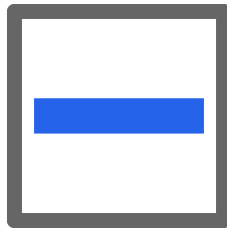


Juosan Interactive Playground: Design, Solve, Conquer!

A guide to using the Juosan Interactive Playground application.



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

Telkom University, Indonesia
School of Computing

15th July 2023

Contents

1	Introduction	3
1.1	Application Overview	3
1.2	About Juosan Puzzles	3
2	Functionalities	4
2.1	General Usage	4
2.2	Drawing Mode	6
3	Important Links	7
4	Appendix	7

1 Introduction

Thank you for using the Juosan Interactive Playground app! This manual will guide you through the application's features and usage to design and solve Juosan puzzles effectively.



1.1 Application Overview





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

1. **Puzzle Selection:** Choose from a selection of example puzzles with varying difficulty levels. These puzzles not only provide challenges for users but also serve as demonstrations of the app's solver, showcasing its ability to solve even large and complex puzzles.
2. **Create Custom Puzzles:** Utilize the “Draw Your Puzzle” feature to create your own Juosan puzzles. 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 you need assistance, the app offers an “Autocomplete” button that employs a built-in solver to complete the puzzle for you.

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

1.2 About Juosan Puzzles

Juosan puzzles, created by Nikoli (the mastermind behind Sudoku), provide an engaging pencil-and-paper challenge. They have been proven to be NP-complete, meaning their complexity increases as the puzzle size grows. Even computers find these puzzles difficult to solve. In a Juosan puzzle, you are presented 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 Figure 1.

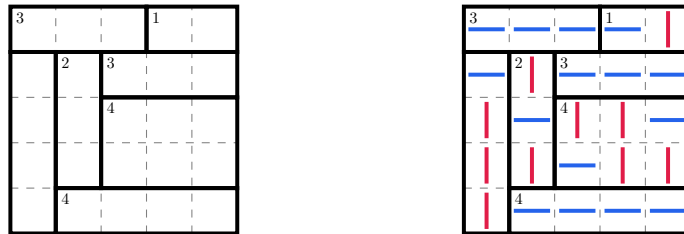


Figure 1: An example of a Juosan puzzle and its solution.

2 Functionalities

2.1 General Usage

This section discusses the features and tools that are available in the application.

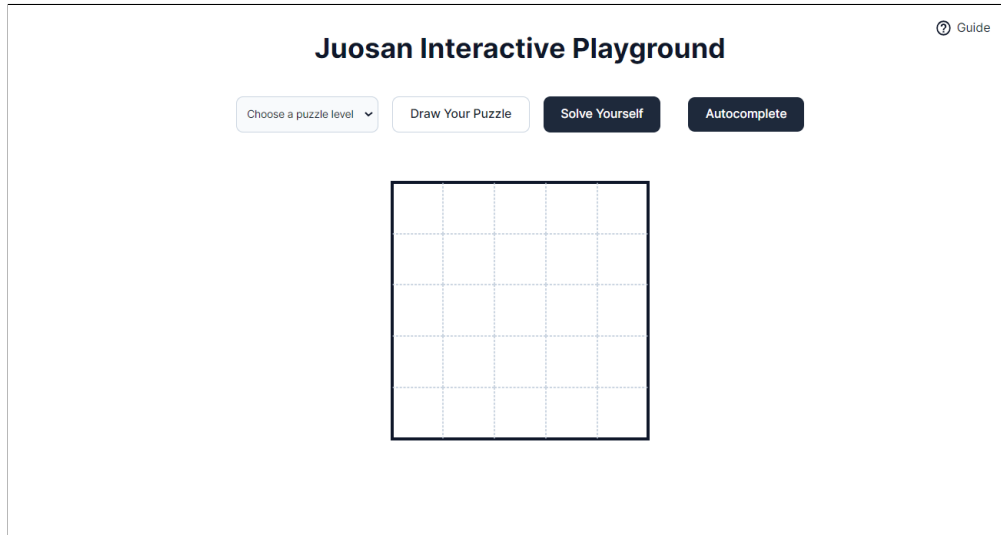


Figure 2: The app interface.

To get started, you 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 you 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 sourced the majority of these puzzles from [Janko](#), a renowned puzzle collection platform. Alternatively, you can draw your own Juosan puzzle by clicking the “Draw Your Puzzle” button. We discuss the tools in the drawing mode in section 2.2.

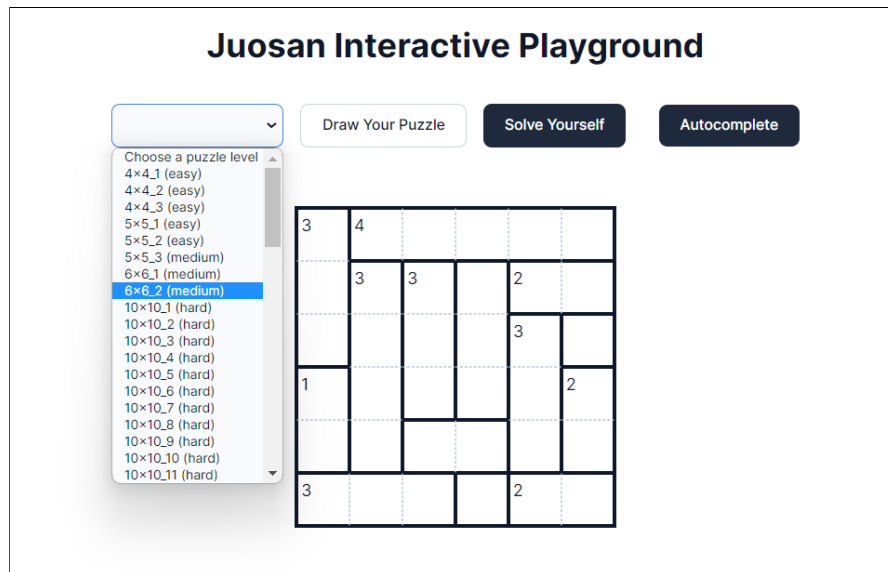


Figure 3: Choosing a puzzle level.

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

1. If you want to solve the puzzle yourself, click the “Solve Yourself” button. You can fill a cell by clicking on it multiple times to cycle through the available symbols. In case you encounter difficulties and decide to give up, you can press the “Give Up” button. Once you have finished solving, press the “Submit” button to check your solution. The app will give you some hints if you fill in the puzzle incorrectly. Please see figs. 4 to 6 for illustrations of the “Solve Yourself” mode.
2. If you prefer to use our solver, click the “Autocomplete” button. The app’s solver will attempt to complete the puzzle for you. Additionally, the app will display the status of the puzzle, indicating whether it has a solution or not.

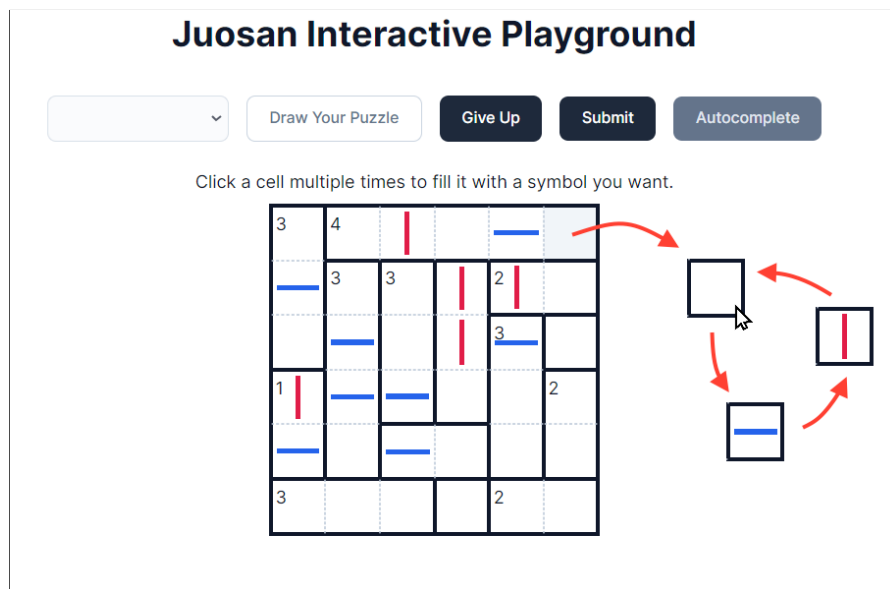


Figure 4: Filling a cell in the Solve Yourself mode.

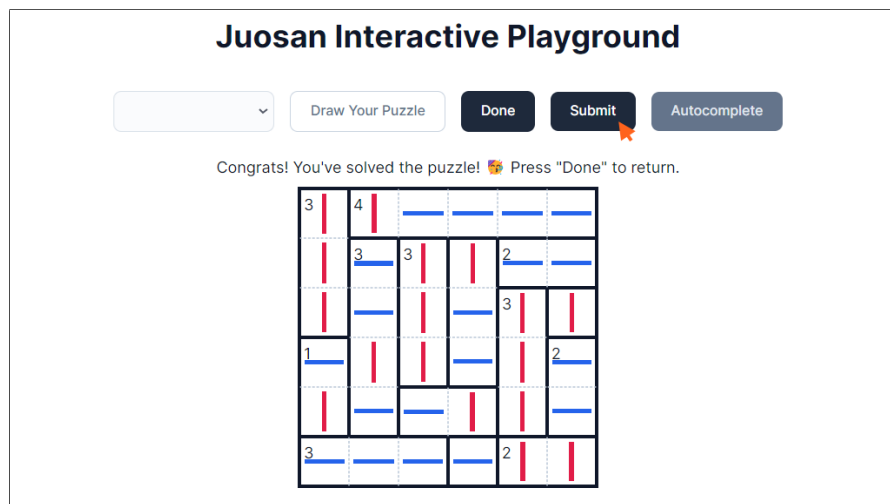


Figure 5: After solving the puzzle correctly.

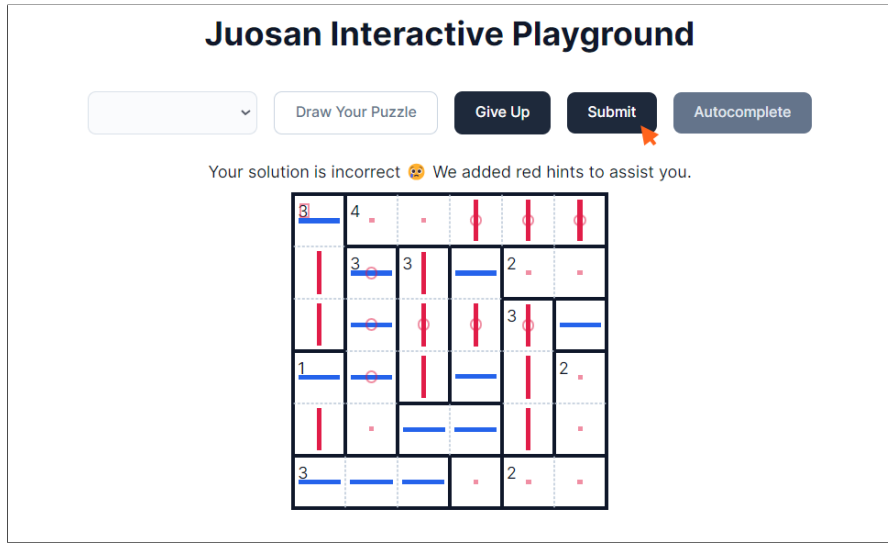


Figure 6: Hints shown when the user filled in the puzzle incorrectly.

2.2 Drawing Mode

To enter the drawing mode, click the “Draw Your Puzzle” button. Then, some tools are provided for you to draw your custom puzzle as follows:

Adjusting the Puzzle Size: At the top of the app interface, you’ll find the M and N fields. These allow you to adjust the size of the puzzle grid. Enter the desired values for M and N to set the size accordingly. Please note that in this app, we limit M and N to a maximum of 25 to ensure reasonable solving times.

Height (M):

Width (N):

Figure 7: M and N fields.

Adding Territories: To add a territory, click on two opposite corners of the desired rectangular area within the grid. This action will mark the selected cells as a territory. Repeat this step to create additional territories as needed.

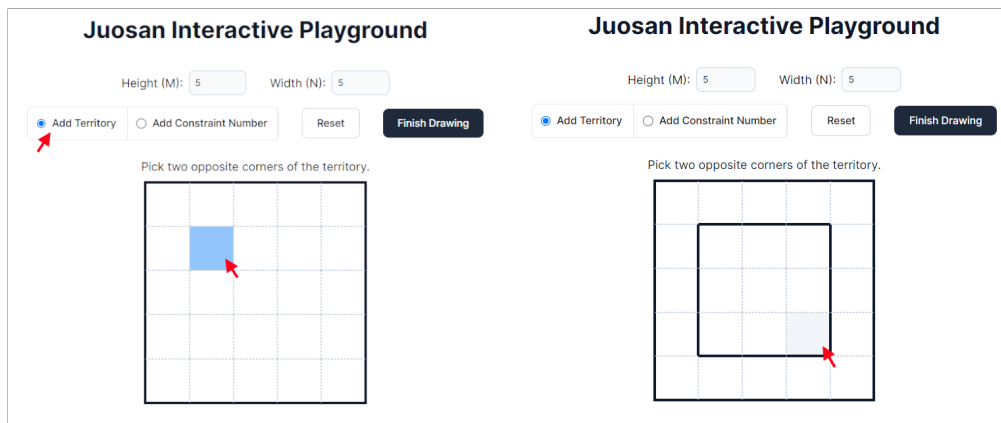




Figure 8: Using the “Add Territory” tool.

Adding Constraint Numbers to a Territory: Click on a territory within the grid to select it. Once selected, you can enter a number to add a constraint to the territory. This number indicates how many  or  symbols must be present within the territory.

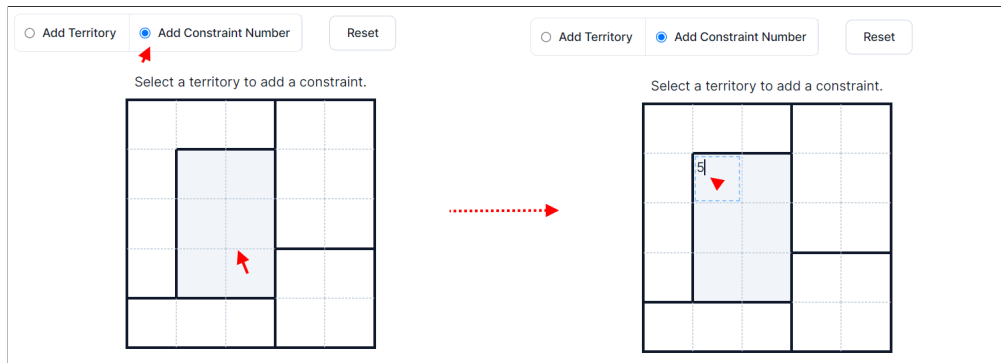


Figure 9: Using the “Add Constraint Number” tool.

Resetting the grid: If you want to start over or clear the grid, you can use the “Reset” button. Clicking on this button will clear the current puzzle and allow you to begin anew.

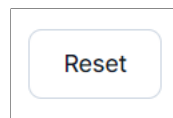


Figure 10: “Reset” button.

It is worth noting that this manual is also summarized in the “Guide” button located at the top right corner of our interface. In case you forget something, you can simply refer to it there for a quick recap. Enjoy using the Juosan Interactive Playground app, and have fun cracking those challenging puzzles!

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

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 the aforementioned repository link.

```
1 from pysat.solvers import Glucose3
2 from pysat.card import CardEnc
3
```

```

4 def distribute(v, formula):
5     return [clause + [v] for clause in formula]
6
7 class Solver:
8     def __init__(self, m, n, r, N, R):
9         self.m = m
10        self.n = n
11        self.r = r
12        self.N = N
13        self.R = R
14        self.region_cells = {region: [] for region in range(r)}
15        for i in range(m):
16            for j in range(n):
17                self.region_cells[R[i][j]].append(self.J(i,j))
18
19    def J(self, i, j):
20        return i*self.n + j + 1
21
22    def solve_tractable_less_than_3(self):
23        S = [[None for j in range(self.n)] for i in range(self.m)]
24
25        WIN_SYMBOL = '-' if self.m <= 2 else '|'
26        LOSE_SYMBOL = '|' if self.m <= 2 else '-'
27
28        vis = [False for _ in range(self.r)]
29        D = [0 for _ in range(self.r)]
30        P = [0 for _ in range(self.r)]
31
32    def valid(i, j):
33        return i >= 0 and i < self.m and j >= 0 and j < self.n
34
35    for i in range(self.m):
36        for j in range(self.n):
37            if not vis[self.R[i][j]]:
38                S[i][j] = WIN_SYMBOL
39                vis[self.R[i][j]] = True
40            else:
41                adj_x, adj_y = None, None
42                for tmp_x, tmp_y in [[i - 1, j], [i, j - 1]]:
43                    if valid(tmp_x, tmp_y) and S[tmp_x][tmp_y] != None
44                        and self.R[i][j] == self.R[tmp_x][tmp_y]:
45                        adj_x, adj_y = tmp_x, tmp_y
46                        break
47
48                assert(adj_x != None)
49                S[i][j] = '-' if S[adj_x][adj_y] == '|' else '|'
50
51            if S[i][j] == '-':
52                D[self.R[i][j]] += 1
53            else:
54                P[self.R[i][j]] += 1
55
56    to_change = [0 for _ in range(self.r)]

```



```

57     for t in range(self.r):
58         if self.N[t] == -1: continue
59         win_count = D[t] if WIN_SYMBOL == '-' else P[t]
60         lose_count = P[t] if WIN_SYMBOL == '-' else D[t]
61         to_change[t] = self.N[t]-win_count if self.N[t] > win_count
62             else lose_count-self.N[t]
63
64     for i in range(self.m):
65         for j in range(self.n):
66             if to_change[self.R[i][j]] > 0 and S[i][j] == LOSE_SYMBOL:
67                 S[i][j] = WIN_SYMBOL
68                 to_change[self.R[i][j]] -= 1
69
70     return {
71         "is_solvable": True,
72         "solution": S,
73     }
74
75 def solve_tractable_no_constraint(self):
76     return {
77         "is_solvable": True,
78         "solution": [
79             ['-'] if (i + j) % 2 == 0 else '|' for j in range(self.n)]
80             for i in range(self.m)
81         ],
82     }
83
84 def solve_with_sat(self):
85     solver = Glucose3()
86
87     # Configure rule three vertically consecutive
88     for i in range(self.m - 2):
89         for j in range(self.n):
90             solver.add_clause([-self.J(i,j), -self.J(i+1,j),
91                 -self.J(i+2,j)])
92
93     # Configure rule three horizontally consecutive
94     for i in range(self.m):
95         for j in range(self.n - 2):
96             solver.add_clause([self.J(i,j), self.J(i,j+1),
97                 self.J(i,j+2)])
98
99     prev_top = self.J(self.m - 1, self.n - 1)
100
101     # Configure territory rules
102     for i in range(self.r):
103         if self.N[i] != -1:
104             h = prev_top + 1
105             dashEqual = CardEnc.equals(
106                 lits=self.region_cells[i],
107                 bound=self.N[i],
108                 top_id=h
109             )

```

```

110     dashEqualWithH = distribute(h, dashEqual.clauses)
111     solver.append_formula(
112         formula=dashEqualWithH,
113         no_return=False
114     )
115
116     barEqual = CardEnc.equals(
117         lits=self.region_cells[i],
118         bound=len(self.region_cells[i])-self.N[i],
119         top_id=max(dashEqual.nv, h)
120     )
121     barEqualWithNegH = distribute(-h, barEqual.clauses)
122     solver.append_formula(
123         formula=barEqualWithNegH,
124         no_return=False
125     )
126
127     prev_top = max(barEqual.nv, h)
128
129     # Running the SAT solver
130     sat = solver.solve()
131
132     if sat:
133         solution = solver.get_model()
134         S = [[0 for j in range(self.n)] for i in range(self.m)]
135         for i in range(self.m):
136             for j in range(self.n):
137                 S[i][j] = "-" if self.J(i,j) in solution else "|"
138
139         return {
140             "is_solvable": True,
141             "solution": S,
142         }
143     else:
144         return {
145             "is_solvable": False,
146             "solution": [],
147         }
148
149     def solve(self):
150         if self.m <= 2 or self.n <= 2:
151             return self.solve_tractable_less_than_3()
152         elif all(c == -1 for c in self.N):
153             return self.solve_tractable_no_constraint()
154         else:
155             return self.solve_with_sat()

```

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

```

1 def validate_solution(m, n, r, N, R, S):
2     # Check numerical constraints within territories
3     bad_territories = []
4     D = [0 for _ in range(r)]
5     P = [0 for _ in range(r)]

```

```

6   for i in range(m):
7       for j in range(n):
8           if S[i][j] == '-': D[R[i][j]] += 1
9           else: P[R[i][j]] += 1
10
11  for t in range(r):
12      if N[t] != -1 and D[t] != N[t] and P[t] != N[t]:
13          bad_territories.append(t)
14
15  # Check for three consecutive cells
16  bad_cells = set()
17  for i in range(m - 2):
18      for j in range(n):
19          if S[i][j] == S[i+1][j] == S[i+2][j] == '-':
20              bad_cells.update([(i,j), (i+1,j), (i+2,j)])
21
22  for i in range(m):
23      for j in range(n - 2):
24          if S[i][j] == S[i][j+1] == S[i][j+2] == '|':
25              bad_cells.update([(i,j), (i,j+1), (i,j+2)])
26
27  empty_cells = [(i,j) for i in range(m) for j in range(n) if S[i][j] == '']
28  return {
29      "correct": len(bad_territories) == len(bad_cells)
30      == len(empty_cells) == 0,
31      "bad_territories": bad_territories,
32      "bad_cells": list(bad_cells),
33      "empty_cells": empty_cells,
34  }

```