CX 4230: Project 2 - Traffic Jams

Jessica Hernandez

Task 1: A First CA Model

1.1

The CA model described in 8.1 assumes the following conditions:

1. 300 cells
2. A fixed number of cars with random velocities and positions
3. Once the car exceeds the boundaries, it simply reappears at the beginning
4. V_max is 5
5. The distance between two cars in adjacent cells is 0

My simulation is based the assumptions above and on the algorithm below:

Algorithm 8.1 (The rules of the CA model).
Update for vehicle i:

1. **Accelerate**: $v_i := \min\{v_i + 1, v_{\max}\}$
2. **Decelerate**: $v_i := d(i, i+1)$, if $v_i > d(i, i+1)$
3. **Move**: vehicle i moves v_i cells forward

In the textbook, each step was done for all the cars before moving on to the next step, so I implemented each step as a separate helper function. The snippet of code below is an implementation of the accelerate step. It starts by iterating through all the cells, and checks if the cell contains a car. If so, then it updates the velocity of the car using the formula described in the algorithm. The code then saves the velocity in a matrix for plotting.

```
36          j = 0
37          while j != len(cells):
38              if isinstance(cells[j], car):
39                  curr_car = cells[j]
40                  curr_car.curr_vel = min(curr_car.curr_vel + 1, v_max) #accelerate
41                  matrix[i, j] = cells[j].curr_vel
42              j = j + 1
43          j = 0
```

I implemented the decelerate function in a similar way. After the accelerate step, I iterate though all the cells and apply the formula given in 8.1.

```
43          j = 0
44          while j != len(cells):
45              if isinstance(cells[j], car):
46                if cells[j].curr_vel > distance(cells, j):
47                    cells[j].curr_vel = distance(cells, j) #deccelerate
48                matrix[i, j] = cells[j].curr_vel
49
50              j = j + 1
```

The distance function works by starting from the next cell and then looping until it reaches the next car. When it returns the distance, 1 is subtracted to account for starting at distance 1 in line 149. This way, when two cars are in adjacent cells, the distance between them is 0 and not 1.

```
148 def distance(cells, index, num_cells):
149   for dist in range(1, num_cells):  # start from the next cell
150       next_index = (index + dist) % num_cells
151       if isinstance(cells[next_index], car):
152           return dist-1  # return the distance to the next car
153   return num_cells-1  # if no car is found, return the maximum distance
```

Lastly, the move step involves calculating the new position of a given car and moving it. To implement this, I created a new list and copied the cars to their new positions. In this way, there's always the same number of cars and there is no accidental overwriting.

```
75 def move(cells, num_cells):
76   cells2 = init_road(num_cells)
77   j = 0
78   while j != len(cells):
79       if isinstance(cells[j], car):
80           cells[j].position = (j + cells[j].curr_vel) % num_cells
81           cells2[cells[j].position] = cells[j]
82       j = j + 1
83   return cells2
```

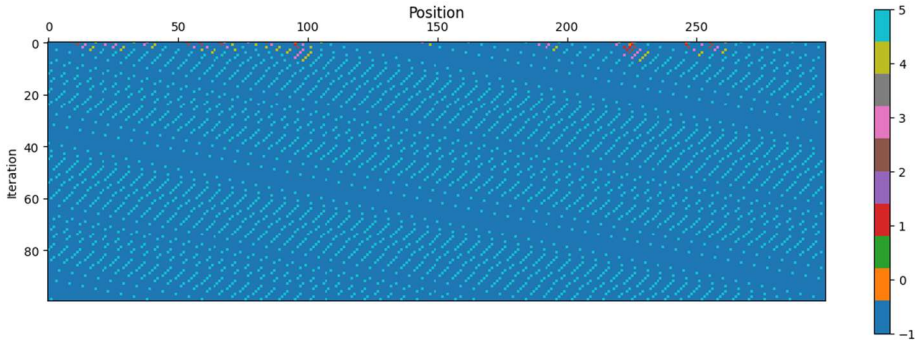
The rest of the code initializes the road and the cars inside

```
 7 ########################################
 8 num_cars = 240 # number of cars on the road(10% = 30 cars, 25% = 75 cars, 50% = 150 cars, 80% = 240 cars)
 9 p = 0.2 #randomizer probiblity
10 num_cells = 300
11
12 v_max = 5 #for one lane only
13 left_vmax = 5
14 right_vmax = 4
15
16 ########################################
17 #               CAR CLASS
18 ########################################
19 class car:
20     def __init__(self, curr_vel, position):
21         self.curr_vel = curr_vel
22         self.position = position
23
24 ########################################
25 #             INITALIZATION
26 ########################################
27 matrix = np.full((100, num_cells), -1)
28 matrix2 = np.full((100, num_cells), -1)
29
30 def init_road(num_cells):
31   cells = []
32   for i in range(num_cells):
33     cells.append(i)
34   return cells
35
36 def add_cars(num_cars, cells, num_cells):
37   count = 0
38   while count != num_cars:
39     spot = random.randint(0, num_cells - 1)
40     if isinstance(cells[spot], int):
41         cells[spot] = car(random.randint(0, 5), spot)
42         count = count + 1
43   return cells
```

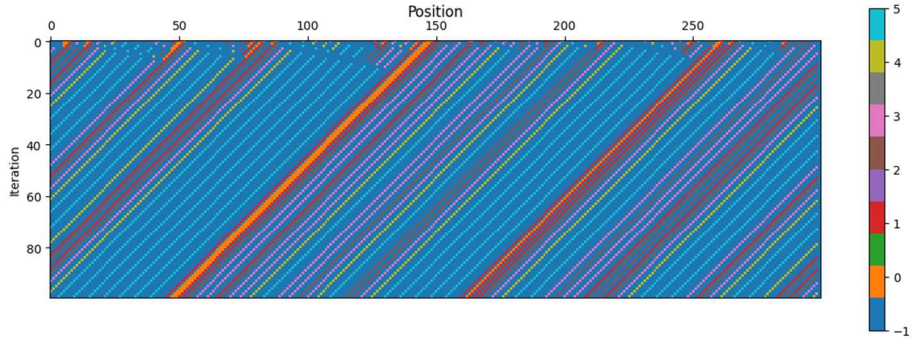
1.2 – Plots for Different Occupancies

To generate these plots, I put the velocities in a matrix and plotted the matrix. I initialized the matrix with -1 because the cars will never have a velocity of -1.

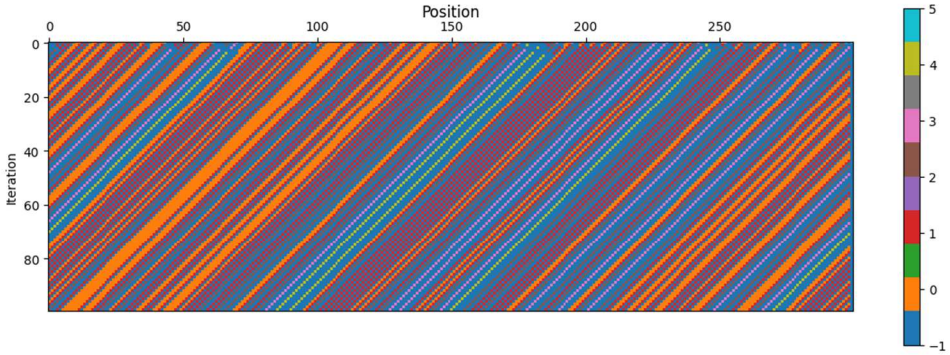
10% occupancy (30 cars)



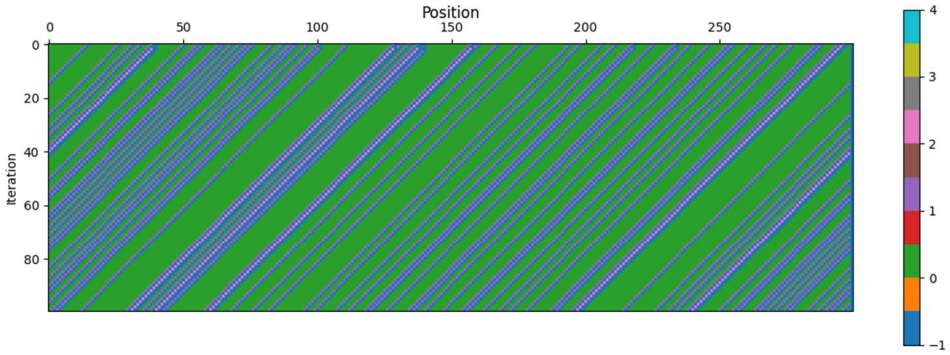
25% occupancy (75 cars)



50% occupancy (150 cars)



80% occupancy (240 cars)

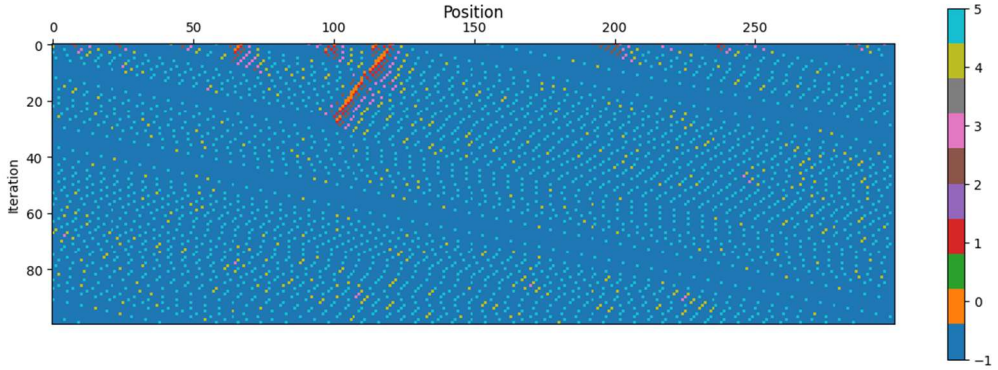


The plots above match the claims of 8.2. More specifically, when for an occupancy of 10%, we see that after a few iterations, all the cars reach the maximum velocity. For occupancies higher than 18% (1/6 of all cells occupied), we see traffic jams form and they move uniformly for the rest of the simulation. This is the case for the 25%, 50%, and 75% case. In the 25% case, there are still a few cars that are able to go full speed, but by 75% occupancy, we do not see this anymore.

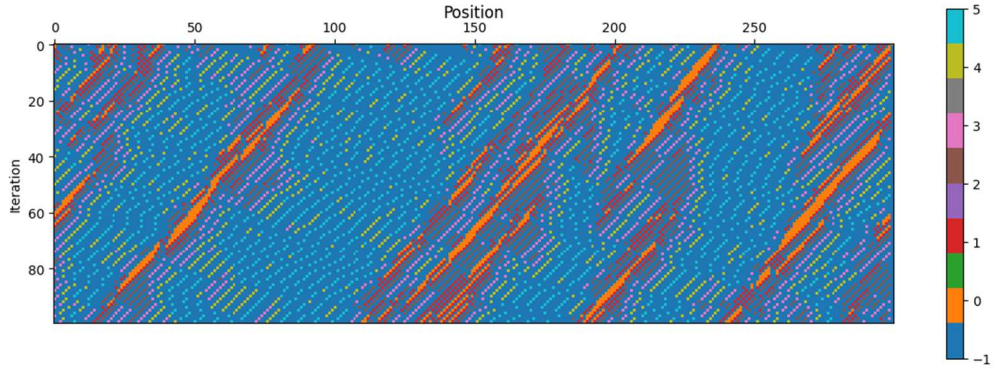
Task 2: Stochastic Behavior

P = 0.1

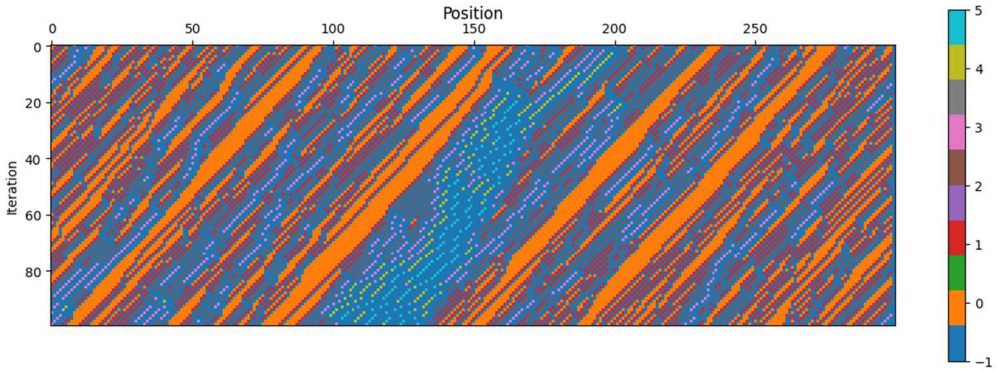
Occupancy = 10%



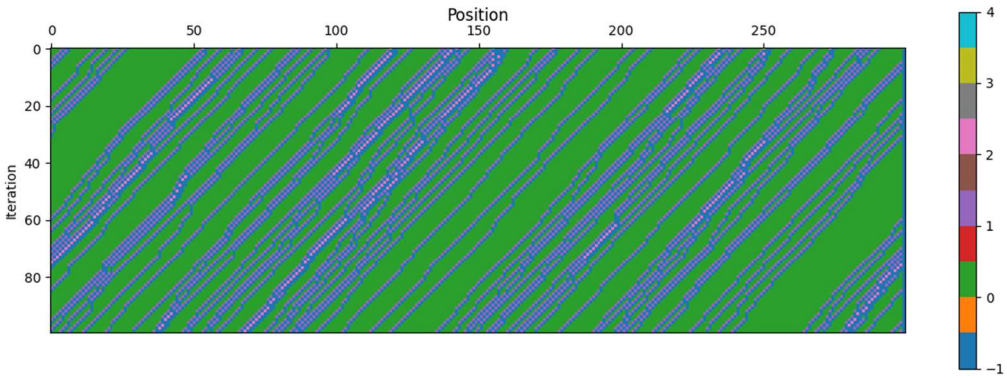
Occupancy = 25%



Occupancy = 50%

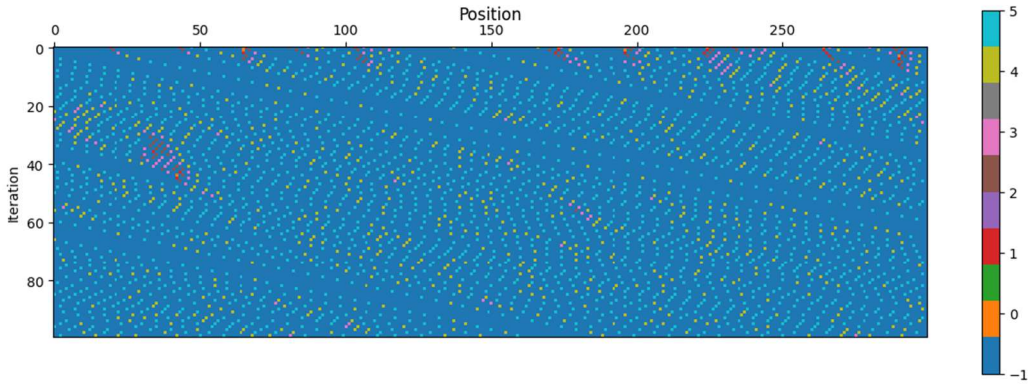


Occupancy = 80%

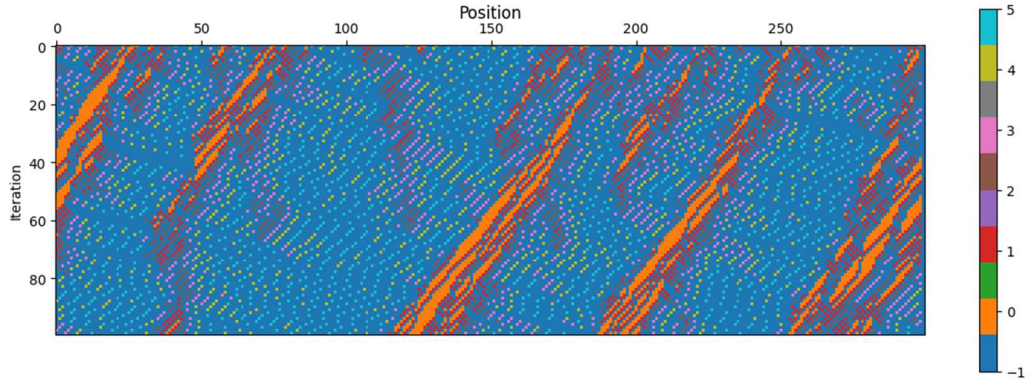


P = 0.2

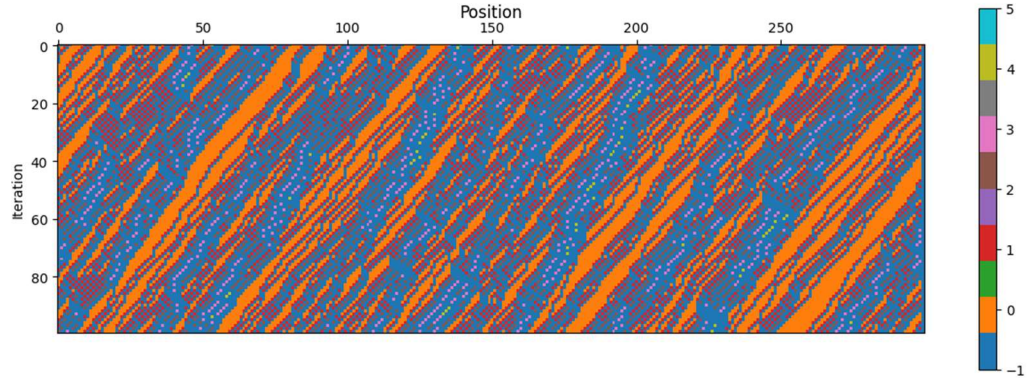
occupancy = 10%



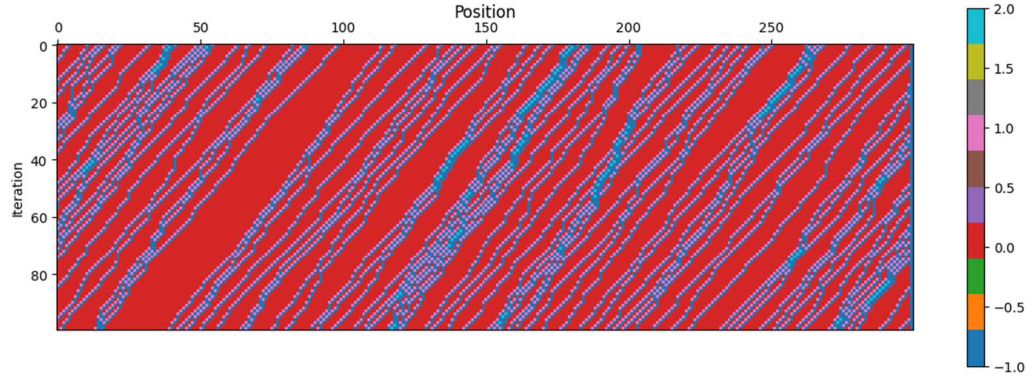
occupancy = 25%



occupancy = 50%

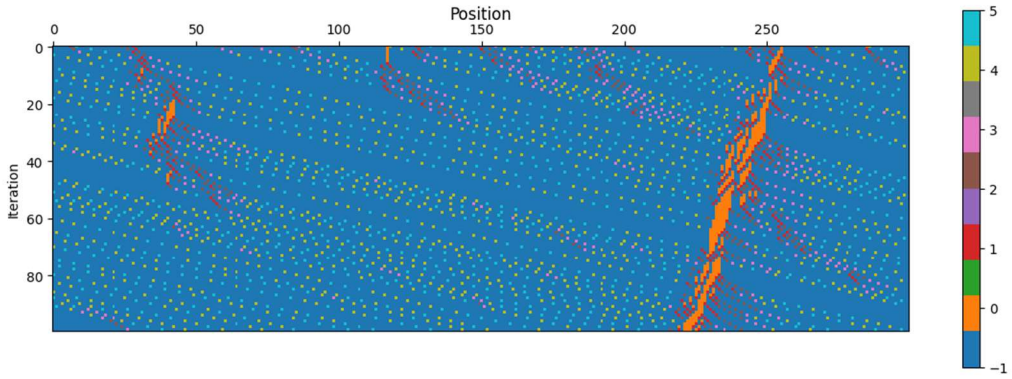


occupancy = *80%*

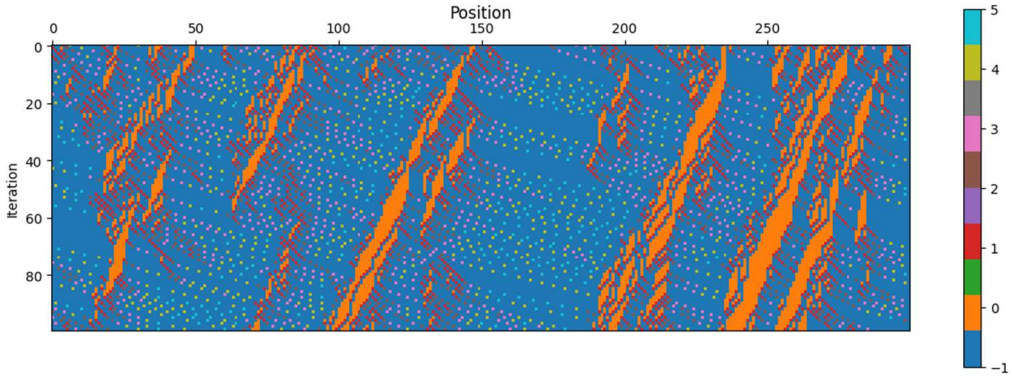


P = 0.5

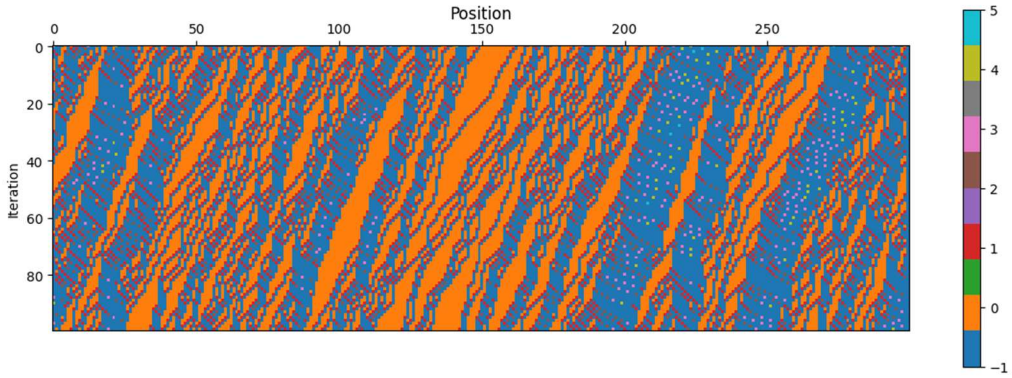
occupancy = 10%



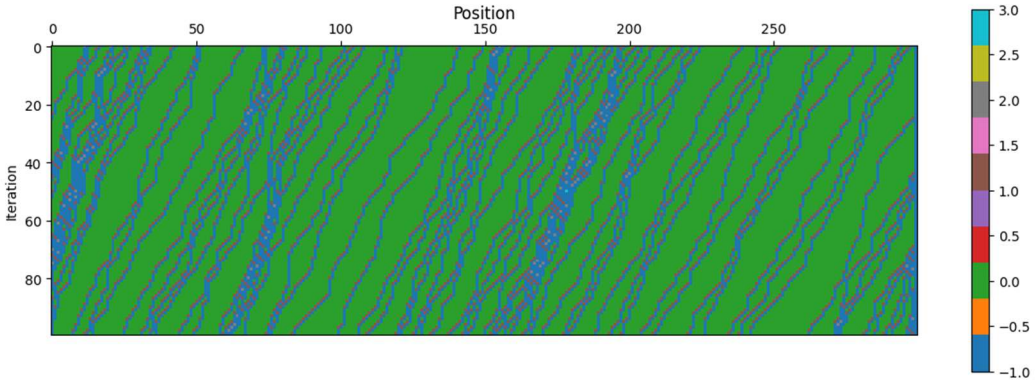
occupancy = 25%



occupancy = 50%



occupancy = 80%



1- Dally Factor of p=0.2 and Observations

At first glance, adding a dally factor or 0.2 gets rid of the uniform look for all occupancies. There looks to be random traffic jams at 10% and it increases as the occupancy rate increases.

2 - Experiments

All of the graphs in the first CA model tend to be very uniform: there are diagonals of the same speeds that form over the course of 100 iterations. This indicates portions of traffic traveling uniformly across the road.

However, with p=0.1and 10% occupancy, we see that there's occasionally a traffic jam. This is different from the first CA model because there was never a traffic jam beyond the first iterations. In higher occupancies, we see a decreasing number of cars that are able to travel at the speed limit. There are also a higher number of traffic jams the higher the occupancy.

A similar trend can be seen with p=0.2 and p=0.5. With p=0.2, we see the start of more traffic jams in 10% occupancy and this continues for higher occupancies. More notably, we see that in 80% occupancy with p=0.2, most of the cars have a velocity of 0. This trend continues in p=0.3.

Figure 8.5 from the reading has p=0.2 and 16% occupancy. My graphs of 10% and 25% occupancy and p=0.2 look most like figure 8.5. In both of these graphs, we see traffic jams out of nowhere that dissolve.

Figure 8.6 has p=0.5 and 16% occupancy. This graph has regions of very high traffic density and regions with very low density. We see this beginning of this in my p=0.5, 10% occupancy figure and more clearly in the 25% occupancy figure. It kind of looks like mountains with stretches of land in between.

Figure 8.7 has p=0.2 and 25% occupancy. My p=0.2, 25% occupancy looks quite similar. In both figures, we see that the traffic jams are more persistent and do not readily dissolve after a few iterations. These can be seen by the long stretches of orange.

Task 4: Two Lane Traffic

Algorithm

Update for vehicle i:

1. **Accelerate**: $v_i = \min\{v_i + 1, v_{\_MAX\_LANE}\}$ # same as before
2. **Change lanes**:

```
If v_i > d(i, i+1), then check_both_lanes() # deceleration condition
given in textbook
   if check_both_lanes == True
       switch lanes
   else
        don't switch lanes

def check_both_lanes():
   check if v_i cells forward are free
   check if v_MAX_LANE cells backward are free
       for i until v_MAX_LANE:
         if car.speed in cell_i = i-1:
           return false
   return true
```

3. **Decelerate**: $v_i = d(i,i+1)$, if $v_i > d(i,i+1)$ # same as before
4. **Move vechicle**: i moves v_i cells forward # same as before

Explanation

The first, third, and fourth steps are the same as the one-lane case except the v_max depends on the lane.

The biggest change was in step 2. If the car needs to decelerate, then it first checks to see if it's safe to switch lanes. It first checks v_i cells forward in the other lane to see if it is free. The car then starts checking the $v_{\_MAX\_LANE}$ cells behind to see if there's a possible collision with the cars behind. For example, if $v_{\_MAX\_LANE}$ is 4, then it checks 4 cells immediately behind. If the immediate cell contains a car with a speed of 0, then the current car cannot switch because we must allow that car to accelerate and move. If the cell behind that one contains a car with a speed of 1, then the current car cannot switch. We repeat this until we get to the $v_{\_MAX\_LANE}$ cells behind our current car. If forward and behind are free, then the car switches lanes. Otherwise, it stays in the same lane

Implementation & Verification

To create two lanes, I created two lists and initialized a random number of cars in each lane. Each car was initialized at a random spot and with random velocity. To implement the first, third, and fourth steps, I simply passed in the corresponding parameters for each lane. (i.e, left_vmax vs right_vmax).

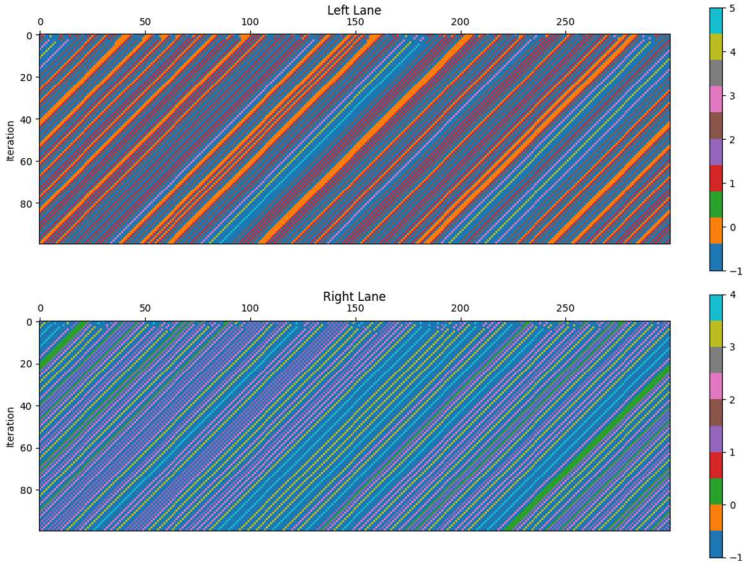
To verify the lane switching logic, I first tested out cases on paper to see if the logic held up. If it did not, then I went back to fix the issue. An initial mistake I had while coding was modding by 299. However, since the index starts at 0 and ends at 299, this meant that it never got to index 299. In addition, I assumed that there were an even number of cars on each lane and that the cars were spread out randomly at random speeds. Lastly, I tested my code on a smaller scale (fewer cells, fewer cars, fewer iterations) and verified that the plotting was correct.

Observations

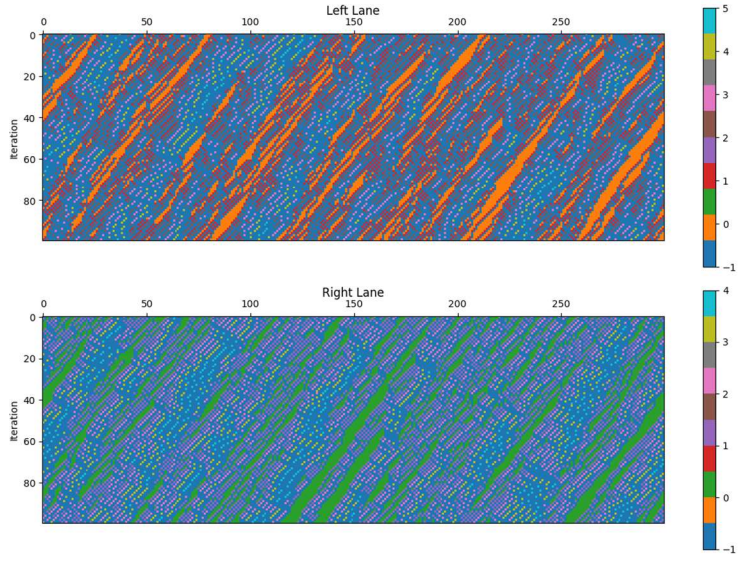
In the single-lane CA model, all vehicles are confined to a single lane, which means that any slow-moving vehicle or blockage affects all following vehicles, leading to potential traffic jams. The congestion in this model is made worse by the fact that vehicles have no alternative routes or lanes to bypass slower traffic, thus any variability in speed or sudden stops can quickly lead to a buildup of cars.

Introducing a second lane changes the dynamics of traffic flow. With two lanes available, the number of cars in each lane is roughly halved, assuming an even distribution of vehicles across both lanes. This reduction in vehicle density per lane directly contributes to a decrease in the likelihood of traffic jams. One reason is that a second lane allows faster vehicles to overtake slower ones, which can help prevent the formation of "platoons" of cars that often lead to traffic congestion. This overtaking mechanism is crucial for maintaining a smoother flow of traffic, as it enables vehicles to move more freely and at speeds closer to their desired speed.

In addition, with a dally factor of 0, the cars tend to make most of the switches toward the earlier iterations. After a couple of iterations, the traffic stabilizes and looks like the first CA model.



With a dally factor > 0, we get a graph that looks like this:



After experimenting with different numbers of cars and dally factors, it seems to reflect the one lane situation. From the textbook, adding the dally factor accounts for three different behaviors: delay when accelerating, delay on an open road, overreaction when decelerating. There's sections where cars are free to accelerate and shortly after, the cars must decelerate when they encounter a traffic jam.