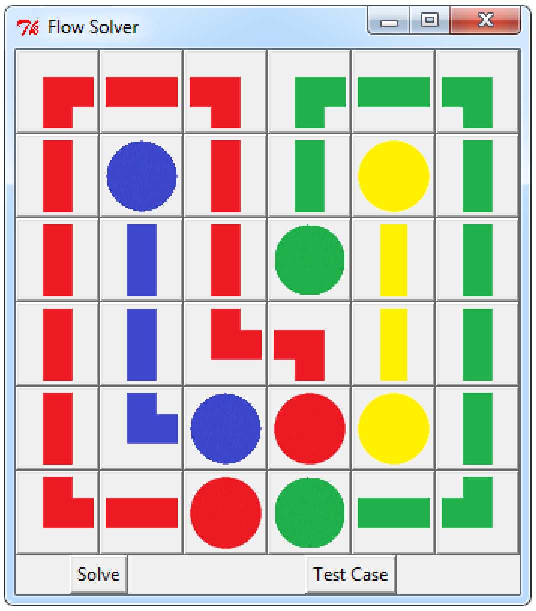
HW9-Flow Solver, Version 3 ~~1~~

20 points

Assignment: Write a program to solve (not play!) the game of Flow.

Due: Friday November 8, 5PM

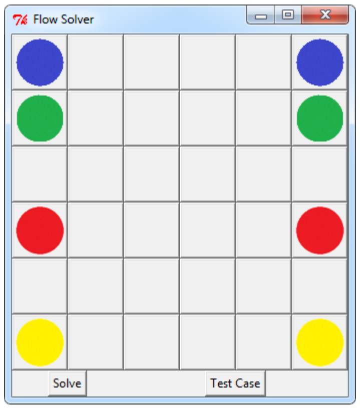
Turn In: Submit your sources to BBV.



The game “Flow” is available free at
http://html5games.com/2012/07/flow-free/
or http://moh97.us/flow/
There are also free iPhone and Android apps.

Try playing the game, if you’re not familiar with it. You’ll need to know how to play, but you will NOT have to be particularly good at it to write a good solver. So try not to get addicted!

I’ll provide a folder called “Cell images” containing pictures of all of the possible cell contents. Download this from BBV, and place that folder into the folder containing your HW9 solution.



When you start the program, the cells should be empty. The cells are really buttons; each button should contain the “Empty.gif” image initially. As you click on buttons, the endpoint circles should appear. Each pair of button clicks should use the same color. The example shown on the right is the result of clicking 8 times, and is a good test case to use when starting.

Yours may show different colors than the example on the right, but this is a good easy test case to use when starting to write your program.

I’ll provide a detailed algorithm starting on the next page.

As we’ll discuss in class, it is possible to associate extra information (“attributes”) with objects in Python. In particular, we’ll be associating attributes with each photo (from the “Cell Images” folder) as they are read in. With each photo we’ll associate that photo’s color (as a string), and whether that photo “goes” (connects) up, down, left, or right (4 pieces of information, each `True` or `False`).

Also, with each button, we’ll associate the photo that that button contains. While Tkinter provides a config() mechanism for changing he photo on a button, it’s otherwise not easy to find out later which photo it contains.

Algorithm:

0) Standard stuff: put your name/hw# at the top. Do your Tkinter import, and create a window with title.
1) We'll first **read in all of the photos** from the "Cell images" folder.
   a) Create a PhotoImage from the file "Cell images/Empty.gif". Call your variable "`empty`". Since the empty cell is allow to connect to any other cell, set empty.goesUp, empty.goesDown, empty.goesRight, and empty.goesLeft all to `True`.
   b) Likewise, create a variable "`wall`" from "Cell images/Wall.gif", but since a wall can't connect to any other cell, set its goesDown, goesRight, goesLeft, and goesUp attributes all `False`.
   c) Read the endPoint photos. Rather than make a separate variable for each one, put all of these photos into a list called "endPoints". This list will be needed when the user clicks on the cells (buttons) during setup, but will not be needed during the solution phase. Since endPoints can connect in any direction, set each photo's goesDown, goesRight, goesLeft, and goesUp attributes to `True`. And, set the photo's color attribute to the appropriate string.
   d) Read the remaining (Horizontal, Vertical, Q1, Q2, Q3, Q4 of all colors) images, putting all of them into a list called "tileList". This list will be used during the solution phase; these are the possible photos we'll put into the buttons that are initially empty. Set each one's four goes… attributes and color attribute as appropriate.

If you haven't run your program yet, run it. It shouldn't produce any output, but may have syntax errors or may crash when run. Fix any crashes that occur.

I recommend that you run your program frequently as you develop it, so that if there are crashes you'll know it's probably caused by something you recently did.

Note: below we'll be changing the image associated with each button. Since during the solve phase we'll need to know which buttons contain which images, we'll add attributes to the buttons like we did to the photos. We'll only need one attribute: `image`.

2) **Construct the board.** This will be a 2D list of buttons, and is somewhat similar to the TicTacToe demo.
   a) Make a variable called "`gridSize`", with an initial value of 6. Make sure to use this variable every time you need to know the size of the grid; we'd like to be able to use larger grids later and only have to change this one variable. The list of buttons that you create below will be actually gridSize+2 rows and columns, because of the hidden walls on the outside border of your 2D list.
   b) Proceed to make your 2D list (call it "cells"), each item in the 2D list should be a button.
      i) The buttons on the top and bottom row, or the far left and right column, should use the `wall` photo. We won't actually show the outer walls (don't use button.grid() on them). These buttons won't need a "command". Add an `image` attribute to your edge buttons, setting it of course to **`wall`** (no quotes).
      ii) For the non-wall buttons: the initial image should be "empty". Use button.grid() to place them into the appropriate row and column. Clicking the button needs to change the image on that button from empty to one of the images in the endPoint list. Since each button will need a different "command" function, we'll need to generate that function there inside of your row/column loops like the TicTacToe demo did. Create a function, with the argument (b=button) that does the following:
         (1) Refer to a global "clickCount" variable. Use clickCount/2 to decide which image from the "endPoints" list to place into this button. (The integer division by 2 will ensure that each *pair* of buttons has the same color):
         `button.config(image=endPoints[clickCount/2])`
         Now, and every time you modify the image on a button, also set the buttons "image" attribute to be that same image, so we can check it later.
         (2) Increment the clickCount variable.

         Create your button using that function as the command. Place the button into the appropriate grid row and column. Add the appropriate image attribute to your button: `button.image=endPoints[clickCount/2]`

At this point, create a window.mainloop(), and you should be able to run your program and place pairs of like-colored circles by clicking the buttons. Do not proceed until this part works.

There will be several functions involved in the solver. One will be a recursive function called solve(), that looks for an empty cell. When an empty cell is found, solve() will call take() to try to place the possible photos into that cell. For each trial, solve will call checkCell() to see if that trial photo is allowed there.

In general, "solve()" 's job is to attempt to solve the board, by making guesses into the remaining empty cells. It should return `True` or `False` depending on whether it was able to solve the board.

3) Write the basic solver.

a) Let's do some easy stuff first. In class you were given the `isAllowedRight(curr,right)` function. Add that to your program. Write the similar functions `isAllowedAbove(curr,up)`, `isAllowedBelow(curr, down)`, and `isAllowedLeft(curr, left)`.

b) Write a function "checkCell(row,col)" that checks whether the photo in the cell at that row or column is OK or not. Return `True` or `False` accordingly. Do this by calling your four isAllowed functions. The "current" cell of course is the image in the cell at cells[row][col]. Right, left, up and down are the four adjacent cells, at row±1 and col±1. The current cell is OK if only if ALL of the isAllowed functions return True.

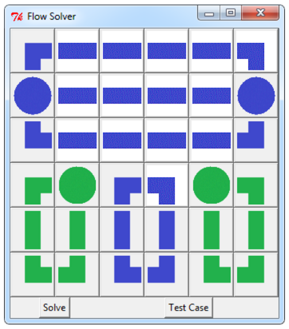
c) Write the solve() function. This function should search through all of the rows and columns of the cells 2D list, looking for a button whose image attribute is equal to your variable `empty`. This call to solve will only attempt to work on this one row and column. When the empty cell is found, call a new function called `take(row, col)` (we'll write this below). Take() will attempt to place photos into that row, col, and return True if it succeeds or False if it fails. If it succeeds, then solve() should return True. But if take() fails, then return False from solve().
If your solve() function never finds any empty button, then the entire board must have been solved already. Return **`True`**.

d) Write the `take(row,col)` function. This should loop through all of the photos in the `tileList`. For each one, place that photo in the button at [row][col]. Then, call checkCell() to see if that photo works.

   i) If checkCell() returns true, then call solve() to attempt to solve the rest of the board. If solve() returns true, then you've succeeded (!!) so return True from take(). But if take() returned false, or if solve returned false, then continue trying the rest of the images in tileList. If none of the images work, set the current button back to empty and return False from take().

e) To make the solving process show its work, call `window.update_idletasks()` every time you change the photo on a button. This will cause the window to be updated visually, even though you haven't returned back to window.mainloop() yet.

At this point your program should almost work. Some puzzles will be solved correctly, but some will have problems. This is because there is no check on the number of cells connecting to an endPoint. So, you might get solutions like this one:



Also, it might take a long time to solve the puzzle.

We'll solve these problems next.

4) Finishing touches.

a) To speed up the program, don't call window.update_idletasks() so often. When a button photo changes, call a function of your own called "show()". There, increment a global counter, and if the counter reaches a multiple of some large number (500 should be a good starting point) then call window.update_idletasks(), otherwise simply return without updating the window. Actually updating the window takes a long time. A final update will still happen automatically when the function (command) called by the Solve button returns.

b) To avoid the problem of an unrestricted number of cells connecting to an endpoint, we'll first need a function that counts the number of cells entering an endpoint.

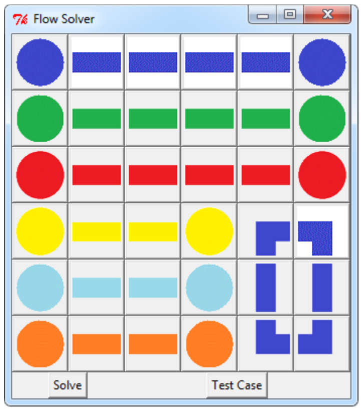
   i) Write a function that scans the cells (all rows and columns) for endpoints. For each endpoint, count the number of adjacent (left, right, up down) cells that are *empty*. Also, count the number of cells that *enter* this endpoint (cell above that goes down, cell on the right that goes left, etc). ~~If the number of entries is > 2, return false. If the number of entries is 1, return true. If the number of entries is zero, return true if there is at least one empty cell adjacent, else return false.~~If the number of entries is ≥2, return false. If the entries and empties are both 0, return false. Otherwise let the row/col loops continue. If the loops finish, return True since no problems were found with any of the endpoints

   ii) Call this function from take, between the current calls to checkCell and solve. So, "take" should return true if and only if checkCell, checkEndpoints (above), and solve() all return True.

c) Add a "Test Case" button, that sets up your favorite starting position. If you don't have one (!) use the one at the top of page 1 of this document.

d) Smile.

There is a remaining problem, a flaw in the algorithm, that my own program doesn't handle and that I haven't found a simple solution for. There is nothing to prevent loops, like the one on the bottom right.



We'll have to live with it.

If you find a nice solution to detect or prevent loops, please let me know!