Path Puzzles App User Manual

Joshua Erlangga Sakti

June 24, 2023

Contents

1 Introduction

1.1 Path puzzles Overview

Path puzzle is a NP-complete puzzle introduced by Roderick Kimball in 2013. The puzzle is played on a rectangular grid of cells with two openings on the edge and constraint numbers on some of the rows and columns. The objective is to find a solution by creating a single continuous line, or path, that connects the two openings while satisfying the given constraints. Each cell in the grid can only be passed through at most once, and the number of cells passing through a specific row or column must equal the constraint numbers.

1.2 App Overview

The Path Puzzles App is a tool designed to assist you in exploring Path puzzles. The app simplifies the process of constructing and solving these puzzles by providing a user-friendly interface and useful features. This user manual will guide you through the process of using the app, covering everything from puzzle construction to challenging yourself with solving it on your own. The layout of the app interface is depicted in Figure 1.

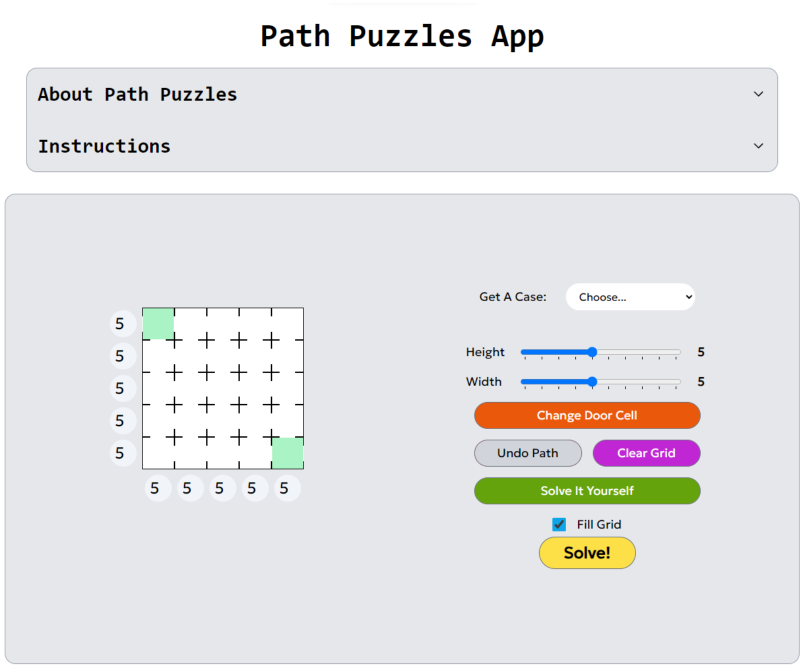


Figure 1: The layout of Path puzzles App.

1.3 Source Code

The application is built using the Django framework. The puzzle solver in the app is implemented in Python using the PySAT library, while the puzzle solution verifier is implemented in Javascript. The source code for the project can be found on `https://github.com/joshuagatizz/path-puzzles-app`.

2 Functionalities

2.1 Overview

The upper section of the page contains information about Path puzzles and provides instructions for utilizing the app. You have the option to expand the sections if you wish to read the details. The app has two primary functionalities: puzzle creation and puzzle solving. The middle of the page has the puzzle grid and a variety of different inputs. You have the flexibility to create Path puzzles using the diverse inputs provided. When it comes to solving the puzzles, you have the option to utilize our implemented solver or challenge yourself to solve them independently—the choice is yours! The puzzles available for solving can be selected from our pre-existing cases, or if you prefer, you can create your own Path puzzles from scratch.

2.2 Getting A Case

If you're up for a challenge or interested in testing the difficulty of Path puzzles, you can select a case from the "Get A Case" dropdown menu, as shown in Figure 2. This will automatically map the chosen case onto the grid for you. Most of these cases are taken from the font-pathpuzzles GitHub repository, created by Erik Demaine. The cases are conveniently divided into three categories based on their difficulty: easy, medium, and hard. Feel free to choose the category that suits your preference!

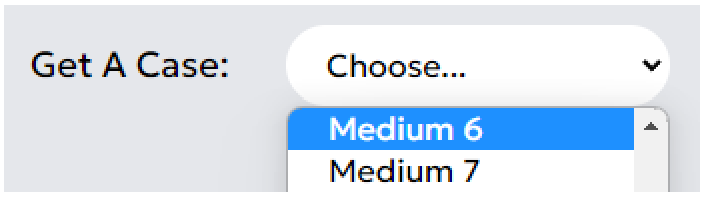


Figure 2: The "Get A Case" dropdown.

2.3 Adjusting the Puzzle Grid

To change the dimension of the puzzle grid, use the “Width” and “Height” sliders as in Figure 3. Slide them to increase or decrease the grid size according to your preference. The minimum dimension for both width and height is 1 while the maximum is 10. Note that the 1×1 grid is invalid since one of the puzzle’s rules is that the door cells must be distinct, which is impossible in a 1×1 grid.

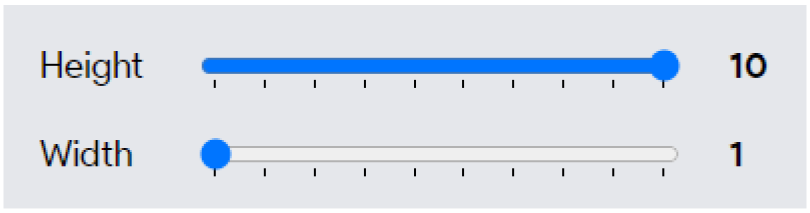
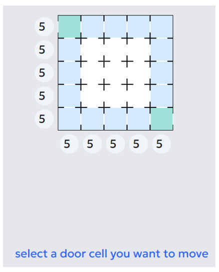


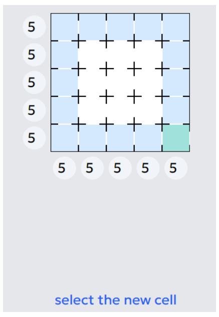
Figure 3: The “Width” and “Height” sliders.

2.4 Changing Door Cells

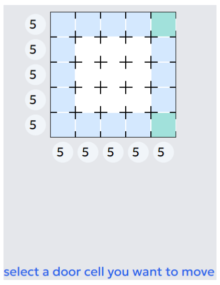
The door cells are the starting and ending points of the path in the puzzle grid. They are marked with a green color. Press the “Change Door Cell” button to move the door cells. It will allow you to select and move the door cells to different cells on the grid. After pressing the button, the edge cells of the grid will change color to blue as in Figure 4, highlighting the possible cells to be a door cell. When you are done moving the door cells, press the button again to confirm.



(a) “Change Door Cell” button clicked.



(b) A door cell is chosen.



(c) The new cell is chosen.

Figure 4: The process of moving a door cell.

2.5 Modifying Clue Numbers

Each row and column in the puzzle grid may have clue numbers on the sides as in Figure 5, which indicate the number of cells that must be passed through that particular row or column. To modify the clue numbers, directly change their values. You can enter specific numbers or leave the value empty if you want no restrictions on a row or column.

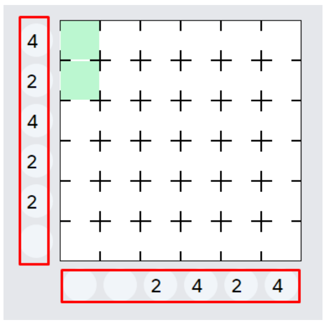


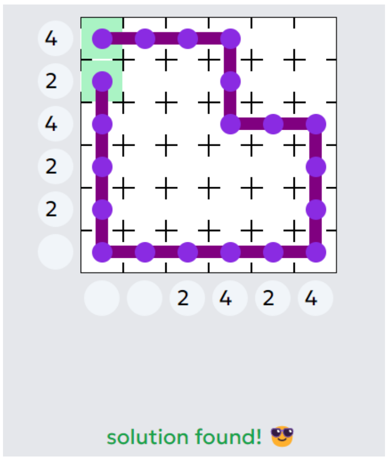
Figure 5: The clue numbers for each row and column.

2.6 Solving the Puzzle

Press the “Solve!” button to instruct the solver to find the solution for the puzzle. If a solution exists, it will be drawn on the grid. The solver will create a path connecting the two door cells, ensuring each cell is passed at most once. It will also satisfy the clue numbers for each row and column. If no solution is found, you may need to adjust the clue numbers or door cell positions to make the puzzle solvable. The output from the “Solve!” button is depicted in Figure 6.

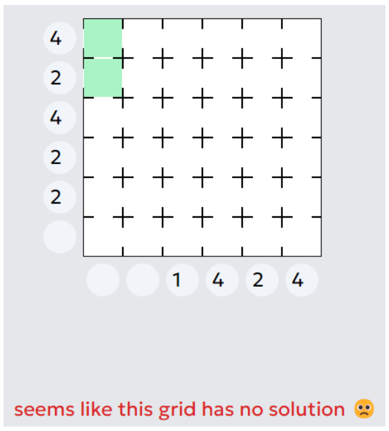
Furthermore, you have the option to uncheck the “Fill Grid” checkbox, which will transform the “Solve!” button into a “Check!” button. Despite the change in name, the button serves the same purpose of finding the solution. However, this time the solver will not draw the grid with the solution. Instead, it will only provide you with a verdict indicating whether a solution exists or not.

This feature proves particularly useful when you prefer to solve a grid by yourself but remain uncertain about the presence of a valid solution. By using the “Check!” button, you can determine the existence of a solution without being directly provided with the solution itself. The output from the “Check!” button is depicted in Figure 7.



solution found! 😎

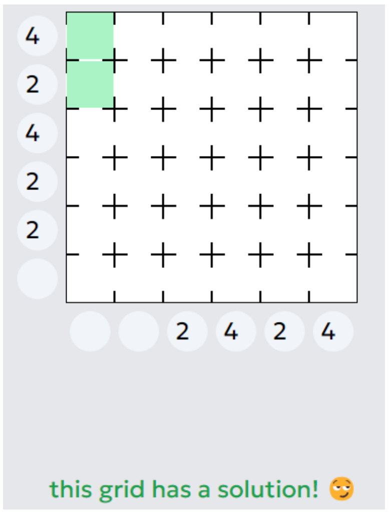
(a) A solution exists.



seems like this grid has no solution 🙁

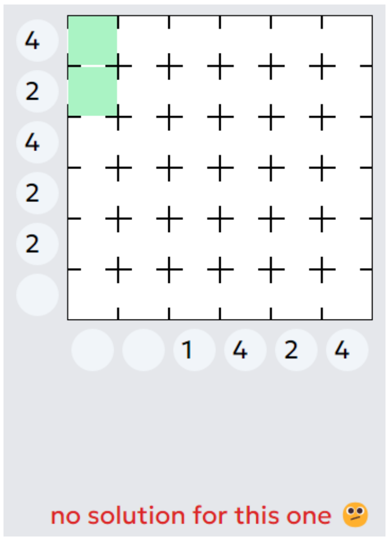
(b) There is no solution.

Figure 6: The output from pressing the “Solve!” button.



this grid has a solution! 😌

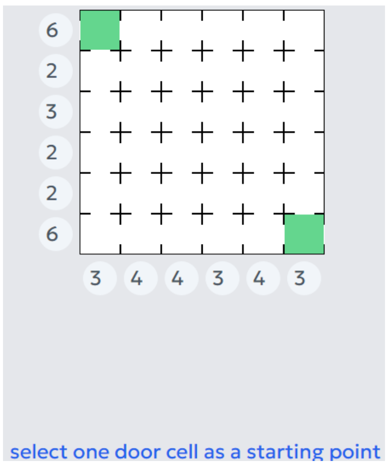
(a) A solution exists.



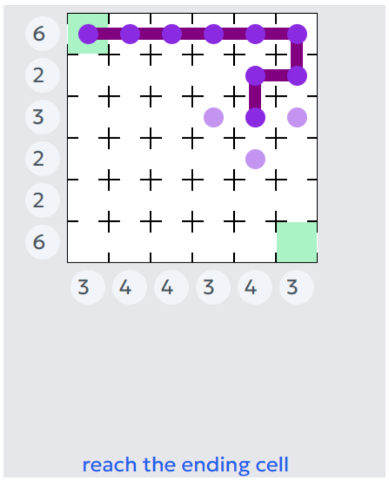
no solution for this one 🫤

(b) There is no solution.

Figure 7: The output from pressing the “Check!” button.



(a) Choosing the starting cell.



(b) Constructing the path to the ending cell.

Figure 8: The traversal process in creating the solution for a Path puzzle.

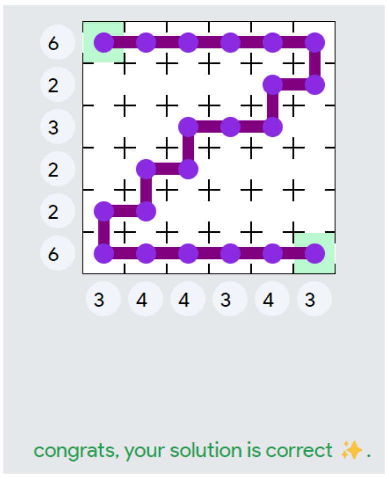
2.7 Solving the Puzzle Yourself

By pressing the “Solve It Yourself” button, you can try to solve the puzzle by yourself. To begin solving the puzzle, click on one of the door cells marked with a green color. Then, continue to its valid adjacent cells marked by semi-opaque purple circles, following the rules of the path puzzle. The process is depicted in Figure 8. Once you reach the other door cell, you will receive a verdict indicating whether your solution is correct as depicted in Figure 9. In case your solution is incorrect, certain clue numbers will be highlighted with a blinking red effect. This visual indication serves to highlight the unsatisfied clue numbers, allowing you to pinpoint the mistake in your current solution more easily.

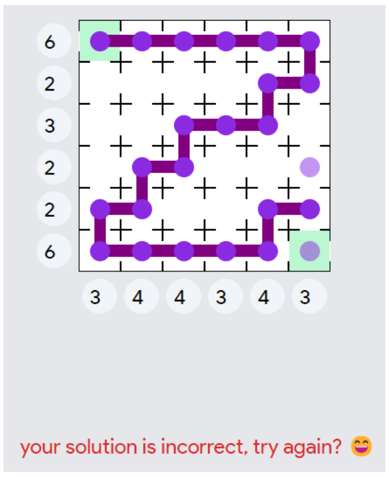
2.8 Undo and Clear

If you make a mistake while constructing the path, you can backtrack one cell by pressing the “Undo Path” button. This feature will remove the last cell you traversed, allowing you to correct your solution seamlessly. To restart from scratch, simply press the “Clear Grid” button. It will clear the entire grid and allow you to start over.

It’s worth noting that the “Clear Grid” button serves another purpose as well. If you’ve used the “Solve!” button to generate a solution and wish to remove it, a single press on the “Clear Grid” button will erase the solver’s solution from the grid.



(a) The solution is correct.



(b) The solution is incorrect.

Figure 9: The verdict when the two door cells are connected.

A SAT-based Solver Source Code

```
import itertools
from pysat.solvers import MinisatGH

# Path instance
m, n = -1, -1
start_x, start_y, finish_x, finish_y = -1, -1, -1, -1
cr, cc = [], []

# Helpers
dirs = ['d', 'l', 'r', 'u']
move = { 'd': (1,0), 'l': (0,-1), 'r': (0,1), 'u': (-1,0) }
max_lit = 0 # to track the biggest integer used to represent literals

def is_valid(i, j):
  return 0 <= i < m and 0 <= j < n
def get_adjacents(i, j):
  return [(i+dx, j+dy) for dx, dy in move.values() if is_valid(i+dx, j+dy)]

# Cardinality constraints
def atMost(lits, bound):
  combinations = list(map(list, itertools.combinations(lits, bound+1)))
  return [[-v for v in comb] for comb in combinations]
def equals(lits, bound):
  return atMost(lits, bound) + atMost([-v for v in lits], len(lits)-bound)
```

```python
def solve_puzzle(cfg):
  global m, n
  global start_x, start_y, finish_x, finish_y
  global cr, cc
  global max_lit

  m = cfg.get('m')
  n = cfg.get('n')
  start_x = cfg.get('start_x')
  start_y = cfg.get('start_y')
  finish_x = cfg.get('finish_x')
  finish_y = cfg.get('finish_y')
  cr = cfg.get('cr')
  cc = cfg.get('cc')

  lower_bound_row = sum([val for val in cr if val != -1])
  upper_bound_row = sum([(val if val != -1 else n) for val in cr])
  lower_bound_col = sum([val for val in cc if val != -1])
  upper_bound_col = sum([(val if val != -1 else m) for val in cc])
  ds = abs(finish_y - start_y) + abs(finish_x - start_x)
  lb = max(lower_bound_col, lower_bound_row, ds+1)
  ub = min(upper_bound_col, upper_bound_row)

  found = False
  for path_len in range(lb, ub + 1):
    solver = MinisatGH()

    # A bijective function that maps V(i,j,t) to a unique integer
    def V(i: int, j: int, t: int) -> int:
      return t + j*path_len + i*path_len*n + 1

    max_lit = V(m-1, n-1, path_len-1)

    # Configure start and finish cells
    solver.add_clause([V(start_x, start_y, 0)])
    solver.add_clause([V(finish_x, finish_y, path_len-1)])

    # Configure rule: if true for some cell (i,j), then one of its
    # adjacent cells must be true
    for t in range(path_len - 1):
      for i in range(m):
        for j in range(n):
          adj = get_adjacents(i,j)
          if len(adj) != 0:
            solver.add_clause([-V(i,j,t)] + [V(ni,nj,t+1) for ni,nj in adj])

    # Configure rule: at time t, only one cell must be true
    for t in range(path_len):
      solver.add_clause([V(i,j,t) for i in range(m) for j in range(n)])
      AC = [V(i,j,t) for i in range(m) for j in range(n)]
      for a in range(m*n-1):
        for b in range(a+1, m*n):
          solver.add_clause([-AC[a], -AC[b]])

    # Configure rule: at each cell (i,j), it can be true
    # for at most one time t
    for i in range(m):
      for j in range(n):
        for t1 in range(path_len-1):
          for t2 in range(t1+1, path_len):
            solver.add_clause([-V(i,j,t1), -V(i,j,t2)])

    # Contraint number setup
    memo = {}
    def C(i: int, j: int):
      global max_lit
      if (i,j) in memo: return memo[(i,j)]
      max_lit += 1
      memo[(i,j)] = max_lit
      return memo[(i,j)]

    for i in range(m):
      for j in range(n):
        cur_cell = [V(i,j,t) for t in range(path_len)]
        solver.add_clause([-C(i,j), *cur_cell])
        solver.append_formula([[C(i,j), -x] for x in cur_cell])

    # Configure constraint row
    for i in range(m):
      if cr[i] != -1:
        row_vars = [C(i,j) for j in range(n)]
        constraint_row = equals(lits=row_vars, bound=cr[i])
        solver.append_formula(constraint_row)

    # Configure constraint col
    for j in range(n):
      if cc[j] != -1:
        col_vars = [C(i,j) for i in range(m)]
        constraint_col = equals(lits=col_vars, bound=cc[j])
        solver.append_formula(constraint_col)

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

    if sat:
      solution = solver.get_model()
      def var_is_true(x):
        l, r = 0, len(solution) - 1
        while l <= r:
          mid = (l+r)//2
          if abs(solution[mid]) == x:
            return solution[mid] > 0
          elif abs(solution[mid]) > x:
            r = mid - 1
          else:
            l = mid + 1
        assert(False)
      # Get path
      path = []
      for t in range(path_len):
        for i, j in itertools.product(range(m), range(n)):
          if var_is_true(V(i,j,t)):
            path.append((i,j))
            break
      # Draw grid
      grid = [['.' for _ in range(n)] for _ in range(m)]
      grid[finish_x][finish_y] = 'u'
      for i in range(len(path) - 1):
        for d, delta in move.items():
          adj_x, adj_y = path[i][0]+delta[0], path[i][1] + delta[1]
          if path[i+1] == (adj_x,adj_y):
            grid[path[i][0]][path[i][1]] = d
      # Return
      found = True;
      return {'found': found, 'grid': grid}
      break

  if not found:
    return {'found': found, 'grid': []}
```

B Verifier Source Code

```
function verify() {
    let invalid = []
    let isIncompliant = false
    updateConstraints()
    for (let idx = 0; idx < path.length; idx++) {
        if (cr[path[idx][0]] - 1 == -1) {
            cr[path[idx][0]] = 999
        } else if (cc[path[idx][1]] - 1 == -1) {
            cc[path[idx][1]] = 999
        }
        if (cr[path[idx][0]] != -1) cr[path[idx][0]] = cr[path[idx][0]] -1
        if (cc[path[idx][1]] != -1) cc[path[idx][1]] = cc[path[idx][1]] -1
    }
    for (let i = 0; i < m; i++) {
        if (cr[i] > 0)  {
            invalid.push(`cr-${i+1}`)
            isIncompliant = true
        }
    }
    for (let j = 0; j < n; j++) {
        if(cc[j] > 0) {
            invalid.push(`cc-${j+1}`)
            isIncompliant = true
        }
    }
    updateConstraints()
    return [!isIncompliant, invalid]
}
```