An investigation into forest fire migration via a 'Cellular Automata' model

I was tasked to create a simulation of a forest. I was also given a set of conditions in the creation of this simulation:
The simulation would be a 2D grid, with its cells existing in one of three states, based upon what the cell is supposed to be simulating. The states are 'empty cell', 'tree containing', and 'fire containing'. I decided to associate these states with the numbers '0', '1', and '2' respectively.
The rules of the simulation would be thus;

1. A burning cell in the previous timestep would be an empty cell in the next. 2 turns to 0.
2. A tree would start burning if one of its neighbours (Von Neumann neighbours) was burning in the previous timestep. 1 turns to 2 if the cell above, below, to the left of it, and/or to the right of it is a 2.
3. A tree is set on fire with random probability x, even if no neighbour is burning.
4. An empty cell will randomly grow a tree with probability y.

If none of these rules apply to a cell, the cell will remain as its previous state.

Model Method

My first step in the creation of this model was to attempt the smallest, simplest version. I knew that I had to create a 2D array for this simulation, but had only used 1D simulations before. So, my first experiment used a single 1D array with a single timestep. This code let a cell only look at the cell to the left of it - A cell would only burn if the cell to the left of it was burning. As well as this, a cell that was previously burning would turn into an empty cell at the next timestep. The simulation was essentially a string of burning trees going from left to right - a 'fuse simulation'. At first, the code only showed the initial state and the next step from it, but I later adapted it to allow more timesteps. I did this to show the consistent movement of the fire in a 1D, single direction simulation. Because the cells were checking the cell to the left of itself, I had an issue with the cell at the start of the array. Usually, in such a situation, the array would loop and the cell at the start of the array would check the cell at the **end** of the array. To avoid this, I set that if the index of the cell is less than zero, the cell to the left of it would be zero. A cell with state zero would impose no rules or interactions with any cells around it. So, it would not affect the cell of index zero.

My next experiment increased in levels of complexity. I decided to create code that would create a 2D array containing all the time steps of the initial 1D, and imposed all the rules on the one directional state.I created the one dimensional array, stated that there should be five time steps, and stated the size of the initial array. I knew that the code would need three functions; one to create the 2D array, one to impose the rules upon each timestep of the 2D array, and a function that would create a timestep that would marry the two previous functions together. In greater detail;
The initialise_state function takes the initial array and the number of timesteps. It creates a two dimensional array the same width as the initial array, and the length of the number of timesteps. It then makes the initial array the first row of this new array.
The update_state function imposes changes on a state with each timestep. It takes a given 1D array (say for example, the initial state) and creates a new 1D array. The rules in this program are the same as my first experiment. The fire can only travel one way. However, I added the two extra rules I missed before. If the previous timestep was an empty cell, it had a one in ten chance of becoming a tree. If the previous timestep was a tree, and would not otherwise set on fire (due to a fire to the left of it) it had a one in ten chance of setting on fire.
I did this by using a random number generator. The RNG would create a number ranging from one to 9. If the number was equal to nine, the relevant rule would be imposed on the relevant state.

The run_simulation function uses the last two functions. It takes the initial state, and runs it through initialise_state. It then takes the resulting array, and row by row, uses update_state to create a new state based on the row before it. It returns the array, each row being a new step in time. The code then prints the array.

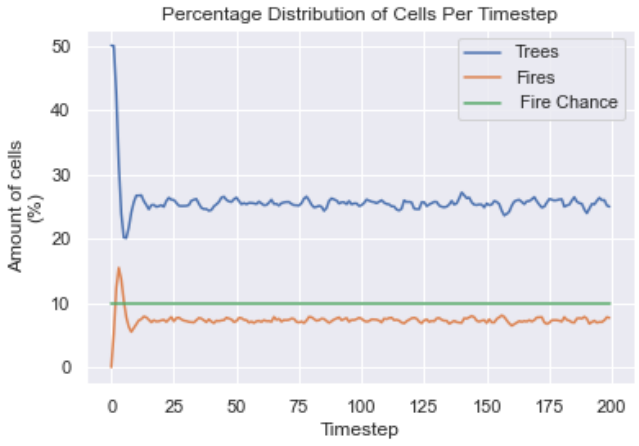
I then created a new python file, and copy pasted the last experiment into it. In this iteration, I would be using a 1D initial array, but allowing fire spread to travel right **and** left. To do this, I had to make some edits in change_state. I defined the index of the tree to the right the same way I did the tree to left, only the index would be one larger, while the index of the tree to left was one smaller. I then set the boundaries at the right of the array in a similar way to the left. However, it wasn't as simple as saying any cell of index smaller than zero was set to state 0. I had to take the length of the array, and minus one from it. If any cell had an index higher than this number, its state would be zero. After this, it was simple enough to say that if the tree to the right was on fire, a cell with state one would be state 2 at the next timestep.

I decided to run a program that would be a single timestep, and imposed two rules upon an initial 2D array. I did this to understand how the program would loop through the 2D array. The code would loop through the array in columns and the rows, with a 2D index showing its position. In this experiment, I did not put any boundaries upon the edges of the array, or allowed a tree to be affected by its neighbour. The only rules I used were 2 turns to 0, and as a test, 1 turns to 2.

I adapted the last code to include all the rules set out by the task. This new code was one timestep. It used the same code mentioned earlier for the rules. The purpose of this code was setting boundaries to the 2D array. The indexes of those cells around a tree were set. Then, like the cells beyond the furthest left and right, the cells beyond the furthest top and bottom were set at state zero. The code successfully updated each state from the first array to the new array.

With this success, I took the code and implemented it into my check_state function.
I created several testing arrays in this time to investigate how well my code was working. One of these was a seeded array, which I then used to investigate the steady state of my simulation. My tree growth and random fires still had a chance of 1 in 9 at this time. I ran the simulation with an initial state size of 100 by 100 (seed 1), for two hundred iterations.

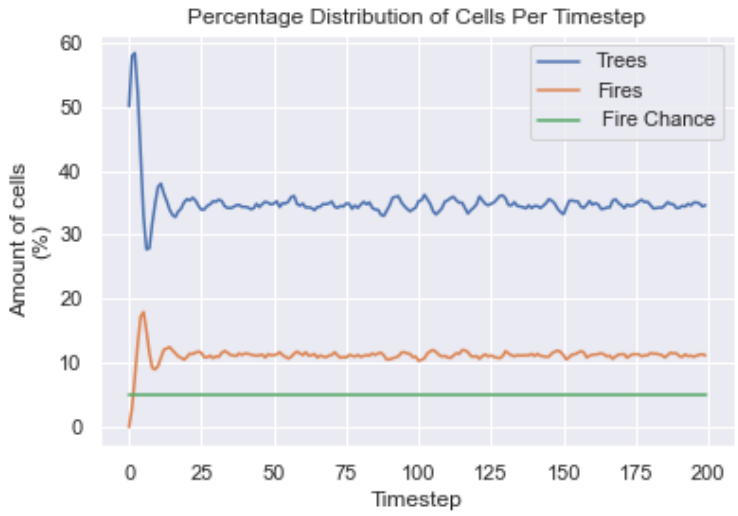
<u>Steady State and Results</u>
My next task was to investigate the relationship between growth rate, chance of random fires. I wanted to achieve a steady state; throughout the simulation, the number of fires and trees should be consistent. In order to obtain this data, I created a python file that would run the Tree simulation python file. It would then take the resulting array, and loop through it. At each timestep, it would count the number of trees and fires present. I then used it to generate this graph.



From the steady state graph, I decided to change the chances of tree growth and random fires. When fire probability was high (50%) and tree growth was low (10%), the number of trees would decrease exceptionally quickly, due to the huge increase in the number of fires. Soon after, both the number of fires and trees would reach a steady state. This would be at around 15% for trees and 9 % for fires. Any tree growth was quickly removed by the fires. However, this was not from continuous spread but from lightning strikes. This was indicated by the high lightning chance and low fire distribution.

I next experimented with high growth probability(50%) and low fire probability(10%). This graph was more promising than the last. A high tree probability meant an early spike in tree numbers. This was quickly followed by the number of fires, but after an equal number of both states (30%) they soon reached a steady position. With this information, I knew that I wanted tree growth to be more likely than lightning strikes. However, I wanted the simulation to have a less sudden increase in the number of trees.

By trial and error, I decided that random tree growth would have a probability of 20%, and random fires would have a probability of 5%. Below is the steady state graph that produces.



Additional Model Feature

I investigated several options in the addition of a new model feature. Implementing a ‘mercy’ rule, where a tree on fire had a chance to be extinguished produced peculiar results. It often produced high percentages of trees, and relatively higher numbers of fires. The high number of trees were due to the lack of fire success, which in turn led to the possibility of better fire spread, and then inevitably more fires. The mercy chance allowed both numbers to be higher.

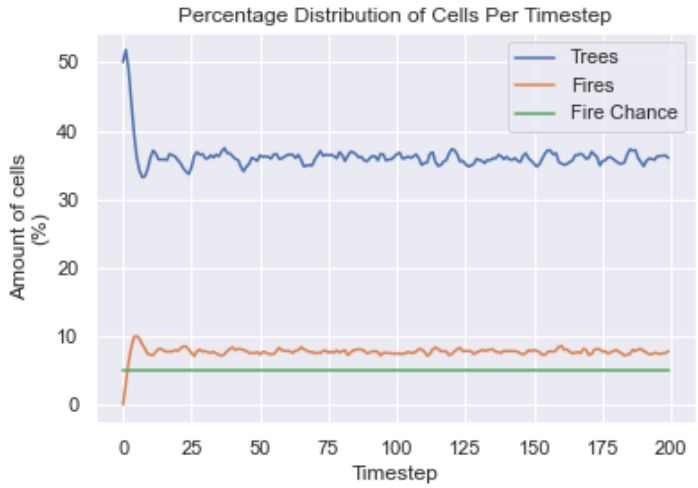
I decided to add the feature “surprise lumberjack” to my code. In this case, a tree would have a one in ten chance of being chopped down. A cell in state 1 would have a one in ten chance of becoming state 0, without first becoming state 2. I experimented as to the order of this rule in relation to others. Would it come before or after fire-spread and lightning strike?

I first experimented with having it after, reasoning realism. Lumberjacks wouldn't go near fire, and avoid lightning storms.

I thought these results were interesting. The number of fires seemed to reach a steady state at just below 10%, and the trees did the same at just below 35%. Both numbers are slightly smaller than the steady states seen in diagram 2. The peak of the fires is significantly reduced, as is the initial dip in the number of trees. The rule just wasn't that effective in changing the distribution.The rule only occurs if the tree is not next to fire and not struck by lightning; this leaves a lower probability of it occurring if it was placed earlier in the code. It has a reduced

chance of occurrence, and so a reduced effect on the model, even with what I thought was a moderately high probability of 1/10.

Having the lumberjack rule before the others had greater changes upon the results. Fires were greatly reduced, and trees had a steady state at a much higher number than the fires. The fire just couldn't spread, as oftentimes a tree was removed before it could be set on fire. Below is a graph of state distribution with lumberjack chances set at 10%.



Conclusion

The investigation of the grid-based simulation provided provocative challenges and engaging interactions between state rules. The involvement of Tree Growth and Lightning Chance acted as mitigators in wild changes in cell numbers. Too many trees, and fire spread will run rampant, leading to too little trees to naturally spread fire. Tree growth chance needed to be higher than lightning chance, as it had to replenish trees lost by fire spread **and** lightning. But not high enough to allow rampant fire spread. This introduces the idea of **ideal tree clustering**. Essentially, the distribution of trees along the grid must allow for sustainable fire spread. Not too many trees in a cluster, or it will reach another cluster and spread too often. Not too few trees in a cluster, or fire cannot spread at all. In the second case, the simulation then relies on lightning to start every fire present in the simulation. The lightning chance was present as a rule to minorly supplement the fire spread. Thus exists a range of possibilities for the two probabilities to achieve a steady state, as I have shown.

The introduction of the lumberjack rule reduces the size of the tree clusters without fire spread. The rule reduced the number of fires but kept the number of trees the same as without the rule. The lumberjacks limited the production of fire, by removing cells the fire could spread to. This reduced the amount of fires. The less fires and the more empty cells the trees could grow led to an increase in trees. The lumberjack rule then acted with the fire rule, creating a steady state where they were in competition with one another.

While not detailed to the point of predicting physical fires in a forest errorlessly, this simulation provides an engaging comprehension of the factors needed to stimulate and maintain a steady state. This report has shown how the various elements interconnect and regulate one another, and how they can compete without extinction.