Hitori Solver

Bachelor Thesis

Matthias Gander

csae1940@uibk.ac.at

Christian Hofer

csae1761@uibk.ac.at

8th April, 2006

Supervisor: Univ.-Prof.Dr. Aart Middeldorp

Abstract

Hitori is a popular Japanese puzzle. In this bachelor thesis we describe the first development and implementation of a Hitori solver. After explaining the rules of Hitori, we present our Hitori solving algorithm. Next we describe a transformation into a propositional satisfiability problem. By applying a modern SAT solver to the resulting formula, we obtain a second Hitori solver for free. We compare the two programs and conclude with a description of the GUI of our solver.

Contents

Chapter 1

Introduction

Our bachelor thesis had the topic "Hitori". We decided to tackle this problem because it promised to be challenging and because there existed no computer programs for it. Hitori was completely new to us, but our supervisor gave us a link to a web site[1] in which the rules and some hints to solve it were mentioned. Based on these hints, we tried to create an algorithm to solve the puzzle. Soon we realized that the hints, which describe some useful patterns, did not bring us even near to our goal, and thus we had to find our own methods. While thinking about how to solve this puzzle we perceived a certain logic behind Hitori. So while finding an algorithm to solve Hitori we meanwhile started to transform Hitori into a propositional satisfiability problem, rendering it solvable by SAT solvers (in our case MiniSat [2]). Both tasks were not easy to manage, but in the end it did work out quite nicely.

Due to the immense complexity of problems like Hitori, the size of the generated boolean formula rapidly becomes huge when increasing the playing field (for a typical 17×17 matrix the size of the generated formula tends to be over a gigabyte and it can take hours to generate), making the SAT approach unsuitable for large matrices. In contrast, our own algorithm works pretty fast even on large matrices.

The remainder of this thesis is organized as follows. In the next chapter we explain the rules of Hitori and present a useful coloring scheme. Our algorithm to solve Hitori is presented in Chapter 3. In Chapter 4 we explain the SAT approach. The two solvers are compared in Chapter 5. We conclude in Chapter 6 with documentation of our program.

[1] `http://www.puzzle.jp/letsplay/hitorirule-e.html`

Chapter 2

Hitori

Here we will explain the rules of the Hitori puzzle. Hitori (Hitori ni shite kure; literally "let me alone") is a logical puzzle from Japan, which is played on an $n \times n$ matrix filled with numbers. It first appeared in Puzzle Communication Nikoli in issue #29 (March 1990). Hitori is very easy to learn since there are only three simple rules. The only thing you must do is paint the right cells of the matrix such that the rules are satisfied. The larger the matrix is, the harder it is to solve. In puzzle books there are 3 levels of difficulty, 8×8 matrices are the easiest, 12×12 are of medium difficulty and 17×17 matrices are the hardest to solve, although a real expert can solve the latter type in less than 10 minutes.

In Figure 2.1 an example 8×8 Hitori matrix is given.

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

Figure 2.1: Example of an 8×8 Hitori matrix.

2.1 Hitori Cell Color Code

Throughout the document parts of playing fields will be presented for demonstration purposes. In these fields cells will be painted in various ways, according to the conventions given in Figure 2.2. These have nothing to do do with Hitori inherently but we find them quite convenient to explain several matters.

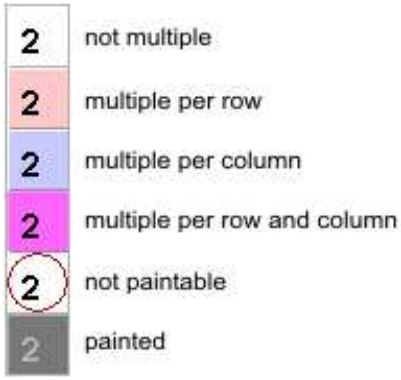

Figure 2.2: Color codes.

2.2 Rules

Now we will explain the three Hitori rules.

2.2.1 Rule 1

> *Numbers must not appear more than once in each row and each column.*

If there are cells with the same number in the same row or column, we have to paint some or sometimes all of them, so that the numbers appear at most once. If we want to fulfill this rule, we first have to search the numbers (cells) which appear more than once per row or column. These special cells we call 'multiples'. To solve Hitori, we only have to paint some (or sometimes all) of these multiples. Figure 2.3 shows an example Hitori matrix. The colored cells are the multiples.

Now we are going to paint (see Figure 2.4) some of the multiples, so that every number appears only once per column and row. Note that to fulfill rule 1 we could also have painted all multiples. If we only consider rule 1, the solution in Figure 2.4 would be correct, but there are two more rules.

Figure 2.3: Hitori matrix with colored multiples.

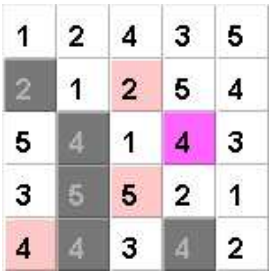

Figure 2.4: Rule 1 is satisfied.

2.2.2 Rule 2

Painted cells are never adjacent in a row or a column.

This means that we cannot paint cells which have a painted neighbor cell. We can easily see that our previous solution is wrong, because we have painted adjacent cells in a column.

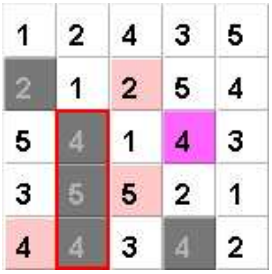

Figure 2.5: Rule 2 is violated.

By painting other cells as shown in Figure 2.6, rules 1 and 2 are both fulfilled. So the solution seems ok now, but beware of the third rule!

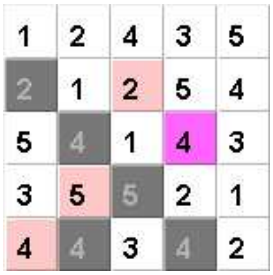

Figure 2.6: Rules 1 and 2 are satisfied.

2.2.3 Rule 3

> *Unpainted cells create a single continuous area, undivided by painted cells.*

In other words, painted cells must not divide the field in separate groups of unpainted cells. Rephrased, there exists a path from every unpainted cell to any other unpainted cell, passing only unpainted horizontal/vertical neighbor cells. We can now see, that our previous solution breaks this new rule twice. Figure 2.7 shows two chains of painted cells, dividing the unpainted cells into two areas

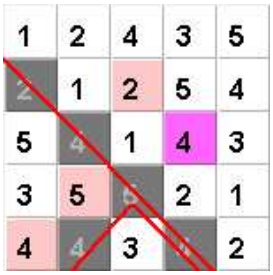

Figure 2.7: Rule 3 is violated.

If we correct our last try, we get a new solution as demonstrated in Figure 2.8. This last solution is correct. Note that for a given Hitori matrix there may be more than one solution, like in our example.

Figure 2.8: Rules 1, 2 and 3 are satisfied.

Chapter 3

Our Algorithm

3.1 Idea

Our algorithm is divided into several parts and all of them have one thing in common:

> *to keep the system in a consistent state and build on that.*

If you encounter a problem as complex as Hitori, the best strategy is to use the complexity against itself and and thereby solving it efficiently. This idea will be discussed in the following sections.

3.2 Inconsistent States

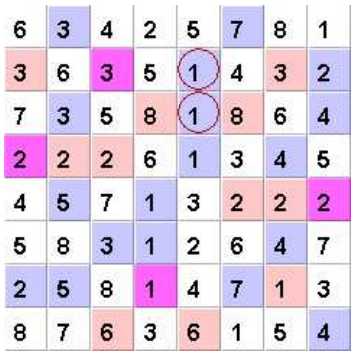

Figure 3.1: An inconsistent state.

Inconsistency, how is it defined? We will call a state inconsistent if it always leads to a point where there are no solutions left. To find inconsistent states we developed another rule which found its implementation in the so-called function "isRule4Conform". What this function does is to look if two multiples in the same row or column do have the same number and if they

are both set unpaintable.[1] Figure 3.1 shows what is meant by the above definition.

The algorithm can bring a field into an inconsistent state by painting a wrong multiple and then letting the standard cycle pattern run. An explanation of this is given by the fields in Figure 3.2.

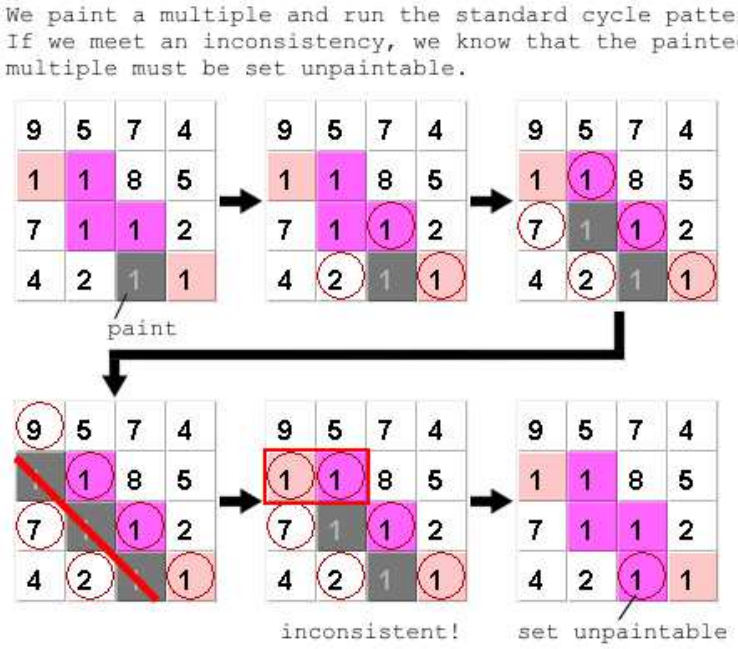

Figure 3.2: Reaching an inconsistent state.

3.3 Part 1: Standard Patterns

We start by using standard patterns. These standard patterns are obviously correct and thus the system remains in a consistent state after applying them. We use these patterns at several places in our algorithm. The following figures show the standard patterns that we use. Some of these are applicable everywhere on a playing field, whereas others are only applicable at a border or in a corner. This is indicated in the captions of the figures.

[1] Having set two multiples of the same number to unpaintable does lead in most cases to a "disastrous" avalanche effect in which many elements of the field might be painted and set unpaintable at the same time.

Figure 3.3: Pattern 1: Applicable everywhere.

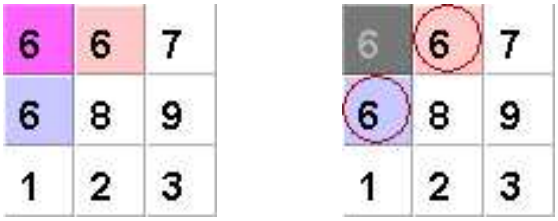

Figure 3.4: Pattern 2: Applicable only at corners.

Figure 3.5: Pattern 3: Applicable only at corners.

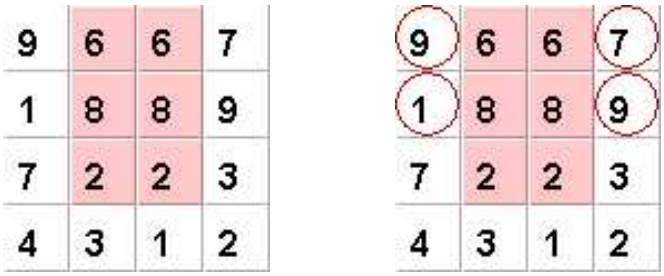

Figure 3.6: Pattern 4: Applicable only at borders.

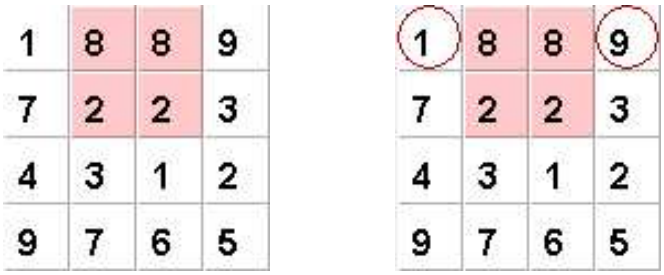

Figure 3.7: Pattern 5: Applicable only at borders.

Figure 3.8: Pattern 6: Applicable everywhere.

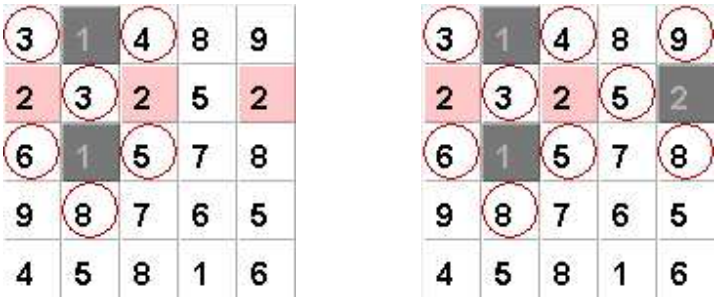

Figure 3.9: Pattern 7: Applicable everywhere.

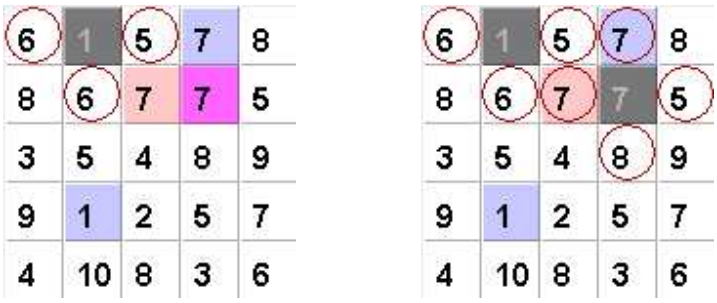

Figure 3.10: Pattern 8: Applicable only at borders.

3.3.1 Important Function: StandardCyclePattern

As the name suggests, StandardCyclePattern is a kind of pattern. It is terminating and correct so it keeps the system consistent. It is used in almost every action which comprises the painting of a multiple. It works as follows. When a multiple is painted, all cells adjacent to it have to be set unpaintable. That is of course trivial and does not need further explanation. Another thing it does is to set every multiple in the same row or column that has the same number, as the multiple which was just set unpaintable, to painted. The third thing it does is to prevent rule 3 to be broken. This means that every multiple which may lead to an inconsistent state by rule 3 is made unpaintable. As mentioned before this method leaves the system consistent. If the wrong multiple has been painted prior to calling this function, an error may occur. That is used to filter unpaintable cells. Let's assume we paint a multiple and execute the StandardCyclePattern function. The process is visualized in Figures 3.11–3.13.

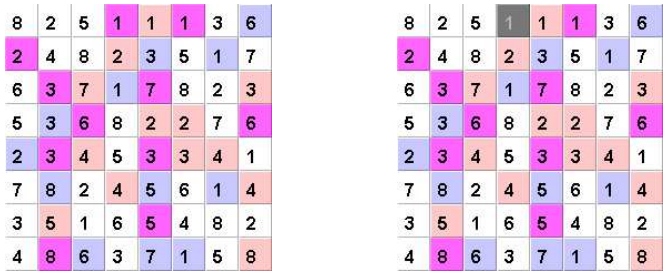

Figure 3.11: Painting a multiple on an empty example field.

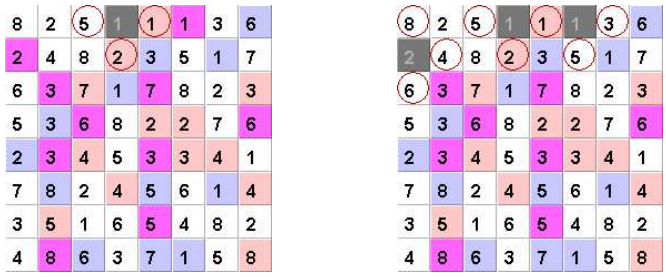

Figure 3.12: Adjacent cells are colored, leading to a chain reaction.

At this stage the algorithm determines for each multiple whether rule 3 is broken by painting the multiple. If there is such a multiple it has to be set unpaintable. In this example the number 3 in row 2 and column 5 has to be set unpaintable (Figure 3.13).

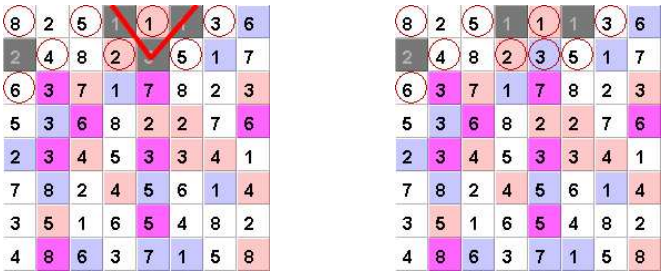

Figure 3.13: Multiples which break rule 3 are set unpaintable

3.4 Part 2

If the standard patterns do not suffice to solve the puzzle, we switch to the second part of the algorithm. What is done here, is that the multiples are stored into a vector **M**. The first multiple A in **M** is painted and then the StandardCyclePattern is run. If the field enters an inconsistent state we know that multiple A cannot be painted. After setting A unpaintable the StandardCyclePattern is run again. After this step **M** has to be calculated again since there is the possibility that some multiples were painted. If the field does not encounter an inconsistent state, nothing is known about the status of A. These steps have to be done from the first until the last multiple in the vector. When no more multiples can be painted, part 2 will terminate. The following pseudo code and the visualization in Figure 3.14 explain this process.

```
Vector {M}; field {b}; bool FLAG
 M := getRemainingMultiples();
 b := field;
 do
  while FLAG == true
  {
        FLAG = false;
        for <i> from 0 to #{M}  <i++>
        {
            paint( M[i] );
            standardCyclePattern(field);
            if <inconsistent == field> then
                field := b;
                setNonPaintable( M[i] );
                standardCyclePattern(field);
                M := getRemainingMultiples();
                i := 0;
                b := field;
                FLAG := true;
```

```
                break;
            else
                field := b;
        }
    }
end do
```

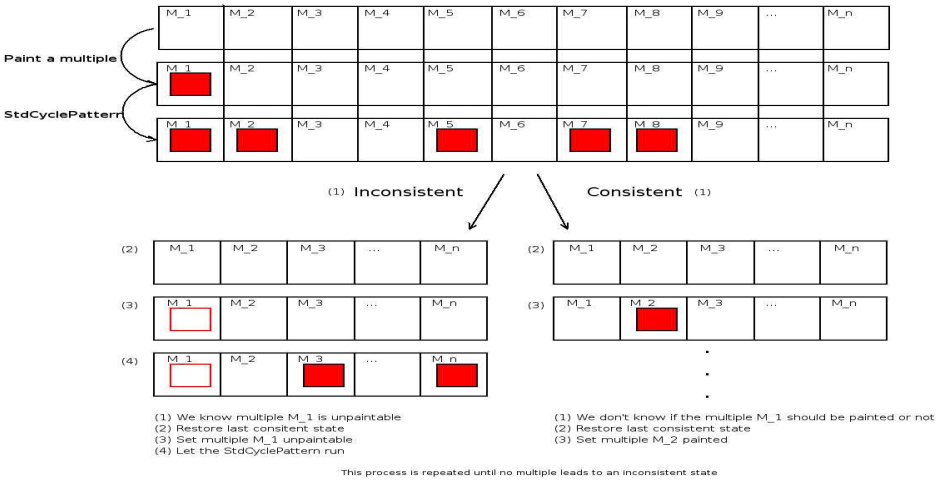

Figure 3.14: Visualization of part 2.

3.5 Part 3

Many of the simpler Hitori matrices can be solved by parts 1 and 2, but if we encounter some harder[2] matrices, part 3 comes into play. Part 3 is in principle a backtracking algorithm. What we do here is calling a slightly altered part 2 function recursively until no unpainted multiples are left. If the matrix isn't solved at the beginning, the backtracking begins, which has the effect of trying to paint every possible combination of multiples. To let the algorithm not degenerate to brute force, we used 2 optimizations: first the standard cycle pattern and second the function described in part 2. A slightly simplified implementation of the backtracking function can be seen in the following lines of pseudo code.

```
Field backtracking(Field hitoriField)
{
    backup = hitoriField;
```

[2]To be precise, harder for our algorithm, but often easier for human solvers.

```
    repeat:
    for every multiple m in remainingMultiples
    {
        paint multiple m;
        standardCyclePattern(hitoriField);
        part2(hitoriField);
        backtracking(hitoriField);   //recursion

        if ( hitoriField is solved )
            return hitoriField;      //we are finished

        if ( hitoriField is inconsistent )
        {
            hitoriField = backup;
            set multiple m unpaintable;
            standardCyclePattern(hitoriField);
            part2(hitoriField);
            backup = hitoriField;
            jump to repeat:
        }
        else
            hitoriField = backup;
    }

    return hitoriField;
}
```

Since the backtracking function tries to paint all of the remaining multiples in any possible combination, we are assured that part 3 solves any Hitori matrix.

Chapter 4

Logical Approach

4.1 Hitori as Boolean Satisfiability (SAT)

Since there are many highly optimized SAT solvers, we translate Hitori into a SAT problem in order to obtain another Hitori solver to which we can compare the algorithm described in the previous chapter. Our advisor recommended to use MiniSat [2], which takes as input a file in DIMACS CNF format. So our goal is the automatized generation of such a DIMACS CNF file from a Hitori matrix. In the following sections, we explain the method.

4.2 The DIMACS CNF File Format

It is recommended to understand the DIMACS CNF (conjunctive normal form) file format for the further sections. Here you can see the structure of the DIMACS CNF format:

```
p cnf n m
1 -3 5 ... 0
2 -6 8 ... 0
...
```

Here n denotes the number of variables and m the number of clauses. First we have a line, showing the number of variables and clauses. The lines following are the clauses, each clause is terminated by the character '0'. The variables are numbered 1 to m. Inverted variables begin with a '-'. Lines beginning with the character 'c' are comment lines.

Example: The term

$$(x_1 \vee x_3 \vee \bar{x}_4) \wedge x_4 \wedge (x_2 \wedge \bar{x}_3)$$

1	2	4	3	5
2	1	2	5	4
5	4	1	4	3
3	5	5	2	1
4	4	3	4	2

Figure 4.1: Hitori example matrix.

is translated into the following CNF format:

```
c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0
2 -3 0
```

4.3 Translating Hitori to SAT

We will explain our translation approach on the following example Hitori matrix of Figure 4.1. We use the following correspondence:

value	cell
1	painted
0	unpainted

The squares in a Hitori matrix are numbered from left to right, top to bottom, as illustrated in Figure 4.2.

Figure 4.2: Variable names (red numbers).

Rule 1

Rule 1 tells us that unpainted cells with the same numbers never appear more than once in each row and column. First we search for numbers that appear more than once in a row or a column (we call these numbers "multiples"). This makes it easier to create the logical formula because we know that only some (or sometimes all) of these multiples must be painted to solve Hitori.

Looking at the first two multiples in the second row in Figure 4.2, there are only 3 possibilities:

1. cell 6 unpainted and cell 8 painted,
2. cell 6 painted and cell 8 unpainted,
3. cell 6 painted and cell 8 painted.

This gives rise to the following clause in our SAT formula:

$$(\bar{x}_6 \vee \bar{x}_8)$$

In the same way, the three multiples labeled 4 in the last row in Figure 4.2 give rise to

$$(\bar{x}_{21} \vee \bar{x}_{22}) \wedge (\bar{x}_{21} \vee \bar{x}_{24}) \wedge (\bar{x}_{22} \vee \bar{x}_{24})$$

We create such clauses for every multiple in each row and each column, so that rule 1 is fulfilled.

Since non-multiples should not be painted in order to solve Hitori, we add for every non-multiple a unit clause which encodes this. For instance,

$$x_1$$

for the Hitori matrix in Figure 4.2.

Rule 2

Rule 2 states that if a cell (multiple) is painted, its horizontal and vertical neighbors must be unpainted. So if cell x_6 in Figure 4.3 is painted, the cells marked with a red circle must be unpainted. This gives rise to the following three clauses:

$$(x_6 \vee x_1) \wedge (x_6 \vee x_7) \wedge (x_6 \vee x_{11})$$

We do this for every multiple in the matrix, in order to fulfill rule 2.

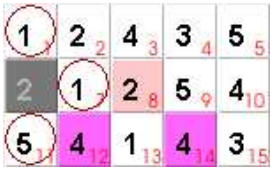

Figure 4.3: Neighbors of a painted cell must be unpainted.

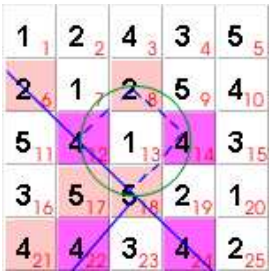

Figure 4.4: All possible chains (blue) and cycles (green).

Rule 3

Rule 3 states that the unpainted cells create a single continuous area, undivided by painted cells. This means that we must ensure that no chains or cycles can be created by painting multiples. Figure 4.4 show what we mean by chains (blue color) and cycles (green color). We generate all possible chains and cycles that can be created by painting some of the multiples. Chains that contain cycles need not be taken into account, so in the figure we ignore chains that contain the dashed cells. For the chains in our example we obtain

$$(x_6 \vee x_{12} \vee x_{18} \vee x_{22}) \wedge (x_6 \vee x_{12} \vee x_{18} \vee x_{24}) \wedge (x_{22} \vee x_{18} \vee x_{24})$$

and for the cycle

$$(x_8 \vee x_{12} \vee x_{18} \vee x_{14})$$

It should be remarked that the required calculation can be very time consuming, especially for 17×17 Hitori matrices. Moreover, since there can be extremely many chains and cycles in a Hitori matrix, the size of the output can quickly reach gigabytes.

Output Example

Here we give the complete CNF output computed by the above approach for our example Hitori matrix. We have added some comments for better comprehension.

```
c CNF header
c 25 - number of variables (5x5 Hitori matrix)
c 60 - number of clauses
p cnf 25 60
c Rule 1
c note that every clause is terminated by the character '0'
-6 -8 0
-12 -14 0
```

-17 -18 0
-21 -22 0
-21 -24 0
-22 -24 0
-12 -22 0
-14 -24 0
c *non-multiples should not be painted*
1 0
2 0
3 0
4 0
5 0
7 0
9 0
10 0
11 0
13 0
15 0
16 0
19 0
20 0
23 0
25 0
c *Rule 2*
6 1 0
6 11 0
6 7 0
8 3 0
8 7 0
8 13 0
8 9 0
12 7 0
12 11 0
12 17 0
12 13 0
14 9 0
14 13 0
14 19 0
14 15 0
17 12 0
17 16 0
17 22 0
17 18 0
18 13 0

```
18 17 0
18 23 0
18 19 0
21 16 0
21 22 0
22 17 0
22 21 0
22 23 0
24 19 0
24 23 0
24 25 0
c Rule 3
c chains
6 12 18 22 0
6 12 18 24 0
22 18 24 0
c cycles
c note that every cycle is generated twice
c we keep both copies because of performance reasons
8 12 18 14 0
8 14 18 12 0
```

4.4 Feasibility

The method described in this chapter works well for Hitori matrices of size 12×12 or smaller. Such matrices are translated into a Boolean formula in a few seconds and Minisat can find a solution for these formulas in less than 1 second. However, as indicated above, 17×17 matrices are problematic because the formula generation process can take several hours and the size of the output file can reach several gigabytes. Moreover, it seems that MiniSat is unable to process formulas which are larger than the available RAM; in our experiments we always got a segmentation fault. So it is pointless to wait hours to get a huge formula that is bigger than the RAM of your computer; MiniSat will not solve it!

To get an impression of how many chains and cycles there can be in a $n \times n$ matrix where every cell is a multiple (which is the worst case), we computed the data in Table 4.1. Note that the entries for matrices greater than 14×14 are estimates (due to processing time reasons).

Table 4.1: Number of chains and cycles.

$n \times n$ MATRIX	# POSSIBLE CHAINS	# POSSIBLE CYCLES
2	2	0
3	10	0
4	30	0
5	82	5
6	230	12
7	710	26
8	2.466	70
9	10.622	260
10	57.466	1.508
11	402.158	11.052
12	3.533.738	96.170
13	41.062.038	953.645
14	618.182.266	11.369.346
15	$\sim$ 12.300.000.000	$\sim$ 159.170.000
16	$\sim$ 319.800.000.000	$\sim$ 2.546.733.000
17	$\sim$ 10.553.400.000.000	$\sim$ 45.841.200.000

Chapter 5

Comparison

5.1 Test Data

We tested 116 Hitori matrices, given to us by our supervisor [1] and some additional ones that we downloaded from the internet. Of these 116 matrices, 41 have size 8×8, 43 have size 12×12 and 32 have size 17×17. These matrices were solved by using the algorithm described in Chapter 3 as well as the logical approach with MiniSat described in Chapter 4. We used a Pentium 4 processor with a 3.2 Ghz CPU and 1 GB of RAM.

5.2 Test Results

The results are illustrated in various tables and diagrams below. In order to interpret the data correctly, the following explanation is useful. We mainly compared the execution times of our algorithm and the SAT approach. For the SAT approach we have to consider two different times: the SAT formula generation time and the time MiniSat needs to solve the generated formula. Table 5.1 shows in the first column the average time in seconds our tool needed to generate the formulas. In the second column you can see the average file size and the third column indicates the average number of clauses in the formulas. The last column shows the average time in seconds needed by MiniSat to solve the formulas.

Note that our tool could only generate formulas for 20 out of the 32 17×17 matrices in acceptable time, for the other 12 we aborted the generation process after 1 hour since the formulas would have grown larger than the available RAM, which means that MiniSat couldn't solve them anyway. So the numbers in the third row are based on the 20 'easy' matrices. As already mentioned in Chapter 4, the SAT approach is only suitable for matrices whose dimension doesn't exceed 12×12.

Table 5.2 shows the average solving times for our algorithm and the SAT approach. Again, the SAT values in the last row are for 20 out of 32 17×17

Table 5.1: Average SAT values.

$n \times n$	GENERATION TIME	FORMULA SIZE	# CLAUSES	MINISAT
8×8	0.030 s	4 KB	310	0.018 s
12×12	0.505 s	1280 KB	20636	0.116 s
17×17	28.713 s	59250 KB	504000	3.262 s

matrices. It is clear that our algorithm is quite a bit faster than the SAT approach. It is interesting to remark here that we didn't perform any code optimizations for our algorithm.

Table 5.2: Average solving times.

	SAT approach			
$n \times n$	GENERATION	MINISAT	TOTAL	OUR ALGORITHM
8×8	0.030 s	0.018 s	0.048 s	0.002 s
12×12	0.505 s	0.116 s	0.620 s	0.093 s
17×17	28.713 s	3.262 s	31.975 s	1.499 s

The diagrams Figures 5.1–5.6 speak for themselves. We conclude this chapter by stating that our algorithm is considerable faster than the SAT approach and, moreover, applicable to larger Hitori matrices.

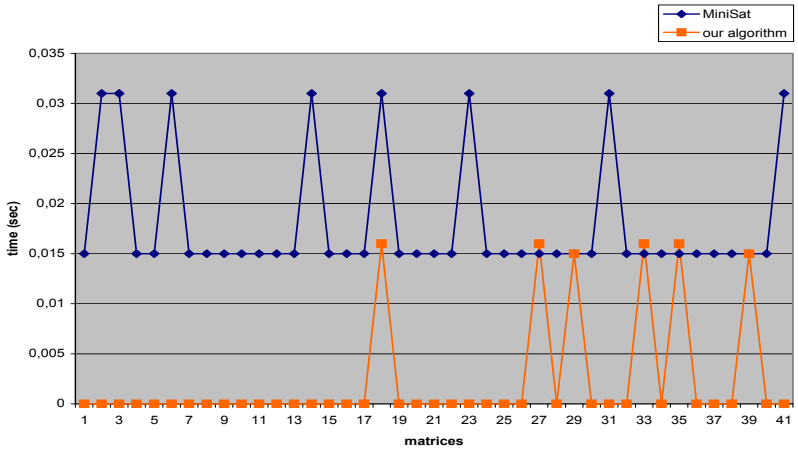

Figure 5.1: Solving times for 8×8 matrices.

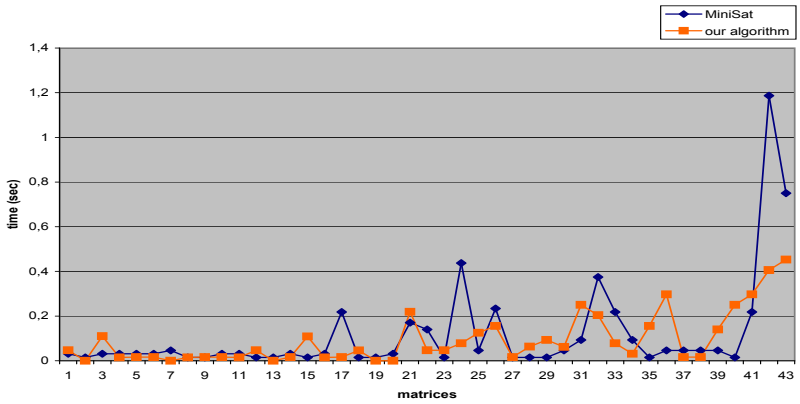

Figure 5.2: Solving times for 12×12 matrices.

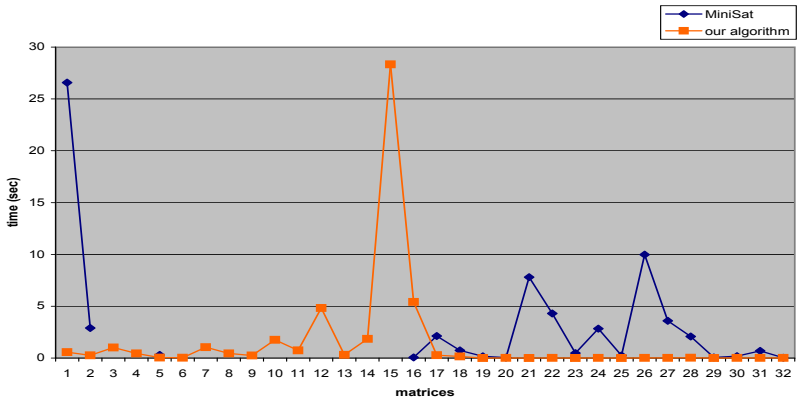

Figure 5.3: Solving times for 17×17 matrices.

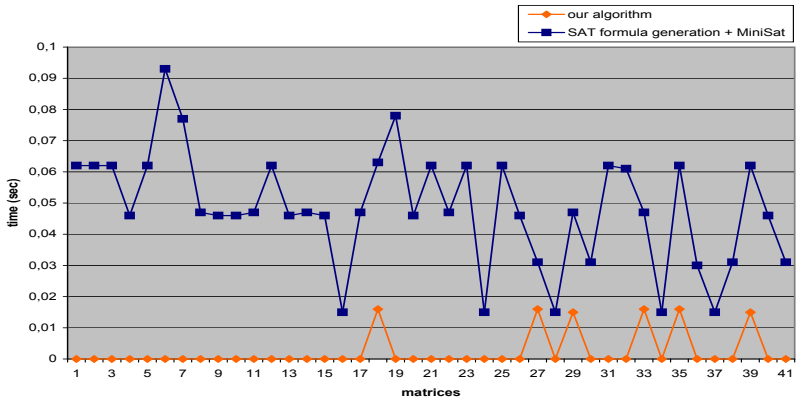

Figure 5.4: Solving times for 8×8 matrices (2).

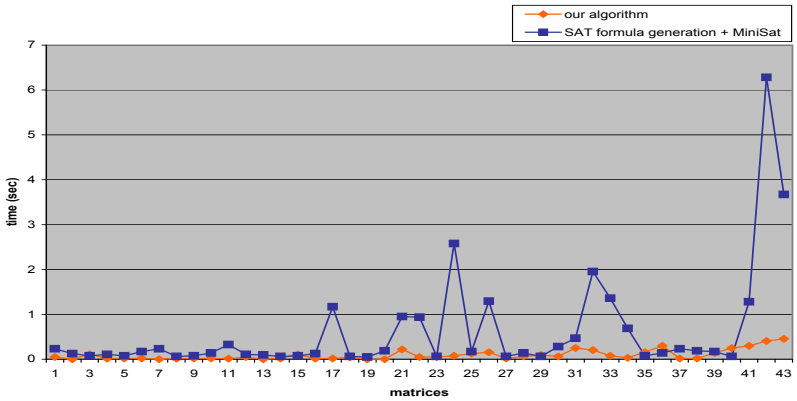

Figure 5.5: Solving times for 12×12 matrices (2).

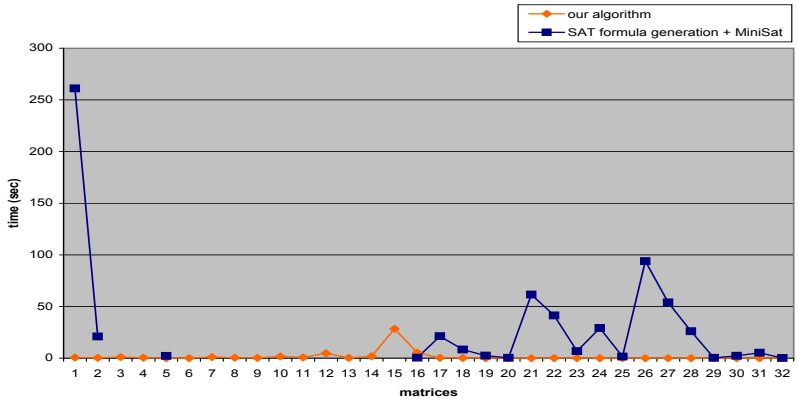

Figure 5.6: Solving times for 17×17 matrices (2).

Chapter 6

Hitori Solver Tool Documentation

In this final chapter we explain the functionality of our "Hitori Solver" tool. First we describe briefly the features of the tool, then we explain how to compile the source code, and finally we describe the GUI elements.

6.1 Features

Here is a list of features the Hitori Solver provides:

- loading your own (quadratic) Hitori matrices,
- solving any (quadratic) Hitori matrix,
- finding the multiples in a Hitori matrix,
- generating a boolean formula from a Hitori matrix,
- loading the solution file MiniSat produces for a matrix which was previously transformed into a SAT formula and solved by MiniSat,
- playing Hitori in debug mode and check at any time if it's solved or not.

6.2 Compilation

The Hitori Solver is written in Java, so if you wish to compile and run it, you will need the Java SDK. Use the following command to compile the Hitori Solver source code:

```
javac *.java
```

After you compiled it, the tool can be started with the following command:

```
java HitoriMainFrame
```

6.3 The Hitori Solver main frame

Figure 6.1 shows the "Hitori Solver" main frame. At the very top is a

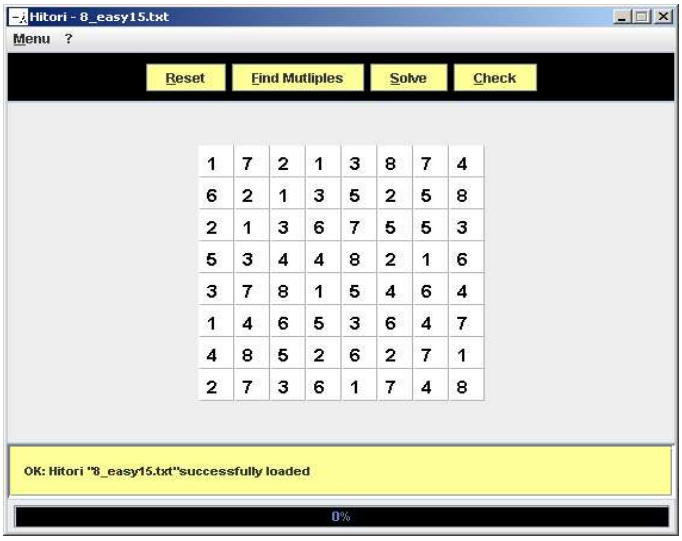

Figure 6.1: The Hitori Solver main frame.

menu bar like in most applications with a GUI. Then follows a panel (black background) with 4 buttons. The large panel in the middle shows the Hitori matrix. Below that is a text area where status messages are displayed. At the very bottom is a progress bar displaying the progress of processes.

Figure 6.2 shows the expanded menu.

Figure 6.2: The Hitori Solver main menu.

6.4 Loading a Hitori Matrix

To load a Hitori matrix you simply click on 'Menu', 'Load Playfield', select a file containing the matrix, and finally click 'Open'. Here is the content of a possible input file:

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

The coefficients must be non-negative integers. Each line in the input file must contain all numbers in a row of the matrix, separated by spaces.

6.5 Solving a Hitori Matrix

First load a matrix, then click 'Menu' followed by 'Solve'. The solution will be shown directly in frame.

6.6 Generating a SAT Formula

Load a Hitori matrix, click 'Menu', 'Generate SAT Formula', choose a file in which the formula should be written and then click 'Save'. It may takes some time to generate the formula, depending on the size of the loaded matrix. It is recommended to generate formulas only for matrices not larger than 12×12.

6.7 Solving the Generated SAT formula

Run MiniSat with the name of the file that contains the generated SAT formula and the name of the file in which the SAT solution should be written as parameters. For example:

```
./MiniSat formula.sat formulaSolution.txt
```

MiniSat can be downloaded from the MiniSat home page [2].

6.8 Loading SAT Solution

You can visualize the solution MiniSat computed by loading the matrix corresponding to the solution. Afterwards click 'Menu', 'Load SAT Solution', choose the solution file and then click 'Open'. The solution will be shown directly on top of the loaded matrix.

6.9 Playing Hitori

If you simply want to play Hitori, you can switch to debug mode, which allows one to paint cells on the loaded matrix. So first load a matrix, click 'Menu' and then select 'Debug Mode'. If you want to see the multiples, click 'Find Multiples'. If you want to check if you solved the loaded Hitori matrix

click ‘Check’. It will display ‘true’ in the status text area if the matrix is solved, otherwise ‘false’. If you want to reset the field to its original state, click ‘Reset’. If you left click on a cell it will toggle between painted and unpainted. Right clicking on a cell will toggle between paintable and non-paintable (red circle). Figure 6.3 shows the debug mode.

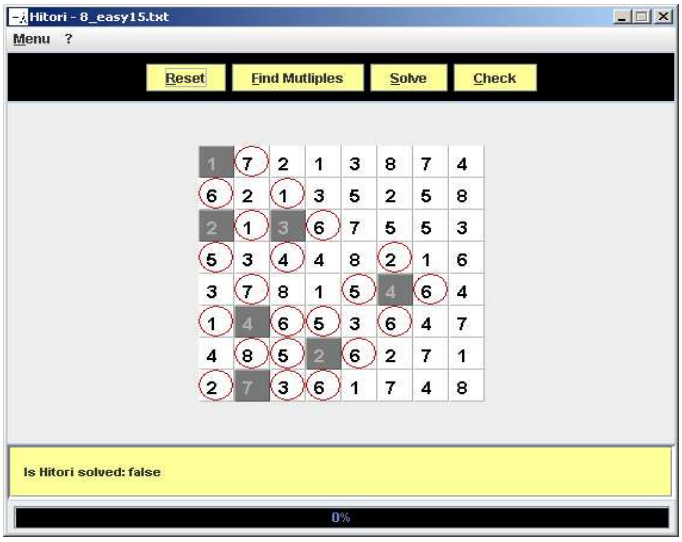

Figure 6.3: Debug mode.

Bibliography

[1] Hitori ni shite kure 1, Pencil Puzzle Book 55, Nikoli, 1999. *In Japanese.*

[2] `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

List of Figures

List of Tables