Interactive Suguru Playground 1.0
User Manual

Butrahandisya
Muhammad Arzaki (editor)
July 5, 2023

Contents

1 Introduction **2**
1.1 Interactive Playground Overview 2

2 Functionalities **2**
2.1 Play Mode 3
2.2 Creative Mode 3
2.3 Level Selection 3
2.4 Adjustable Grid Size 4
2.5 Draw Region 4
2.6 Fill Cell 5
2.7 Validate Configuration 5
2.8 Autocomplete 6
2.9 Clear Regions 7
2.10 Empty Cells 7
2.11 Guide 7

A SAT-based Solver Source Code **8**

B Verifier Source Code **10**

C Important Link **11**

1 Introduction

1.1 Interactive Playground Overview

Interactive Suguru Playground 1.0 is a web application designed to provide puzzle enthusiasts with a platform to construct and solve Suguru grids. Its user-friendly interface and powerful features offer an engaging and interactive experience for puzzle lovers of all skill levels. Whether you are a beginner or an experienced puzzler, Interactive Suguru Playground 1.0 is a good destination for hours of brain-teasing fun. Figure 1 depicts a screenshot of the main page of the Interactive Suguru Playground 1.0.

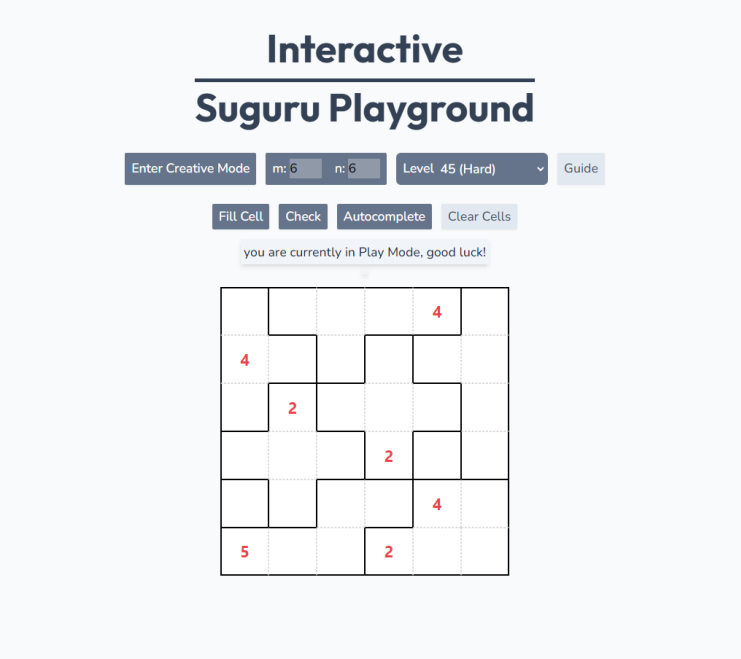

Figure 1: Main page of the Interactive Suguru Playground 1.0.

To explore further or access the complete source code of the Suguru Solver, please visit: `https://github.com/abcqwq/suguru-solver-app`.

2 Functionalities

This section provides an in-depth exploration of key functionalities provided in the Suguru Playground.

2.1 Play Mode

The Play Mode allows the users to play the Suguru puzzle independently. See Figure 2 for illustration.

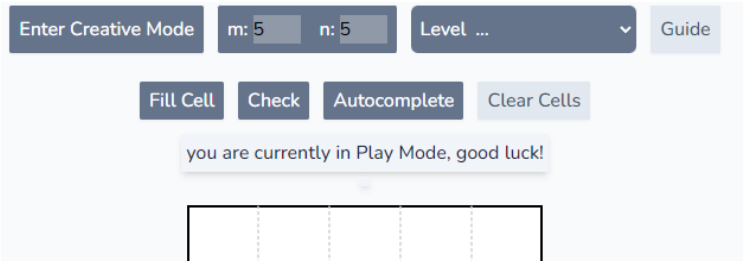

Figure 2: Layout of the playground when the user is in Play Mode.

2.2 Creative Mode

The Creative Mode allows the users to create a custom Suguru puzzle grid. For a better experience, the users are advised to use a desktop-based browser. See Figure 3 for illustration.

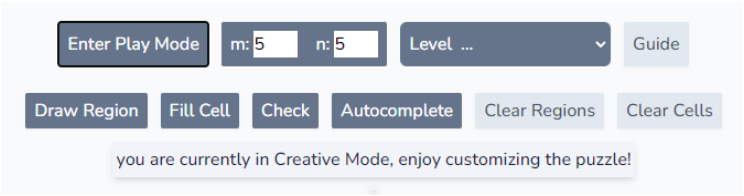

Figure 3: Layout of the playground when the user is in Creative Mode.

2.3 Level Selection

The Level Selection drop-down allows the user to select previously prepared Suguru puzzles ready to be solved. These puzzles are mostly obtained from the curated instances by Otto Janko, visit https://www.janko.at/Raetsel/Suguru/index.htm for details. All instances with medium or harder difficulty are guaranteed to have exactly one solution. See Figure 4 for illustration.

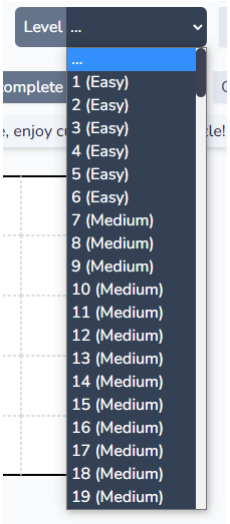

Figure 4: Level Selection drop-down.

2.4 Adjustable Grid Size

These Adjustable Grid Size fields allow the users to adjust the row and column size of the grid, providing flexibility in customizing the puzzle layout. The maximum grid size may depend on the software or machine specification used to implement this interactive application. See Figure 5 for illustration.

Figure 5: Grid size input fields.

2.5 Draw Region

The user must enter grid-brushing mode by pressing the Draw Region button to create regions within the grid. Once in this mode, the user can draw regions directly on the grid. See Figure 6 for illustration.

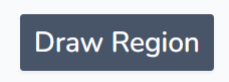

Figure 6: Draw Region button.

2.6 Fill Cell

To fill a cell in the grid, the user needs to be in cell-filling mode, which can be activated by pressing the Fill Cell button. Once in this mode, the user can input an integer value, limited to three digits, into any desired cell. See Figure 7 for illustration.

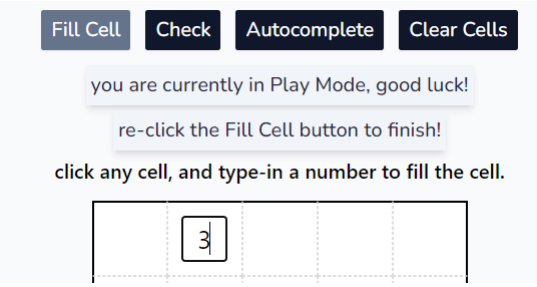

Figure 7: Fill Cell button.

2.7 Validate Configuration

By clicking the Check button, the user can verify the current puzzle configuration and check if it complies with the rules and constraints of the Suguru puzzle. See Figure 8. Pressing the Check button can lead to two possible outcomes: a message indicating that the current configuration fails to meet the constraint rules or confirming that the configuration is correct.

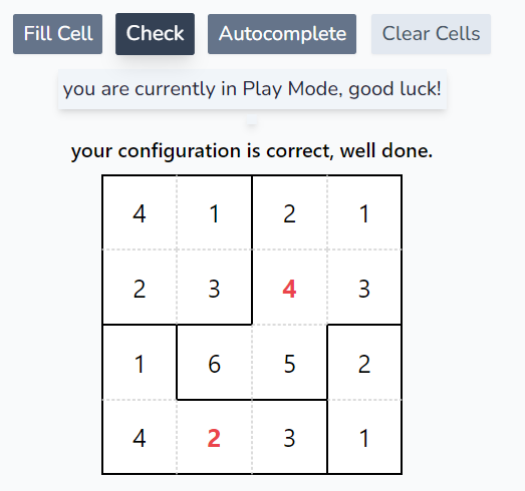

Figure 8: Validate button.

2.8 Autocomplete

When the user clicks the Autocomplete button, the Suguru Playground promptly solves the puzzle in the grid and provides a solution or notifies the user if the puzzle is determined to be unsolvable (i.e., the puzzle has no solution). See Figure 9 and Figure 10 for illustration. Behind the scenes, the Suguru Playground utilizes a SAT-based approach to solve the puzzle. This approach is empirically stable for solving a general $m \times n$ Suguru puzzle.

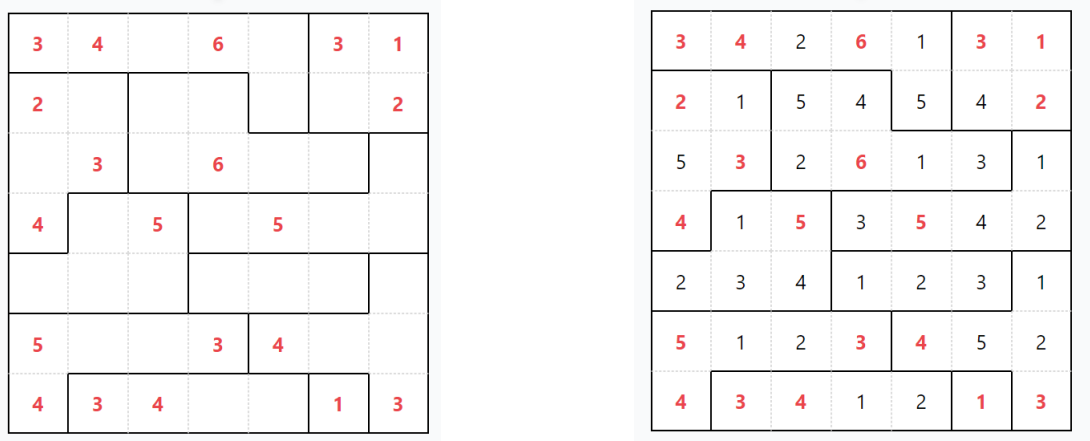

Figure 9: All cells in the grid are filled upon pressing the Autocomplete button.

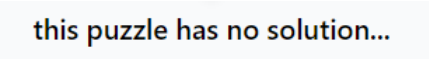

Figure 10: Message notifying the user that the puzzle has no solution.

2.9 Clear Regions

The Clear Regions button provides a convenient way to remove all previously drawn regions on the grid. By clicking this button, users can easily start fresh or make adjustments to the puzzle structure.

2.10 Empty Cells

The Clear Cells button serves the purpose of emptying all cells from the grid with a single click.

2.11 Guide

The playground offers a simplified manual that contains a condensed version of the instructions. See Figure 11 for illustration.

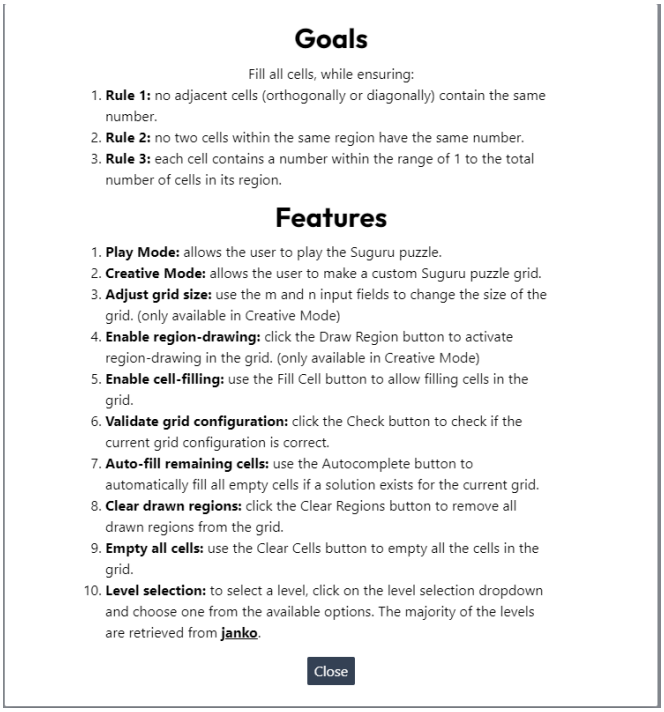

Figure 11: Modal showing the condensed version of this manual book.

A SAT-based Solver Source Code

```
from pysat.solvers import Glucose3
from pysat.formula import CNF

def g(i, j, v):
    return (i*n + j)*s_max + v

# Rule 1: All cell must be filled with exactly a single value
in range 1 to s[region[i][j]].
def phi_1():
    phi_1 = CNF()
    for i in range(m):
        for j in range(n):
            for v in range(1, s[region[i][j]]):
                for v_ in range(v+1, s[region[i][j]]+1):
                    phi_1.append([-g(i, j, v), -g(i, j, v_)])
    return phi_1

# Rule 2: For each region k, all numbers in the set {1, 2, ..., s[k]}
must exist in any cell of region k.
def phi_2():
    phi_2 = CNF()
    for k in range(len(s)):
        for v in range(1, s[k]+1):
            clause = []
            for i in range(m):
                for j in range(n):
                    if region[i][j] == k:
                        clause.append(g(i, j, v))
            phi_2.append(clause)
    return phi_2

def inside_grid(i, j):
    return i >= 0 and i < m and j >= 0 and j < n

# Rule 3: No two adjacent cells, either orthogonally or diagonally,
can share a value.
def phi_3():
    dir = [[0, 1], [1, -1], [1, 0], [1, 1]]
    phi_3 = CNF()

    for i in range(m):
        for j in range(n):
```

```
            for [dx, dy] in dir:
                if inside_grid(i + dx, j + dy):
                    for v in range(1, min(s[region[i][j]],
                    s[region[i+dx][j+dy]]) + 1):
                        phi_3.append([-g(i, j, v), -g(i + dx, j + dy, v)])
    return phi_3

def hint_constraint():
    constraint = CNF()
    for i in range(m):
        for j in range(n):
            if hint[i][j] > 0:
                constraint.append([g(i, j, hint[i][j])])
    return constraint

def clean_region():
    queue = []
    is_visited = [[False for c in range(n)] for r in range(m)]
    region_counter = 1

    def bfs(reg):
        while len(queue) > 0:
            [i, j] = queue.pop()
            is_visited[i][j] = True
            region[i][j] = region_counter
            for [dx, dy] in [[1, 0], [-1, 0], [0, 1], [0, -1]]:
                if inside_grid(i+dx, j+dy) and region[i+dx][j+dy] == reg
                and not is_visited[i+dx][j+dy]:
                    queue.append([i+dx, j+dy])

    for i in range(m):
        for j in range(n):
            if not is_visited[i][j]:
                queue.append([i, j])
                bfs(region[i][j])
                region_counter += 1

def retrieve_data(config):
    global m, n, hint, region, s, s_max, R
    m = config.get('m')
    n = config.get('n')
    hint = config.get('hint')
    region = config.get('region')
    clean_region()
```

```
    s = [0 for _ in range(max(max(row) for row in region) + 1)]
    for row in region:
        for col in row:
            s[col] += 1
    s_max = max(s)
    R = len(s)-1

def solve(config):
    retrieve_data(config)
    if s_max > 50:
        return {'solve_status': 'unsolved', 'message': 'to use
        the autocomplete feature, a region can only contain up to 50 cells.'}
    solver = Glucose3()
    solver.append_formula(phi_1().clauses)
    solver.append_formula(phi_2().clauses)
    solver.append_formula(phi_3().clauses)
    solver.append_formula(hint_constraint().clauses)

    hint = [[-1 for j in range(n)] for i in range(m)]
    solve_status = 'unsolvable'
    message = 'this puzzle has no solution...'

    if solver.solve():
        solve_status = 'solved'
        message = 'success'
        solution = solver.get_model()
        for i in range(m):
            for j in range(n):
                for v in range(1, s[region[i][j]] + 1):
                    if g(i, j, v) in solution:
                        hint[i][j] = v
    return {'hint': hint, 'solve_status': solve_status, 'message': message}
```

B Verifier Source Code

```
def validate(config):
    retrieve_data(config)

    messages = []
    adjacent_error_message = lambda i, j: f'cell ({i+1}, {j+1})
    shares a number with its adjacent cell(s).'
    region_error_message = lambda i, j: f'cell ({i+1}, {j+1})
    shares a number with cell(s) in within the same region.'
```

```
    range_error_message = lambda i, j, sk: f'cell ({i+1}, {j+1})
    has a number exceeding {sk}.'

    cv = [[0 for _ in range(s[k]+1)] for k in range(len(s))]
    for i in range(m):
        for j in range(n):
            if hint[i][j] <= s[region[i][j]]:
                cv[region[i][j]][hint[i][j]] += 1


    def validate_cell(i, j):
        dxdy = [[0, 1], [0, -1], [1, 0], [-1, 0],
        [1, 1], [-1, 1], [1, -1], [-1, -1]]
        if hint[i][j] < 1 or hint[i][j] > s[region[i][j]]:
            messages.append(range_error_message(i, j, s[region[i][j]]))

        has_adjacent_cell = False
        for [dx, dy] in dxdy:
            if inside_grid(i+dx, j+dy):
                has_adjacent_cell = has_adjacent_cell or
                hint[i][j] == hint[i+dx][j+dy]
        if has_adjacent_cell:
            messages.append(adjacent_error_message(i, j))

        if hint[i][j] <= s[region[i][j]] and cv[region[i][j]][hint[i][j]] > 1:
            messages.append(region_error_message(i, j))

    for i in range(m):
        for j in range(n):
            validate_cell(i, j)

    return {'hint': hint, 'solve_status': 'validation_success'
    if len(messages) == 0 else 'unsolved', 'message': f'there
    are {len(messages)} constraint(s) not met'
    if len(messages) > 0 else 'your configuration is correct, well done.'}
```

C Important Link

To explore further or access the complete source code of the Suguru Playground, please visit: https://github.com/abcqwq/suguru-solver-app.