Yin-Yang Solver
User Manual

Made Indrayana Putra
September 16, 2022

Contents

1 Introduction

This user manual provides a detailed explanation of the functionalities of the Yin-Yang Solver. The Solver can be accessed using the following link: https://computing-telu.com/products/yin-yang/

1.1 Main Interface

The following is the main interface of the Solver. The interface consists of:

1. Row : Shows and adjust the number of rows.
2. Column : Shows and adjust the number of columns.
3. Clear : Clears the puzzle into empty cells.
4. Build Puzzle : Sets the puzzle into a pre-determined solvable puzzles.
5. Solve : Solves or Verifies a puzzle.
6. Puzzle Board : Represents the Yin-Yang Puzzle.
7. Description : Simple description of the application.

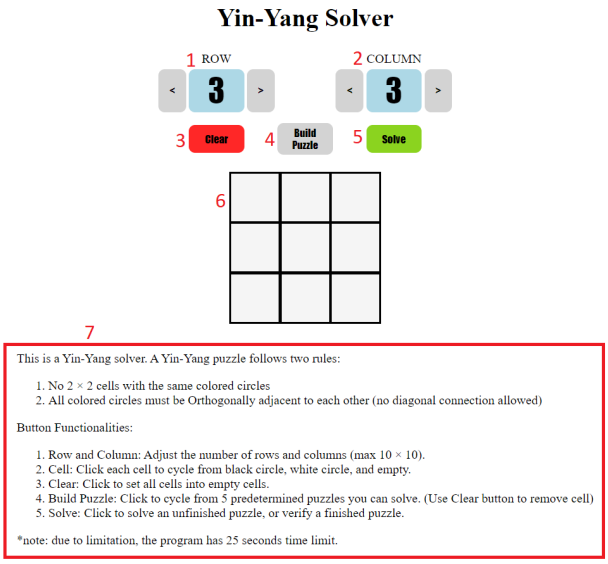


Figure 1: Main interface of the Yin-Yang solver.

1.2 Answer Interface

The following is the answer interface of the Solver when solving a puzzle. The interface consists of:

1. Description : Shows if the input has a solution or not
2. Solution : Shows a Yin-Yang solution based on the input.
3. Return : Returns to main interface

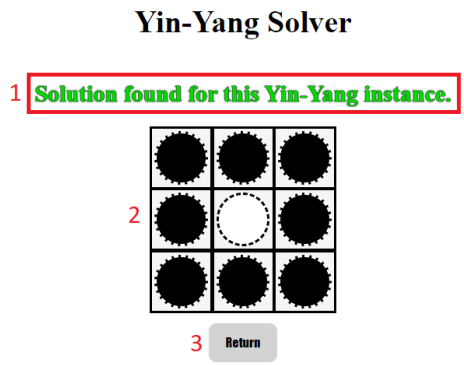


Figure 2: Answer interface of the Yin-Yang solver.

2 Functionalities

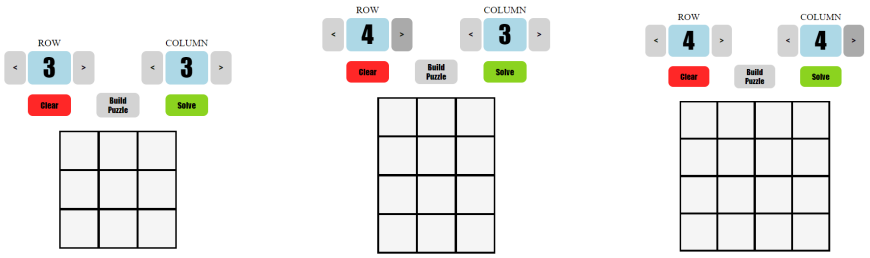
This section describes some functionalities of the application and the oputput of said functionalities.

2.1 Row and Column

These buttons allows the app to adjust the number of rows and column of the puzzle.

2.2 Cells

Each cell of a puzzle can be changed by the user. When clicked, the cell cycles from black circle, white circle, and empty cell.



(a) Initial state. (b) Add 1 row. (c) Add 1 column.

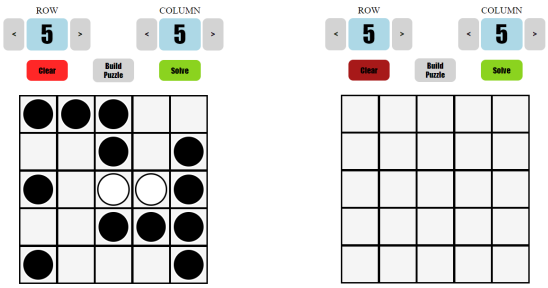
Figure 3: Example of changing row and column.

(a) Hovered empty cell. (b) Black Circle (c) White Circle

Figure 4: Example of changing row and column.

2.3 Clear Button

This button sets all available cells into cells.

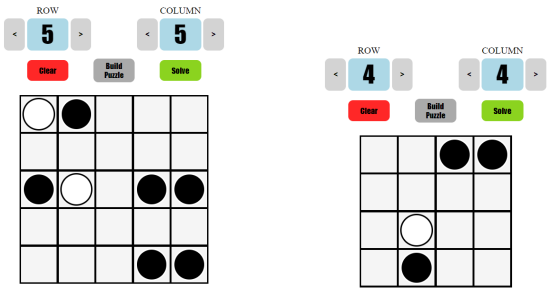


(a) A partially filled Yin-Yang puzzle. (b) Cleared using clear button.

Figure 5: Example of clearing a puzzle with clear Button

2.4 Build Puzzle Button

This button allows the user to try some pre-determined puzzles that can be solved. The button replaces the previous puzzle with a new one.



(a) Build Puzzle example.

(b) Another example overwriting the previous.

Figure 6: Example of building puzzles.

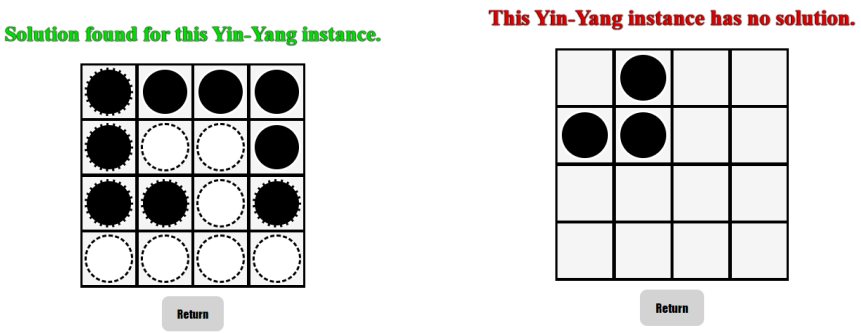
When using build puzzle button, some of the hints can not be changed. To remove the hints, use the clear button.

2.5 Solve Button

When clicking the solve button, the program does either two things:

1. Solves an unfinished puzzle.
2. Verifies a finished puzzle.

When solving an unfinished puzzle, there are two possible outcome. Either the puzzle is solvable or not. If it is solvable, the program gives the solution in the Answer interface. Otherwise it shows that the puzzle has no solution.

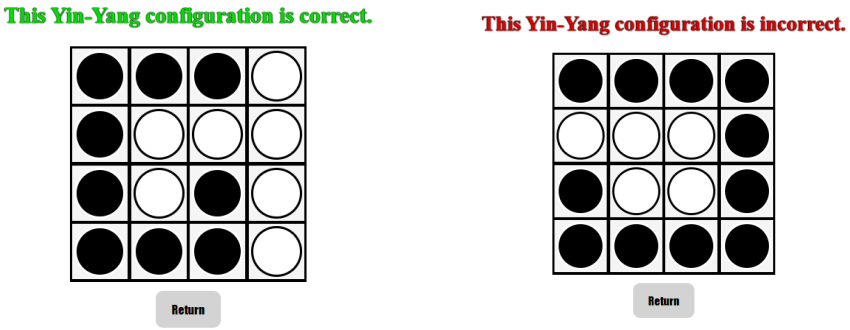


(a) A solution is found when solving the puzzle

(b) The puzzle has no solution.

Figure 7: Output of solving a Yin-Yang puzzle.

When verifying a finished puzzle, there are two possible outcome. Either the puzzle is correct or some of the rules has been broken. The program shows the result in the description line of answer interface.



(a) The Puzzle is correct (b) The puzzle is incorrect.

Figure 8: Output of verifying a Yin-Yang puzzle.

There is a time limit when solving a Yin-Yang puzzle. If the program exceeds 25 seconds time limit, the program tells the user with the answer interface.

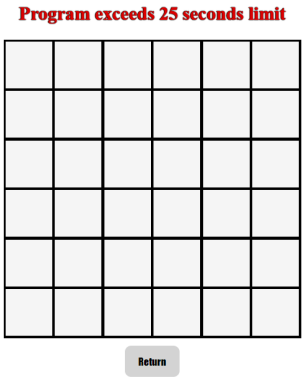


Figure 9: Output when the program exceeds the time limit

3 Source Code

The following is the source code of the solver program. For the source code of the Yin-Yang Solver website, please visit:
https://github.com/MadetheMeep/Yin-Yang-Website

```
import json
from django.http import HttpResponse
from django.shortcuts import render
import pycosat
from copy import copy, deepcopy
import time

m = 3
n = 3

#Rotation and Reflection

def identical(A, B):
    #checking whether A and B are identical matrices
    row_A = len(A)
    col_A = len(A[0])
    row_B = len(B)
    col_B = len(B[0])
    if((row_A != row_B) or (col_A != col_B)): return False
    else:
        for i in range(row_A):
            for j in range(col_A):
                if(A[i][j] != B[i][j]): return False
    return True

def transpose(A):
    #transposing a matrix A, i.e., Atrans[i][j] = A[j][i]
    row_A = len(A)
    col_A = len(A[0])
    row_A_trans = col_A
    col_A_trans = row_A
    A_trans = [[0]*col_A_trans for i in range(row_A_trans)]
    for i in range(row_A_trans):
        for j in range(col_A_trans):
            A_trans[i][j] = A[j][i]
    return deepcopy(A_trans)

def reverse_row(A):
    #reversing every row of A
  #Arevrow[i][j] = A[i][n-1-j], where n is the number of column of A
    row_A = len(A)
    col_A = len(A[0])
    A_rev_row = deepcopy(A)
    for i in range(row_A):
```

```
        A_rev_row[i].reverse()
    return deepcopy(A_rev_row)

def reverse_col(A):
    #reversing every column A
  #Arevcol[i][j] = A[m-1-i][j], where m is the number of row of A
    row_A = len(A)
    col_A = len(A[0])
    A_rev_col = [[0]*col_A for i in range(row_A)]
    for i in range(row_A):
        for j in range(col_A):
            A_rev_col[i][j] = A[row_A-1-i][j]
    return deepcopy(A_rev_col)

def CW_rotate(A):
    #rotating a matrix A 90 degree in clockwise direction
  #transpose the matrix and reverse each row
    A_CW = reverse_row(transpose(A))
    return deepcopy(A_CW)

def CCW_rotate(A):
    #rotating a matrix A 90 degree in counter-clockwise direction
  #reverse each row and transpose the matrix
    A_CCW = transpose(reverse_row(A))
    return deepcopy(A_CCW)

def one_eighty(A):
    #rotating a matrix A 180 degree in clockwise (or counter-
    clockwise) direction
  #we can compose CWrotate or CCWrotate twice
    A_one_eighty = CW_rotate(CW_rotate(A))
    return deepcopy(A_one_eighty)

def CW_rotate_horizontal_ref(A):
    #rotating a matrix A 90 degree in clockwise direction and
    reflecting the result horizontally
    A_CW = CW_rotate(A)
    A_CW_hor = reverse_col(A_CW)
    return deepcopy(A_CW_hor)

def horizontal_ref_CW_rotate(A):
    #reflecting a matrix A horizontally and then rotating the result
    90 degree in clockwise direction
    A_hor = reverse_col(A)
    A_hor_CW = CW_rotate(A_hor)
    return deepcopy(A_hor_CW)

#CNF

def mapping(r, c):
```

```
    return n * (r - 1) + c;

def check_all_same(A):
    ele = A[0]
    print(ele)
    if (ele == ['*'] or ele[0] == '*'):
        return False
    if(m == 1):
        ele2 = ele[0]
        for x in ele:
            if ele2 != x:
                return False
    if(n == 1):
        for x in A:
            if ele != x:
                return False
    return True

def get_1_solution(A):
    cnf = []
    clause = []
    if (find_first(A,0)):
        color = -1
    else:
        color = 1
    if (m == 1):
        for c in range(1,n+1):
            clause.append(mapping(m,c)*color)
    elif (n == 1):
        for r in range(1,m+1):
            clause.append(mapping(r,n)*color)
    cnf.append(clause)
    return cnf

def rule_2by2():
    cnf = []
    for r in range(1, m):
        for c in range(1, n):
            clause = []
            clause.append(-mapping(r,c))
            clause.append(-mapping(r+1,c))
            clause.append(-mapping(r,c+1))
            clause.append(-mapping(r+1,c+1))
            cnf.append(clause)
            clause = []
            clause.append(mapping(r,c))
            clause.append(mapping(r+1,c))
            clause.append(mapping(r,c+1))
            clause.append(mapping(r+1,c+1))
            cnf.append(clause)
```

```
    return cnf

def rule_alternating():
    cnf = []
    for r in range(1, m):
        for c in range(1, n):
            clause = []
            clause.append(-mapping(r,c))
            clause.append(mapping(r+1,c))
            clause.append(mapping(r,c+1))
            clause.append(-mapping(r+1,c+1))
            cnf.append(clause)
            clause = []
            clause.append(mapping(r,c))
            clause.append(-mapping(r+1,c))
            clause.append(-mapping(r,c+1))
            clause.append(mapping(r+1,c+1))
            cnf.append(clause)
    return cnf

def add_hints(A):
    cnf = []
    for r in range(m):
        for c in range(n):
            if(A[r][c] != '*'):
                cnf.append([mapping(r+1,c+1) * ((A[r][c] * 2) - 1)])
    #Map Positive for 1, Map Negative for 0
    return cnf

def translate_to_array(cnf):
    config = deepcopy(cnf)
    A = []
    for r in range(m):
        row_list = []
        for c in range(n):
            if (config.pop(0) > 0): #Positive Value
                row_list.append(1)
            else:                   #Negative Value
                row_list.append(0)
        A.append(row_list)
    return A

def translate_to_clause(A):
    clause = []
    for r in range(m):
        for c in range(n):
            clause.append(mapping(r+1,c+1) * ((A[r][c] * 2) - 1))
    return clause

def negate_clause(clause):
```

```
    return [-x for x in clause]

#Verifier

def compare_2by2(s):
    return ((s == '0000') or (s == '1111') or (s == '0110') or (s == '1001'))

def check_2by2(playboard):
    for i in range(m-1):
        for j in range(n-1):
            #if all cells in the 2x2 box are all the same (all 0 or all 1) the sum will be 0 or 4, plus if it creates alternating pattern (total 2 and (cell i j and i+1 j+1 are the same value))
            s = str(playboard[i][j]) + str(playboard[i+1][j]) + str(playboard[i][j+1]) + str(playboard[i+1][j+1])
            if (compare_2by2(s)): return False
    return True

def valid_cell(r,c):
    return ((r >= 0) and (r < m) and (c >=0) and (c < n))

def check_connectivity_BFS(A, r, c, color):
    #Initialize 2 dimentional array of 0 for checklist
    checklist = [[0]*n for _ in range(m)]
    checklist[r][c] = 1
    #Initialize queue for breath first search
    queue = []
    queue.append([r,c])
    #Initialize Direction
    dr = [0,1,0,-1]
    dc = [1,0,-1,0]
    #initialize count
    count = 1
    #run until the stack is empty
    while queue:
        #pop queue
        row, col = queue.pop(0)
        for i in range(4):
            adj_row = row + dr[i]
            adj_col = col + dc[i]
            if (valid_cell(adj_row, adj_col)) and (checklist[adj_row][adj_col] == 0):
                checklist[adj_row][adj_col] = 1
                if (A[adj_row][adj_col] == color):
                    queue.append([adj_row,adj_col])
                    count += 1
    #return count of all connected cells from the starting cell
    return count

```

```
def check_connectivity_DFS(A, r, c, color):
    #Initialize 2 dimentional array of 0 for checklist
    checklist = [[0]*n for _ in range(m)]
    checklist[r][c] = 1
    #Initialize queue for breath first search
    stack = []
    stack.append([r,c])
    #Initialize Direction
    dr = [0,1,0,-1]
    dc = [1,0,-1,0]
    #initialize count
    count = 1
    #run until the stack is empty
    while stack:
        #pop stack
        row, col = stack.pop()
        for i in range(4):
            adj_row = row + dr[i]
            adj_col = col + dc[i]
            if (valid_cell(adj_row, adj_col)) and (checklist[adj_row
    ][adj_col] == 0):
                checklist[adj_row][adj_col] = 1
                if (A[adj_row][adj_col] == color):
                    stack.append([adj_row,adj_col])
                    count += 1
    #return count of all connected cells from the starting cell
    return count

def show_config(A):
    print("")
    for x in A:
        print(*x, sep=" ")

def find_first(A, color): #Finds the first cell containing color, if
    none found it returns empty array
    for i in range(m):
        for j in range(n):
            if(A[i][j]==color):
                return [i, j]
    return []

def verify(A):
    b_count = sum(row.count(0) for row in A) #count black (0)
    w_count = sum(row.count(1) for row in A) #count white (1)
    b_start = find_first(A, 0)
    w_start = find_first(A, 1)
    if b_start: #checks if b_start is an empty array or not
        b_valid = check_connectivity_DFS(A, b_start[0], b_start[1],
    0)
    else:
```

```
        b_valid = 0
    if w_start: #checks if w_start is an empty array or not
        w_valid = check_connectivity_DFS(A, w_start[0], w_start[1],
    1)
    else:
        w_valid = 0
    if (b_valid == b_count) and (w_valid == w_count):
        return True
    else:
        return False

def verify_with_2by2(A):
    if(check_2by2(A)):
        return verify(A)
    return False
#SAT Solver

def check_duplicate_clause(neg_clauses,clause):
    for x in neg_clauses:
        if (x == clause): return neg_clauses
    neg_clauses.append(clause)
    return neg_clauses

def find_solution(cnf):
    total_iter = 0
    start = time.time()
    while True:
        solution = pycosat.solve(cnf)
        total_iter += 1
        if isinstance(solution, list):
            A = translate_to_array(solution)
            check = verify(A)
            if (check):
                print(total_iter)
                return A
            end = time.time()
            if (end - start > 25):
                return ['time']
            neg_sol = generate_negations(A, solution)
            cnf.extend(neg_sol) #Negate Solution
        else:
            return []

def generate_negations(A, solution):
    neg_clauses = []
    neg_clauses.append(negate_clause(solution))

    #Reflection only + 180 Rotate
    neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(
    translate_to_clause(reverse_row(A))))
```

```
    neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(reverse_col(A))))
    neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(one_eighty(A))))

    if (m == n):
        #Rotation and both
        neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(CW_rotate(A))))
        neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(CCW_rotate(A))))
        neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(CW_rotate_horizontal_ref(A))))
        neg_clauses = check_duplicate_clause(neg_clauses,negate_clause(translate_to_clause(horizontal_ref_CW_rotate(A))))

    return neg_clauses

def convert_2d_array(list):
    temp = deepcopy(list)
    print(temp)
    print(m)
    print(n)
    A = []
    for i in range(m):
        row = []
        for j in range(n):
            val = temp.pop(0)
            if (val == '*'):
                row.append(val)
            else:
                row.append(int(val))
        A.append(row)
    return A

def find_empty(A):
    for i in range(m):
        for j in range(n):
            if(A[i][j]=='*'):
                return True
    return False

def answer(request):
    global m
    global n
    cnf = []
    m = int(request.GET.get("row"))
    n = int(request.GET.get("column"))
    grid = request.GET.getlist("grid")
    print(m)
```

```
    print(n)
    print(grid)
    playboard = convert_2d_array(grid)
    print(m)
    print(n)
    print(playboard)
    sol = []
    res = ""
    indicator = ""
    if(find_empty(playboard)):
        if ((m == 1) or (n == 1)):
            # if(check_all_same(playboard)):
                # sol = playboard
            # else:
            cnf.extend(get_1_solution(playboard))
        else:
            cnf.extend(rule_2by2()) #adding 2by2 rule to CNF
            cnf.extend(rule_alternating()) #adding 2by2 alternating
rule to CNF
            default_count = 0
        cnf.extend(add_hints(playboard))
        if(not sol):
            sol = find_solution(cnf)
        print(sol)
        if(not sol):
            res = "This Yin-Yang instance has no solution."
            indicator = "F"
        elif(sol[0] == 'time'):
            sol = []
            res = "Program exceeds 25 seconds limit"
            indicator = "F"
        else:
            res = "Solution found for this Yin-Yang instance."
            indicator = "T"
    else:
        if (verify_with_2by2(playboard)):
            res = "This Yin-Yang configuration is correct."
            indicator = "T"
        else:
            res = "This Yin-Yang configuration is incorrect."
            indicator = "F"
    return render(request, 'App/answer.html', {'instance':json.dumps(
playboard),'solution':json.dumps(sol),'result':res,'indicator':
indicator}, )
```

Listing 1: Python source code of the Yin-Yang Solver