Physics-based Tracking of Moving and Growing Cells in a Colony:

Counting the Number of Cells

David Moe

Research Project (Undergrad Senior)

Professor Wayne Hayes

October 10, 2017

Abstract

The program can count the number of - previously highlighted by another algorithm - bacteria cells in each frame of the sample gif with 99.70% accuracy. It does this by finding and going along the highlighted pixels of a frame in a clockwise direction until it is determined to be a cell by heuristics or until it runs out of paths to explore. The program is built to work specifically with the highlighting techniques of the algorithm and might not work as well using other methods. The completion of this program took about 2 days of coding (approximately 15 hours) and one more day to increase the accuracy of cell counting from 85.5% to 99.7% (approximately 5 hours).

Environment Settings

All the tools that I've used in the program were built-in with the **C# library** except for the image analyzer. I used **Emgu.CV** to be able to read and process images pixel by pixel and obtain their RGB color information. The gif was decomposed into individual picture frames from an online software Ezgif which are then read by Emugu.CV one by one.

Algorithm: Image Processing

The program iterates through the picture frames one by one. In each picture frame, it goes over each pixel left-to-right, top-to-bottom starting from the top-left corner (0,0). Heuristic functions are not hit until it hits the red pixels, which are from the highlighting algorithm. The color of the red was calculated and given a small range of difference for variances in color code. The red pixels are highlighted in another color (bright yellow in this case) once it has been analyzed for the user to see what's going on and for the program to treat it differently.

Figure 1 – Processed Pixels Vs Unprocessed Pixels. Actual image that was processed by the program.

Algorithm: Heuristics for Determining a Cell

The first thing it does when it finds an unprocessed cell is determining how many pixels in the adjacent (all sides and diagonals) pixels are red or yellow pixels. It's important that the yellow pixels are also counted because by not including them, it is possible for the paths to be limited thus making an underestimate of cell counts (more later on this later). In order to make it easier to select the next pixel to explore, all the pixels with 3 or more neighbors and their children are recursively marked as processed pixels. You can see in figure 1 of the second cell

that the first pixel to hit is the top-left most corner and it marked surrounding 3 as processed. After this process, it reduces the number of paths and can go down one direction more accuracy.

Pixel chooses the next pixel mainly based on the clockwise direction. Since all the cells are ellipses, if it keeps going along the main path, it'll eventually circle around. Clockwise direction was chosen since the picture frames are being read from left-to-right so it continues in a rightward direction.

The iterative adjacent pixel checking after the first red or yellow pixel in that round has a few additional heuristics. Most of it has to do with checking one more pixel further than adjacent pixels that seems to be help stay in the clockwise direction of the circle. Another one is if there are too many adjacent pixels, then it has come to an intersection with another cell. In this case, it recursively marks all the pixels with 3 or more neighbors just like it did on the first red or yellow pixel. It then chooses the path that seem to go in the clockwise direction.

Goal test is evaluated on each pixel moved. To do this, a few variables are kept track of: the average pixel-size of cells initially set to 40, a unique set of traveled pixels since the encounter of first red or yellow pixel in that iteration, and the number of times the path has traversed on already traveled pixels. The condition for determining if it is a cell is if the current pixel is back on the initial pixel after traveling a number of pixels around the average pixel-size of cell or if it has been traveling on already traveled pixels for about the average pixel-size of cell number of times, which means it has looped the same path twice.

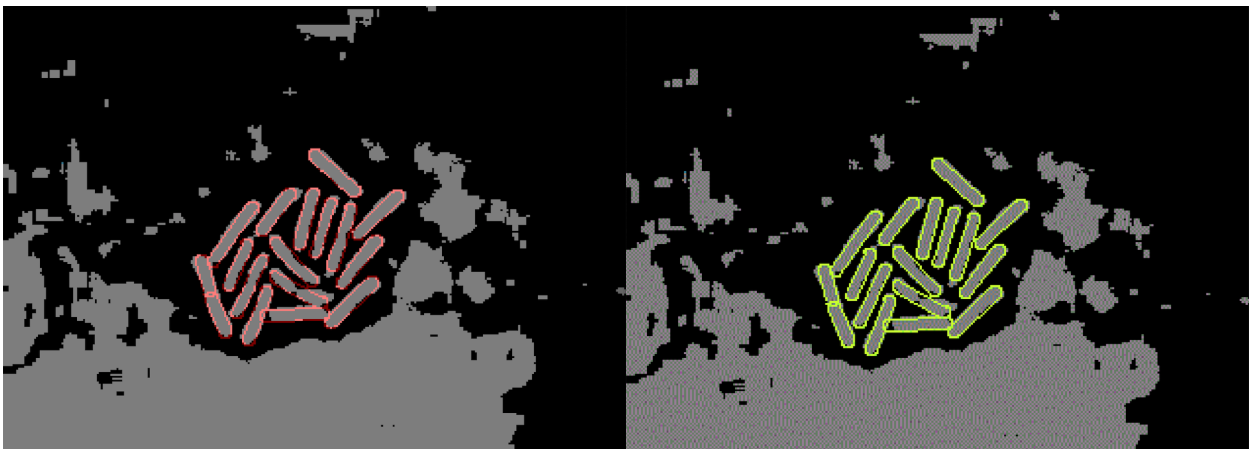

Figure 2 – Left is original frame. Right is after the program finishes counting.

Improving Accuracy

Initially, the program accuracy was about 85%. The main reason was from low accuracy was from the red pixels not being completely adjacent to each other and occasionally skipping 1 pixel. When a check was added to check 2 pixels in cases where there are no more paths or if an adjacent path is going in the opposite direction than clockwise, the accuracy improved by about 11% (see figure 2 below). Removing the goal test in the earlier stages improved the accuracy by 4 count but have no affects later, which leads me to conclusion that it wasn't actually an accuracy improvement and is from other fix, possibility from marking neighbors as visited. All of these issues were found by debugging and looking through where the program is not working properly due to some new environment presented by the frame picture that the program was not prepared for.

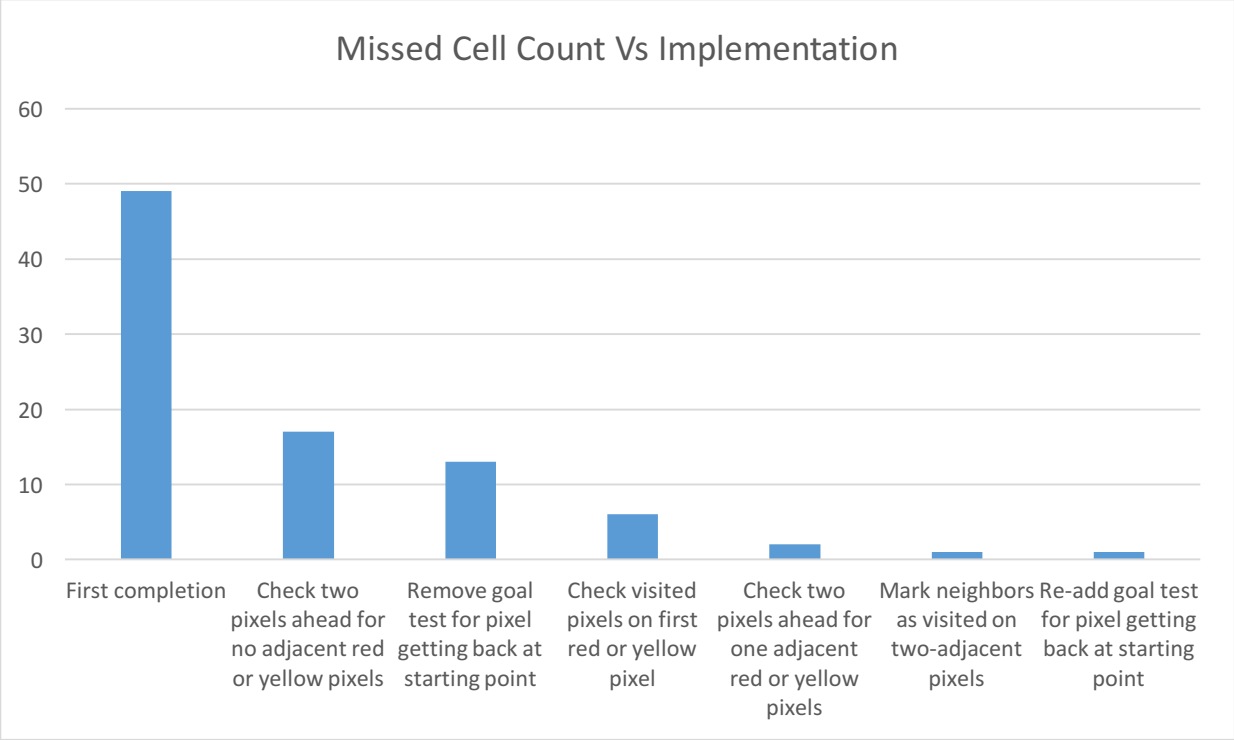

Figure 3 – Missed Cell Count Vs Implementation.

Possible Future Improvements

There are many ways of improving the current count functionality but the time could be better spent on images of cells without highlights. After a bunch of improvements, most of it is towards making the program to be really good at reading the patterns of how the highlighting algorithm counts the cells. It would be more advantages to spend more time working on cells that are not highlighted and actually work on improving the algorithm for an even more accurate counting of cells as the highlighting algorithm itself is not 100% accurate.

A few ways to improve this program would be: use neural networks to learn how the cells grows by running many simulations, improve run-time by reducing complexity of the program, use more images and run them to find areas for improvement, and organize the code for reusability and scalability for large number of inputs.