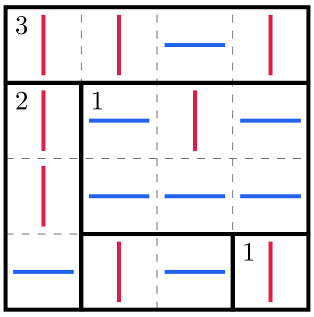
Juosan Interactive Playground: Design, Solve, Conquer!

A guide to using the Juosan Interactive Playground application.



Authors: M. Tsaqif Ammar & M. Arzaki (editor)

Computing Laboratory, School of Computing
Telkom University, Indonesia

July 29, 2023

Contents

1 Introduction

We thank the players for using the Juosan Interactive Playground application. This manual will guide the user through the application features and usage to effectively design and solve Juosan puzzles.

1.1 About Juosan Puzzles

Juosan (縦横さん in Japanese) is a puzzle created by Nikoli (the mastermind behind Sudoku) which provides an engaging single-player challenge. Originally this puzzle is played using only paper and pencil; however, our Juosan Interactive Playground provides an interactive experience for playing and solving Juosan puzzles. These puzzles have been proven NP-complete, meaning their complexity increases exponentially as the size grows. Even computers find these puzzles challenging to solve if the puzzles' sizes are sufficiently large. A Juosan puzzle presents you with an $M \times N$ grid divided into multiple rectangular territories. The objective is to fill each cell with either a ⊟ or ◫ symbol, following specific rules, namely:

- If a territory contains a number, the number of ⊟ or ◫ symbols within it must be equal to that number;
- The ⊟ symbol cannot extend vertically for more than two cells;
- The ◫ symbol cannot extend horizontally for more than two cells.

To further illustrate the Juosan puzzle, please refer to Fig. 1.

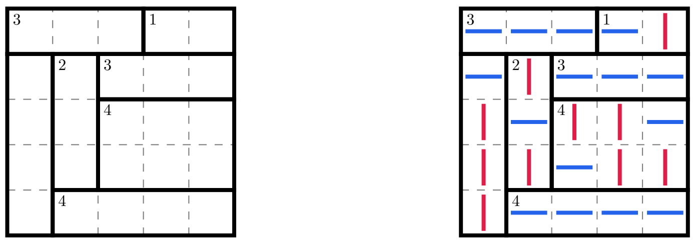


Figure 1: An example of a Juosan puzzle and its solution.

There is no prerequisite to play Juosan puzzles. These puzzles involve very little numerical calculation.

1.2 Application Overview

The Juosan Interactive Playground application is a user-friendly and interactive tool that allows puzzle enthusiasts to design and solve Juosan puzzles. The key features of this application include:

1. **Puzzle Selection**: choose from a selection of example puzzles with varying difficulty levels. These puzzles provide challenges for users and serve as demonstrations of the application's solver, showcasing its ability to solve even large and complex puzzles.
2. **Create Custom Puzzles**: utilize the "Draw Your Puzzle" feature to create customized Juosan puzzles. The players can customize the grid size and add territories and constraint numbers to design unique puzzles.
3. **Solve Yourself**: engage in the puzzle-solving process of the Juosan puzzle.
4. **Autocomplete Feature**: if the players need assistance, the application offers an "Auto-complete" button that employs a built-in solver to complete the puzzle automatically.

The Juosan Interactive Playground application combines the joy of solving puzzles with the convenience of digital technology. Whether the players choose pre-designed puzzles or create their own, this application provides a delightful platform to explore and solve Juosan puzzles interactively.

2 Functionalities

2.1 General Usage

This section discusses the features and tools that are available in the application. The application's main interface is depicted in Fig. 2.

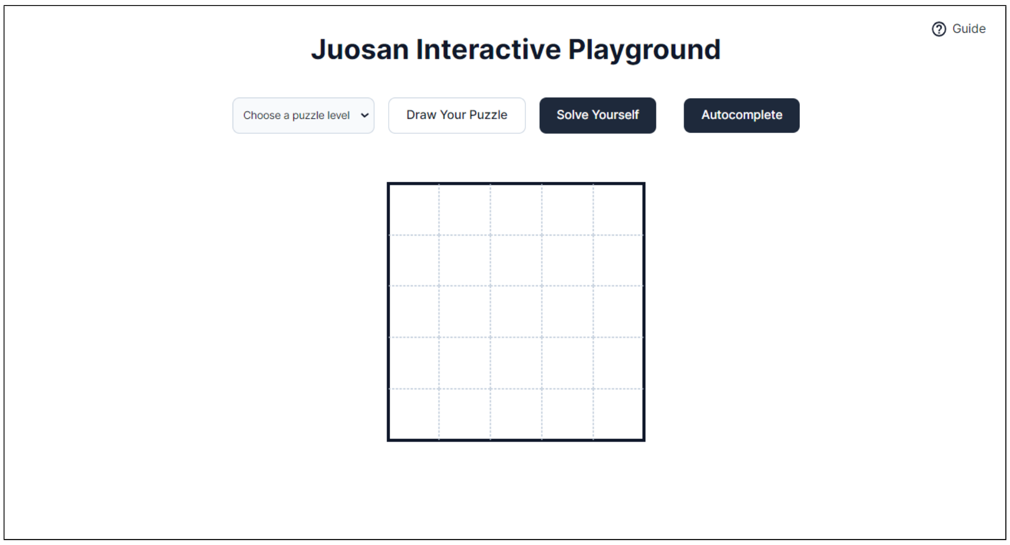


Figure 2: The application main interface.

To play the puzzles independently, the players can choose a puzzle level from the provided example puzzles at the top left (see Fig. 3). This selection of ready-to-solve puzzles allows them to practice solving different puzzle variations, with the levels ranging from easy to advanced, determined by the sizes of the puzzles and the amount of information initially given. We have curated most of these puzzles from Otto Janko Website, a renowned puzzle collection platform. Alternatively, the players can draw their Juosan puzzle by clicking the "Draw Your Puzzle" button. We discuss the tools in the drawing mode in Section 2.2.

After selecting (or drawing) a puzzle, the players have two options to proceed:

1. If they want to solve the puzzle themselves, they click the "Solve Yourself" button. They can fill a cell by clicking on it multiple times to cycle through the available symbols as illustrated in Fig. 4. If they encounter difficulties and decide to give up, they can press the "Give Up" button. Once they have finished solving, they press the "Submit" button to check their solution. If the players fill the grid correctly, a notification as in Fig. 5 appears. The application will give them hints as depicted in Fig. 6 if they fill the puzzle incorrectly.

2. If they prefer to use our built-in solver, they click the "Autocomplete" button. The application's solver will attempt to complete the puzzle for the players. Additionally, the application will display the status of the puzzle, indicating whether it has a solution or not. Fig. 7 demonstrates how the "Autocomplete" feature helps in finding a solution.

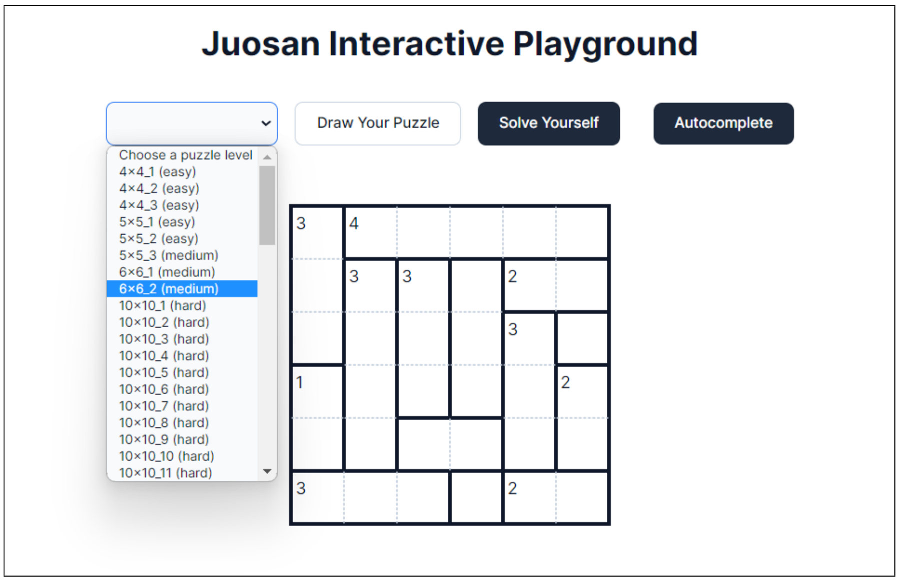
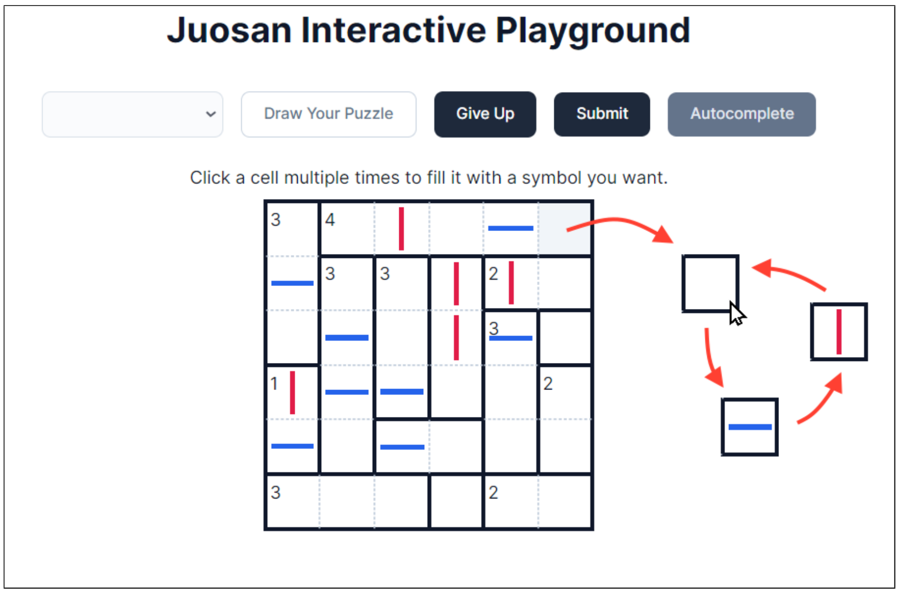
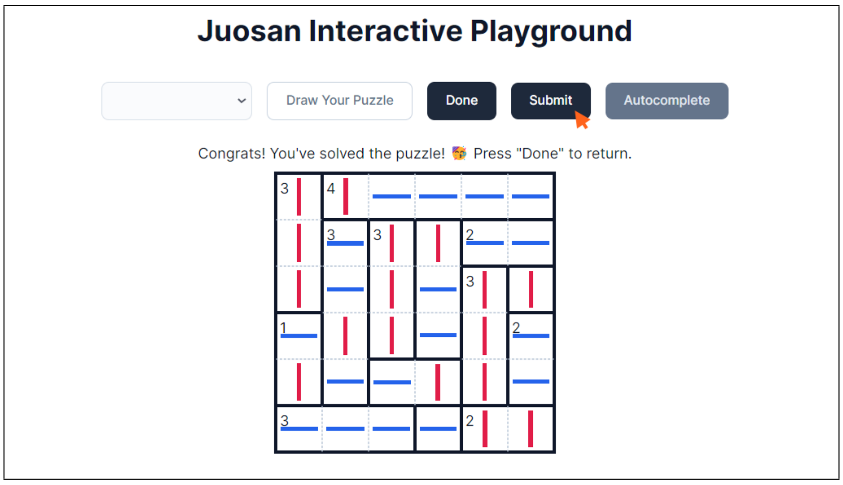


Figure 3: Choosing a puzzle level.



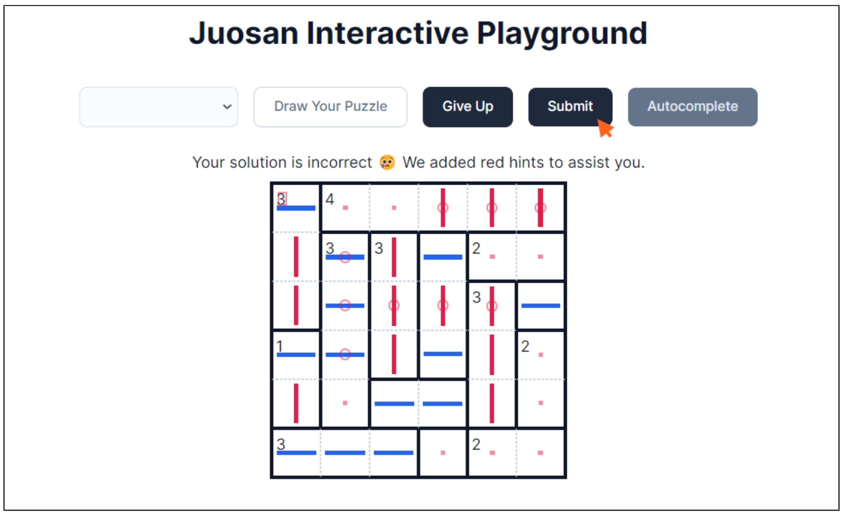
Click a cell multiple times to fill it with a symbol you want.

Figure 4: Filling a cell in the “Solve Yourself” mode.



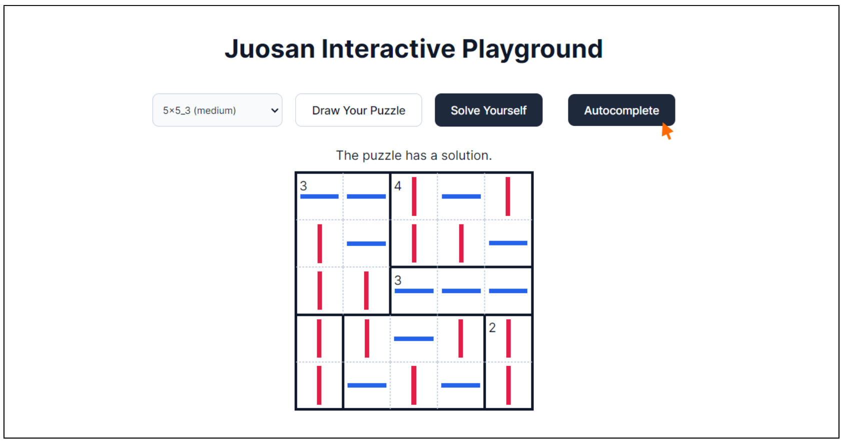
Congrats! You've solved the puzzle! 🥳 Press "Done" to return.

Figure 5: Notification after solving the puzzle correctly.



Your solution is incorrect 😢 We added red hints to assist you.

Figure 6: Hints are shown when the user fills in the puzzle incorrectly.



The puzzle has a solution.

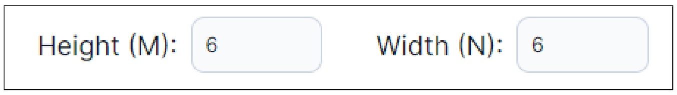
Figure 7: A solution is found after pressing "Autocomplete".

2.2 Drawing Mode

To enter the drawing mode, the players click the “Draw Your Puzzle” button. Afterward, some tools are provided for the players to draw their custom puzzles as follows.

2.2.1 Adjusting the Puzzle Size

At the top of the application interface, the players may find the M and N fields as depicted in Fig. 8. These allow them to adjust the size of the puzzle grid. They may enter the desired values for M and N to set the size accordingly. Please note that in this application the value of M and N are limited to a maximum of 25 to ensure reasonable solving times.

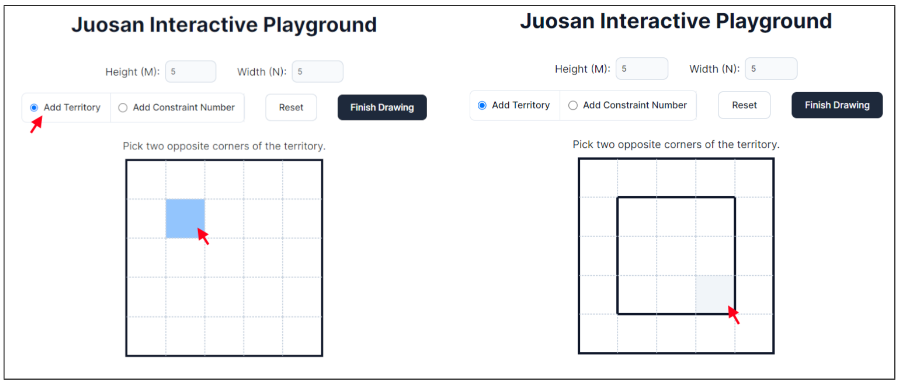


Height (M): 6 Width (N): 6

Figure 8: The M and N fields.

2.2.2 Adding Territories

To add a territory, the players click on two opposite corners of the desired rectangular area within the grid. This action will mark the selected cells as a territory as depicted in Fig. 9. The players may repeat this step to create additional territories as needed.



Juosan Interactive Playground

Height (M): 5 Width (N): 5

[✓ Add Territory] Add Constraint Number

Pick two opposite corners of the territory.

Juosan Interactive Playground

Height (M): 5 Width (N): 5

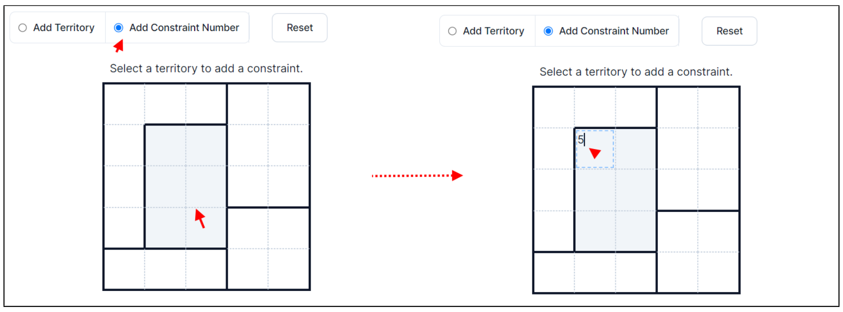
[✓ Add Territory] Add Constraint Number

Pick two opposite corners of the territory.

Figure 9: Using the “Add Territory” tool.

2.2.3 Adding Constraint Numbers to a Territory

The players first click on a territory within the grid to select it. Once selected, they can enter a number to add a numerical constraint to the territory as depicted in Fig. 10. This number indicates how many ▭ or ▯ symbols must be present within the territory.



Add Territory [✓ Add Constraint Number]

Select a territory to add a constraint.

Add Territory [✓ Add Constraint Number]

Select a territory to add a constraint.

Figure 10: Using the “Add Constraint Number” tool.

2.2.4 Resetting the Grid

If the players want to start over or clear the grid, they can use the "Reset" button as depicted in Fig. 11. Clicking on this button will clear the current puzzle and allow the players to begin anew.

Figure 11: The "Reset" button.

It is worth noting that this manual is also summarized in the "Guide" button located at the top right corner of our interface, as shown in Fig. 12. In case the players forget something, they can simply refer to it there for a quick recap. We wish the players to enjoy using the Juosan Interactive Playground application, and have fun cracking those challenging puzzles!

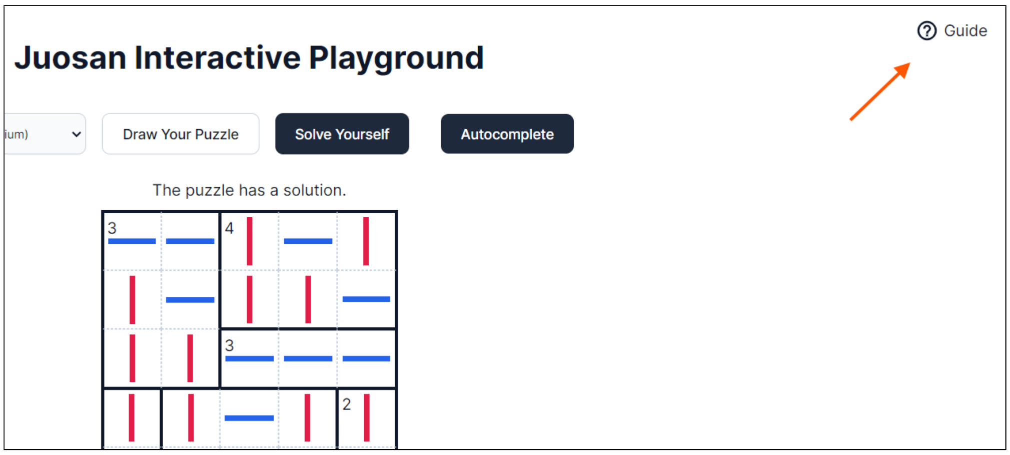


Figure 12: The "Guide" button.

3 Important Links

Below are some important links related to this solver app:

1. **The source code for this app**: https://github.com/tsaqifammar/juosan-interactive-playground.
2. **More about Juosan puzzles**: https://www.janko.at/Raetsel/Juosan/index.htm, https://www.nikoli.co.jp/en/puzzles/juosan/.

4 Appendix

4.1 Verifier Script

Below is the source code we utilize to determine the correctness of a filled-in Juosan puzzle.

```python
def validate_solution(m, n, r, N, R, S):
  # Check numerical constraints within territories
  bad_territories = []
  D = [0 for _ in range(r)]
  P = [0 for _ in range(r)]
  for i in range(m):
    for j in range(n):
      if S[i][j] == '-': D[R[i][j]] += 1
      else: P[R[i][j]] += 1

  for t in range(r):
    if N[t] != -1 and D[t] != N[t] and P[t] != N[t]:
      bad_territories.append(t)

  # Check for three consecutive cells
  bad_cells = set()
  for i in range(m - 2):
    for j in range(n):
      if S[i][j] == S[i+1][j] == S[i+2][j] == '-':
        bad_cells.update([(i,j), (i+1,j), (i+2,j)])

  for i in range(m):
    for j in range(n - 2):
      if S[i][j] == S[i][j+1] == S[i][j+2] == '|':
        bad_cells.update([(i,j), (i,j+1), (i,j+2)])

  empty_cells = [(i,j) for i in range(m) for j in range(n) if S[i][j] == '']
  return {
    "correct": len(bad_territories) == len(bad_cells)
      == len(empty_cells) == 0,
    "bad_territories": bad_territories,
    "bad_cells": list(bad_cells),
    "empty_cells": empty_cells,
  }
```

4.2 Solver Script

The following is the source code of the "solver" part of the program, implemented in Python. For the general case, it uses a SAT-based approach for solving the puzzle. It also handles some particular cases that are tractable uniquely with a more efficient approach. For the full source code, please refer to the aforementioned repository link.

```python
from pysat.solvers import Glucose3
from pysat.card import CardEnc

def distribute(v, formula):
  return [clause + [v] for clause in formula]

```

```python
class Solver:
  def __init__(self, m, n, r, N, R):
    self.m = m
    self.n = n
    self.r = r
    self.N = N
    self.R = R
    self.region_cells = {region: [] for region in range(r)}
    for i in range(m):
      for j in range(n):
        self.region_cells[R[i][j]].append(self.J(i,j))

  def J(self, i, j):
    return i*self.n + j + 1

  def solve_tractable_less_than_3(self):
    S = [[None for j in range(self.n)] for i in range(self.m)]

    WIN_SYMBOL = '-' if self.m <= 2 else '|'
    LOSE_SYMBOL = '|' if self.m <= 2 else '-'

    vis = [False for _ in range(self.r)]
    D = [0 for _ in range(self.r)]
    P = [0 for _ in range(self.r)]

    def valid(i, j):
      return i >= 0 and i < self.m and j >= 0 and j < self.n

    for i in range(self.m):
      for j in range(self.n):
        if not vis[self.R[i][j]]:
          S[i][j] = WIN_SYMBOL
          vis[self.R[i][j]] = True
        else:
          adj_x, adj_y = None, None
          for tmp_x, tmp_y in [[i - 1, j], [i, j - 1]]:
            if valid(tmp_x, tmp_y) and S[tmp_x][tmp_y] != None
             and self.R[i][j] == self.R[tmp_x][tmp_y]:
              adj_x, adj_y = tmp_x, tmp_y
              break

          assert(adj_x != None)
          S[i][j] = '-' if S[adj_x][adj_y] == '|' else '|'

        if S[i][j] == '-':
          D[self.R[i][j]] += 1
        else:
          P[self.R[i][j]] += 1

    to_change = [0 for _ in range(self.r)]
    for t in range(self.r):
      if self.N[t] == -1: continue
      win_count = D[t] if WIN_SYMBOL == '-' else P[t]
      lose_count = P[t] if WIN_SYMBOL == '-' else D[t]
      to_change[t] = self.N[t]-win_count if self.N[t] > win_count
        else lose_count-self.N[t]

    for i in range(self.m):
      for j in range(self.n):
        if to_change[self.R[i][j]] > 0 and S[i][j] == LOSE_SYMBOL:
          S[i][j] = WIN_SYMBOL
          to_change[self.R[i][j]] -= 1

    return {
      "is_solvable": True,
      "solution": S,
    }

  def solve_tractable_no_constraint(self):
    return {
      "is_solvable": True,
      "solution": [
        ['-' if (i + j) % 2 == 0 else '|' for j in range(self.n)]
        for i in range(self.m)
      ],
    }

  def solve_with_sat(self):
    solver = Glucose3()

    # Configure rule three vertically consecutive
    for i in range(self.m - 2):
      for j in range(self.n):
        solver.add_clause([-self.J(i,j), -self.J(i+1,j),
          -self.J(i+2,j)])

    # Configure rule three horizontally consecutive
    for i in range(self.m):
      for j in range(self.n - 2):
        solver.add_clause([self.J(i,j), self.J(i,j+1),
          self.J(i,j+2)])

    prev_top = self.J(self.m - 1, self.n - 1)

    # Configure territory rules
    for i in range(self.r):
      if self.N[i] != -1:
        h = prev_top + 1
        dashEqual = CardEnc.equals(
          lits=self.region_cells[i],
          bound=self.N[i],
          top_id=h
        )
        dashEqualWithH = distribute(h, dashEqual.clauses)
        solver.append_formula(
          formula=dashEqualWithH,
          no_return=False
        )

        barEqual = CardEnc.equals(
          lits=self.region_cells[i],
          bound=len(self.region_cells[i])-self.N[i],
          top_id=max(dashEqual.nv, h)
        )
        barEqualWithNegH = distribute(-h, barEqual.clauses)
        solver.append_formula(
          formula=barEqualWithNegH,
          no_return=False
        )

        prev_top = max(barEqual.nv, h)

    # Running the SAT solver
    sat = solver.solve()

    if sat:
      solution = solver.get_model()
      S = [[0 for j in range(self.n)] for i in range(self.m)]
      for i in range(self.m):
        for j in range(self.n):
          S[i][j] = "-" if self.J(i,j) in solution else "|"

      return {
        "is_solvable": True,
        "solution": S,
      }
    else:
      return {
        "is_solvable": False,
        "solution": [],
      }

  def solve(self):
    if self.m <= 2 or self.n <= 2:
      return self.solve_tractable_less_than_3()
    elif all(c == -1 for c in self.N):
      return self.solve_tractable_no_constraint()
    else:
      return self.solve_with_sat()
```