15-110: Principles of Computing, Spring 2018

Programming Assignment 9

Due: Tuesday, April 10 by 9PM

IMPORTANT ANNOUNCEMENT
You cannot drop this assignment even if it is your lowest PA score.
Failure to submit this assignment on time will result in a 0 which will be included in your PA total.
We only drop the lowest PA score from PA1-PA8. Please submit whatever you can before the deadline, even if it is not completely done.

Note: You are **responsible for protecting your solutions** to the following problems from being seen by other students both physically (e.g., by looking over your shoulder or verbal discussion) and electronically. In particular, since the lab machines use the Andrew File System (AFS) to share files worldwide, you need to be careful that you do not put files in a place that is publicly accessible.

If you are doing the assignment on the Gates-Hillman Cluster machines we use in the lab or on `unix.andrew.cmu.edu`, please remember to have your solutions inside a `private` folder (which is under your home directory). Our recommendation is that you create a `pa9` folder under `~/private/15110` for this assignment. That is, the new directory `pa9` is inside the directory named `15110`, which is inside the `private` directory.

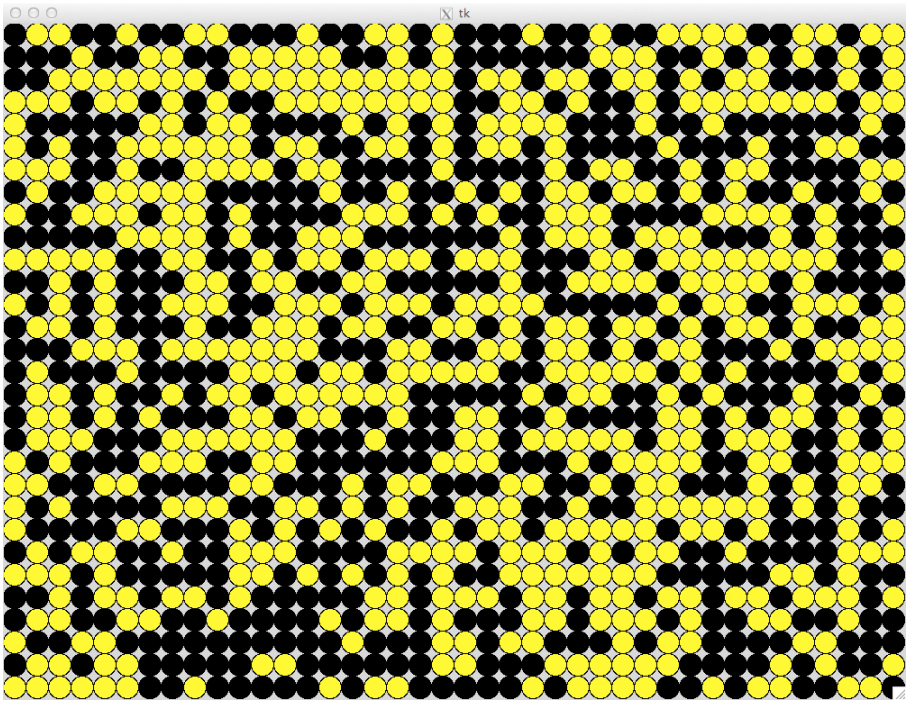
Setup

For this assignment, you will create a Python file for each of the problems below. You should save all of these files in a folder named `pa9`. Once you have every file, you should zip up the `pa9` folder and submit the zipped file on Autolab.

This assignment will help you put the principles you learned in class to creating a new simulation.
NOTE: If you use your own laptop and have Python 3 installed, you should have the tkinter module available. See a TA if you wish to install Python 3 on your laptop. If you ssh into the Andrew servers, tkinter is now available to import, but you need to use the –X option when connecting.

Overview

In this assignment, you will develop a simple graphical simulation that simulates a collection of cells that live and die on a microscope slide. The cells are arranged in a grid of size 30 (rows) X 40 (columns) on canvas of size 1000 pixels X 750 pixels. Each cell will be displayed by a circle that is 25 pixels wide by 25 pixels high. Cells that are alive are colored yellow; cells that are dead are colored black. Here is a sample grid of cells:



The simulation you create will show the state of all of the cells from one second to the next as a series of snapshots, like the flu virus simulation shown in class. Each second is known as a "generation".

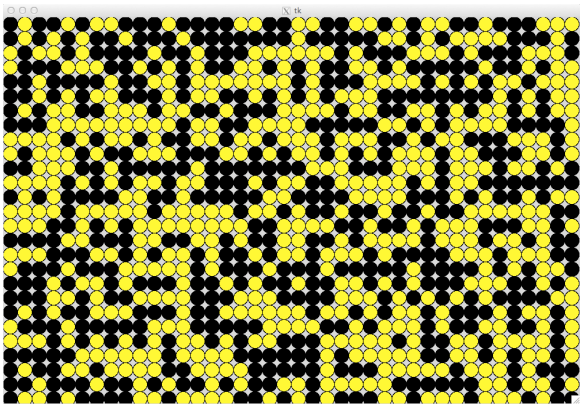
Cells live and die using the following rules:

- If a live cell has fewer than 2 live neighbors in the current generation, the cell will die in the next generation due to isolation.
- If a live cell has more than 3 live neighbors in the current generation, the cell will die in the next generation due to overpopulation.
- If a dead cell has 3 live neighbors in the current generation, the cell will become alive in the next generation (birth).
- If none of these conditions above apply, then the cell remains in its current state in the next generation.

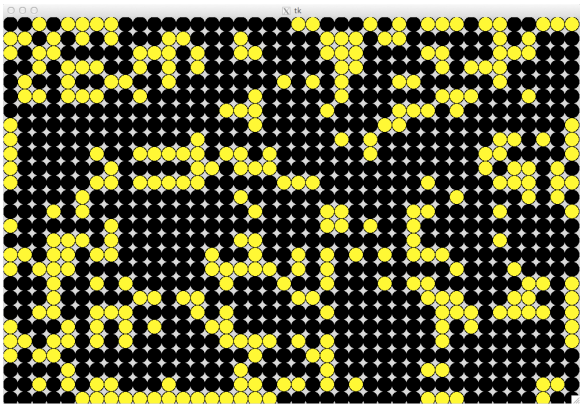
A neighbor is defined as a cell that is directly adjacent to a given cell, either horizontally, vertically or diagonally. Most cells will have 8 neighbors, but some cells will have fewer.

This simulation will show how the cell population evolves over time, represented by a matrix of 30 X 40 circles. We will represent the cells internally as a list of lists of integers, with two different integers used to represent the states of alive and dead.

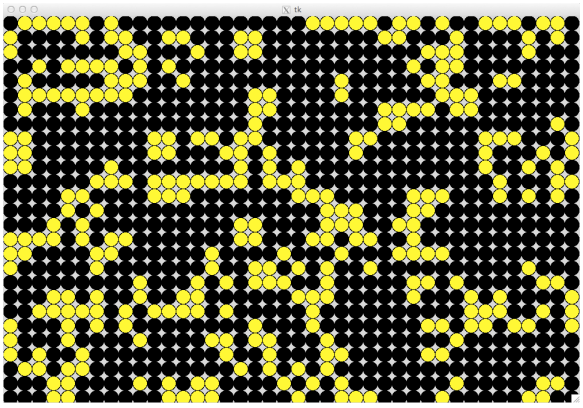
Here is a sample of what the first three simulated seconds might look like:



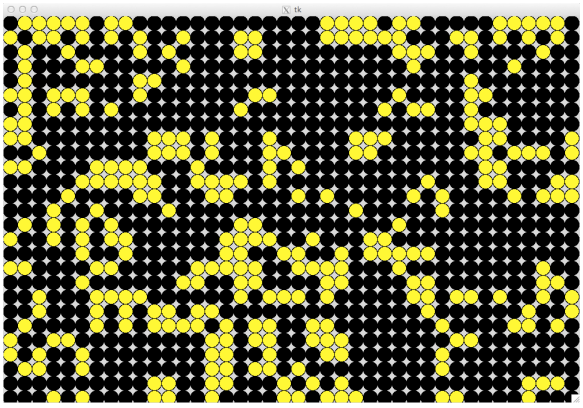
Initial state



after 1 simulated second



after 2 simulated seconds



after 3 simulated seconds

Each snapshot above is a time step of the simulation shown in a window that is 1000 pixels wide by 750 pixels high. Each circle represents one cell.

Complete the problems below to build this cell simulation in Python 3. Refer to the simulation we covered in class for guidance on how to put together this new simulation. **Since there are only two states for the cells, do NOT use any global constants for this programming assignment like we did for the flu virus simulation.**

NOTE: If you have any issues with colorblindness or sight concerns, please contact your instructor for accommodations.

Problems (PLEASE NOTE CORRECTION IN PROBLEM 1)

1. (3 points) Write a function `display(c, matrix)` in the file `display.py` where the parameter `matrix` is a 30 X 40 matrix (i.e. list of lists) representing a culture (collection) of 1200 cells for the simulation. Each value in the matrix has an integer in the range 0 through 1 (inclusive), which encodes a cell's state as follows:

   1 = alive (yellow) 0 = dead (black) <<< CORRECTED!

   Your function should go through the entire matrix and display each "cell" on the canvas `c` as a circle in the proper position with a diameter of 25 pixels in the indicated color based on its state.

   General algorithm:
   I. For each row and column of the matrix, do the following:
   A. Set color equal to the corresponding color as given in the information above depending on the value at that row and column in the matrix.
   B. Draw a circle with a diameter of 25 pixels on the canvas based on the current row and column with a fill color as specified above and an explicit outline drawn in black.
   Notes:
   - Think about how to draw a circle and how this corresponds to the given diameter.
   - Be sure to map rows and columns to the correct dimension in the window.

   Test your function using the Python function `test_display` (and its helper) below that creates a matrix of size 30 X 40 and fills each cell with a random integer between 0 and 1 inclusive.

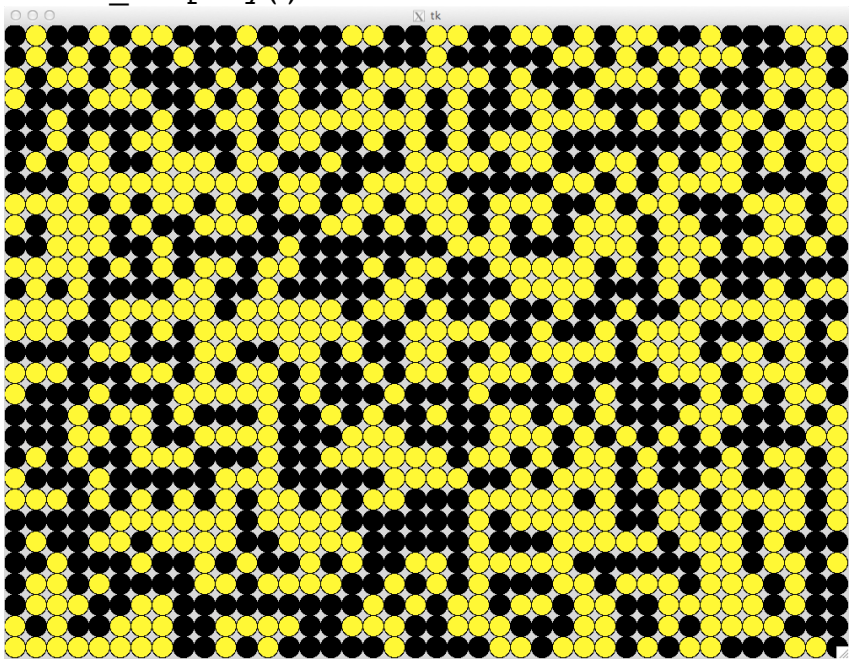
```
def create_matrix():
    matrix = []
    for i in range(0,30):
        row = []
        for j in range(0,40):
            row.append(randint(0,1))
        matrix.append(row)
    return matrix

def test_display():
    window = Tk()
    c = Canvas(window, width=1000, height=750)
    c.pack()
    seed(15110)
    matrix = create_matrix()
    display(c, matrix)
    c.update()
    window.mainloop()    # wait until user clicks close button
    return matrix
```

Put this function in the same file as your `display` function. Be sure to import the necessary modules for your code to run correctly.

For this problem, we have a random seed in the `test_display` function. This means that the random number generator always start at the same place in its sequence every time you run the function. So you should get the same test display every time (returned matrix not shown):

```
> python3 –i display.py
>>> test_display()
```



2. (3 points) Next, you will write some support functions to determine if a cell is alive or dead and to count the number of live neighbors a cell has.

   a. Write a function `alive(matrix, row, column)` in the file `tests.py` that returns `True` if the integer at the given row and column in the matrix represents a cell that is alive, `False` otherwise.

   Be sure to test your function in the Python interpreter. You don't have to test this function with a matrix of size 30 X 40. You can test it with a much smaller matrix.

   Sample usage:

```
> python3 -i tests.py
>>> matrix = [ [ 1, 1, 0, 1, 1], [0, 0, 1, 1, 1], [1, 0, 1, 0, 0]]
>>> alive(matrix, 2, 2)
True
>>> alive(matrix, 1, 0)
False
```

b. Write another function `dead(matrix, row, column)` in the file `tests.py` that returns `True` if the integer at the given row and column in the matrix represents a cell that is dead, `False` otherwise.

Be sure to test your function in the Python interpreter. You don't have to test this function with a matrix of size 30 X 40. You can test it with a much smaller matrix.

Sample usage:

```
> python3 -i tests.py
>>> matrix = [ [ 1, 1, 0, 1, 1], [0, 0, 1, 1, 1], [1, 0, 1, 0, 0]]
>>> dead(matrix, 2, 2)
False
>>> dead(matrix, 1, 0)
True
```

c. Write a function `neighbors(matrix, row, column)` in the file `tests.py` that returns the number of live neighbors for the cell represented in the matrix at the given row and column. BE CAREFUL: Not all cells have 8 neighbors!

General algorithm: Set up a counter initialized to 0. If the cell is not along the left edge and that neighbor is alive, add 1 to the counter. If the cell is not along the top edge and that neighbor is alive, add 1 to the counter. Continue this for the other 6 possible neighboring positions. Then return the final value of the counter. (You can make use of the function `alive` that you wrote for part a.)

Be sure to test your function in the Python interpreter. You don't have to test this function with a matrix of size 30 X 40. You can test it with a much smaller matrix.

Sample usage:

```
> python3 -i tests.py
>>> matrix = [ [ 1, 1, 0, 1, 1], [0, 0, 1, 1, 1], [1, 0, 1, 0, 0]]
>>> neighbors(matrix, 1, 2)
4
>>> neighbors(matrix, 1, 1)
5
>>> neighbors(matrix, 1, 2)
4
>>> neighbors(matrix, 1, 3)
5
>>> neighbors(matrix, 0, 1)
2
>>> neighbors(matrix, 1, 0)
3
>>> neighbors(matrix, 1, 4)
3
>>> neighbors(matrix, 0, 0)
1
>>> neighbors(matrix, 0, 4)
3
>>> neighbors(matrix, 2, 0)
0
>>> neighbors(matrix, 2, 4)
2
```

3. (3 points) Write a function `nextsecond(matrix)` in the file `simulation.py`, which takes a matrix (i.e. list of lists) named `matrix`, representing our culture of cells as described above for the current time step of the simulation. This function returns a new matrix (i.e. list of lists) of the same size representing our culture of cells during the next time step of the simulation, after one simulated second has passed.

   In this file, you will need copies of the `display`, `alive`, `dead` and `neighbors` functions you wrote from the previous problems. Be sure to test these carefully since these must work correctly for the next steps to work correctly.

   The basic idea here is that we start with a representation of the current culture as a list of lists named `matrix` containing integers 0 and 1 and create a new snapshot of the culture after one time step in a new list of lists named `newmatrix`. Each value `newmatrix[row][column]` represents the updated state of that cell in the culture from `matrix[row][column]`. Refer to the simulation done in class. This function will be very similar to the function we wrote to advance one day in the flu virus simulation.

   General algorithm:

   I. Create a new matrix of the same size as matrix, with each value initialized to 0.
   II. For each row and column of the matrix, do the following:
      a. Compute the number of live neighbors of the given cell. (Use your function from part 2 of this assignment!)
      b. Apply the rules for the simulation (see page 2) and store the updated state of this cell in the new matrix.
   III. Return the new matrix as your final result of this function.

   Test your `nextsecond` function using the following function `run_simulation` (and its helper), which creates a random culture of cells given the initial seed and runs the simulation for the given number of simulated seconds. Put these functions in the same file `simulation.py`.

```
def create_matrix():
    matrix = []
    for i in range(0,30):
        row = []
        for j in range(0,40):
            row.append(randint(0,1))
        matrix.append(row)
    return matrix

def run_simulation(init_seed, num_seconds):
    # create a canvas of size 1000 X 750
    window = Tk()
    c = Canvas(window, width=1000, height=750)
    c.pack()   # continued on next page!
```

```
    seed(init_seed)
    matrix = create_matrix()
    display(c, matrix)
    c.update()
    sleep(2)
    for i in range(num_seconds):
        matrix = nextsecond(matrix)
        display(c, matrix)
        c.update()
        sleep(2)
    window.mainloop()    # wait until user clicks close button
    return matrix
```

Sample usage:

```
> python3 -i  simulation.py
>>> run_simulation(15110,50)
```

The first 4 results of this simulation function are shown in the beginning of this document, using an initial random number seed of 15110 and whatever number of seconds you wish to simulate. For example, you can run it this way: `run_simulation(15110,50)` to use an initial seed for the random number generator of 15110 and 50 simulated seconds. (Returned matrix is not shown.)

4. (1 point) Once you have your general simulation working, we want you to see what happens when the percentage of initial live cells varies. Copy all of your code in `simulation.py` into `simulation2.py` and remove the `test_display` function in `simulation2.py`. Be sure your previous simulation is working the way you want, since any changes in `simulation2.py` will not be reflected automatically in `simulation.py`.

Update the `create_matrix` function in `simulation2.py` so that it has a parameter `p` representing the probability that any cell is initially alive as a percentage chance. The parameter is assumed to be an integer between 0 and 100 inclusive (i.e. 0% chance up to 100% chance). Change the function so that a cell is alive with chance p%. HINT: Use `randint` as shown in class to do this.

Also change the `run_simulation` function so that it accepts a third parameter `p` for the probability that any cell is alive. Also, change the call to `create_matrix` so that `p` is sent to the function:

- `def run_simulation(init_seed, num_seconds, p):`
- `matrix = create_matrix(p)`

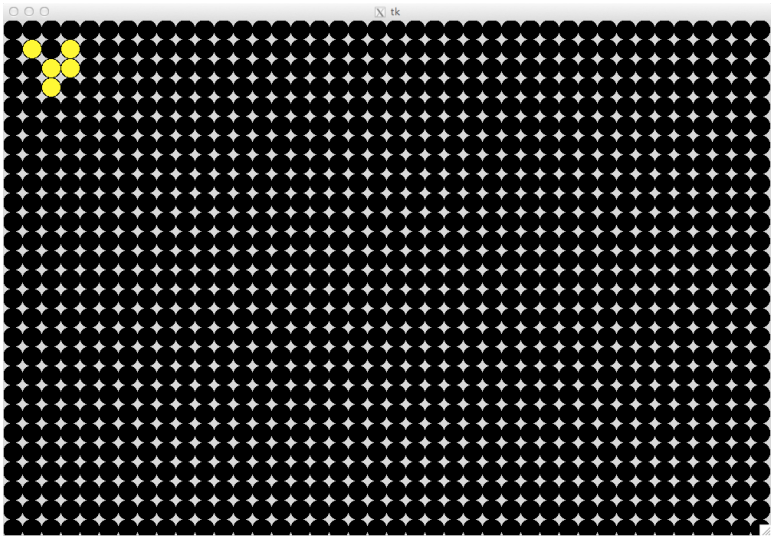
Now run the simulation again to test your work with percentage chances 10, 40, 60, and 90.

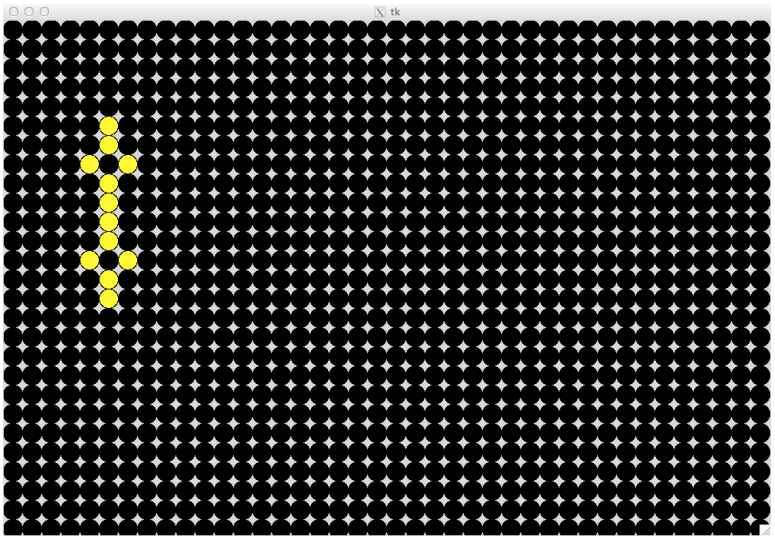
Sample usage:

```
> python3 -i simulation2.py
>>> run_simulation(15110, 50, 10)
>>> run_simulation(15110, 50, 40)    etc.
```

FOR FUN (NO CREDIT):

You can make additional simulations that start with other matrices that are not random. You do not need to hand these in, but you can try them out if you have the time or the curiosity! Each one does something very interesting! Simply make a new simulation file based on task 3 (no probabilities) and update the `create_matrix` so it returns one of these two initial cultures, as a list of lists, of course.





Submission

You should now have the `pa9` folder that contains the following Python files:

`display.py` `tests.py` `simulation.py` `simulation2.py`

You may have additional simulation files but these will not be graded.
Zip up the folder and submit the zipped file named as `pa9.zip` on Autolab.
Be sure to check your submission to see that you submitted the correct code.