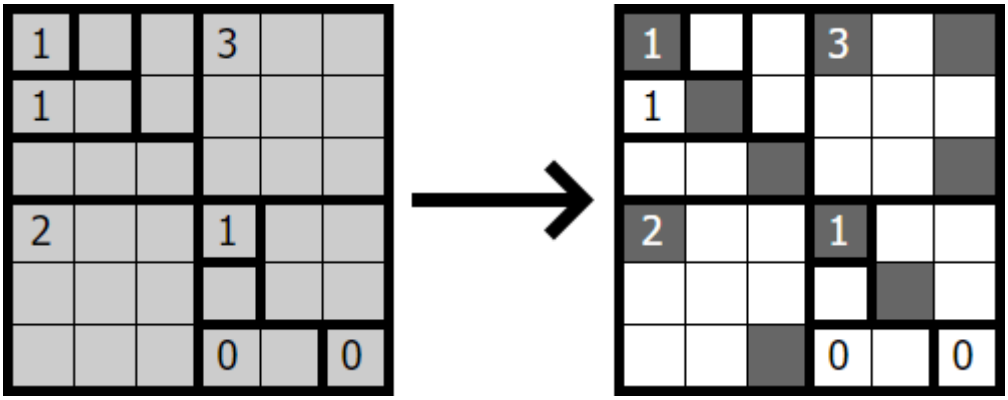
Heyawake Solver Documentation

Heyawake Puzzle Introduction



A Heyawake puzzle is a logic puzzle where each cell of a grid must be coloured black or white. Regions are identified as the groups of cells surrounded by thick lines. Some regions have an associated number. The rules are:

1. Regions with a number *must* have that number of black cells within it
2. Two black cells *cannot* be adjacent to each other
3. A straight line of white cells *cannot* span more than 2 regions
4. All white cells *must* be connected in a single group

All puzzles are sourced from www.puzzle-heyawake.com. This program allows the user to input a Heyawake puzzle and solve it with Gurobi. It solves square grids of size 4x4 all the way up to 35x35 using constraints for the first 3 rules and lazy constraints for the 4th rule. The program comes preloaded with 10 puzzles, 3 of which are 35x35.

Small grids such as 6x6 are solved near instantly. Testing on 3 different 35x35 grids, the runtimes were on average 0.19s, 0.11s and 0.21s respectively. This really shows the power of Gurobi as no other solver I found online even comes close to this speed. The solvers I found online and their runtimes are below:

GitHub - FloThinksPi/Heyawake-Solver: This solver calculates the solution for the japanese puzzle "heyawake", like sudoku solvers do. Study Project. solves an 8x8 grid in 1.78s (solver on Github from 2016)

FULLTEXT01.pdf (diva-portal.org) solves 11x11 grids within 13-31s (Bachelor's Thesis from 2014)

Heyawake Solver in Copris (cspsat.gitlab.io) solves three different 31x45 grids in 18.25s, 18.22s and 38.68s respectively (solver program created in 2013 and updated in 2020)

Applying Swarm Intelligence to Solve Heyawake Puzzles (psu.edu) solves three different 10x10 grids in (on average) 433.26s, 1.78s and 0.55s respectively. This program utilises swarm intelligence so the runtimes vary, with the shortest runtime being 0.22s and the longest runtime being 1481.28s (honour's thesis from 2010)

Solver Explanation

Assuming we have an $n \times n$ grid, the variables and constraints for the Gurobi solver are as follows:

Variable:

$$X[i,j,c] \in [0,1] \qquad \forall i,j \in 1,\dots,n \quad \forall c \in [0,1]$$

In essence, the grid point at (i,j) is black if $X[i,j,1] = 1$ and white if $X[i,j,0] = 1$.

Constraints:

Select one colour only for each grid point:

$$X[i,j,0] + X[i,j,1] == 1 \qquad \forall i,j \in 1,\dots,n$$

Adhere to the region number where necessary (rule 1):

$$\sum_{i,j \in \text{region}} X[i,j,1] == \text{regionNumber} \qquad \forall \text{ regions such that regionNumber exists}$$

Black cells must not be adjacent (rule 2):

$$\sum_{ii,jj \in \text{Neigh}(i,j)} X[ii,jj,1] \leq \text{len}\big(\text{Neigh}(i,j)\big) \times (1 - X[i,j,1]) \qquad \forall i,j \in 1,\dots,n$$

Where $\text{Neigh}(i,j)$ is the list of all orthogonal neighbours of (i,j)

If a white cell exists, then there clearly must exist at least another white cell in the adjacent cells (otherwise rule 4 is violated). This constraint is not necessary but it improves runtime significantly:

$$\sum_{(ii,jj) \in \text{Neigh(i,j)}} X[ii,jj,0] \geq X[i,j,0] \qquad \forall i,j \in 1,\dots,n$$

A vertical line of cells spanning across 3 regions must contain at least 1 black cell (rule 3a). The program implements this by scanning through all cells with a “southern border” (i.e. a bold line below it). For each of these cells, a list is generated via the function vertNeigh(i, j) including the original cell and all cells below it such that *exactly* 3 regions are spanned. If the function fails to find such a list, then we move onto the next cell:

$$\sum_{ii,jj \in \text{vertNeigh}(i,j)} X[ii,jj,1] \geq 1 \qquad \forall i,j \in 1,\dots,n \text{ such that } (i,j) \text{ has southern border and vertNeigh(i, j) exists}$$

A horizontal line of cells spanning across 3 regions must contain at least 1 black cell (rule 3b). This is exactly the same as above except it applies to cells with an “eastern border” and iterates to the right:

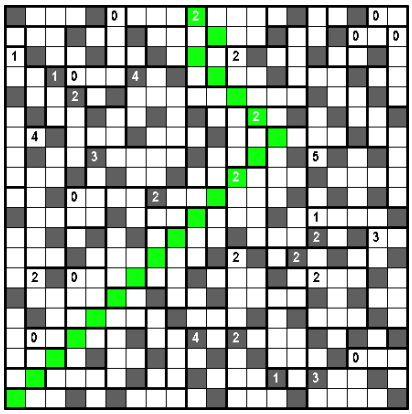
$$\sum_{ii,jj \in \text{horNeigh}(i,j)} X[ii,jj,1] \geq 1 \qquad \forall i,j \in 1,\dots,n \text{ such that } (i,j) \text{ has eastern border and horNeigh(i, j) exists}$$

Lazy Constraints

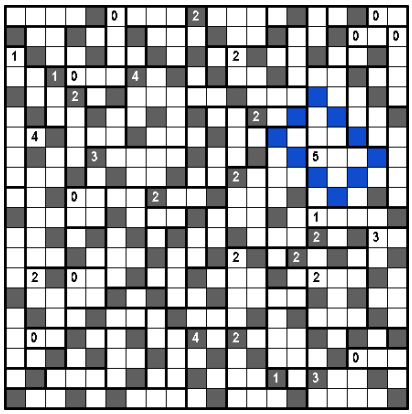
The 4th rule is dealt with via lazy constraints. Upon each solution adhering to the above constraints, we scan the grid to make sure all white cells are connected. A naïve way to approach this is to start at a white cell and iterate through all connected white cells to see if we can reach all other white cells. If there are white cells that cannot be reached, a lazy constraint can be added to prevent this solution from happening. This approach is relatively slow.

A better way to approach the problem is to consider what constitutes all white cells *not* being connected. If all white cells are *not* connected, then there must exist either:

1. An impenetrable wall. That is, a diagonally connected line of black cells which touch 2 boundary points:



2. A closed loop. That is, a diagonalled connected line of black cells which forms a loop:

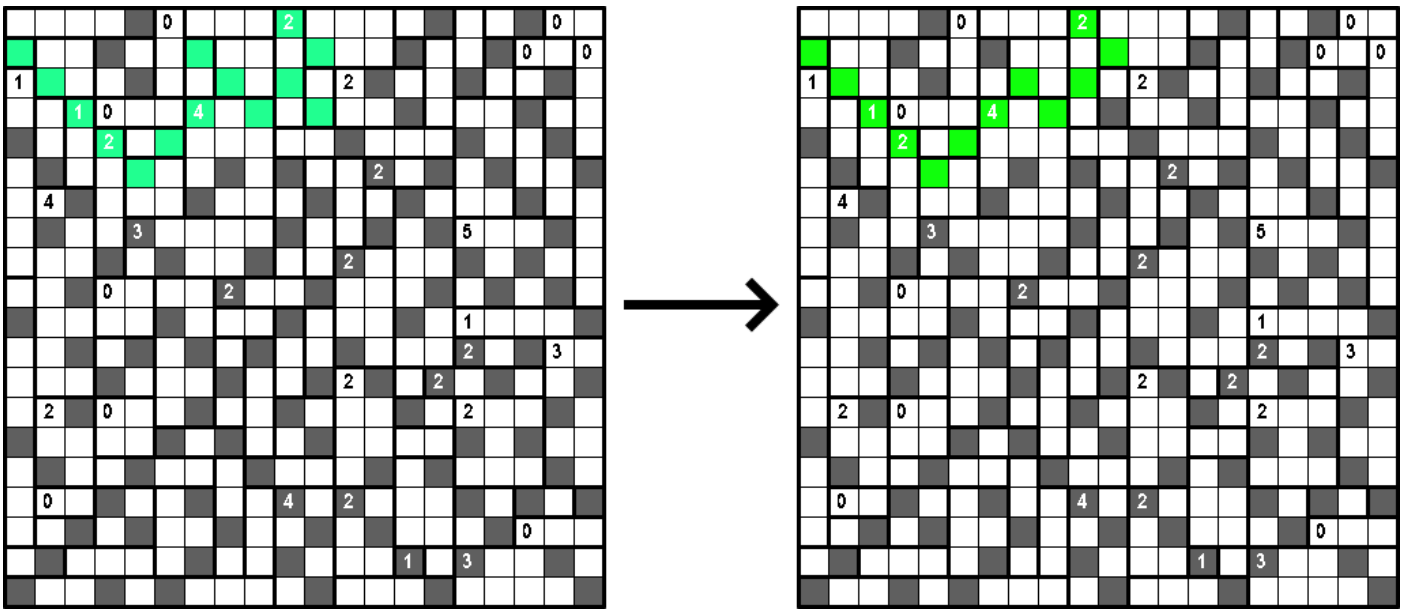


Given a solution, the program attempts to find one impenetrable wall and one closed loop. If it finds such a rule violation, a lazy constraint is added to prevent that group of black cells from forming again. This repeats until no impenetrable wall and loop is found, giving the final correct solution. More specifically, given a group of cells visitedCells which contains cells forming an impenetrable wall or a loop, the lazy constraint is given as:

$$\sum_{(i,j)\in \text{visitedCells}} X[i,j,1] \leq \text{len}(\text{visitedCells}) - 1$$

I experimented and found that trying to find just one impenetrable wall and one closed loop on each callback was the most efficient. If I tried to find all impenetrable walls and all closed loops on each callback, the program would identify the same walls and loops multiple times, thus wasting time.

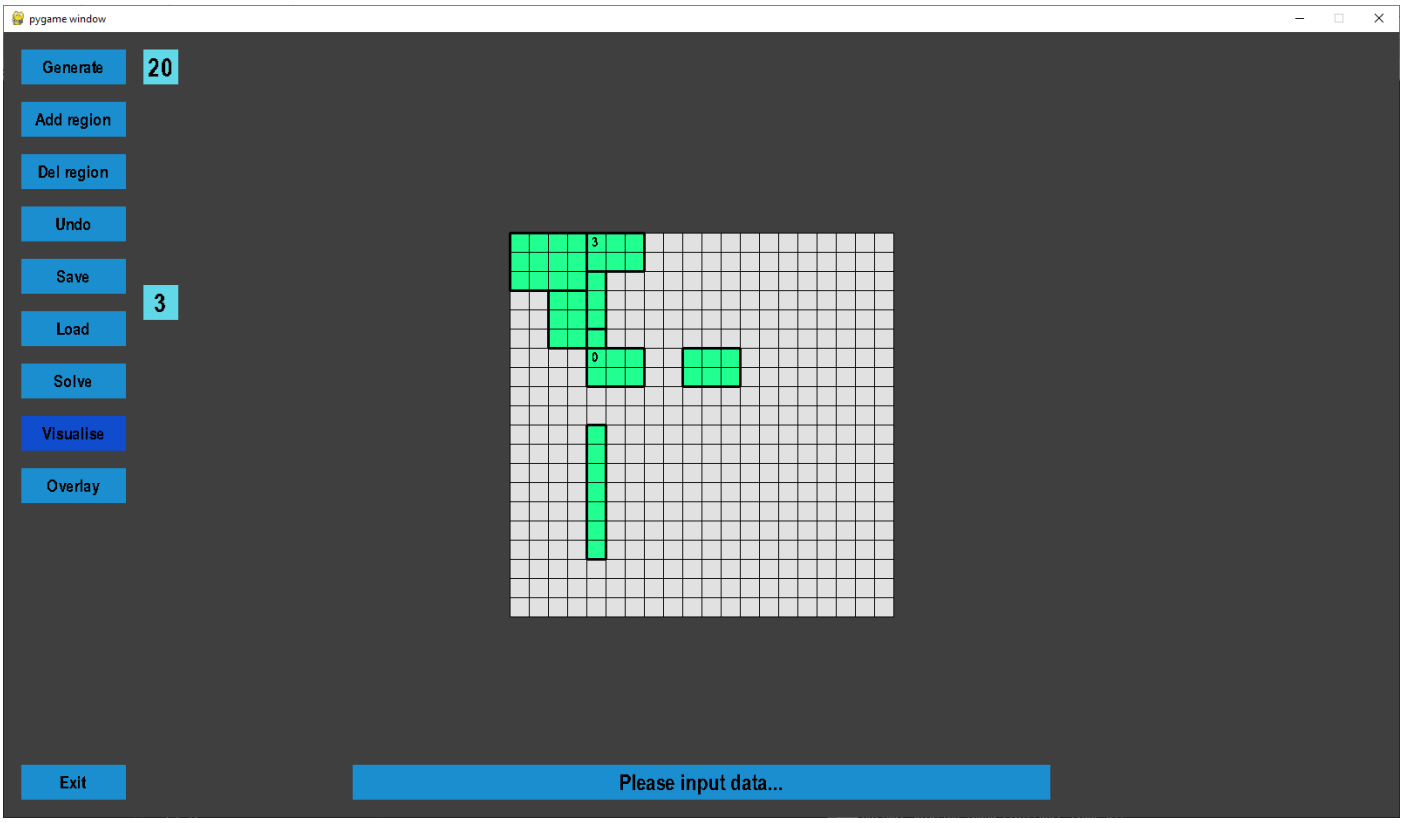
The algorithm for finding an impenetrable wall involves a recursive function which starts with a black boundary cell and iterates diagonally through other black cells until another boundary cell is found. By design, this recursive function yields a list containing this impenetrable wall but also may contain cells which look like “offshoots” of the wall, where the algorithm was searching prior to finding the second boundary point (this is the group highlighted in *light green* below). It is perfectly okay to add a lazy constraint to prevent this exact group of black cells from appearing again, but the runtime is improved quite significantly if we run a second algorithm which “chops off” these offshoots. This second algorithm runs until all offshoots have been cut off and only the impenetrable wall remains (this is the group highlighted in *green* below).



A similar process is run for the closed loops. Both of these processes are visualised within the program (if the **Visualise** option is toggled). Impenetrable walls are identified by green and closed loops are identified by blue.

Program Explanation

The program utilises pygame for the GUI and allows the user to easily enter in any square Heyawake puzzle. Required modules are: pygame, sys, time, pickle, gurobipy. The solver is shown below:



The user can right click on the grid to start building regions which will be highlighted in *light green*. The region can be finalised by clicking the **Add region** button. Then, if applicable, the user can left click within the region and enter the region number. Numbers can be erased using backspace. Additionally, if an entire region is right clicked, it will be highlighted in red and can be removed with the **Del region** button.

Buttons:

Generate: This generates a $n \times n$ grid where n is the number inserted in the box beside the button

Add region: This turns any selection of cells into a region. A shortcut for this button is the spacebar

Del region: This deletes any regions highlighted in red

Undo: This will undo the previous action whether it be selecting a cell or adding a region

Save/Load: This allows the user to save a completed puzzle template to file under any number from 0-9, where the number can be inserted in the box beside the two buttons

Solve: This will solve the currently input puzzle

Visualise: This toggles the visualisation of lazy constraints being added

Overlay: This will overlay a screenshot of any puzzle onto the grid. Simply take a screenshot of the grid from a website and put it in the program's directory named *puzzle.png*. The program will automatically resize it and make it transparent on top of the grid

Exit: Exit the program

Additionally, the textbox at the bottom of the screen will display useful information for the user.