Algorithm Discussion

The algorithm for this project is contained in the file cs-sketch.js. This file uses p5.js to create the canvas and cella150.html displays the input textboxes/labels and the canvas onto the screen. This project implements 5 new functions (displayGrid, clearGrid, generateNthLevel, calculateNextState, and setMaxLevel) and 1 modified function (draw) to solve the Cella-150 Rule Problem. The HTML file waits for the user to input the number of generations (from 0 to 401) and then performs the Cella-150 Rule for that number of generations.

The function, setMaxLevel(), either prints that it is a bad input or it sets the maxLevel value. The if statement takes a O(1) time to do because it takes constant time to check if it is null, not a number, or not between 0-401. Printing and the setting takes O(1), so it takes T(n) = O(1) + max{O(1), O(1)} = O(1).

The function, displayGrid(), shows what the canvas looks like on the nth level. It goes through each column in the canvas and sets that box to white or black. Assuming that it takes a O(1) time to set a pixel, to check if it is white or black takes O(1) time for n columns inside of the grid. Thus, it would take this function a total run-time of: T(n) = n \* max{O(1), O(1)} = O(n).

The function, clearGrid(), sets all of the grid tiles to the initial state, 1’s all in the first row and 0’s everywhere else and at the (0, 200) position. To do this, it goes through each row and column and checks if that is row 0, which takes constant time to do. If it is the first row it sets it to 1, otherwise it sets it to 0. At the end, it sets the grid at (0, 200) to 0. The check for the row being 0 takes a O(1) time to do and O(1) time to set the item to 0 or 1. There are n number of rows and n number of columns to go through, so it would take a total run time of T(n) = n \* n \* max{O(1), O(1)} + O(1) = O(n2).

The most important function is calculateNextState() which finds the next generation’s value at the (r, c) position. It takes the maximum number of rows (401) and checks to see what is happening in the upper-left, above, and upper-right positions from the point (r, c). If those locations have a 1 (Black) then they get counted, otherwise they do not. In the Cella-150 Rule, the counter must be 1 or 3 in order to set this next generation pixel to 1. Thus, it takes O(1) time to get the number of columns (401), O(1) to check the top-left corner, O(1) to check the above value, O(1) to check the top-right corner, and O(1) to check the count as 1 or 3. Thus the total time is at most T(n) = O(1) + O(1) + O(1) + O(1) + O(1) = O(1).

The function, generateNthLevel(), figures out all of the cells in the nth row and shows it to the screen. It checks to make sure the value is valid (like in setMaxLevel) and then goes through all columns to calculate the next state for position (n, c). Afterwards, it displays the grid onto the screen using displayGrid(). The check takes O(1) time, calculateNextState() takes a O(1) time repeated n number of times for n number of columns. Finally, it takes O(n) time to display the grid at that nth row. Which at worst case takes T(n) = O(1) + n \* O(1) + O(n) = O(n).

On every update, the program utilizes the draw() function, which draws each generation one at a time. If the maxLevel is set and this is the starting level (n = 0), then we clear the grid which takes O(n2), initializes the grid for O(n2) time, and increments the level. Otherwise, it increments a g\_frame\_cnt that holds the current frame number and checks to see if it is a frame number that the system should update on and n is less than the maxLevel. If so, then it generates the nth level which takes O(n) time and updates n and maxLevel if needed. This draw() function takes O(n2) on the first iteration and O(n) for each update up to the maxLevel. This means, in total, it takes T(n) = O(n2) + O(n) \* n = O(n2) for all operations. This is expected because it compares and accesses all elements in the matrix at least once.