Tilepaint Puzzles App
User Manual

Vincentius Arnold Fridolin

June 27, 2023

Contents

1 Introduction

1.1 Tilepaint Puzzle Overview

Tilepaint (タイルペイント) is a logic pencil-and-paper puzzle created by Nikoli and invented by Toshihari Yamamoto in Japan. According to Nikoli manager Jimmy Goto, Tilepaint first appeared in issue 53 of Nikoli's quarterly Puzzle Communication magazine in 1995 and has been published regularly ever since This puzzle considers an $m \times n$ grid of cells. The cells are divided into tiles that are separated by bold lines. Each cell in every tile must be either colored or uncolored. There are numbers at the top and left of the grid indicating the number of colored cells in the corresponding row and column. The problem is to find any configuration that matches the number of colored cells according to the constraint described for each row and column (if any).

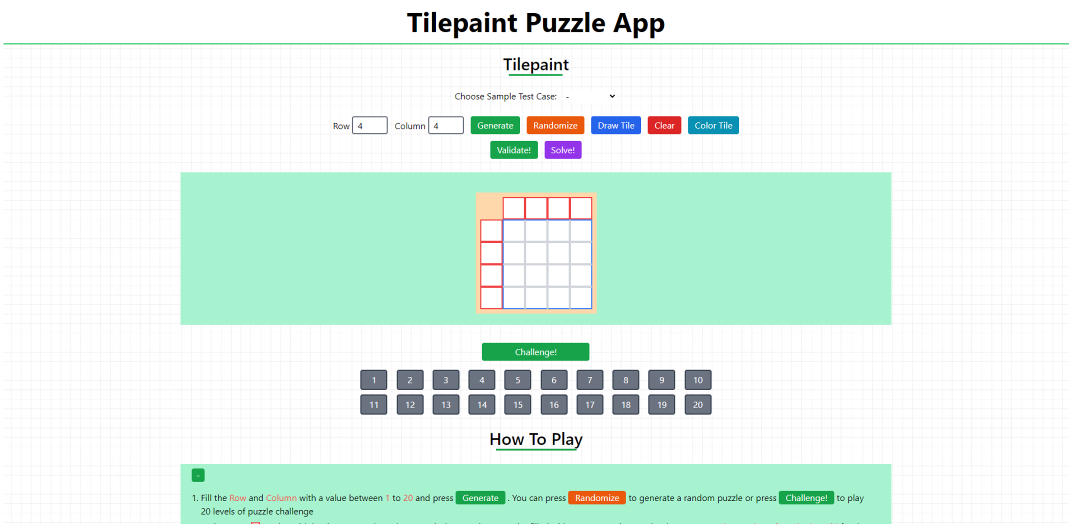
1.2 App Overview

The Tilepaint Puzzle Solver App is designed to solve any Tilepaint puzzle, offering a range of features for solving experience. This app is also can validate a configuration of coloring tiles in the grid that you create with some tiles and constraints. The app provides a user-friendly interface and some useful tools to tackle the solving of Tilepaint puzzles. This manual will guide you to use the app step-by-step of generating the puzzle grid, create some tiles, modify the constrain, solve the puzzle, and color the tile to solve puzzles on your own. Figure 1 shows the app interface.

2 Functionalities

2.1 Adjusting the Puzzle Grid

At the top left of the app, users can adjust the puzzle size by entering values for the number of rows and columns, respectively. The valid range for the row and column inputs is from 1 to 20, inclusive. The default values for the row and column inputs are set to 4. To construct or reconstruct the grid, users need to press the "Generate" button. Figure 2 illustrates the row and column input interface. Another way to construct the instance is by choosing the sample test case on the top of the "Generate" button. Choosing one of them will automatically generate the grid. Figure 3 shows the example of choosing the sample test case.



1. Fill the Row and Column with a value between 1 to 20 and press Generate . You can press Randomize to generate a random puzzle or press Challenge! to play 20 levels of puzzle challenge

Figure 1: The layout of the Tilepaint puzzle App.

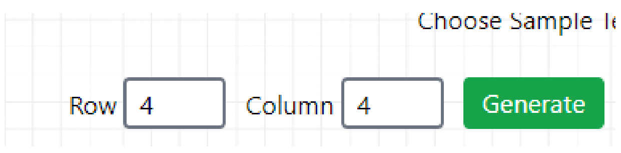


Figure 2: The Row and Column input.

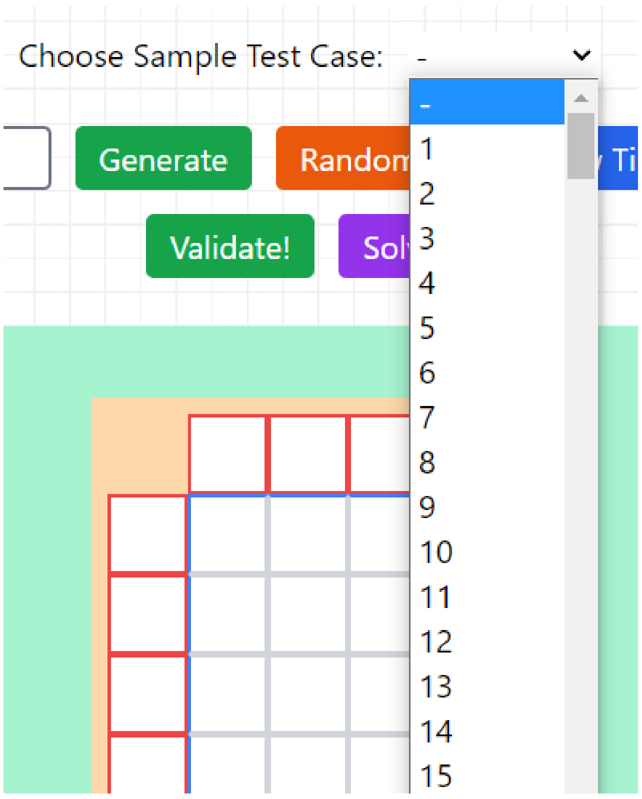


Figure 3: An example of choosing an example test case.

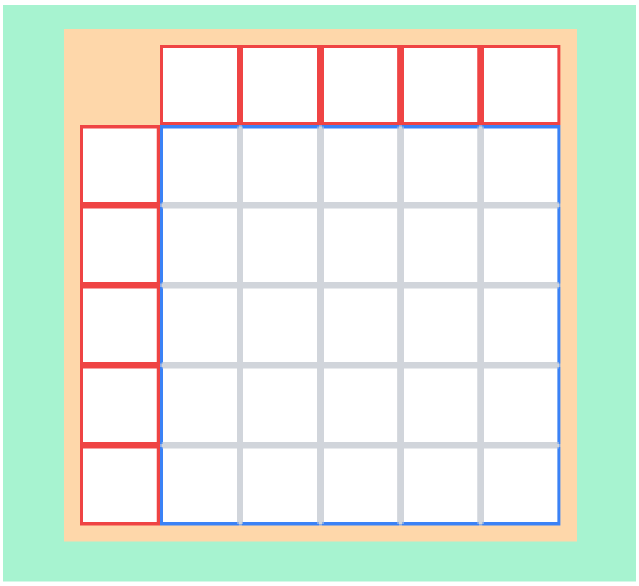


Figure 4: An example of the constructed 5 $\times$ 5 grid.

2.2 Construct a Randomize Instance

The “Randomize” button automatically generates a random instance of a Tilepaint puzzle. This feature is particularly useful when you want to solve a puzzle with a randomly generated tile pattern. The generated instance is guaranteed to have at least one valid solution to the given constraints. Users can press the “Randomize” button multiple times to generate new Tilepaint instances.

2.3 Adding and Modifying the Constraint Numbers

Once the grid is generated, the user will find it displayed in the center of the page. The top and left cells of the grid are bordered in red, indicating that they can be filled with non-negative numbers representing the column and row constraints, respectively. To fill or modify a constraint, simply click on the red-bordered cell in the grid and enter a number. If users want to have no restrictions on coloring cells in a row or column, you can leave the value empty. Figure 4 illustrates an example of the red-bordered cells in the grid.

2.4 Create Some Tiles in the Grid

Besides the "Randomize" button, there is a "Draw Tile" button. To create a single tile, the user needs to press the "Draw Tile" button to activate the editing

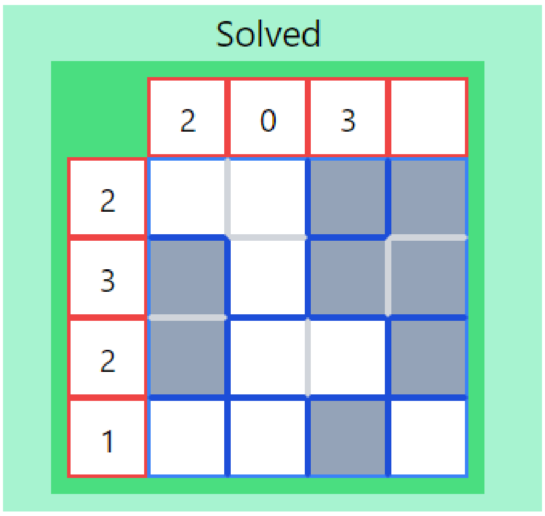


Figure 5: A solution grid to the Tilepaint instance.

mode. Once in this mode, the user can left-click and hold the mouse button while moving the cursor across adjacent cells to create a tile. Releasing the left-click mouse button will stop drawing the tile. The user can repeat this process to create additional tiles with different shapes. To create a new tile, the user can repeat the aforementioned steps. Pressing the "Draw Tile" button will deactivate the grid editing mode.

2.5 The Color Tile Button

The user can try to solve the puzzle by coloring some of the tiles in the instance that was constructed by clicking the button “Color Tile”. By activating this mode, the user now can press on a cell to color or uncolor the set of cells in the same tile to which the pressed cell belongs. If the solution is found, the orange background will turn to green and a “Solved” string will appear on the top of the grid. The apps will automatically verify the grid, so the user does not need to press any button to verify the configuration of the colored tile. However, the user can verify the configuration by pressing the “Validate!” button. Pressing the “Validate!” button will show the unmatched number of colored cells in a specific row or column. The example of the solution grid is depicted in Figure 5.

2.6 Verifying the Puzzle

The user has the option to validate whether the coloring configuration is a solution to the grid. By pressing the “Validate!” button, the user will receive

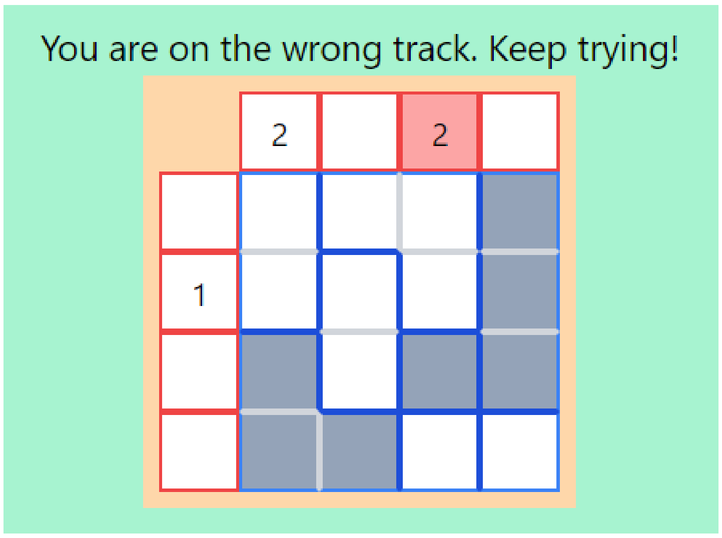


Figure 6: An example of hints by pressing “Validate!” button.

hints indicating the columns and rows where the count of colored cells does not match the specified constraints. If the configuration is indeed a solution to the grid, the orange background will turn green, and a “Solved” string will appear at the top of the grid. However, if the configuration is not a valid solution, the grid will highlight the mismatched number of colored cells in a specific row or column, allowing the user to identify and correct the errors. An example of such a hint is shown in Figure 6.

2.7 Solving the Puzzle

The “Solve!” button is used to find a solution to the grid. If there exists a solution to the instance, some tiles in the grid will be colored, and the orange background will turn green. Additionally, the string “Solved!” will appear at the top of the grid, indicating a successful solution. However, if there is no solution to the instance, the string "No Solution" will appear instead. In such cases, the user can modify the constraint numbers or change the tile shape to make the puzzle solvable. The solution and no solution grids generated by the solver are illustrated in Figure 5 and Figure 7, respectively.

2.8 Play a Challenge level

The user can engage in a challenging gameplay experience by selecting a stage from levels 1 to 20. As the user progresses through the levels, the puzzle difficulty increases, featuring distinct grid sizes and varying numbers

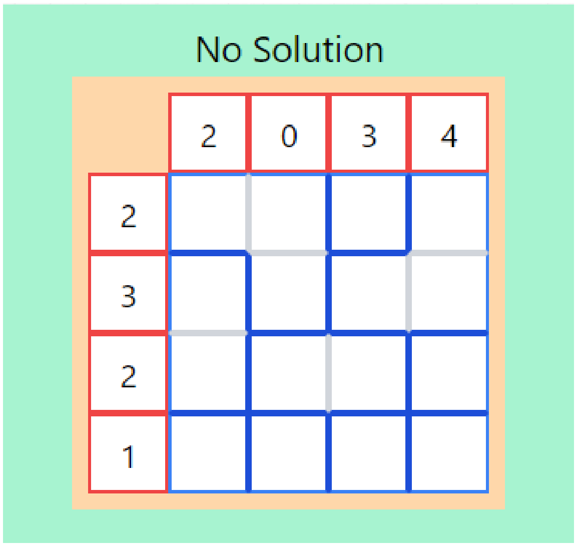


Figure 7: An example of a no solution grid by pressing the “Solve!” button.

of constraints. By clicking the “Challenge!” button, the user can select any number displayed below the button, with the grid remaining unchanged until the user decides to give up or commence a new challenge session. Upon successfully solving a grid for a specific level, the level number will turn green, accompanied by the appearance of the word “Solved”. The “Validate!” button can also be utilized during these challenges. In the event that the user decides to give up, they can employ the "Solve!" button to reveal the grid’s solution or attempt the level again. It’s important to note that the grid remains unaltered until the user initiates another challenge, allowing for independent attempts to find the solution outside the challenge session. The example of the challenge session is depicted in Figure 8.

2.9 Clear Button

To remove numbered cells, constraints, and colored cells, press the “Clear” button. It will clear all of it without removing the shape of all tiles in the grid. This button is useful when the user wants to solve the puzzle after knowing the solution or starts solving with the same tile shape but with a different constraint number.

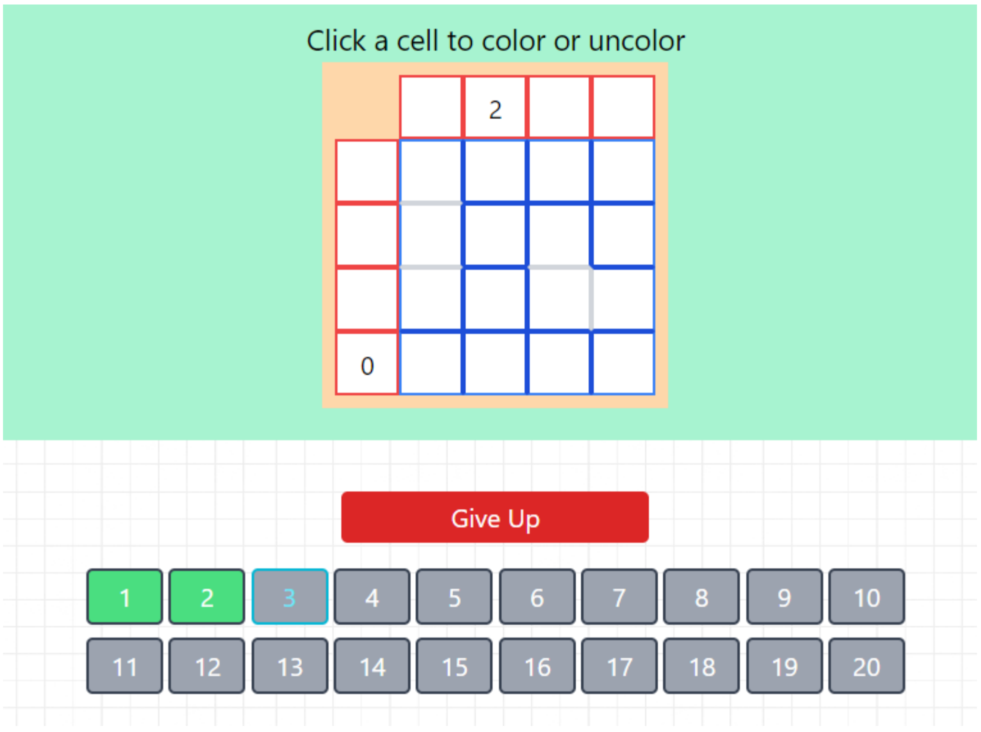


Figure 8: An example of a challenge session with level 1 and 2 is already solved.

3 Source Code

The app was built using HTML, Javascript, and Python with the Django framework. Interested users can find the source code of this project on `https://github.com/vincentiusar/Tilepaint-Solver-App`.

The solver source code is implemented in Python using pysat which uses a SAT-based approach for solving the puzzle. For the full source code, please refer to the aforementioned repository. The following program is the solver code.

```python
from pysat.solvers import Glucose3
from pysat.formula import CNF
from pysat.card import *
import itertools

m = 0
n = 0

def h(i, j):
    return i*n + j + 1

def atMost(A, k):
    combi = list(itertools.combinations(A, k+1))
    combi_list = [[*a] for a in combi]
    combi_neg = [[-a for a in combi] for combi in combi_list]

    return combi_neg

def atLeast(A, k):
    return atMost([-a for a in A], len(A) - k)

def equals(A, k):
    cnf = []
    atleast, atmost = [*atLeast(A, k)], [*atMost(A, k)]
    if len(atleast) != 0:
        cnf.append(atleast)
    if len(atmost) != 0:
        cnf.append(atmost)

    return cnf

def phi_1(R, p):
    cnf = []
    for k in range(1, p+1):
        for i in range(len(R[k])):
            for j in range(i+1, len(R[k])):
                cnf.append([-1 * h(R[k][i][0], R[k][i][1]),
                            h(R[k][j][0], R[k][j][1])])
                cnf.append([h(R[k][i][0], R[k][i][1]),
                            -1 * h(R[k][j][0], R[k][j][1])])

    return cnf

def phi_2(cc):
    cnf = []
    for j in range(n):
        if (cc[j] != -1):
            lits = []
            for i in range(m):
                lits.append(h(i,j))
            cnf.extend(equals(lits, cc[j]))

    return cnf

def phi_3(cr):
    cnf = []
    for i in range(m):
        if (cr[i] != -1):
            lits = []
            for j in range(n):
                lits.append(h(i,j))
            cnf.extend(equals(lits, cr[i]))

    return cnf

def solve(M, N, CG, CC, CR):
    global m
    global n

    m = M
    n = N

    MaxT = 0
    for i in range(len(CG)):
        MaxT = max(MaxT, max(CG[i]))
    R = [[] for _ in range(MaxT+1)]

    for i in range(M):
        for j in range(N):
            R[CG[i][j]].append((i, j))

    for i in range(M):
        if (CR[i] > N):
            return {'Model': None}

    for j in range(N):
        if (CC[j] > M):
            return {'Model': None}

    s = Glucose3()

    s.append_formula(phi_1(R, MaxT))
    for item in (phi_2(CC)) :
        s.append_formula(item)
    for item in (phi_3(CR)) :
        s.append_formula(item)

    s.solve()
    model = s.get_model()

    return {'Model': model}
```

The following source code is the verifier for whether the configuration is the solution.

```
m = 0
n = 0

def isSumEqual(CC, CR):
    sum_row, sum_col = 0, 0
    for item in CC:
        if (item == -1):
            return True
        sum_col += item

    for item in CR:
        if (item == -1):
            return True
        sum_row += item

    return sum_col == sum_row

def complyConstraint(Cell, CC, CR, m, n):
    coloredCol, coloredRow = [0 for _ in range(n)],
                             [0 for _ in range(m)]

    cr, cc = [], []

    for i in range(m):
        for j in range(n):
            if (Cell[i][j] == 1):
                coloredCol[j] += 1
                coloredRow[i] += 1

    for j in range(n) :
        if (CC[j] != -1 and coloredCol[j] != CC[j]) :
            cc.append(j+1)

    for i in range(m) :
        if (CR[i] != -1 and coloredRow[i] != CR[i]) :
            cr.append(i+1)

    return len(cc) == 0 and len(cr) == 0, cc, cr

def IsAllSame(CG, Cell, m, n, p):
    colored, cellTotal = [0 for _ in range(p+1)],
                         [0 for _ in range(p+1)]

    for i in range(m):
        for j in range(n):
            cellTotal[CG[i][j]] += 1
            if (Cell[i][j] == 1):
                colored[CG[i][j]] += 1

    for i in range(1, p+1):
        if (colored[i] != cellTotal[i] and colored[i] != 0):
            return False

    return True

def color(CG, Cell, t, c):
    for i in range(len(CG)):
        for j in range(len(CG[i])):
            if (CG[i][j] == t) :
                Cell[i][j] = c

def validate(M, N, CG, S, CC, CR):
    MaxT = 0
    for i in range(len(CG)):
        MaxT = max(MaxT, max(CG[i]))

    Cell = [[0 for _ in range(N)] for _ in range(M)]

    for item in S:
        color(CG, Cell, item, 1)

    rule1 = isSumEqual(CC, CR)
    rule2 = IsAllSame(CG, Cell, M, N, MaxT)
    rule3, cc, cr = complyConstraint(Cell, CC, CR, M, N)

    return {
        'Model': rule1 and rule2 and rule3,
        "cc": cc,
        "cr": cr
    }
```