose `r_0_0` as the root, thus giving it a value of 1. Now because this is chosen, there are a few other effects that will happen. First off all, because of the line

```
(iff (= z_0_0 1) (= r_0_0 1))
```

the variable `z_0_0` which could have had a value between 1 and 4, will now have value 1. Also, because all of the ‘r’-variables together can only have a value of 1, every other ‘r’-variable except `r_0_0` has a value of 0 and thus the ‘z’-variables will not have a value of 1. Now because of the lines

```
(=> (= r_5_0 0) (or (> v_0_0 0)))
```

and

```
(=> (= r_6_6 0) (or (> v_0_6 0)))
```

we see that `v_0_0` will be larger than 0 and `v_0_6` will also be larger than 0, meaning that both these bridges have a direction downwards. Now we only need the direction of the horizontal bridge. This is deducted from the line:

```
(=> (= r_0_6 0) (or (< v_0_6 0) (> h_0_0 0)))
```

Since `r_0_6` equals 0 either `v_0_6` must be smaller than 0 or `h_0_0` must be larger than 0. We just established that both vertical bridges run downwards and have a value greater than 0, so `v_0_6` is not smaller than 0, thus `h_0_0` must be larger than 0 and consequently this bridge has a direction to the right. Now we have all the directions we need and we can look at the final part of building a tree. The lines

```
(=> (> v_0_0 0) (< z_0_0 z_5_0))
(=> (> h_0_0 0) (< z_0_0 z_0_6))
(=> (> v_0_6 0) (< z_0_6 z_6_6))
```

tell us the last pieces of information. In all the lines of the building of the tree, we have not given any values to 'z'-variables of the nodes, except we chose the root to have value 1. We still need to assign numbers to the other three 'z'-variables corresponding to the other three nodes. In the three statements above the first part is true, all the variables are larger than 0. We can see that `z_0_0 < z_5_0`, which should not be hard to do since `z_0_0` equals 1, so we, perhaps, assign a value of 2 to `z_5_0`. This is acceptable since the 'z'-variables all have values between 1 and the number of nodes, in this case 4. Other values are also possible as long as it is a value between 1 and the number of nodes and a path from the root of the tree to a leaf will only encounter increasing numbers. The next two lines show us that `z_0_0 < z_0_6 < z_6_6`, which is also doable within the rules. Assign a value of 2 to `z_0_6` and a value of 3 to `z_6_6`. Now we have a graph with directions and values increasing in value the further away from the root they are, see Figure 6. Here we see that we have a spanning tree to all nodes and thus we have a single connected group within the solution.

In general, a spanning tree can be constructed (thus having a connected group) precisely when the formulas can be satisfied.

4.2 Using Sugar

After the file is parsed, it is time to start solving the puzzle. One of the more interesting things of using Sugar is that you also need to use a SAT-Solver. In

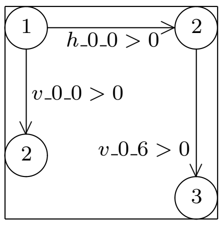


Figure 6: The spanning tree corresponding to the puzzle in Figure 4, with the ‘z’-values in the nodes

our case we used MiniSat [3], a very easy to use solver with no configuration needed. When that is set up, it is time to let Sugar encode the input file to a file that MiniSat can use to solve your puzzle. Using the command

```
java -jar ./sugar.jar -encode <file.csp> <file.cnf> <file.map>
```

will do that for you; here `sugar.jar` is the current version of the java archive from which Sugar is run. In this case Sugar is installed in the same directory as the rest of the files. The `.csp` file is the input file containing the problem you want to solve. The `.cnf` file is an output file from Sugar used as an input file for MiniSat to solve the problem. The `.map` files used here are files used by Sugar to store the variable data with the constraints, so called mapping files. After encoding the files a simple command

```
./MiniSat <file.cnf> <file.out>
```

will solve the puzzle. The `.out` file will be used again by Sugar as an input file. You will immediately see if the puzzle was solvable or not. Now you have a couple of files with the solution to the puzzle, which are still not easily readable. To make it clear and immediately understandable what the solver did you need to decode the files again with Sugar using the command

```
java -jar ./sugar.jar -decode <file.out> <file.map>.
```

This will give you a list with the variables you declared in your file with their value behind it.
To give you a bit of a clearer idea, here are the `.cnf`, `.map` and `.out` files correspeonding to Figure 4. First the `.cnf` file:

```
p cnf 0 0
```

Secondly the `.map` file:

```
int x_00_06 1 1..1

int x_00_50 1 1..1

int x_06_66 1 1..1
```

Last the `.out` file:

```
SAT

0
```

4.3 Expectations Sugar vs own solver

Sugar will probably be able to solve most to all puzzles given, but we are not sure at the timeframe. Inputting the three different commands does take a bit of time, which is not really efficient. On the other hand with larger puzzles the programmed solver will probably have to take a couple of guesses randomly which can be completely wrong and then it will take a lot of time to find the correct solution, so then the time you need to enter the three Sugar commands will be faster. In the chapter Results, you will find the results of the test to see which is faster.

5 Making a Puzzle Generator

Another part of this project was to make a puzzle generator. Copying puzzles from internet sites is a good way to know you have uniquely solvable puzzles, but it is time consuming. To ensure that the puzzles the generator makes are within the rules of the puzzle the generator followed these steps:

1. Put a single random node somewhere on the board.
2. From a random node already on the board, draw a single or double bridge (decided randomly) in a random direction. This will not happen if the node has no more options to draw a bridge and a new node from where to draw a bridge will randomly be chosen.
3. During this traversal, on every grid point decide whether to go on drawing the bridge or to stop and put an island on that spot. If you encounter a crossing bridge at the grid point, place the node on the bridge, if possible, or otherwise place the node one step back, if possible.

4. Continue steps 2 through 4 until you have reached the number of nodes you'd like to have (given in the command line when starting the generator).

5. Check at each node how many bridges are connected and give the node the proper value.

6. Remove the bridges from the puzzle and it is complete.

What you see in Figure 7 is a puzzle with three nodes being generated step by step. In step I a single island is put on the board. In step II the generator will choose automatically that one node and decide to either draw a single or double bridge. In this case a double bridge was chosen and it will go down to when it decides to stop and put down a node. In step III another node is chosen to draw a bridge from and this time it's the lower of the two. A single bridge is drawn to the right, until it stops again and a node is placed. In step III the values of the nodes are also inserted as a final step in making the puzzle complete. Step IV, the final step, shows the puzzle as it will be saved. No bridges, just nodes that need solving. In this example, the puzzle is uniquely solvable. Furthermore, the resulting solution has a single connected group.

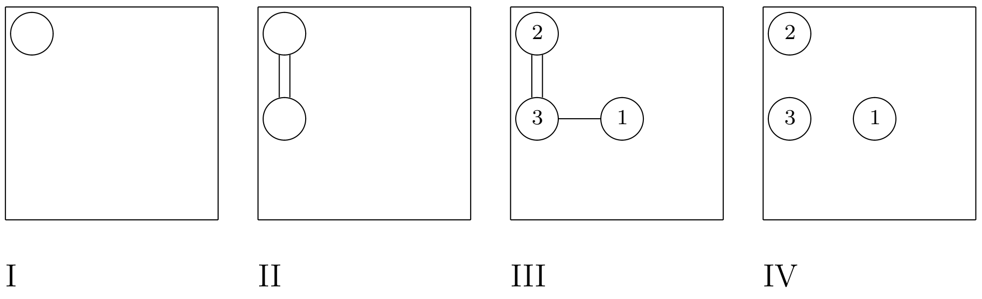


Figure 7: Generating a puzzle

6 Results

In this section, we will show how the self built solver (called just solver in the next part) did against the Sugar program. We would like to show how efficient the solver is in time and number of solved puzzles.

6.1 Experiments

For the experiments we needed to test the generator, the self written solver and Sugar. This was done in two different ways. First we generated and solved with our self built solver 7×7 puzzles, 10×10 puzzles and 15×15 puzzles. The 7×7 puzzles were generated with a maximum number of 7 nodes, and thus usually with 7 nodes in the puzzle. The 10×10 puzzles were generated with a maximum of 15 nodes, usually generating puzzles with 15 nodes. The 15×15 puzzles were generated with a maximum of 30 nodes, again usually generated with 30 nodes. In generating the puzzles there is a random chance which could cause, for example, to let the generator put nodes in all 4 corners of the puzzle and connecting them all with one bridge between them. In this case you have only 4 nodes and the generator will try 10,000 times to put another bridge from a node, which will fail, thus, for example, a 15×15 puzzle with only 4 nodes is created. There are more scenarios in which this can happen. That is why we say that puzzles are usually generated with the number of nodes specified.

Secondly we solved the same puzzles with Sugar. First the 10,000 7×7 puzzles were generated and solved with the solver and then with sugar. After this was done the same was done for the 10,000 10×10 puzzles and the 10,000 15×15 puzzles. Every run made a lot of output files, merging them into a few at the end. We measured a few interesting key points like time taken to do a job of 10,000 puzzles and how many were solved. Also we examined how the generator generated the puzzle and if the solver and Sugar solved it in the same way or that there were different solutions to a puzzle.

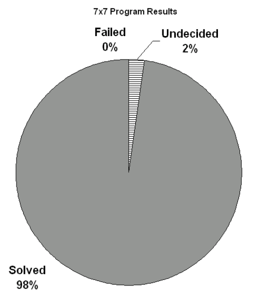
6.2 Results

6.2.1 Time

The average time it took the generator to generate 10,000 puzzles and the solver to solve these puzzles and generate output, was about 25 minutes on an AMD Turion 64 Mobile machine (791 MHz) with 512 MB of internal memory. The fact that the 15×15 puzzles were much larger than the 7×7 did not result in any larger run time of the batch. The average time it took the generator to generate the same 10,000 puzzles and then let Sugar solve them took a lot longer, about 1h45m. This is mainly due to the fact that the way Sugar solves puzzles creates a lot more files than the self built solver. Moving, reading and copying these files took most of this time. If you look at a single puzzle being solved by MiniSat [3], it took less than a second, as well as the encoding and decoding by Sugar.

6.2.2 Own solver

To see how our own solver did, we have some statistics to review. As can be seen in Figure 8 (left) in the 7×7 puzzles only 2% was undecided by the solver, i.e., it did not solve the puzzle. As one can see in Figure 8 (right) from all the solved puzzles 15% was solved differently than the generator intended.



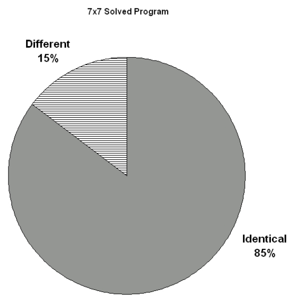
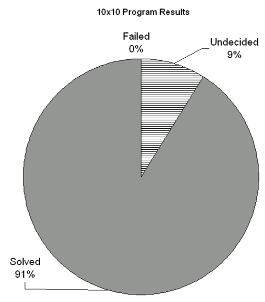


Figure 8: Diagrams of (left) percentages of how much the program solved of the 7×7 puzzles and (right) the percentages of puzzles being solved differently than the generator intended them.

As we look at the 10×10 puzzles, we notice that the percentage of undecided puzzles has grown to 10%, see Figure 9 (left). The number of puzzles that is solved differently is almost doubled to 28%, as you can be seen in Figure 9 (right).



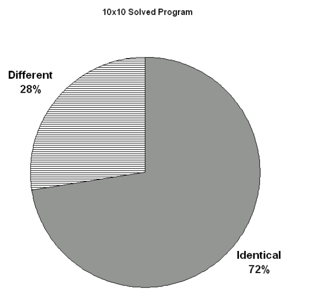


Figure 9: Diagrams of (left) percentages of how much the program solved of the 10×10 and (right) the percentages of puzzles being solved differently than the generator intended them.

Finally we look at the 15×15 puzzles. The percentage of undecided puzzles is now 12%, which is not that big an increase, see Figure 10 (left).

We assume that the number of undecided puzzles will increase still with the larger puzzles. Also with increased size of the puzzles, the percentage of puzzles solved differently from the generator rises to a value of 38%, see Figure 10 (right). This will also be common the larger the puzzle gets.

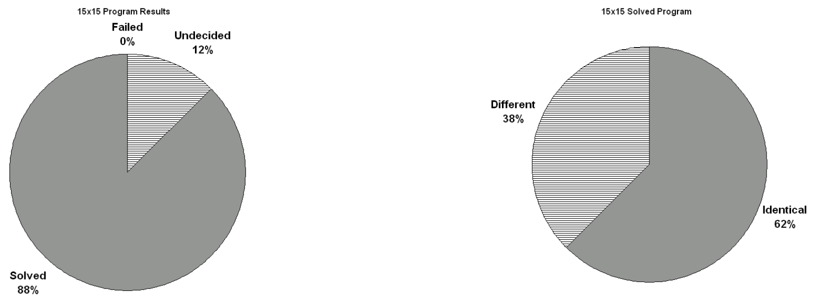


Figure 10: Diagrams of (left) percentages of how much the program solved of the 15×15 and (right) the percentages of puzzles being solved differently than the generator intended them.

6.2.3 Sugar

Sugar solved every single puzzle given by the generator. This did not come as a surprise, since the generator generates only single connected group puzzles and Sugar solves these with ease. It did however took a considerable time to solve the puzzles.

6.2.4 Generator vs own solver vs Sugar

Last but not least we checked how many of the puzzles were solved the same way by either the generator, solver or Sugar. This was done by comparing the output files of the jobs they were given. Both the generator solution files and the solver program solution files had to be rewritten in the same way as Sugar outputs the solutions. In Figure 11 one can find table of a comparison of the way the programs solved the puzzles. Every possibility has a label, described in the list below. one can also find examples for each possibility, in Sugar output form, in the appendix.

In reading the output there were eight distinct possibilities:

1. All three solutions were identical. The solver solved the puzzle without having to guess a bridge, making the puzzle a puzzle with a unique solution. This is labeled `1=2=3, unique` in Figure 11. For an example of a solution in Sugar output form where this is the case, see Figure 13.

2. All three solutions were identical. The solver guessed and so we can not be sure that the solution is unique. This is labeled `1=2=3, guess` in Figure 11. For an example see Figure 14.

3. The generator and Sugar had the same solution, but the solver program found a different one. This is mainly due to the guesses the solver takes. See Figure 11, labeled `1=2, not 3`. For an example see Figure 15.

4. The generator and the solver program had the same solution, but Sugar found a different one. See Figure 11, labeled `1=3, but not 2`. For an example see Figure 16.

5. The generator had a solution, but Sugar and the solver program found another, both the same, solution. See Figure 11, labeled `2=3, not 1`. For an example see Figure 17.

6. All three programs have a different solution to the puzzle. See Figure 11, labeled `All different`. For an example see Figure 18.

7. The solver program could not solve the puzzle, but the generator and Sugar had the same solution. See Figure 11, labeled `3 fail, 1=2`. For an example see Figure 19.

8. The solver program could not solve the puzzle, and the generator and Sugar had a different solution. See Figure 11, labeled `3 fail, 1 not 2`. For an example see Figure 20.

As can be seen in Figure 11, the part where all three programs have the same solution and the solver does not guess, shrinks as the puzzles get bigger. This is because in larger puzzles the chances of a puzzle with cycles, which are solvable in multiple ways, become higher. For an example of such a cycle see Figure 12. The number of puzzles that are solved the same way but with some guesses from the solver, does increase with the puzzle size. In the rest of the cases, where the solver program doesn't fail, we can speak of puzzles with more than one solution. In the case that the solver program fails to give a solution there are, as mentioned in the list above, two possibilities. The generator and Sugar agree on the solution or disagree. In both cases, assuming that the solver program can solve all puzzles with a unique solution, we can speak of puzzles with more than one solution. If the generator and Sugar still give the same solution, this might be just a coincidence.

When comparing the solutions of the eight examples shown in the appendix, we notice that a certain situation occurs rather frequently. In a cycle of four nodes it happens that one program solves it a certain way, and the

Situation	7 × 7	10 × 10	15 × 15
`1=2=3, unique`	7267	5181	4000
`1=2=3, guess`	562	851	902
`1=2, not 3`	718	1371	1635
`1=3, not 2`	528	566	570
`2=3, not 1`	587	587	641
`All different`	126	554	1010
`3 fail, 1=2`	114	419	447
`3 fail, 1 not 2`	98	471	795
Total	10000	10000	10000

Figure 11: Table with data comparisons of the eight possibilities mentioned in the list above. Here 1 is the generator, 2 is Sugar and 3 is the solver.

other program solves it with 2 bridges "turned" 90°. So one bridges turns 90° counterclockwise and the opposite bridge in the cycle turns 90° clockwise.

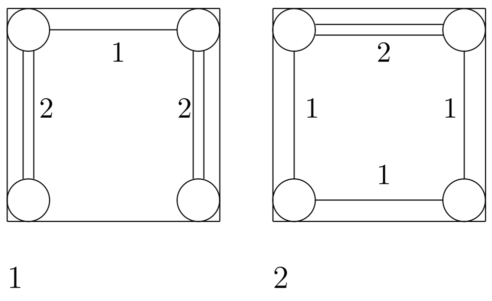


Figure 12: An example of how in a cycle the bridges can "flip" to create a different solution. Values along bridges are the number of bridges.

As can be seen in Figure 12 there are two different solutions. In Figure 12.1 there are two sets of nodes connected with vertical double bridges and a set of nodes with a horizontal single bridge. In this case the two vertical bridges "flip" into a single bridge, for example, the left clockwise and the right one counterclockwise. This creates Figure 12.2, and thus a different solution while still all nodes having the same amount of bridges attached to them. This is a noticably common situation between two different solutions of the same puzzle.

7 Conclusion

As we can see in Section 6 our own solver was really a bit quicker in solving the puzzles. The only problem is that it did not solve all puzzles, someting that the program Sugar did.

As can be seen in Figure 11 the generator does generate a fair number of puzzles with a unique solution. Due to the random factor in generating these puzzles, we see that the larger the puzzles get the more difficult it is to generate such a puzzle. We also see that the larger puzzles have a larger chance that all programs find a different solution. This is because the chance of getting cycles that can be "flipped" is larger. From the perspective of a person who wants to solve the puzzle, having a puzzle with more than one solution can create a more difficult, or even an easier puzzle, depending on what choices have to be made.

The own solver proved to be quite a good solver. Having the numbers of puzzles it could not solve increased when the puzzle size increased is unfortunate, but it did not grow into a really large percentage (with 15×15 it was only 12%). The larger the puzzles get, the better it is to use Sugar to solve them instead of the own solver. It does take more time, but the results are a lot better.

As further research, we mention a further investigation into the relation between the quantities in Figure 11.

8 Acknowledgements

I'd like to thank Naoyuki Tamura for his help and quick email replies, containing useful scripts and example data. Without that the project would have been a lot harder (if not impossible).

Also I'd like to thank my supervisor Dr. Walter Kosters, who made it seem he had all the time in the world to help me, while juggling his own busy schedule. Without his constantly giving hints and ideas I'd still be at Section 1.

Last but not least I'd like to thank Sandra, my girlfriend, for giving me hope, energy and hugs to continue on this project.

References

[1] Website Nikoli, `http://www.nikoli.co.jp/en/` [retrieved August 18, 2009]

[2] Website Sugar, http://bach.istc.kobe-u.ac.jp/sugar/ [retrieved August 12, 2009]

[3] Website MiniSat, http://minisat.se/ [retrieved August 12, 2009]

Appendix

In this appendix we give examples of the eight possibilities obtainable by comparing the generator, Sugar and solver solutions. With this output it is sometimes not possible to obtain the positions of all nodes, but for our purposes the examples seem sufficient.

```
generator:
c Decoding puzzle0.out
s
a h_1_0 1
a v_1_2 1
a h_1_2 1
a v_1_4 1
a h_1_4 2
a v_3_4 1
Sugar:
c Decoding puzzle0.out
s SATISFIABLE
a h_1_0 1
a v_1_2 1
a h_1_2 1
a v_1_4 1
a h_1_4 2
a v_3_4 1
solver:
c Decoding puzzle0.out
s
a h_1_0 1
a v_1_2 1
a h_1_2 1
a v_1_4 1
a h_1_4 2
a v_3_4 1
```

Figure 13: Example of a 7×7 puzzle, with generator (top), Sugar (middle) and solver (bottom) having the same solution. Also, the solver did not guess, so this is a puzzle with a unique solution

```
generator:
c Decoding puzzle55.out
s
a h_0_1 1
a h_0_3 1
a v_0_6 1
a h_2_3 1
a v_2_6 2
a h_6_4 1
Sugar:
c Decoding puzzle55.out
s SATISFIABLE
a h_0_1 1
a h_0_3 1
a v_0_6 1
a h_2_3 1
a v_2_6 2
a h_6_4 1
solver:
c Decoding puzzle55.out
s
a h_0_1 1
a h_0_3 1
a v_0_6 1
a h_2_3 1
a v_2_6 2
a h_6_4 1
```

Figure 14: Example of a 7×7 puzzle, with generator (top), Sugar (middle) and solver (bottom) having the same solution. Also, the solver did guess, so this is a puzzle with more than one solution.

```
generator:
c Decoding puzzle8.out
s
a v_0_0 1
a h_0_0 2
a v_0_3 1
a h_0_3 1
a h_3_0 1
a v_3_3 2
a h_5_0 1
Sugar:
c Decoding puzzle8.out
s SATISFIABLE
a v_0_0 1
a h_0_0 2
a v_0_3 1
a h_0_3 1
a h_3_0 1
a v_3_3 2
a h_5_0 1
solver:
c Decoding puzzle8.out
s
a v_0_0 2
a h_0_0 1
a v_0_3 2
a h_0_3 1
a v_3_3 2
a h_5_0 1
```

Figure 15: Example of a 7×7 puzzle, with generator (top) having same solution as Sugar (middle), but not with solver (bottom).

```
generator:
c Decoding puzzle21.out
s
a v_0_0 2
a h_0_0 1
a v_0_3 1
a v_3_0 1
a h_3_0 1
a v_3_3 1
a h_3_3 2
Sugar:
c Decoding puzzle21.out
s SATISFIABLE
a v_0_0 1
a h_0_0 2
a v_3_0 1
a h_3_0 2
a v_3_3 1
a h_3_3 2
solver:
c Decoding puzzle21.out
s
a v_0_0 2
a h_0_0 1
a v_0_3 1
a v_3_0 1
a h_3_0 1
a v_3_3 1
a h_3_3 2
```

Figure 16: Example of a 7×7 puzzle, with generator (top) having same solution as solver (bottom), but not with Sugar (middle).

```
generator:
c Decoding puzzle26.out
s
a h_1_0 2
a h_1_3 1
a v_1_5 1
a h_3_1 2
a v_3_3 1
a h_3_3 2
Sugar:
c Decoding puzzle26.out
s SATISFIABLE
a h_1_0 2
a v_1_3 1
a v_1_5 2
a h_3_1 2
a v_3_3 1
a h_3_3 1
solver:
c Decoding puzzle26.out
s
a h_1_0 2
a v_1_3 1
a v_1_5 2
a h_3_1 2
a v_3_3 1
a h_3_3 1
```

Figure 17: Example of a 7×7 puzzle, with Sugar (middle) having same solution as solver (bottom), but not with generator (top).

```
generator:
c Decoding puzzle22.out
s
a v_0_0 2
a h_0_0 1
a v_0_6 1
a v_3_0 1
a h_3_0 2
a v_3_6 1
a h_6_0 2
Sugar:
c Decoding puzzle22.out
s SATISFIABLE
a v_0_0 1
a h_0_0 2
a v_3_0 2
a h_3_0 2
a v_3_6 2
a h_6_0 1
solver:
c Decoding puzzle22.out
s
a v_0_0 2
a h_0_0 1
a v_0_6 1
a v_3_0 2
a h_3_0 1
a v_3_6 2
a h_6_0 1
```

Figure 18: Example of a 7×7 puzzle, with every program having different solution.

```
generator:
c Decoding puzzle53.out
s
a v_0_0 1
a h_0_0 1
a v_0_2 1
a h_0_2 2
a h_2_0 1
a v_2_2 2
a h_2_2 1
Sugar:
c Decoding puzzle53.out
s SATISFIABLE
a v_0_0 1
a h_0_0 1
a v_0_2 1
a h_0_2 2
a h_2_0 1
a v_2_2 2
a h_2_2 1
solver FAILS:
c Decoding puzzle53.out
s
a h_0_2 2
a h_2_0 1
a v_2_2 2
a h_2_2 1
```

Figure 19: Example of a 7×7 puzzle, with the solver (bottom) failing and generator (top) and Sugar (middle) having the same solution.

```
generator:
c Decoding puzzle253.out
s
a h_0_0 2
a v_0_2 2
a v_0_5 2
a h_2_2 1
a v_2_5 1
a h_5_2 1
Sugar:
c Decoding puzzle253.out
s SATISFIABLE
a h_0_0 2
a v_0_2 1
a h_0_2 1
a v_0_5 1
a v_2_2 1
a h_2_2 1
a v_2_5 2
solver FAILS:
c Decoding puzzle253.out
s
a h_0_0 2
a v_0_2 1
a h_2_2 1
a v_2_5 1
a h_5_2 1
```

Figure 20: Example of a 7×7 puzzle, with the solver (bottom) failing and generator (top) and Sugar (middle) having a different solution.