Solving the Kakurasu Puzzle using Answer Set Programming*

Johan van Nispen

November 16, 2023

Abstract

Answer set programming (ASP) is a declarative programming paradigm used primarily to solve difficult search problems. In this project the declarative ASP paradigm will be used to solve a logical puzzle called Kakurasu. It will be shown that formalizing and encoding the problem statement of the Kakurasu puzzle using the rules of ASP can be done in a very natural and concise manner. It will also be shown that solving increasingly more difficult versions of the Kakurasu puzzle scales well and does not significantly increase the solution search time, despite the exponential growth in the search space.

1 Introduction

Answer set programming (ASP) is a declarative programming paradigm, and is based on the stable model semantics of logic programming. It is primarily used for solving difficult search problems. In ASP a problem is formulated in such a way that that the models (referred to as the answer sets) of the problem correspond to the solutions of the problem [JN16].

In this project, which is part of the course "Model-Based Artificial Intelligence", a type of logical puzzle called *Kakurasu* will be examined. By using the ASP paradigm and by making use of a widely used answer set system called *clingo*, a logic program will be written to solve this puzzle. To guide the investigations made in this project, the following research questions have been formulated:

A. **Is it possible to solve the Kakurasu puzzle using answer set programming?**

B. **What is the effect of the Kakurasu puzzle size on the time needed to calculate the solution using clingo?**

This report is structured as follows: in section 2 some brief background information is provided on ASP and the Kakurasu puzzle. Section 3 shows how the problem can be formulated using ASP, and provides the results of solving increasingly larger (i.e. more difficult) Kakurasu puzzles. In section 4 the conclusions are presented.

2 Background

2.1 ASP and the workflow of ASP

To arrive at a solution for a specific problem using ASP, the workflow shown in figure 1 is followed [KLPS16, GKKS22]. First, the problem is modelled and encoded into a logic program, which consists of *facts*, *rules* and *constraints*. The logic program typically consists of statements about how the solution of the problem should look like, and about which specific conditions the solution should satisfy [JN16]. In a second step, the *grounder* transforms the logic program into a equivalent representation without variables, and in a third step a *solver* computes stable models of the problem from which the problem solution can be read (if any).[GKKS22].

For a short overview of the language of ASP the reader is referred to [Lif16], and a more complete overview of ASP is given in the book [Lif19]. In the book, many examples of problems and on how

*I, Johan van Nispen (student number: 83654127), confirm that this is my own work. Any material taken from other sources has been fully referenced in the text of the work. All sources used in the preparation of this work have been listed in the References section.

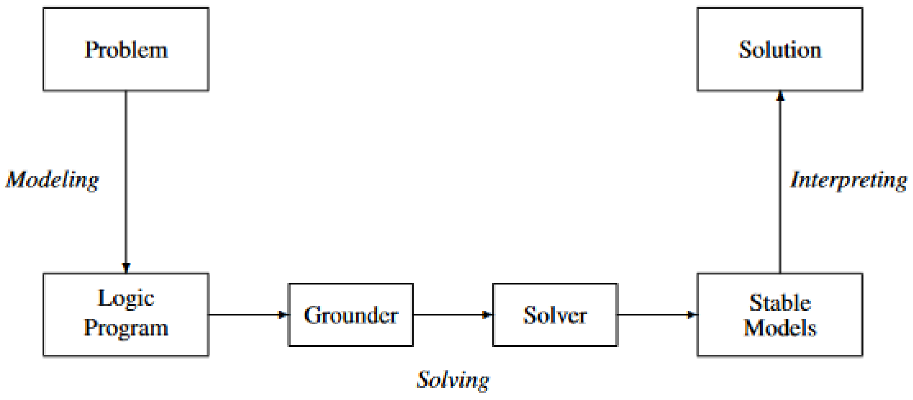


Figure 1: The ASP workflow [KLPS16].

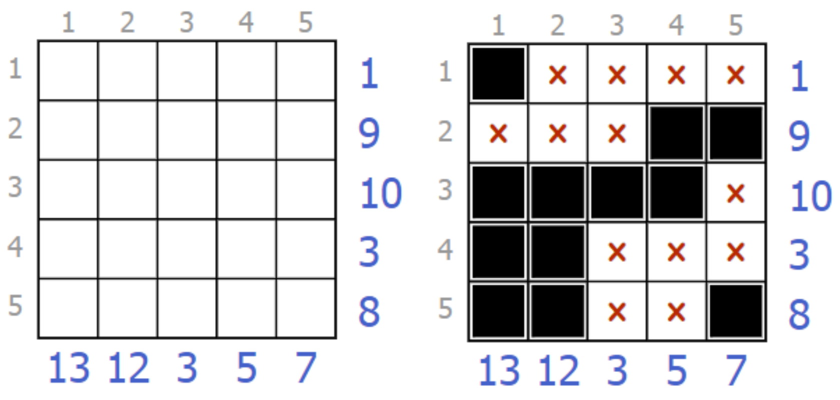
to solve them using ASP are given. The book uses a widely used answer set solver system called *clingo* [GKKS14], which is the software tool that will also be used here. A comprehensive manual for the clingo system can be found at: https://github.com/potassco/guide/releases/.

2.2 The Kakurasu logic puzzle

Kukurasu (also called *"Index Sums"*) is a logic puzzle played on a rectangular grid of N x N squares. The goal is to make some of the squares black in such a way that:

1. The black squares on each row sum up to the number on the right;
2. The black squares on each column sum up to the number on the bottom;
3. If a black cell is first on the row or column, the value is 1. If it is second the value is 2, etc.

The puzzle is usually provided on a 4 x 4 up to a 9 x 9 grid, but there is no limit in theory. An example of a 5 x 5 Kakurasu puzzle is given in figure 2. Figure 2(a) shows the puzzle in the initial state, where only the clues are provided on the right and bottom, figure 2(b) shows the solved puzzle with all the black squares filled in. As can be seen from the figure on the right, the first row sums up to 1, the second row up to 9, etc. (satisfying rule #1), and the first column sums up to 13, the second column up to 12, etc. (satisfying rule #2).



(a) Empty puzzle (b) Solved puzzle

Figure 2: Example of a 5 x 5 Kakurasu puzzle (https://www.puzzle-kakurasu.com).

The total number of combinations for a N x N grid, where each cell can either black or white grows exponentially. For each row we have 2^N combinations, which leads to 2^{N^2} possible combinations for the N x N grid. To illustrate this, for $N = 4$ there are $2^{16} = 65536$ combinations, and for $N = 9$ there are already $2^{81} \approx 2,4 \times 10^{24}$ combinations. Of course, in terms of satisfying solutions, this number also decreases as fast again when the constraints on the row and column sums are brought into the equation, ultimately leading to a single solution.

3 Results

3.1 Solving the puzzle

We now focus on solving the Kakurasu puzzle by following the general ASP workflow as presented earlier (see 2.1). We start with modelling the relevant concepts in the problem domain.

For the puzzle we are given an N x N grid, consisting of N rows and N columns where each square in the grid can either be black or white. Next to that we are also provided with the sums of the rows and columns. In the logic program we can encode these *facts* in the following way:

```
#const n=5.
color(white; black).

%% clues for the rows and columns
clues_row(1,1; 2,9; 3,10; 4,3; 5,8).
clues_col(1,13; 2,12; 3,3; 4,5; 5,7).
```

color(black) indicates the color black, clues_row(3,10) indicates that the sum of the black squares on row 3 is 10, etc. With the facts about the grid size, the cell color and the row/column clues we have specified the initial state of the puzzle as shown in figure 2(a).

The next step is to add a rule which generates a list of all possible grid sets (i.e. all potential answer sets for the problem), where each square in the grid can either be black or white, i.e. to generate all possible N x N grid sets having all squares white, to all squares black, and every possible combination in between. This can be achieved by adding a *choice rule*:

```
{pos(R,C,COL) : color(COL)} = 1 :- R=1..n, C=1..n.
%% achieved: every square of the grid is either black or white
```

The rule introduces the predicate pos(R,C,COL), where pos(3,2,black) would indicate that the square on row 3 and column 2 is colored black. When run for a 3 x 3 grid this generates a total of 512 possible sets (see 2.2). Clingo displays the following results (only a partial output is shown):

```
...
Answer: 509
pos(1,1,white) pos(1,2,white) pos(1,3,white) pos(2,1,black) pos(2,2,black)
pos(2,3,black) pos(3,1,black) pos(3,2,black) pos(3,3,white)
Answer: 510
pos(1,1,white) pos(1,2,white) pos(1,3,white) pos(2,1,black) pos(2,2,black)
pos(2,3,black) pos(3,1,black) pos(3,2,white) pos(3,3,white)
...
```

In order to constrain the solution space, each generated set has an associated set of sums of the rows/columns which need to be compared to the sums of the row/column clues provided in as facts. For the sum there are two possible situations: (a) a row/column contains one or more black squares, and (b) the row/column does not contain any black squares and is empty. For the sums we introduce two predicates, row_tot(R,SUM) and col_tot(C,SUM).

```
row_tot(R,SUM) :- pos(R,_,COL), COL=black, SUM=#sum{C : pos(R,C,COL)}.
col_tot(C,SUM) :- pos(_,C,COL), COL=black, SUM=#sum{R : pos(R,C,COL)}.
%% achieved: calculate the sum of the rows/columns containing black squares

row_tot(R,0) :- pos(R,_,COL), COL=white, #count{C : pos(R,C,COL)} = n.
col_tot(C,0) :- pos(_,C,COL), COL=white, #count{R : pos(R,C,COL)} = n.
%% achieved: add the the sum of the rows/columns containing only white squares
```

The final step is to add a rule to disallow any answer sets which do not match with the sums of the row/columns in the clues provided. This is achieved by adding the following *constraint* to the program:

```
:- row_tot(R,SUM1), clues_row(R,SUM2), SUM1 != SUM2.
:- col_tot(C,SUM1), clues_col(C,SUM2), SUM1 != SUM2.
%% achieved: if the sum of the rows/columns is not equal to the sum
%%           provided in the clue, then this is not a valid solution
```

Adding these two constraints fully specifies the puzzle solution. Because the puzzle solution has been specified in a generic way, the program will be able to solve a Kakurasu puzzle of any given size. The full code listing of the Kakurasu puzzle solver has been added as Appendix A.

For reference, the clingo output of the solver when run using the row/column clues for the puzzle as shown in figure 2(a) is listed below:

```
$ clingo kakurasu.lp 0
clingo version 5.6.2 (d469780)
Reading from kakurasu.lp
Solving...
Answer: 1
pos(1,1,black) pos(1,2,white) pos(1,3,white) pos(1,4,white) pos(1,5,white)
pos(2,1,white) pos(2,2,white) pos(2,3,white) pos(2,4,black) pos(2,5,black)
pos(3,1,black) pos(3,2,black) pos(3,3,black) pos(3,4,black) pos(3,5,white)
pos(4,1,black) pos(4,2,black) pos(4,3,white) pos(4,4,white) pos(4,5,white)
pos(5,1,black) pos(5,2,black) pos(5,3,white) pos(5,4,white) pos(5,5,black)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.013s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.013s
```

Reading from the output we see that the solvers finds a unique solution to the problem, and also that the solution matches the one shown in figure 2(b).

3.2 Effects on the solution calculation time

As was shown in section 2.2, the number of potential puzzles grows exponentially with the grid size. For $N = 4$ there are $2^{16} = 65536$ combinations possible, but for $N = 17$ this number has already grown to $2^{289} \approx 10^{87}$, which is orders of magnitude higher than the estimated number of protons in the observable universe ($\approx 10^{80}$)[1].

To get a feeling of how the execution time needed to solve the puzzle scales with increasing grid size, a series of puzzles of increasing size was solved using the Kakurasu solver. For each grid size three puzzles of comparable difficulty were used, and the mean CPU execution time to solve them was measured. For puzzles of sizes $N = 4 - 9$, puzzles taken from the "Hard Kakurasu" category from https://www.puzzle-kakurasu.com were used.

For larger grid sizes ($N > 9$), a simple Python script to generate Kakurasu puzzles of user configurable grid size and density was created[2]. It was noticed that for larger grid sizes and lower puzzle densities it becomes increasingly more difficult to generate puzzles with a unique solution. In practice, for puzzles ($N > 9$) a density of $0.85 - 0.9$ was used. All puzzles used in the measurements have a unique solution. The puzzles used in obtaining the results have been added as Appendix B.

The result of this experiment is shown in table 1. All tests were executed on a desktop PC running a VirtualBox VM (Ubuntu 22.04.3 LTS, 16GB/128GB SSD/Intel(R) Core(TM) i5-2400 CPU@3.10GHz x 4).

NxN	Time (s)	NxN	Time (s)
4 x 4	0.008	12 x 12	0.073
5 x 5	0.012	15 x 15	0.164
7 x 7	0.018	18 x 18	0.268
9 x 9	0.030	21 x 21	0.521

Table 1: Grid size (NxN) and mean CPU execution time (s) reported in solving the puzzle.

The results from table 1 are plotted in figure 3. Reading from the figure we see that the time needed to solve the puzzle increases non-linearly, most probably exponentially.

It is important to observe that the search space per row/column is determined by the number of ways in which the sum per row/column can be formed as compared to the total number of

[1] https://en.wikipedia.org/wiki/Eddington_number

[2] The script can be found as part of the project deliverables.

possible configurations per row/column. In general, if the sum of a row/column is low (i.e. near zero) then there are only a limited number of ways to form this sum, and the same is true for a sum which is near the maximum sum of the row/column. In contrast, values near the middle of the maximum sum per row/column can be formed in many more ways, so the search space for these values is larger.

The previous observation would imply that for puzzles with a high percentage of row/column clues near the middle values of the maximum sum per row/column the search space is bigger and the time needed to solve the puzzle would be higher, as compared to the time needed to solve puzzles with a high percentage of the clues near the minimum or maximum of the total sum per row/column. However, to investigate this further lies outside of the scope of the current project.

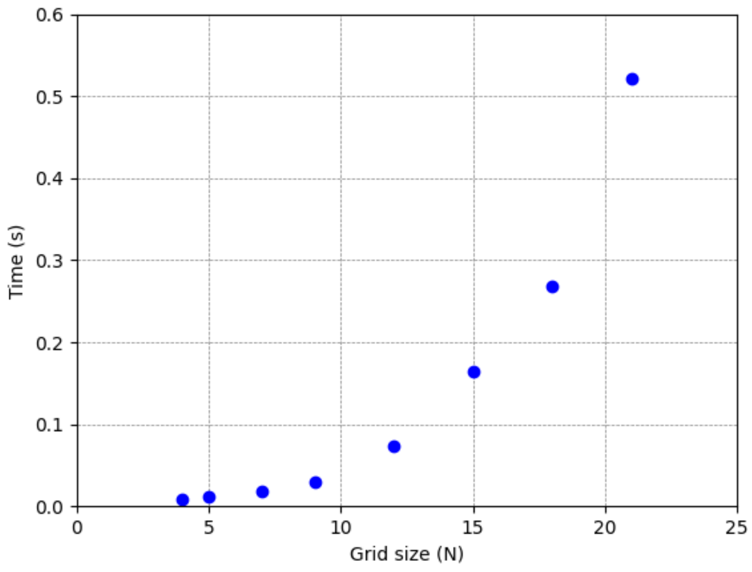


Figure 3: Grid size (N) vs. solving time (s).

4 Conclusion

After deriving a logic program to solve the Kakurasu puzzle using the ASP paradigm, and after investigating the influence of the puzzle size on the execution time required to solve the puzzle, we now return to our initial research questions to draw the conclusions.

A. **Is it possible to solve the Kakurasu puzzle using answer set programming?**

   As was shown in section 2.2, it is possible to solve the Kakurasu puzzle using ASP. The encoding of the problem into logic statements which define the form of the puzzle solution can be done in only a few lines of code.

B. **What is the effect of the Kakurasu puzzle size on the time needed to calculate the solution using clingo?**

   For all the "standard" Kakurasu puzzle sizes ($N = 4 - 9$) it was found that the solver is able to find the puzzle solution in a short time. Even for the largest puzzle size tested ($N = 21$), the time needed to solve the puzzle remains short on a human scale. It appears that the solving time grows in an exponential fashion.

References

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= asp+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[GKKS22] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer set solving in practice*. Springer Nature, 2022.

[JN16] Tomi Janhunen and Ilkka Nimelä. The answer set programming paradigm. *AI Magazine*, 37(3):13–24, 2016.

[KLPS16] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI magazine*, 37(3):25–32, 2016.

[Lif16] Vladimir Lifschitz. Answer sets and the language of answer set programming. *AI Magazine*, 37(3):7–12, 2016.

[Lif19] Vladimir Lifschitz. *Answer set programming.* Springer Heidelberg, 2019.

A Code Listing

```
%% Kakurasu puzzle solver
%%
%% Kakurasu is played on a rectangular grid N x N
%% 1. The black squares on each row sum up to the number on the right.
%% 2. The black squares on each column sum up to the number on the bottom.
%% 3. If a black cell is first on its row/column its value is 1. If it is
%%    second its value is 2, etc.

#const n=5.
color(white; black).

%% clues for the rows and columns
clues_row(1,1; 2,9; 3,10; 4,3; 5,8).
clues_col(1,13; 2,12; 3,3; 4,5; 5,7).

{pos(R,C,COL) : color(COL)} = 1 :- R=1..n, C=1..n.
%% achieved: every square of the grid is either black or white

row_tot(R,SUM) :- pos(R,_,COL), COL=black, SUM=#sum{C : pos(R,C,COL)}.
col_tot(C,SUM) :- pos(_,C,COL), COL=black, SUM=#sum{R : pos(R,C,COL)}.
%% achieved: calculate the sum of the rows/columns containing black squares

row_tot(R,0) :- pos(R,_,COL), COL=white, #count{C : pos(R,C,COL)} = n.
col_tot(C,0) :- pos(_,C,COL), COL=white, #count{R : pos(R,C,COL)} = n.
%% achieved: add the the sum of the rows/columns containing only white squares

:- row_tot(R,SUM1), clues_row(R,SUM2), SUM1 != SUM2.
:- col_tot(C,SUM1), clues_col(C,SUM2), SUM1 != SUM2.
%% achieved: if the sum of the rows/columns is not equal to the sum
%%           provided in the clue, then this is not a valid solution

#show pos/3.
```

B Puzzles used for time measurements

```
% 4 x 4
clues_row(1,7; 2,7; 3,7; 4,2).
clues_col(1,4; 2,8; 3,2; 4,6).
clues_row(1,3; 2,7; 3,3; 4,3).
clues_col(1,1; 2,1; 3,9; 4,2).
clues_row(1,7; 2,5; 3,6; 4,8).
clues_col(1,9; 2,3; 3,8; 4,7).

% 5 x 5
clues_row(1,3; 2,5; 3,9; 4,8; 5,12).
clues_col(1,4; 2,9; 3,11; 4,8; 5,9).
clues_row(1,3; 2,6; 3,12; 4,8; 5,4).
clues_col(1,7; 2,7; 3,5; 4,8; 5,7).
clues_row(1,7; 2,3; 3,11; 4,9; 5,6).
clues_col(1,5; 2,14; 3,8; 4,10; 5,3).

% 7 x 7
clues_row(1,25; 2,16; 3,13; 4,22; 5,5; 6,13; 7,24).
clues_col(1,17; 2,8; 3,16; 4,4; 5,22; 6,18; 7,20).
clues_row(1,13; 2,20; 3,26; 4,22; 5,15; 6,14; 7,17).
clues_col(1,13; 2,13; 3,27; 4,15; 5,21; 6,12; 7,20).
clues_row(1,6; 2,10; 3,21; 4,10; 5,13; 6,14; 7,17).
clues_col(1,21; 2,6; 3,16; 4,22; 5,25; 6,9; 7,7).

% 9 x 9
clues_row(1,38; 2,26; 3,31; 4,5; 5,25; 6,10; 7,18; 8,40; 9,31).
clues_col(1,8; 2,33; 3,16; 4,16; 5,33; 6,10; 7,29; 8,23; 9,35).
clues_row(1,30; 2,27; 3,23; 4,44; 5,2; 6,2; 7,17; 8,19; 9,31).
clues_col(1,25; 2,27; 3,11; 4,14; 5,29; 6,18; 7,8; 8,26; 9,24).
clues_row(1,25; 2,27; 3,23; 4,41; 5,27; 6,21; 7,10; 8,20; 9,15).
clues_col(1,39; 2,35; 3,1; 4,29; 5,21; 6,13; 7,9; 8,22; 9,32).

% 12 x 12
clues_row(1,66; 2,78; 3,67; 4,70; 5,57; 6,69; 7,69; 8,54; 9,57; 10,69; 11,65; 12,60).
clues_col(1,66; 2,51; 3,49; 4,69; 5,57; 6,70; 7,62; 8,71; 9,65; 10,66; 11,64; 12,59).
clues_row(1,57; 2,63; 3,75; 4,69; 5,66; 6,76; 7,73; 8,57; 9,63; 10,68; 11,44; 12,65).
clues_col(1,59; 2,65; 3,66; 4,57; 5,67; 6,77; 7,67; 8,67; 9,66; 10,57; 11,61; 12,58).
clues_row(1,76; 2,61; 3,69; 4,78; 5,44; 6,57; 7,68; 8,78; 9,66; 10,77; 11,49; 12,78).
clues_col(1,51; 2,69; 3,78; 4,49; 5,78; 6,55; 7,67; 8,58; 9,75; 10,62; 11,65; 12,78).

% 15 x 15
clues_row(1,114; 2,115; 3,108; 4,86; 5,111; 6,115; 7,118; 8,89; 9,94; 10,95;
11,106; 12,93; 13,97; 14,82; 15,96).
clues_col(1,105; 2,113; 3,107; 4,79; 5,111; 6,99; 7,92; 8,89; 9,96; 10,109;
11,96; 12,82; 13,99; 14,92; 15,112).
clues_row(1,120; 2,101; 3,113; 4,120; 5,75; 6,108; 7,82; 8,111; 9,100; 10,111;
11,101; 12,93; 13,113; 14,103; 15,103).
clues_col(1,120; 2,120; 3,101; 4,97; 5,88; 6,85; 7,85; 8,102; 9,93; 10,120;
11,95; 12,92; 13,103; 14,113; 15,120).
clues_row(1,106; 2,104; 3,114; 4,106; 5,102; 6,120; 7,112; 8,111; 9,120;
10,106; 11,92; 12,77; 13,100; 14,104; 15,120).
clues_col(1,106; 2,118; 3,107; 4,107; 5,108; 6,117; 7,102; 8,113; 9,100;
10,94; 11,115; 12,120; 13,120; 14,82; 15,94).
```

```
% 18 x 18
clues_row(1,171; 2,146; 3,171; 4,165; 5,163; 6,171; 7,156; 8,113; 9,137; 10,169;
11,171; 12,144; 13,163; 14,171; 15,171; 16,159; 17,161; 18,171).
clues_col(1,171; 2,161; 3,171; 4,163; 5,171; 6,158; 7,171; 8,153; 9,171; 10,137;
11,171; 12,145; 13,157; 14,159; 15,156; 16,171; 17,163; 18,162).
clues_row(1,160; 2,157; 3,148; 4,166; 5,168; 6,153; 7,169; 8,157; 9,152; 10,162;
11,171; 12,167; 13,167; 14,171; 15,167; 16,156; 17,147; 18,147).
clues_col(1,150; 2,146; 3,163; 4,109; 5,158; 6,171; 7,162; 8,136; 9,171; 10,163;
11,170; 12,162; 13,171; 14,151; 15,135; 16,171; 17,171; 18,165).
clues_row(1,171; 2,153; 3,133; 4,146; 5,131; 6,166; 7,159; 8,171; 9,158; 10,153;
11,167; 12,169; 13,152; 14,146; 15,166; 16,167; 17,171; 18,150).
clues_col(1,171; 2,131; 3,153; 4,144; 5,161; 6,171; 7,154; 8,161; 9,167; 10,140;
11,144; 12,159; 13,162; 14,171; 15,171; 16,171; 17,153; 18,152).

% 21 x 21
clues_row(1,159; 2,199; 3,184; 4,215; 5,209; 6,185; 7,205; 8,218; 9,223; 10,194;
11,206; 12,228; 13,223; 14,231; 15,213; 16,206; 17,218; 18,226; 19,224; 20,220;
21,208).
clues_col(1,216; 2,215; 3,193; 4,231; 5,202; 6,207; 7,196; 8,209; 9,231; 10,201;
11,203; 12,227; 13,182; 14,215; 15,219; 16,211; 17,203; 18,230; 19,226; 20,231;
21,221).
clues_row(1,230; 2,176; 3,231; 4,220; 5,206; 6,231; 7,231; 8,217; 9,205; 10,226;
11,226; 12,214; 13,218; 14,209; 15,231; 16,192; 17,214; 18,231; 19,226; 20,217;
21,195).
clues_col(1,210; 2,212; 3,191; 4,192; 5,196; 6,231; 7,231; 8,231; 9,226; 10,231;
11,227; 12,222; 13,210; 14,163; 15,210; 16,212; 17,200; 18,231; 19,231; 20,231;
21,213).
clues_row(1,205; 2,221; 3,206; 4,206; 5,231; 6,201; 7,231; 8,231; 9,212; 10,203;
11,206; 12,159; 13,223; 14,182; 15,225; 16,221; 17,216; 18,185; 19,216; 20,180;
21,212).
clues_col(1,231; 2,208; 3,231; 4,195; 5,221; 6,214; 7,228; 8,206; 9,207; 10,200;
11,220; 12,199; 13,188; 14,208; 15,157; 16,193; 17,231; 18,192; 19,209; 20,211;
21,226).
```