Traveling salesman and disease modeling assignment

1.1 Optimization Problem:

Source of image: https://www.nationsonline.org/oneworld/map/liberia-map.htm

Scenario 1

Disease: Ebola

The problem is distributing vaccines to 15 cities in Liberia (county capitals) while the objective function is minimizing time taken, as we don't want to give the disease time to spread. I will assume a linear relationship between distance traveled and time taken, allowing the objective function (time in hours) to be a function of distance (meters).

In the real world there might be constraints such as fuel, traversability of path, or available mode of transport. I'll assume neither of these are constraints as including them would add too much complexity, weakening the inferences we can make from the result. The distances between cities would be calculated beforehand. This problem is very similar to the well known traveling salesman problem (TSP), the difference being that we can revisit cities and we also don't care about the journey from the final destination back to the start city. Technically we don't need to create a cycle, just an open path (not returning to start), but for the sake of generality I will create an algorithm for the TSP. The decision variable is the order in which we visit the cities, the path. The path will impact the distance we need to travel, since some paths are longer than others, eg a zig-zag vs square path. The objective function would be measured by adding all the known distances for traveling between cities.

1.2 Optimization Technique:

Ant Colony Optimization (ACO) for the Traveling Salesman Problem-Optimization

Problem (TSP-OP)

I'm being specific saying TSP-OP instead of TSP, because TSP could refer to a decision or search problem, for example "Is there a path shorter than x" (YES/NO), or "find a path shorter than x" (PATH). Both of these have results which are trivial to validate. The problem here, TSP-OP, is non-trivial to validate, as knowing if the found path is the shortest is as difficult as the original problem. For TSP, we can't guarantee global minima unless performing a costly a brute force search.

The ACO algorithm is an algorithm that finds near optimal solutions for the TSP-OP. Yet we can't guarantee the global minima. The input for the algorithm is a graph and the weights for each edges, which here will be the euclidian distance between the nodes in the plane. The output will be the path and it's distance. The steps toward the solution can also be seen as an output, as the "certainty" of the algorithm is represented by how quick it finds the solution as well as the

Defining functions

All code below has been fully created by me and me alone

In [44]:
```
# Run this cell to install networkx and plotly library
# Alternatively run "pip install networkx" in your cmd promt
import sys
!{sys.executable} -m pip install networkx
!{sys.executable} -m pip install plotly
```

Importing Libraries

In [1]:
```
# For Visualizations
from plotly.subplots import make_subplots
from IPython.display import clear_output
from matplotlib.pyplot import figure
import plotly.graph_objects as go
import matplotlib.pyplot as plt
import plotly.express as px
from ipywidgets import *

# Library for manipulating graphs
import networkx as nx

# Functional libraries
import random as rnd
import numpy as np
import math
import time
```

Defining Functions for ACO

In [2]:

```python
# Create Network
def CreateNetwork(amountNodes, createNew):
    # Create graph, position nodes randomly
    if createNew:
        G = nx.complete_graph(amountNodes)
        _pos = nx.random_layout(G)
    else:
        # Set graph and positions to global variables to copy last graph
        # Used when doing multiple simulations on the same graph
        G = network
        _pos = pos

    # Give each edge 2 attributes, weight and pheromone
    for edge in G.edges():
        startnode=edge[0]
        endnode=edge[1]

        if createNew:
            # Set weight as euclidian distance according to pos
            distances = round(math.sqrt(((_pos[endnode][1]-_pos[startnode][1])**2)+
                            ((_pos[endnode][0]-_pos[startnode][0])**2)),2)
            # Set pheromones to 1
            G.add_edge(startnode, endnode, distance=distances, pheromone=1)
        else:
            # If not creating new, reset pheromones, distance will already be set
            G.add_edge(startnode, endnode, pheromone=1)

    # Return the network, and nodes positions
    return G, _pos

def GetRandomPathRec(path):
    # If path contains the whole network
    #     return
    if len(path) == len(network.nodes):
        return path

    # Continue from last node
    currentNode = path[-1]
    # Available neighbors
    possibleNext = []
    # Neighbors individual probability weight for selection
    probArr = []

    # Append each available neighbor
    for neighbor in list(network[currentNode]):
        if neighbor not in path:
            possibleNext.append(neighbor)
            # Calculate individual prob using TraverseProb()
            probArr.append(TraverseProb(currentNode, neighbor))

    # Randomly select one neighbor
    selection = rnd.random() * sum(probArr)
    currI = -1
    while selection > 0:
        currI += 1
        selection -= probArr[currI]

    # Append selected neighbor
    path.append(possibleNext[currI])

    # Recursively return the rest of the path
    return GetRandomPathRec(path)

def LengthOfPath(path):
    # Calculate length of path
    length = 0
    # Increment with each distance of each edge
    for i in range(len(path)):
        length += network.edges[path[i], path[(i + 1) % len(path)]]['distance']
    return length

```

```
def IncrementPheromones(path, value):
    # Increment pheromone in each edge from path with value
    for i in range(len(path)):
        network.edges[path[i], path[(i + 1) % len(path)]]['pheromone'] += value

def TraverseProb(nodeA, nodeB):
    # distance^-distBias * pheromone^pheroBias
    # (-)distBias because more distance decrease probability
    # We want bias toward closer nodes
    return (pow(network.edges[nodeA, nodeB]['distance'], -distBias) *
            pow(network.edges[nodeA, nodeB]['pheromone'], pheroBias))
```

Defining Functions for visualizations

In [3]:

```python
# Draw the network
def DrawNetwork(pos, weights=[], text=""):
    # Set size bounds for displaying network
    figure(figsize=(10, 8), dpi=80)

    # get the phermone of each edge
    if weights == []:
        # If weights not given, get weights of current network
        weights = nx.get_edge_attributes(network, 'pheromone')

    # Width of edge correspond to relative pheromone level
    widths = np.array(list(weights.values()))

    # Normalize widths
    widths /= max(widths)
    widths *= maxThickness

    # Get nodes
    nodelist = network.nodes()

    # Draw nodes
    nx.draw_networkx_nodes(network,pos,
                           nodelist=nodelist,
                           node_size=300,
                           node_color='red',
                           alpha=1)
    # Draw Edges
    nx.draw_networkx_edges(network,pos,
                           edgelist = weights.keys(),
                           width=widths,
                           edge_color='black',
                           alpha=1)

    # Draw digits on nodes
    nx.draw_networkx_labels(network, pos, font_size=10, font_color='white')

    # Update title
    if text:
        plt.title(text)

    plt.box(False)
    plt.show()

def DisplayNetworkAnims(pos, index, snapshotInterval, doAnimation=True):
    # Update function for interactive animation (graph).
    # I put the Update function inside DisplayNetworkAnims() to get proper variable scope
    # as this function should only be accesible in this function
    def Update(step):
        DrawNetwork(pos, pheroHistories[index][step],
                    f"Simulation {chr(index +65)}\nAnts simulated: {step * snapshotInterval}")

    if doAnimation:
        # Display animation
        for i in range(len(pheroHistories[index])):
            currentAnt = i * snapshotInterval # How many ants have been simulated at snapshot
            clear_output(wait=True)           # Clear output
            DrawNetwork(pos, pheroHistories[index][i], f"Ants simulated: {currentAnt}") # Redra
            time.sleep(.1) # Make it animate slowly
        time.sleep(1)
        clear_output(wait=True)

    # interact() creates a interactive slider connected to the fuunction Update
    print("You can click the slider and then use the left/right arrows to step through the proc
    interact(Update, step=widgets.IntSlider(min=0, max=len(pheroHistories[index]) - 1, step=1,

# Draws a line graph describing the performance of the algorithm
def PerformanceGraph(antLen, average, bestLen):
    # Create subplots depending on how many simulations have been performed (len(antLen) == amo
    # Have 2 graphs per row if simulations > 1
    fig = make_subplots(rows=math.ceil(len(antLen) / 2), cols=1 if len(antLen) == 1 else 2)
    avgFig = px.scatter()
```

```
72
73      # Boolean for displalying legend
74      displayLegend = True
75
76      for i in range(len(antLen)):
77          # Only display legend for first graph
78          if i == 1:
79              displayLegend = False
80
81          # Calculate current position among graphs
82          rowPos = (i // 2) + 1
83          colPos = i % 2 + 1
84
85          # Set titles for each graph
86          fig.update_yaxes(title_text="<b>Length of path</b><br>Unit is (100 km)s", row=rowPos, c
87          fig.update_xaxes(title_text="<b>Ants simulated</b>", row=rowPos, col=colPos)
88
89          # Add trace for each line
90          # Each ants traversed distance
91          fig.add_trace(go.Scatter(y=antLen[i],
92                                   mode='lines',
93                                   name='Path length for each ant',
94                                   line_color='royalblue',
95                                   showlegend=displayLegend),
96                        row = rowPos, col=colPos)
97
98          # Shortest path so far
99          fig.add_trace(go.Scatter(y=bestLen[i],
100                                  mode='lines',
101                                  name='Shortest path so far',
102                                  line_color='lightgreen',
103                                  showlegend=displayLegend),
104                       row = rowPos, col=colPos)
105
106         # Critical termination line at 5% over the shortest path
107         fig.add_trace(go.Scatter(y=np.array(bestLen[i])*(1+terminationLine),
108                                  name='5% over shortest path',
109                                  line=dict(color='green', dash='dash'),
110                                  showlegend=displayLegend),
111                       row = rowPos, col=colPos)
112
113         # Filled green area for shortest length
114         fig.add_trace(go.Scatter(
115                                  x=list(np.array(range(len(bestLen[i])))) +
116                                      list(np.array(range(len(bestLen[i]))))[::-1],
117                                  y=list(np.array(bestLen[i])*(1+terminationLine))+list(np.array(best
118                                  fill='toself',
119                                  fillcolor='rgba(107,231,103,0.3)',
120                                  line_color='rgba(255,255,255,0)',
121                                  showlegend=displayLegend,
122                                  name='5% region'),
123                       row = rowPos, col=colPos)
124
125         # Running average of traversed distance
126         fig.add_trace(go.Scatter(x=np.array(range(len(antLen[i]))) + runningAvgOf//2,
127                                  y=average[i],
128                                  mode='lines',
129                                  name=f'Running average of path length (average of {runningAvgOf
130                                  line_color='red',
131                                  showlegend=displayLegend),
132                       row = rowPos,
133                       col=colPos)
134
135         # Assign letter to each graph
136         fig.add_annotation(text=chr(i + 65),
137               xref="x domain", yref="y domain",
138               x=0.95, y=0.95, showarrow=False, row = rowPos, col=colPos, font_size=25)
139
140         # Only do running average graph if there are multiple simulations to compare
141         if len(average) > 1:
142             # avgFig is graph displaying each graphs running average
```

```
            avgFig.add_trace(go.Scatter(x=np.array(range(len(antLen[i]))) + runningAvgOf//2,
                                        y=average[i],
                                        mode='lines+text',
                                        name=f'Sim {chr(i + 65)} running average'))

    # Layout settings for performance graphs
    fig.update_layout(height=400*(math.ceil(len(antLen) / 2) + 1),
                      width=1000,
                      legend=dict(x=0, yanchor="bottom", y=-.15),
                      title_text='''<b>Linegraph showing performance of algorithm</b><br>
                                    Distance of paths over amount of ants simulated''')
    fig.show()

    # Only do running average graph if there are multiple simulations to compare
    if len(average) > 1:
        # Set titles for running average graph
        avgFig.update_yaxes(title_text="<b>Running average of paths</b><br>Unit is (100 km)s")
        avgFig.update_xaxes(title_text="<b>Ants simulated</b>")
        avgFig.update_layout(title_text='''<b>Linegraph showing running average of path length
                                           over amount of ants simulated</b>''')

        avgFig.show()
```

Algorithm

In [4]:

```
# Function for running ACO algorithm
def RunACO(nodesAmount,
           distBias,
           pheroBias,
           evaporationRate,
           pheromoneRate,
           simulations = 1,
           newNetwork=True):

    # It is in general a bad practice to make variables global,
    # as that might lead to unintended access/interactions with the variables,
    # which can lead to logical errors which are difficult to find.
    # I still choose to globalize these variables as to
    # avoid being forced to tunnel them to the functions as parameters,
    # this increases the readability of the code
    global network
    global pos

    # These lists continuously store the relevant data for analysing the performance of the alg
    # I globalize these variables for same reason as stated above.
    global pheroHistories
    global maxThickness
    global antLenHistory
    global bestLenHistory
    global pheroHistory
    global runningAvg

    pheroHistories = []
    globAntLenHistory = []
    globBestLenHistory = []
    globRunningAvgHistory = []
    bestLengths = []
    bestPaths = []

    # Defining visual parameters
    # All data later visualized are stored every 'snapshotInterval' ant
    snapshotInterval=50
    # Max thickness of the lines in the network
    maxThickness = 8
    # Set fontsize for network visualization
    plt.rcParams.update({'font.size': 20})

    # Globalized termination line for same reason as stated above
    global terminationLine
    # When the running average distance of the ants hit the line which is
    # 'terminationLine' % above the shortest distance, it terminates. (bestL * (1+terminationLi
    terminationLine = .05

    # Set minimum amount of ants to simulate, in the beginning of the simulation,
    # the termination condition might be fulfilled, as the best path so far can be very high.
    # Therefore we don't want to terminate if less than 'minimumAnts' have been smulated
    minimumAnts = 100
    # If the algorithm doesn't converge, we wish to terminate after 'maxAmountAnts' no matter wh
    maxAmountAnts = 10000

    # A runningAvgOf 50 means that each value in the list runningAvg is the average of the 50 pr
    global runningAvgOf
    runningAvgOf=50

    # For each simulation
    for simiteration in range(simulations):

        # The current 'runningAvgOf' values to average
        # Works like a que, will be updated to always contain the most
        # recent 'runningAvgOf' values from antLenHistory
        currentFocus = []

        # Storing pheromone values
        pheroHistory = []
        # Storing length of best path so far
        bestLenHistory = []
```

```
        # Storing length of each path created
        antLenHistory = []
        # Storing the running average of path lengths
        runningAvg = []
        # Store what the shortest path is
        # Stores the order of the cities, listed by their index
        bestPath = []

        # Create network, only create new network first simulation
        network, pos = CreateNetwork(nodesAmount, simiteration == 0)

        # Set best length as large
        bestL = 1000
        for i in range(maxAmountAnts + 1):

            # Evaporate pheromones each iteration
            for edge in network.edges():
                network.edges[edge]['pheromone'] *= (1 - evaporationRate)

            # Put ant at random node
            # Make ant go random path
            path = GetRandomPathRec([rnd.randrange(nodesAmount)])
            pathL = LengthOfPath(path)

            # Fill up currentFocus until 'runningAvgOf' is reached
            if i < runningAvgOf:
                currentFocus.append(pathL)
            else:
                # After the first iterations
                # Add the average to runningAvg
                runningAvg.append(sum(currentFocus)/len(currentFocus))
                # Add the new value to currentFocus
                currentFocus.append(pathL)
                # Remove the oldest value
                currentFocus = currentFocus[1:]

                # If the difference between the current runing average and the bestlength is
                # within the termination percentage (5% in this case),
                # and we've done at least 'minimumAnts' iterations,
                # Terminate
                if (runningAvg[-1] - bestL) / bestL < terminationLine and i > minimumAnts:
                    break

            # If current path is shorter than shortest path
            if pathL < bestL:
                # Store new best path
                bestL = pathL
                bestPath = path

            # Append current values to history lists
            bestLenHistory.append(bestL)
            antLenHistory.append(pathL)

            # Store snapshot of pheromones at intervals
            if i % snapshotInterval == 0:
                pheroHistory.append(nx.get_edge_attributes(network, 'pheromone'))

            # diff, how close to best seen solution
            # Adding .001 to avoid division by zero
            diff = pathL - bestL + .001
            IncrementPheromones(path, pheromoneRate/diff)

        # Store each history list in lists,
        # So we can draw performance graphs on each simulation's history
        globAntLenHistory.append(antLenHistory)
        globBestLenHistory.append(bestLenHistory)
        globRunningAvgHistory.append(runningAvg)

        pheroHistories.append(pheroHistory)
        bestLengths.append(round(bestL, 6))
        bestPaths.append(bestPath)
```

```
143
144         # Only do the animation if we're doing a single simulation
145         if simulations == 1:
146             DisplayNetworkAnims(pos, 0, snapshotInterval, True)
147 
148     # If simulating multiple times, draw each network, no animation
149     if simulations != 1:
150         for index in range(len(pheroHistories)):
151             DisplayNetworkAnims(pos, index, snapshotInterval, False)
152 
153     # Draw the performance graph, for each simulation
154     PerformanceGraph(globAntLenHistory,
155                      globRunningAvgHistory,
156                      globBestLenHistory)
157 
158     # Return the best paths and their lengths
159     return bestLengths, bestPaths
```

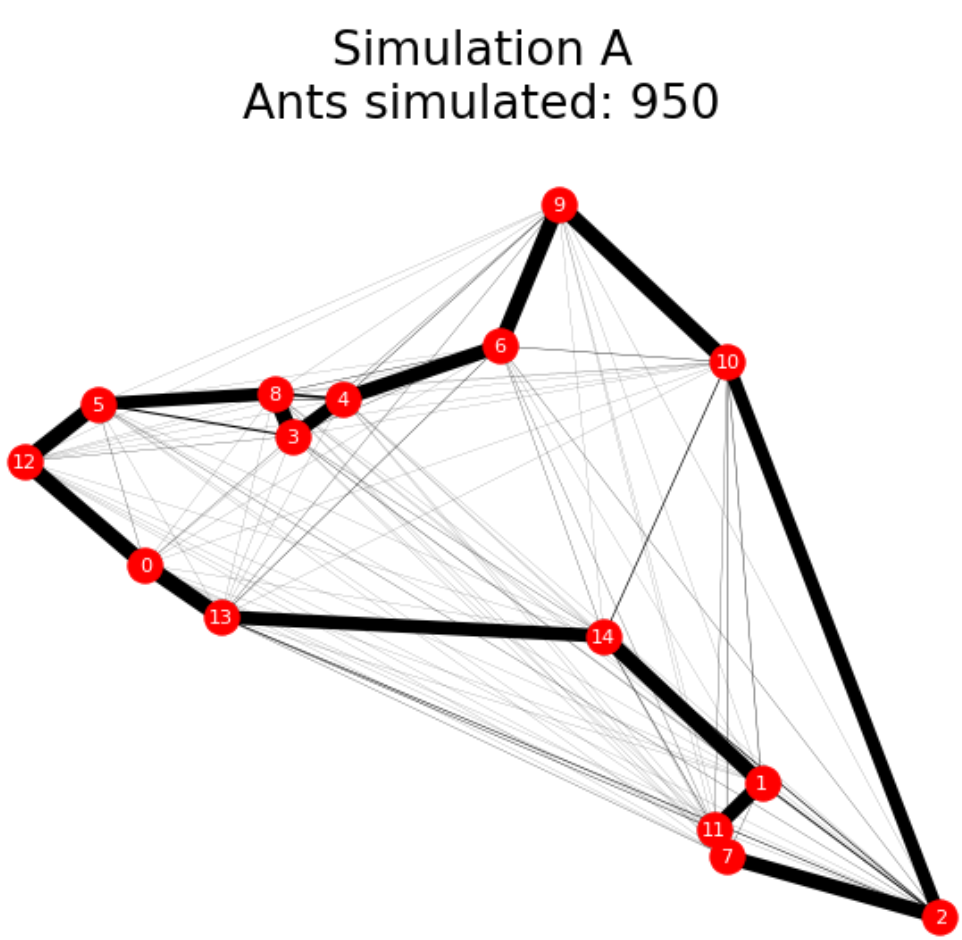
Function calls

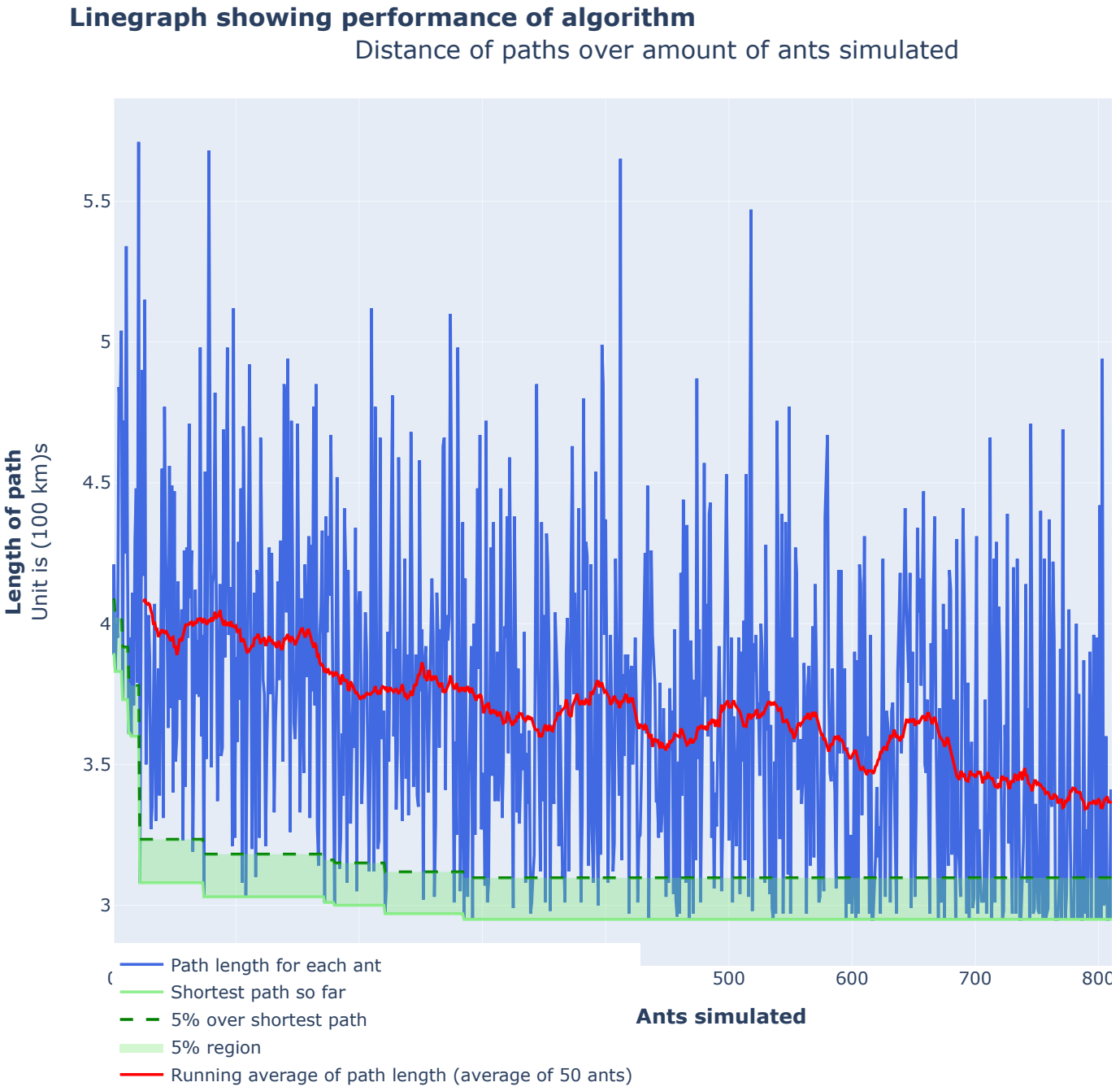
In [6]:

```
#Hyperparameters
nodesAmount=15

# Bias variables toward greedy and pheromone approach
distBias=2 # Increases bias toward choosing closer nodes
pheroBias=1 # Increase bias toward choosing pheromones, edges which have been traversed previous

# percentage of pheromones that evaporates each iteration
evaporationRate = .0001

# Coefficient for adding pheromones
pheromoneRate = .001

# Run this to see animation and one simulation
RunACO(nodesAmount,
       distBias,
       pheroBias,
       evaporationRate,
       pheromoneRate, 1)
```

```
You can click the slider and then use the left/right arrows to step through the process
```

step 19





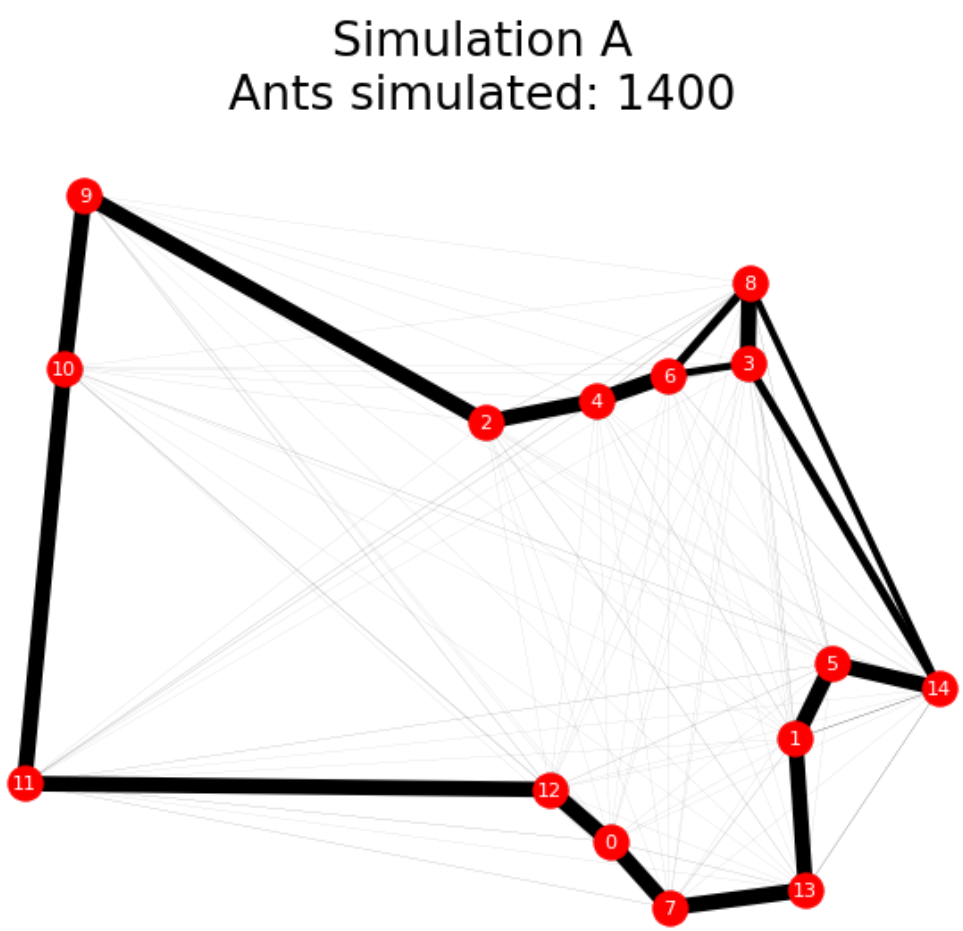
Out[6]: ([2.94], [[10, 9, 6, 4, 3, 8, 5, 12, 0, 13, 11, 7, 2, 1, 14]])

In [7]:

```
# How many simulations to run on the same graph
simAmount = 6

# Run this to see multiple simulations
RunACO(nodesAmount,
       distBias,
       pheroBias,
       evaporationRate,
       pheromoneRate, simAmount)
```

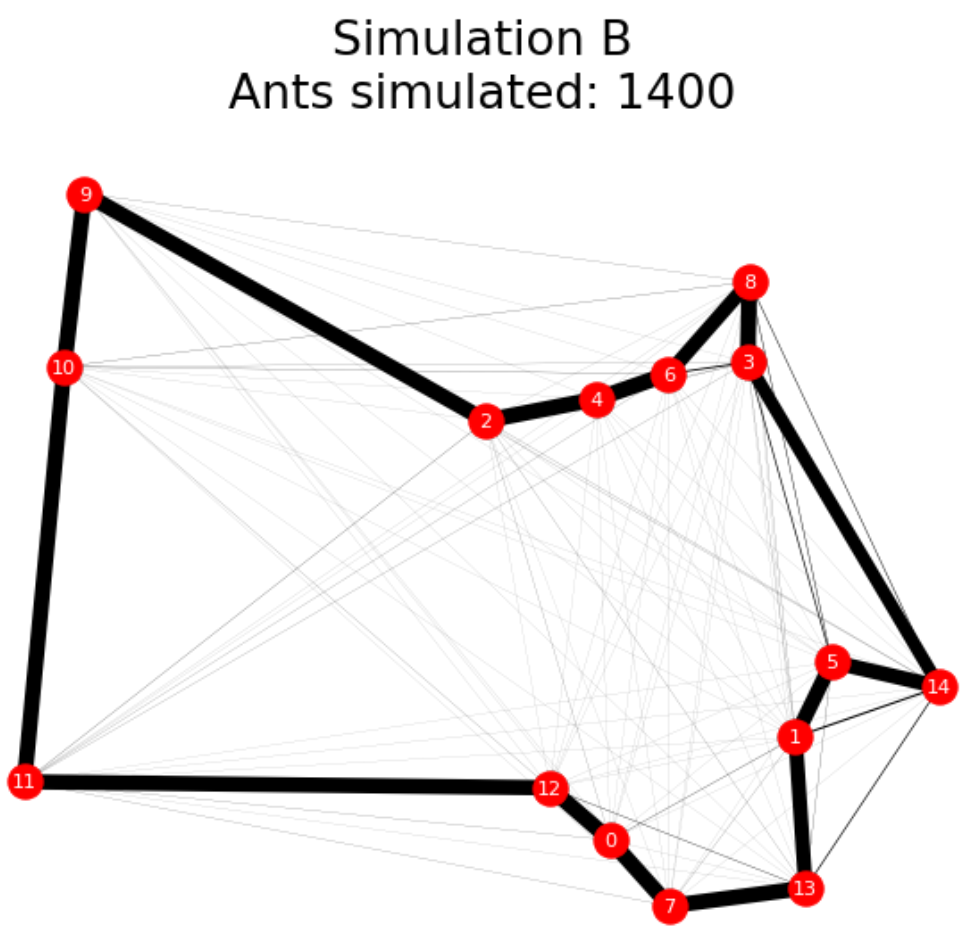
You can click the slider and then use the left/right arrows to step through the process

step 28



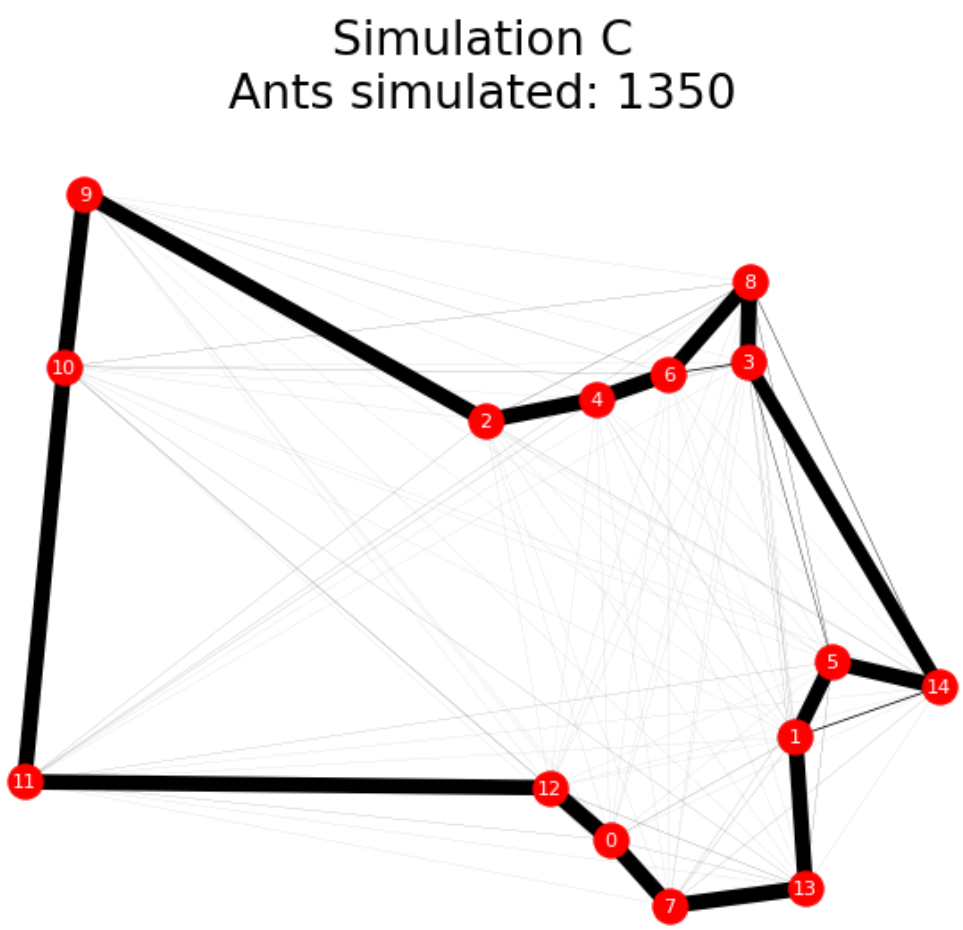
You can click the slider and then use the left/right arrows to step through the process

step 28



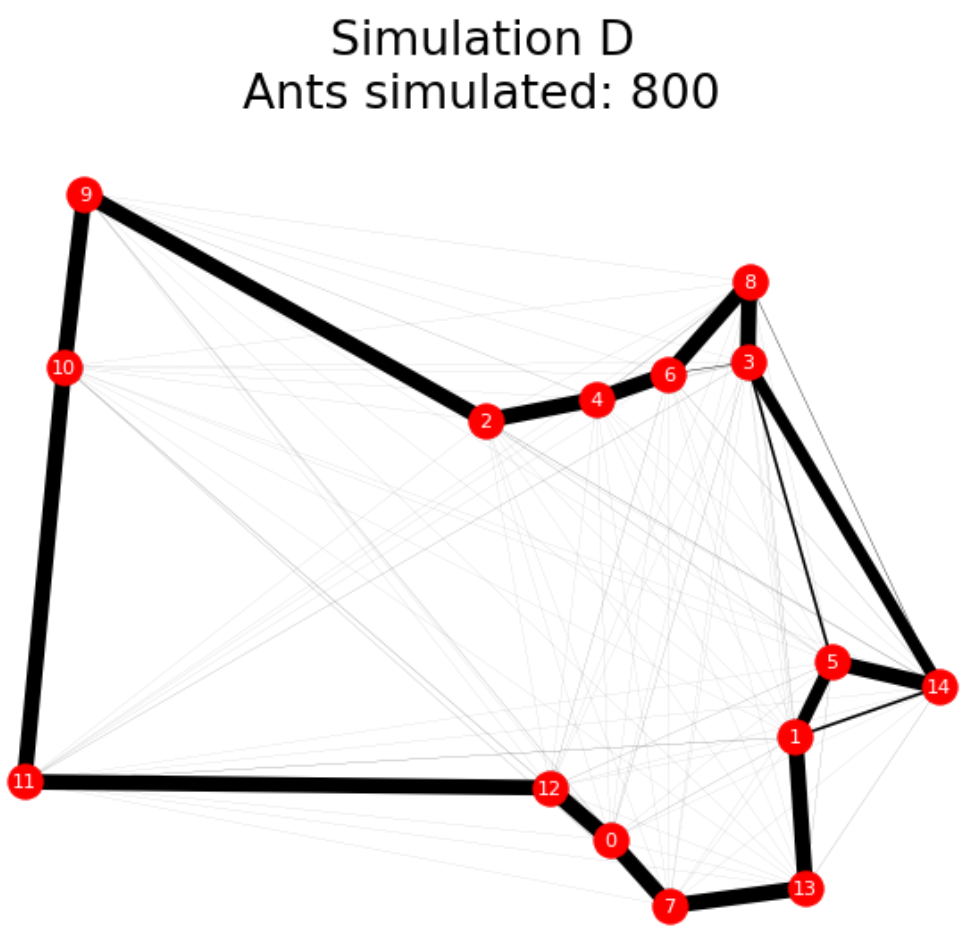
You can click the slider and then use the left/right arrows to step through the process

step 27



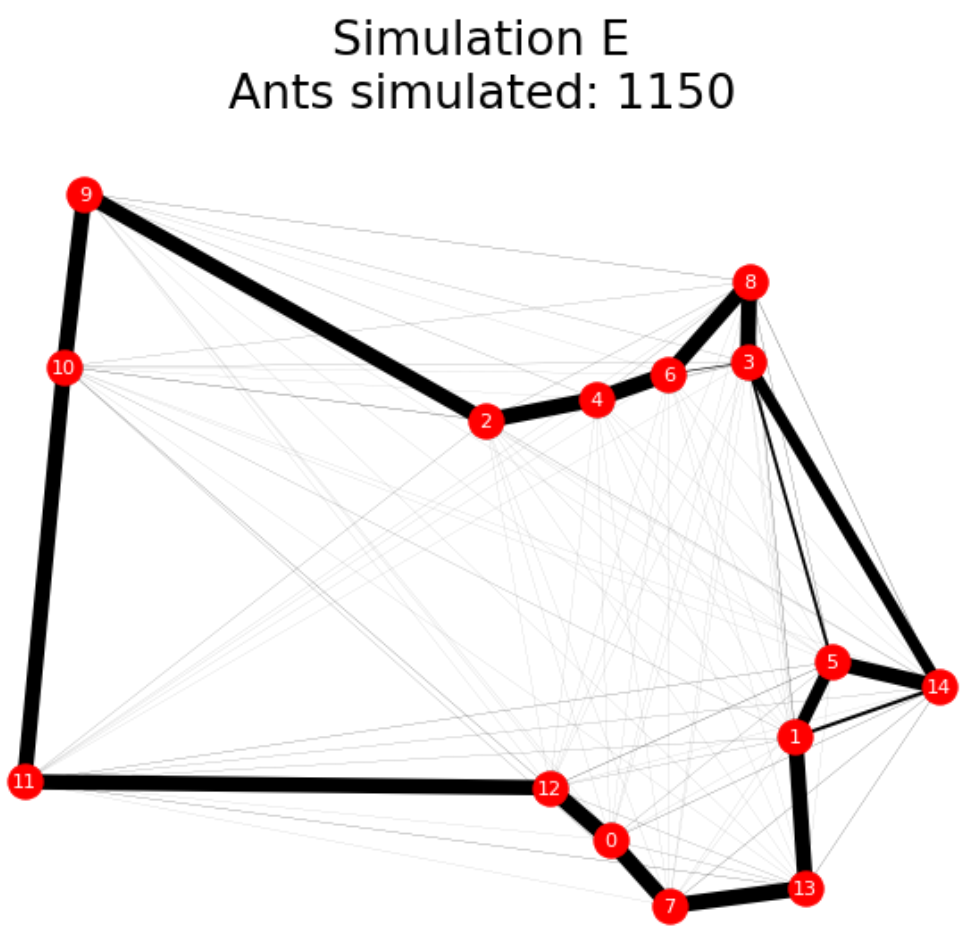
You can click the slider and then use the left/right arrows to step through the process

step 16



You can click the slider and then use the left/right arrows to step through the process

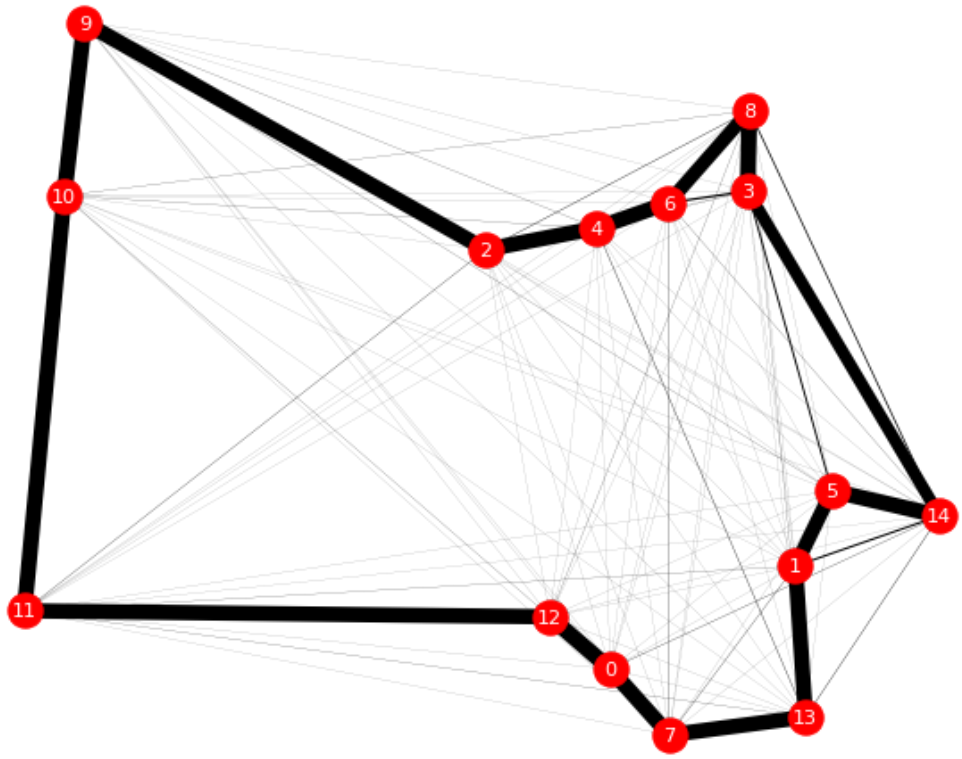
step 23

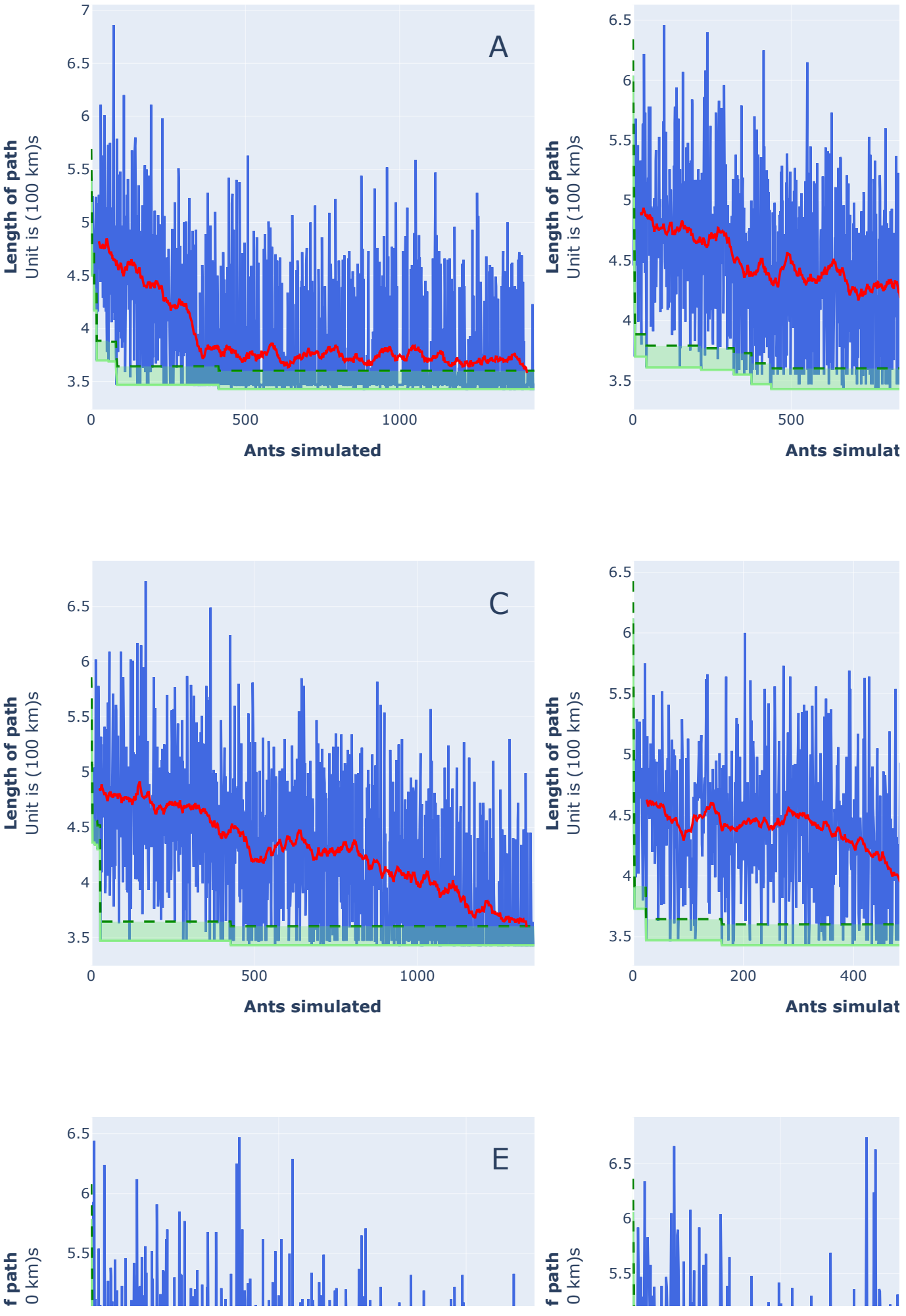


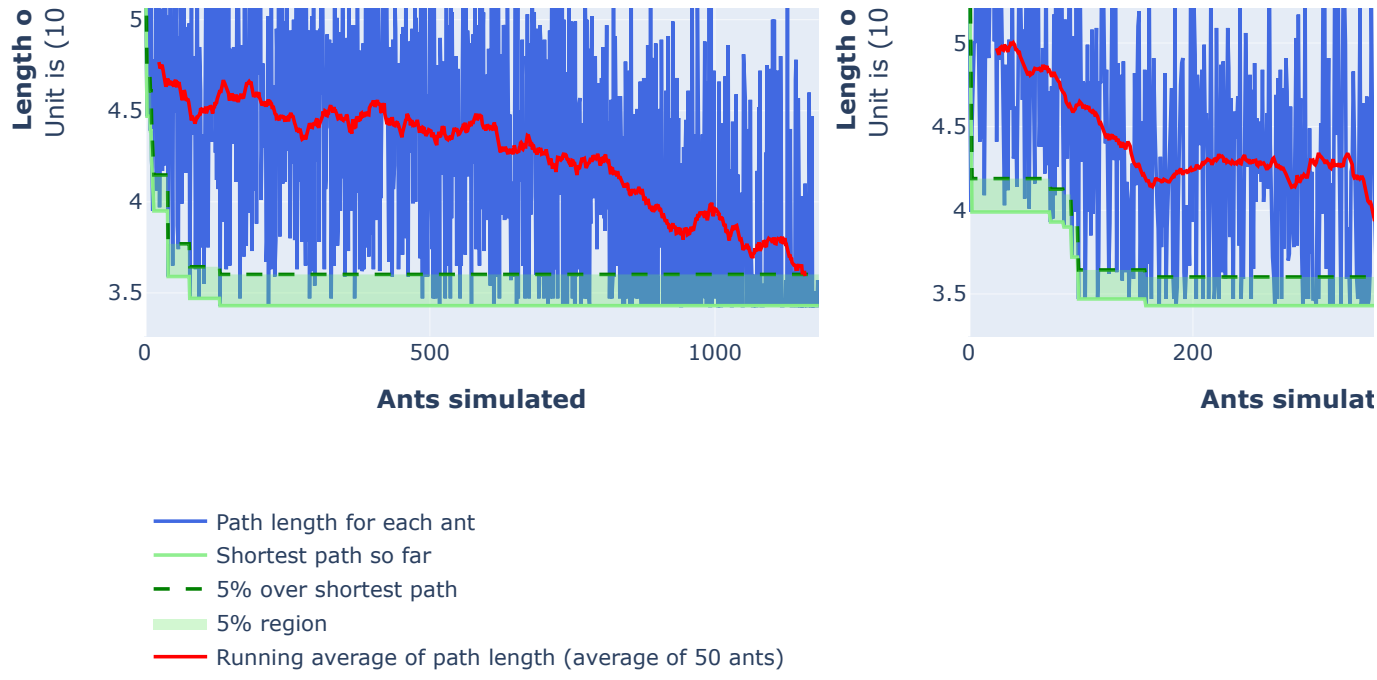
You can click the slider and then use the left/right arrows to step through the process

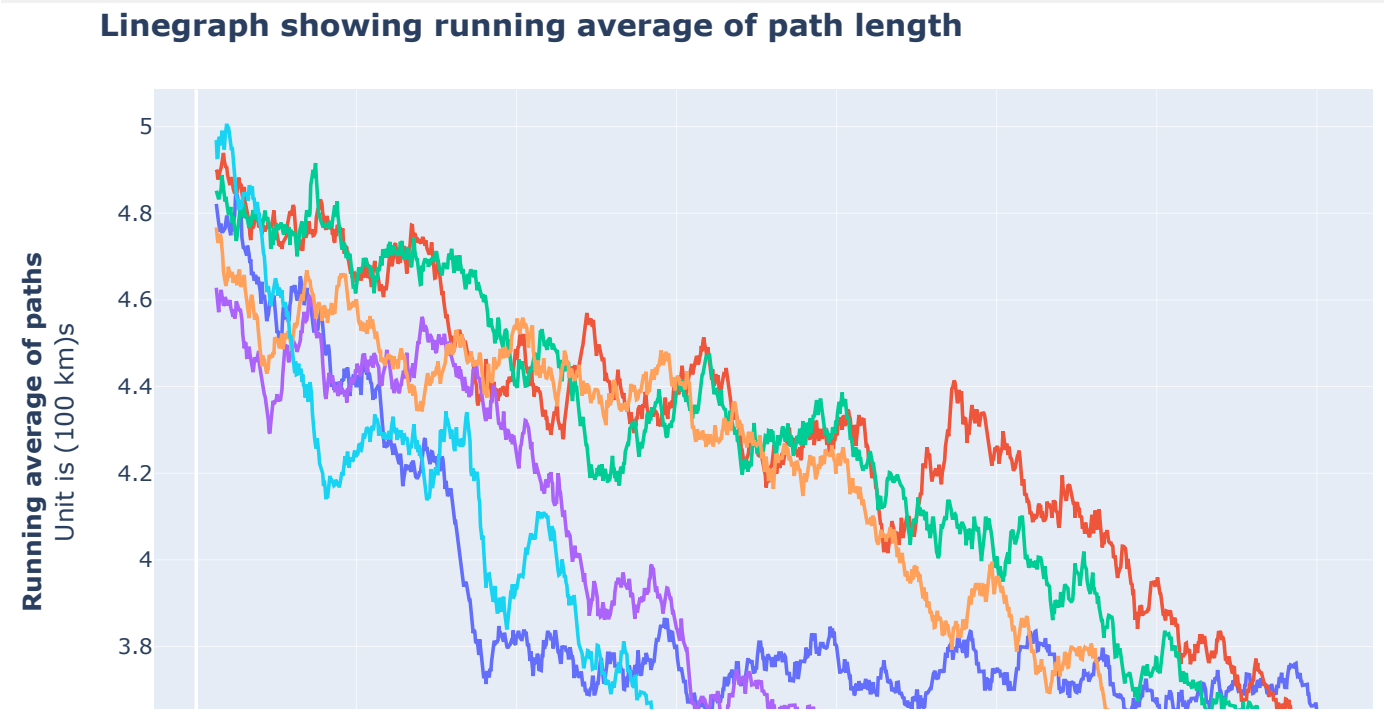
step 12

Simulation F
Ants simulated: 600




Linegraph showing performance of algorithm
Distance of paths over amount of ants simulated
A
C
E
Length of path
Unit is (100 km)s
Ants simulated
0
500
1000
200
400
3.5
4
4.5
5
5.5
6
6.5
7





Out[7]:
```
([3.43, 3.43, 3.43, 3.43, 3.43, 3.43],
 [[12, 0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11],
  [0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11, 12],
  [12, 0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11],
  [0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11, 12],
  [12, 0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11],
  [0, 7, 13, 1, 5, 14, 3, 8, 6, 4, 2, 9, 10, 11, 12]])
```

[#dataviz]

When displaying the performance of the algorithm I display multiple different graphs showing different elements which are relevant for the performance. For example the interactive animation for the graph displays relevant information about how certain the the algorithm is in the different paths over time. The graphs are informative yet easy to follow with correct

labeling and axis names. The graphs allow for focusing at one line at a time and zooming into critical moments, allowing for unique insight about how the algortithm works in its optimization. For example plotting 'individual ant distance' together with 'shortest path so far' gave valuable insight about how the higher level effect of the algorithm 'deciding on a path' can be measured

[#communicationdesign]

In the animated network graph, I effectively use the principles of salience and informative change. As I adjust the salience of the edges by changing their thickness, this shifts the readers focus and intuitively shows the relevant information about the progress of the algorithm. The color difference between the nodes and edges also increases the discriminability between the relevant components of the graph. The change in edge thickness is a good application of informative change, as the visual salience of them are proportionally representative of how much their included in the probability calculations for the ant. This design grants very intuitive interpretation of a critical component of the algorithm

Notes about output

Observe that we can see the length of the best path at the bottom of the output. When running multiple simulations we can see the stochastic component of this algorithm, as the final solution and the iterations required to get there varies for different smulatons on the same graph. We can see that the simulations that find a suboptimal solution also tend to be the ones that terminated early.

All line graphs above are interactive, you can select and deselect lines by clicking them in the legend. By clicking and dragging inside the graphs you can zoom in.

Explaining the algorithm

The algorithm simulates ants and how they in the real world find the best using pheromones. The concentration of pheromones is an emergent property of accumulated knowledge in the system. The core mechanic that makes this pathfinding work is that shorter paths will accumulate more pheromones as ants will walk there more frequently.

The algorithm works as follows:

Initialization:

1. Set each edge's distance attribute to the euclidian distance between its two nodes
2. Set each edge's pheromone value to 1
3. Define termination condition, maximum and mimum amount of ants to simulate

Algorithm

Foreach ant:

1. Put the ant at a random node
2. The ant chooses the next node to traverse depending on each edge's:
   A. **Distance attribute**, representing the euclidian distance to that node
   B. **Pheromone concentration**, represening how much this edge has been traversed and how good(short) the path it resulted in was
   C. The ant takes these two values to the power of their bias (how biased the ant should be toward each value), and then multiplies the result.
   $$distance^{-distBias} \times pheromone^{pheroBias}$$
   This value will be calculated for each available node, and is the weighted probability of that node being selected.
3. Select a node randomly depending on their weighted probability. The ant moves to the selected node
4. Repeat 2-3 until all nodes are visited
5. Calculate the length of the ants path and the differnce between this path and the shortest path found so far (save this path length as shortest path so far, if that's the case)
6. Increase the phermone of each edge traversed by the ant, increase with reciprocal of the difference from step 5 (a small difference would be a good path, which should increase the pheromone more)
7. Take a new ant and go to step 1, repeat until out of ants

Termination condition

There are many ways one could design a termination condition. The goal of my termination condition is to correctly detect when the algorithm has "decided" upon a path, which I define as the probability of the algorithm "changing its mind" is decreasing and accellerating heavily. This event is very clear visibly in my animation, as the chosen paths edges become thicker, and all other edges practically disappear.

This is a good termination condition as running the simulation further would by definition not give a new solution, but only reinforce the one found. A negative is that we can't confirm that this will happen within a reasonable amount of time, for this reason I implemented a maximum amount of ants to simulate, as a secondary termination condition. This hyperparameter, much like the other hyperparameters, are dependent on the amount of nodes in the network, the computational time that is reasonable for the context, and the desired confidence.

When deciding on a termination condition I utilized the linegraph I produced which described the progress of the ants, this graph gave valuable insight on how to detect this event. First I considered only running for a predetermined amount of iterations only, which could be a sufficient condition depending on context, but clearly not effective as it sometimes would perform computation that has no effect on the result, and sometimes it would terminate before a path had been clearly desided upon. After examining the graph I noticed that the distribution of path-distances was "cut-off" approximately at the same time as when the desired effect could be visually seen in the animation. From this I added both the red and green line, the average distance of the paths and the shortet path found so far. With those additions it became apparent how the "deciding" event occured when a more significant portion of the ants took the best path found so far, which further reinforces it and further increases the amount of ants taking that path. When formulating the termination condition I considered:

- "when x ants in a row go on the best path", but I concluded that was to affected by random events
- "When the average range of the distribution of path-distances was cut to 75%", but it would be difficult to define and difficult to interpret how that confidently is connected to a decision being made
- "When ants taking the best path represent the fraction x of the latest n ants", I felt this would be to affected by randomization and It'd be unclear how to specify these new hyperparameters Finally I stopped on
- "When the runnning average (last 50 ants) of the distance is within 5% of the shortest path", This is an accurate proxy for the decision event, as well as being easy to implement, interpret, and cheap to check for. It's also clear how this is directly connected to more ants taking the shortest path and starting the reinforcement loop. This critical line is represented by the green dashed line in the linegraph.

Another termination condition that would be a more accurate representation of "one single path being chosen" is to see if the edges contained in the path are the only edges which are "activated", where not activated would be represented by edges having pheromone levels below some critical level. Where that critical level would be so low compared to the activated edges that they're effectively irrelevant. This would be a good termination condtion, but I judged it to be too costy to compute in each iteration, as I would have to check the levels of each edge every single iteration

Pheromone evaporation

The evaporation rate of the system is proportional to the current pheromone level, while the incrementation of pheromones is a constant value. This means that when the current pheromone levels are low, the impact of increasing pheromones will be relatively larger than the rate of evaporation. Vice versa is also true, if the pheromone levels are large the evaporation will be larger as well. Practically this means that an edge with less pheromones will, after being visited, get a more longer lasting impact than an edge that already has a lot of pheromones, there is a bias towards edges that are not visited that much, promoting any shorter paths found by random exploration. This effect is a balance against the effect that paths with more phermones will get visited a lot more. If the evaporation was not proportional, but had a constant value for decrementation, it would increase the probability that the algorithm would reinforce early solutions too heavily and settle on a local minima.

Discussion

ACO is a stochastic optimization algorithm which, simular to simulated annealing, can find the global minima by allowing for random and temporary decrease in the objective function to fall into other basins that might lead to a better local minima, or perhaps even global minima (assuming we're looking to maximize). This is preferable over greedy and naive algorithms which have no process to avoid getting stuck in a local minima.

This stochastic property of course leaves the possibility for varying results on the same graph, therefore introducing the probability of suboptimal solutions. In ACO there are many small entities, ants, having a small impact on the system probabilities, pheromones. This in combination with the fact that better paths gets more pheromone than worse paths means that the only way for a suboptimal path to get chosen is for many random events "choosing" the less probable option. If path A is 1 units longer than the best path found so far, and path B is 2 units longer, then twice as many ants

would have to randomly choose Path B than A for their pheromone levels to stay the same. Only if B is randomly selected more than twice as much in such succession so that its increased pheromone levels gives it twice the probability of being selected compared to A, only then could the pheromones take over to select a suboptimal path. This issue can be mitigated by decreasing the pheromones that ants spread, requiring mor e inprobable events in a row to hit this critical value. Relevant is also the evaporation rate, with a small evaporation rate, incorrectly enforced paths will stay for longer, increasing the chance of reinforcing the bad path. Following the logic above, we can control and decrease the probability of "accidentally" reinforcing a suboptimal solution by increasing the amount of ants, decreasing their individual pheromone spread, and increasing the fraction of pheromones that are evaporated.

ACO is clearly much more effective than both a greedy best first search (https://en.wikipedia.org/wiki/Best-first_search (https://en.wikipedia.org/wiki/Best-first_search)) and a brute force approach. The greedy best first would be very fast, especially if only run a single time, but has no process to avoid getting stuck in a local minima, which also means it often does not find the optimal solution, depending on the complexity of the problem. Brute force on the other hand will guaranteed find the best solution, although as it has a time complexity of O(n!), this algorithm becomes unfeasibly slow as n increases (n being the amount of nodes in the network). Greedy best first and brute force are on the two extremes on the balance between quality of result and time complexity. ACO has a much better balance, as it quickly eliminates large parts of the solution space while keeping a broad exploration.

Greedy best first search could also get stuck in local minima which are very bad solutions. If ACO gets stuck in local minima, it will with high probability be near optimal. Since the incrementing of the pheromones is proportional to the difference of the solutions and the best solution so far, it is much easier to get out of a local minima which is far away from best so far. Local minima that have a large difference of the objective function compared to the lowest point found so far become more volatile, or unstable. This stays true if the best solution found so far is close to the global minima, which is in general the case for ACO because of how it traverses the solution space. As covered in detail by (Gómez & Barán, n.d.), one of the reasons for ACOs success in TSP is its ability narrow down the soluton space. This can loosely be explained by the fact that near-optimal solutions for the TSP would share many edges, which is in ACOs favor as it's an edge selection based algorithm. Put in more concrete words, it's highly unprobable that there's a path which is significantly better than the best path found so far, which does not share any edges. Another way to phrase it is that the solution space has a convex structure at a larger scale.

Comparison, GA and ACO

Comparing a genetic algorithm (GA) and ACO is interesting as when you look at them conceptually, they in general build on similar principles when it comes to TSP (Gómez & Barán, n.d.). A similarity is that both algorithm can be seen as working with some kind of population, which has direct and combined impact on the further state. This is clear in GA, but also in ACO if you consider the edges as individual chromosomes, subject to high elitism. They performing crossover by increasing the probability that edges connected to a "high-pheromone" edge get searched, meaning that two "high-pheromone" edges can effectively merge, by increasing the amount of ants running over their nodes, and probabilisticly favoring the edges that connect these edges. Mutation, or exploration in the solution space, is performed by the random selections of each individual ant. These similarities are of course very conceptual, the largest practical difference are how this "crossover" and "mutation" is performed. The crossover in ACO is on a local scale, while GA does crossover between two whole paths. In ACO the crossover does not make large steps from the "parents" in the solution space, instead incremental steps which favor improving objective function, the negative of this is that it's not as exploratory as the GA crossover, which can mix entire paths in different ways. The mutation is also different, in ACO it has a very frequent but small effect, while in GA it's more rare but has more extreme effects (Depends on how it's perfromed, but a common approach, switching place of two cities, has a large impact on the objective function).

It has been shown that ACO outperforms GA in Path-finding problems which have similarity to TSP. In the test performed by (Binti Sariff & Buniyamin, 2010), ACO was more than twice as fast in finding the optimized path.

Possible improvements

There are multiple possible improvements for this algorithm. One is simply to run the algorithm multiple times with different hyperparameters, measuring the overall performance and finding the optimal hyperparameters for the specific use case. Another is simply performing the simulation in parallel, conecptually this could be like having multiple colonies of ants which are not affected by eachothers pheromones, this would decrease the probability of getting stuck in local minima (Manfrin et al., 2006). Another interesting example of an improvement is making the pheromone distribution rank-based, similar to rank-based selection in GAs, we would only allow the top n ants spread pheromones, we could also make this pheromone spread proportional to the ants rank instead of its performance (Dorigo, 1999). More developed versions of ACO, and how they work, is covered by (Gómez & Barán, n.d.).

[#emergentproperties]

This algorithm relies on how the pheromone accumulation on the graph edges is an emergent property of learning between the ants. Every single ant walks pseudorandomly and sprouts pheromones in proportion to the reciprocal of distance traveled, no ant has any understanding of the overall path, they also don't learn anything as they don't adapt their behavior, they only follow the pheromones laid by other equally dumb ants. The incremental effect the pheromone has on the probabilites of following ants decisions lead to a positive feedback loop, reinforcing the edges which often take part in paths which end up short. No ant has any sense of the shortest path, but the pheromone concentration represents the collective learning of many ants and their decisions. This emergent property is exactly what we observe in the animation for the graph, where we can intuitively see how this collective learning adapts, changes it mind, has different certainty, and finally converges upon a reinforcing decision.

PART 2: SIMULATION

Part 2.1 Numerical Modeling and Simulation

For this part, I'll consider the SIR model described by the set of differential equations below, and the numerical simulation in Python via Euler's method.

$$\frac{dS}{dt} = -\frac{b}{N}S(t)I(t)$$

$$\frac{dI}{dt} = \frac{b}{N}S(t)I(t) - kI(t) - mI(t)$$

$$\frac{dR}{dt} = kI(t)$$

$$\frac{dD}{dt} = mI(t)$$

2.1.1 Variables and Parameters

I chose to model the disease Ebola.

The variables are **S**, **I**, and **R**, and they represent the amount of people in the population which are **S**usceptible, **I**nfected, and **R**emoved. In the real world these variables will be discrete, as their unit are 'amount of people'. Although in Euler's method we'll use them as continuous functions, eg S(t). We do this because derivation, which Euler's method uses, requires a continuous function. The variable **N**, the size of the population, is discrete, as we don't need to perform derivation on **N**. **t** is the indepedndent variable and represents time, a continuous variable used in a discrete way for Euler's Method, as it relies on an iterative process where we increment **t** with a certain step size.

b and **k** are parameters, which represent the infection and recovery rate respectively. Increasing **k**, would be like giving treatment earlier, as that would decrease the time each infected needs to get recovered. Increasing **b**, for example by not cleaning your hands, would mean that more people get infected per infected person. The unit for **b** is 1/time

Ebola was first reported in Liberia in March 30, 2014. (Nyenswah et al., 2016)
The population at that time was, ~4.36 million (worldbank.org, 2014)
The infection and recovery rates are .8 and .5, the fatality rate being 55-60%.(Pan American Heatlh Organization, n.d.)

S_0 = 4 359 999
I_0 = 1
R_0 = 0
b = 0.8
k = 0.5
time unit is days

2.1.2 Euler's Method Description

Euler's method solves the SIR differential equations numerically, that means that the ouput is not a continuous function that is a solution to the differential equation, instead the output is a series of points which estimate the function. The inputs for Euler's method are an initial point, and a differential equation for the derivative. By calculating the derivative at the initial point, we create a new point one step size away with the assumption that the slope stayed the same between these two points. When the step size **h** is large, this assumption will fail, as the slope will curve between two points, this error will decrease as we decrease **h**.

2.1.3 Euler's Method Implementation

Importing Libraries

In [30]:
```
1 import numpy as np
2 import plotly.graph_objects as go
```

Defining Function

In [8]:

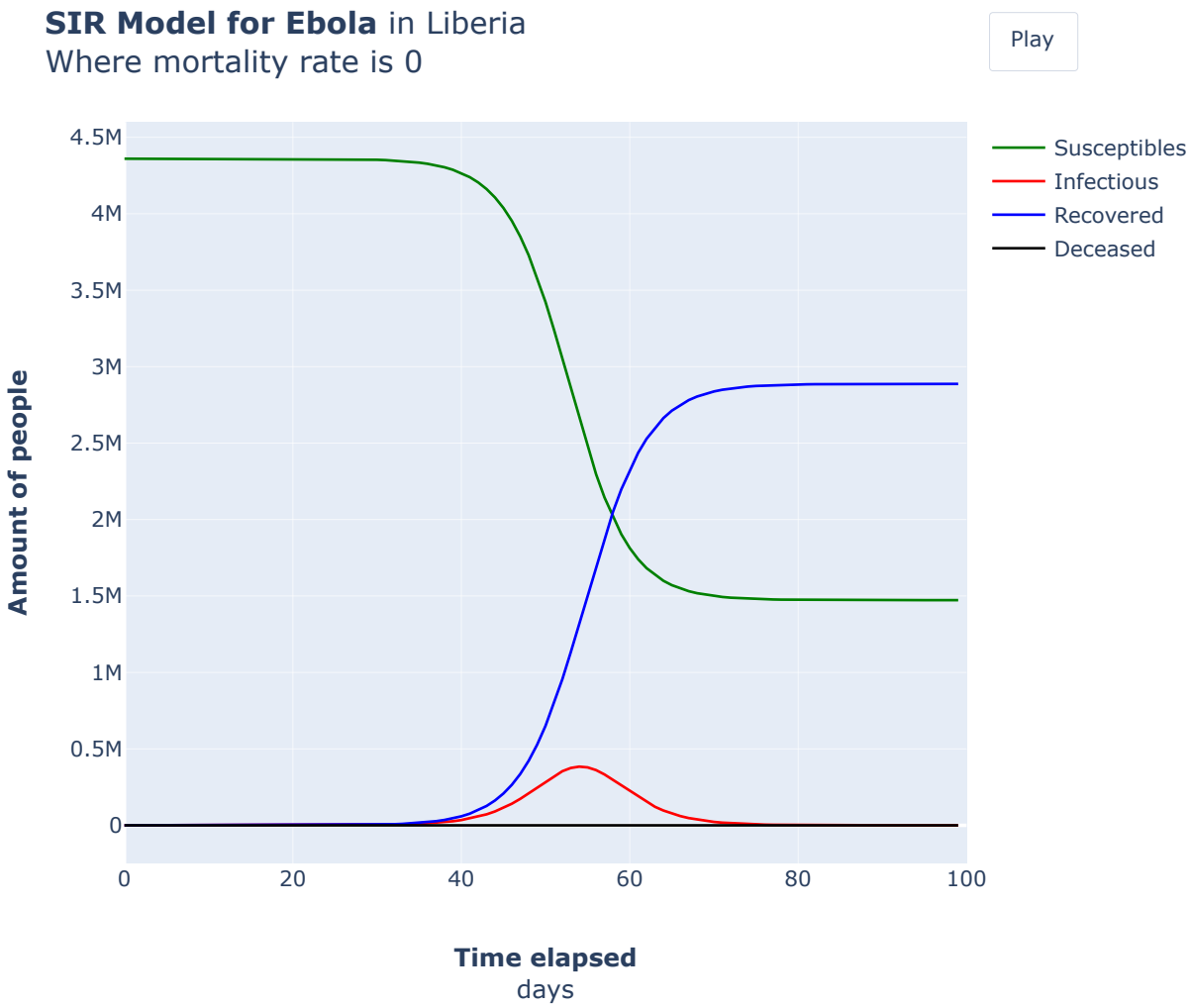
```
def EulerSIR(infRate,
             recRate,
             initialConds,
             mortalityRate = 0,
             titleText = "",
             ss = 1, # step size, I chose 'ss' instead of 'h' for readability
             tEnd = 100):

    # The amount of days (or other time unit) to simulate is tEnd
    # Amount of steps to simulate is days/stepsize + 1
    # (+1 to account for first measurement)
    stepsAmount = tEnd//ss + 1

    # Create arrays for each population category.
    # Creating the full array here is faster computationally
    # than appending values as we calculate them

    # For consistency I usually only use capitalized letters for function names
    # but I make an exception here considering they represent a function, but discretely
    S = np.zeros(stepsAmount)
    I = np.zeros(stepsAmount)
    R = np.zeros(stepsAmount)
    D = np.zeros(stepsAmount)

    # Set the initial conditions
    popSize = sum(initialConds)
    S[0] = initialConds[0]
    I[0] = initialConds[1]
    R[0] = initialConds[2]
    D[0] = initialConds[3]

    for n in range(stepsAmount - 1):
        # I use more variables (more storage) than necessary here
        # because it increases readability
        newRemoved = ss*recRate*I[n]
        newInfected = ss*infRate*I[n]*S[n]/popSize

        S[n+1] = S[n] - newInfected
        I[n+1] = I[n] + newInfected - newRemoved
        R[n+1] = R[n] + newRemoved * (1 - mortalityRate)
        D[n+1] = D[n] + newRemoved * mortalityRate

    GraphSIR(S, I, R, D, titleText)


def GraphSIR(S, I, R, D, titleText):
    # Define the frames for the animation
    frames=[dict(data= [dict(type='scatter',
                             y=S[:k+1]),
                        dict(type='scatter',
                             y=I[:k+1]),
                        dict(type='scatter',
                             y=R[:k+1]),
                        dict(type='scatter',
                             y=D[:k+1])],traces=[0,1,2,3],)for k in range(1, len(S)-1)]

    # The different lines to be animated
    traces = []
    traces.append(go.Scatter(y=S[:2],
                        mode='lines',
                        line=dict(width=1.5),
                        line_color='green',
                        name="Susceptibles"))
    traces.append(go.Scatter(y = I[:2],
                         mode='lines',
                        line=dict(width=1.5),
                        line_color='red',
                        name="Infectious"))
    traces.append(go.Scatter(y = R[:2],
                        mode='lines',
                        line=dict(width=1.5),
```

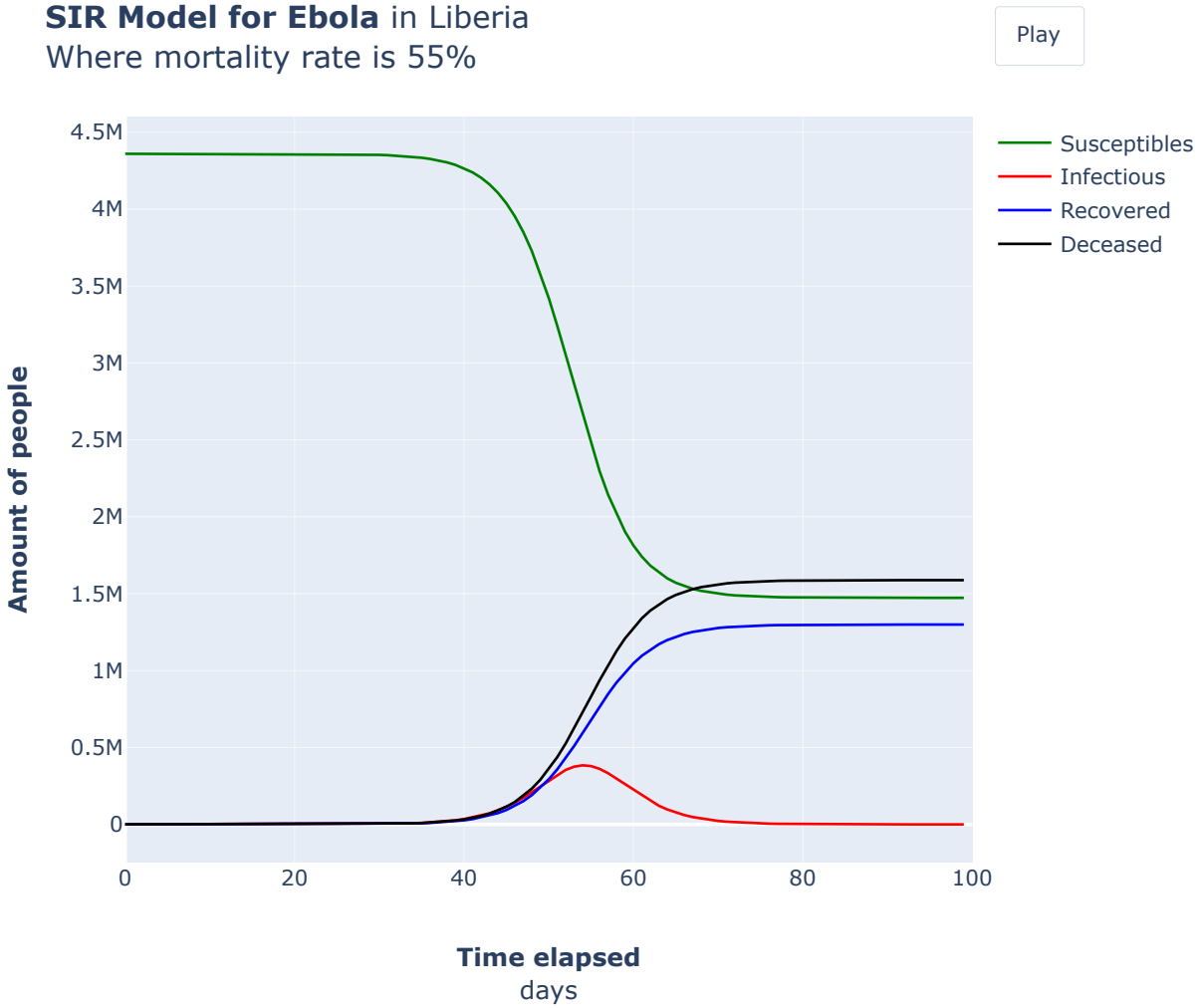
```
                        line_color='blue',
                        name="Recovered"))
    traces.append(go.Scatter(y = D[:2],
                        mode='lines',
                        line=dict(width=1.5),
                        line_color='black',
                        name="Deceased"))

    # Layout settings
    layout = go.Layout(width=700,
                       height=600,
                       showlegend=True,
                       hovermode='x unified',
                       updatemenus=[dict(
                                type='buttons', showactive=False, # Settings for play button
                                y=1.15,
                                x=1.15,
                                xanchor='right',
                                yanchor='top',
                                pad=dict(t=0, r=10),
                                buttons=[dict(label='Play',
                                    method='animate',
                                    args=[None,
                                      dict(frame=dict(duration=1,
                                                      redraw=True),
                                                      transition=dict(duration=0),
                                                      fromcurrent=True,
                                                      mode='immediate')]
                                    )]
                                )]
                      )

    # Write titles
    fig = go.Figure(data=traces, frames=frames, layout=layout)
    fig.update_xaxes(title_text="<b>Time elapsed</b><br>days")
    fig.update_yaxes(title_text="<b>Amount of people</b>")
    fig.update_layout(title_text=f"<b>SIR Model for Ebola</b> in Liberia<br>{titleText}",
                      yaxis_range=[-2.5*10**5, 4.6*10**6],
                      xaxis_range=[0, len(S)-1])
    fig.show()
```

Function Call

In [9]:

```
infectionRate = .8
recoveryRate = .5

totalPop = 4.36*10**6
initialInfected = 1

# Order of data is [S, I, R, D]
initialConditions = [totalPop - initialInfected, initialInfected, 0, 0]

mortalityRate = 0

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where mortality rate is 0")

mortalityRate = .55

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where mortality rate is 55%")

                                        # PRESS PLAY BUTTON TO PLAY ANIMATION
```

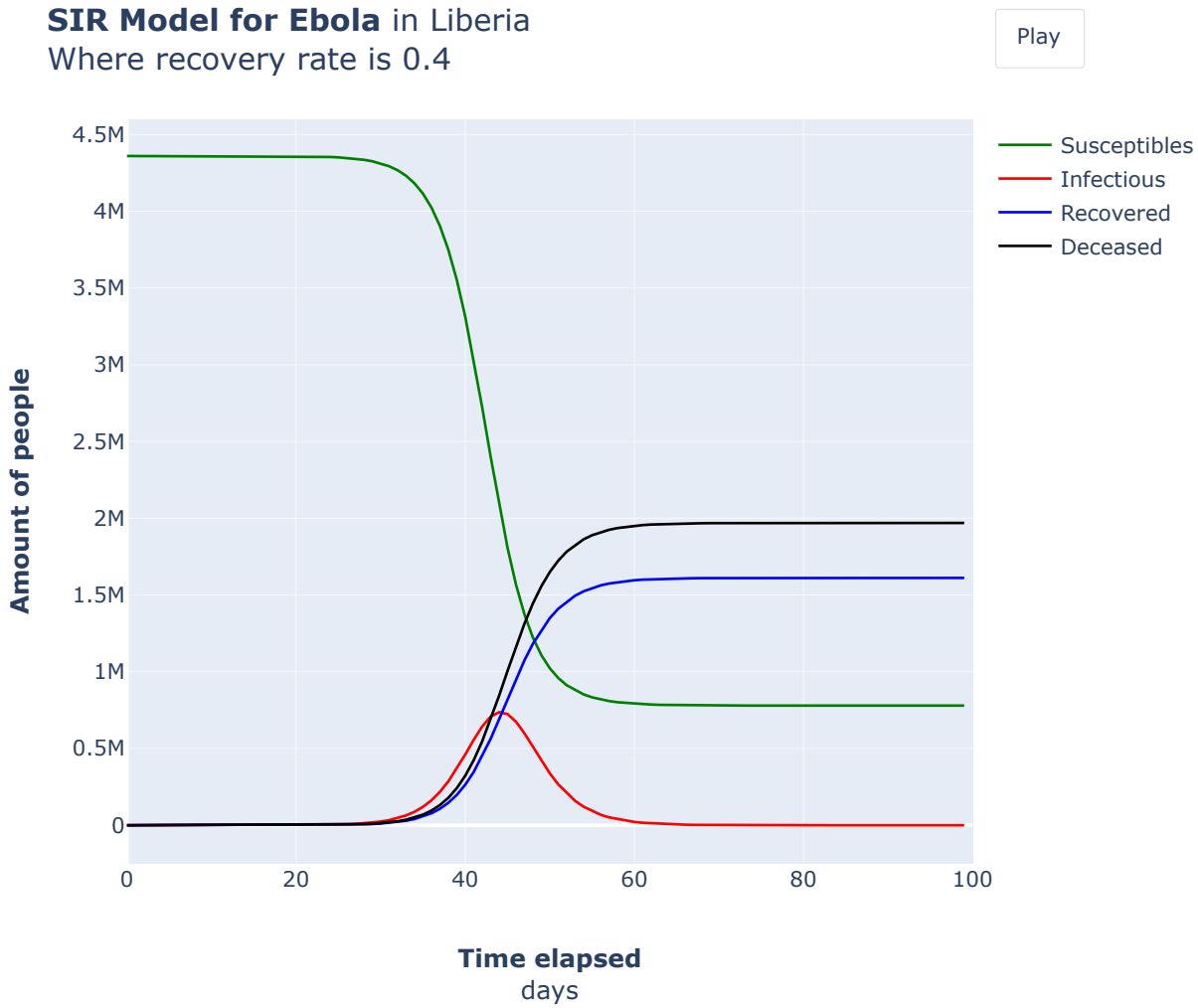


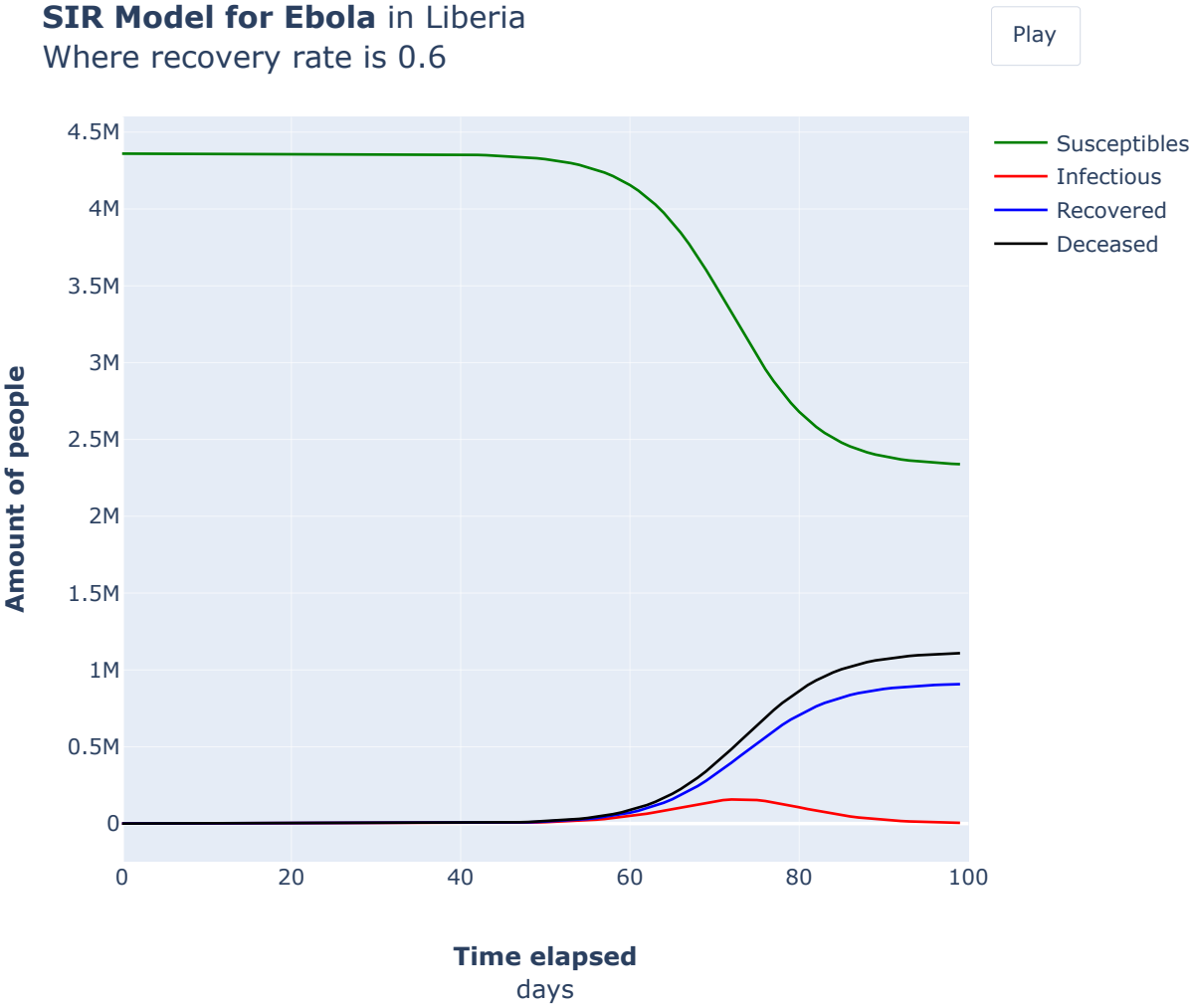


2.1.4 Results and Interpretation

In [10]:

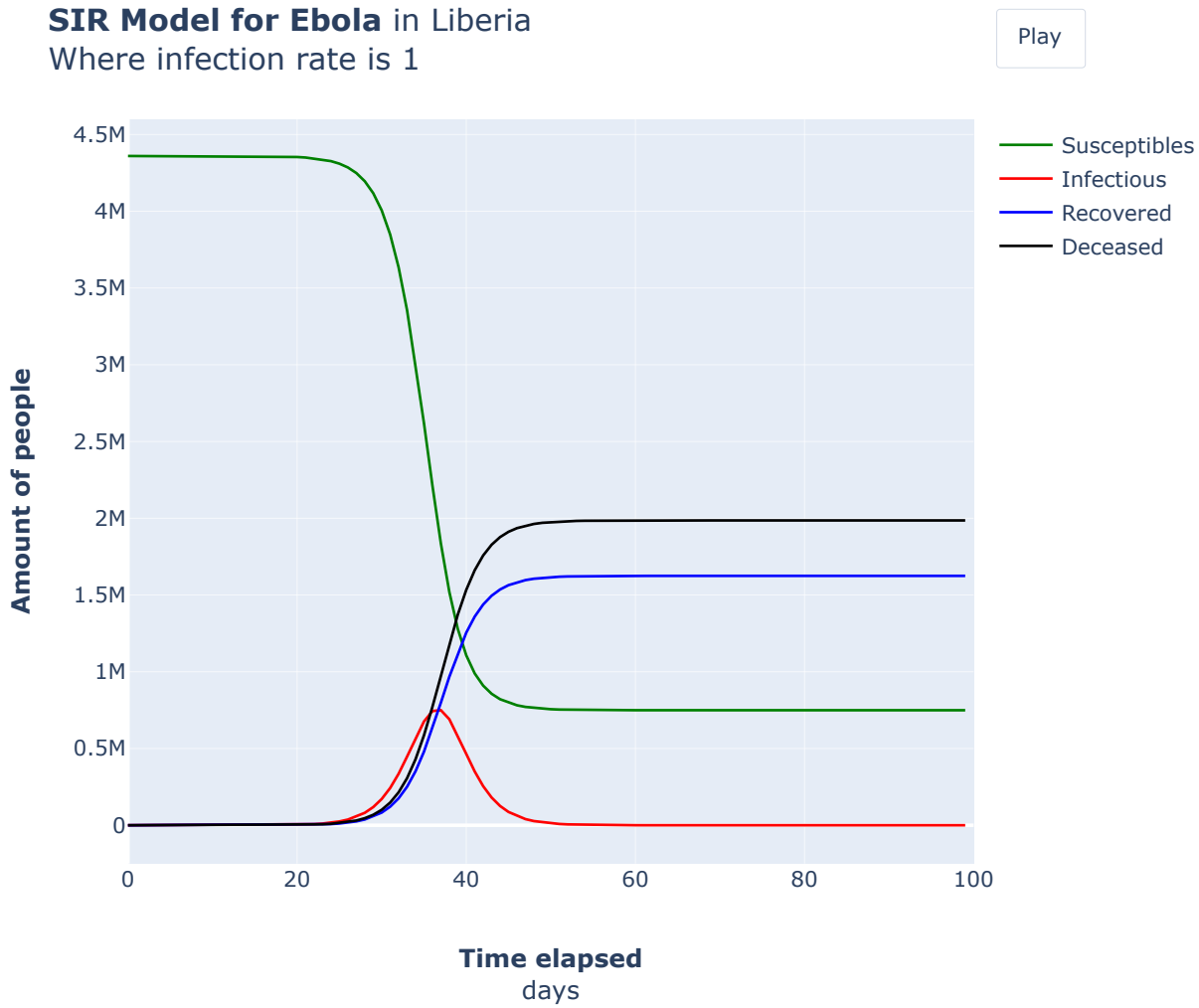
```
totalPop = 4.36*10**6
initialInfected = 1
# Order of data is [S, I, R, D]
initialConditions = [totalPop - initialInfected, initialInfected, 0, 0]

infectionRate = .8
recoveryRate = .4
mortalityRate = .55

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where recovery rate is 0.4")

infectionRate = .8
recoveryRate = .6

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where recovery rate is 0.6")
```

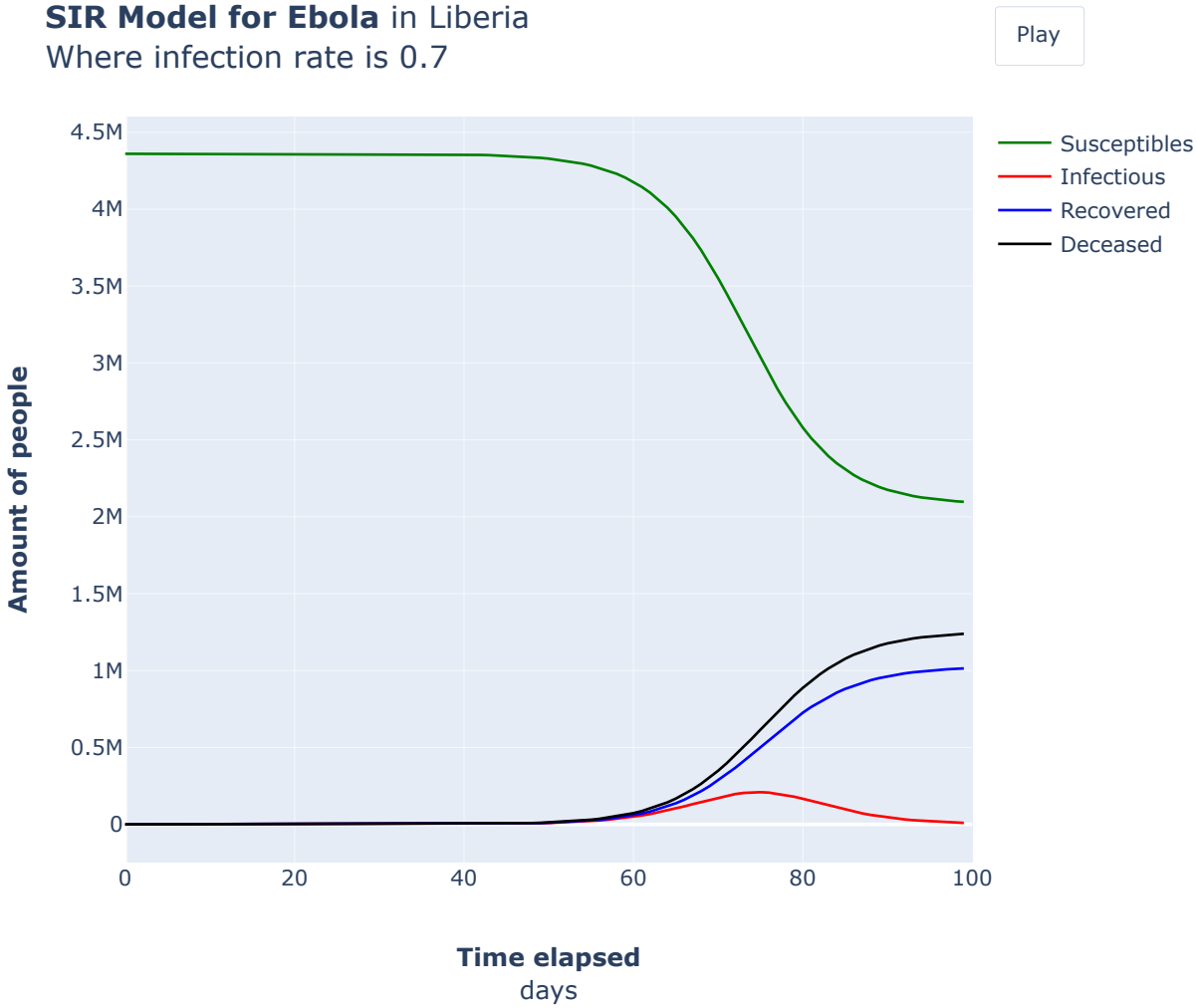



SIR Model for Ebola in Liberia
Where recovery rate is 0.6
Play
Susceptibles
Infectious
Recovered
Deceased
Amount of people
4.5M
4M
3.5M
3M
2.5M
2M
1.5M
1M
0.5M
0
0
20
40
60
80
100
Time elapsed
days

In [11]:

```
totalPop = 4.36*10**6
initialInfected = 1
# Order of data is [S, I, R, D]
initialConditions = [totalPop - initialInfected, initialInfected, 0, 0]

infectionRate = 1
recoveryRate = .5
mortalityRate = .55

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where infection rate is 1")

infectionRate = .7
recoveryRate = .5

EulerSIR(infectionRate,
        recoveryRate,
        initialConditions,
        mortalityRate,
        "Where infection rate is 0.7")
```





In the output we can see the development of the functions **S**, **I**, and **R**, over time. The behavior we see is a single hump of infections, which represents a temporary outbreak, leading to a burst of susceptible people moving to 'removed'. From this graph we can get valuable insight for the real world, how long the outbreak will be, the peak amount of infected people per day, and how many people will end up dying from the outbreak. We can also see what kind of impact real word interventions would have, such as cleaning your hands or improving healthcare, which would be associated with decreasing the infection and recovery rate respectively. We can see from the graphs above that it's the relationship between infection and recovery rate that impacts the length and timing of the outbreak. When the infection rate is about twice the recovery rate, we get the early and high peak, in the other two graphs where infection rate is approximately 1.5 times the recovery rate, we get the slower outbreak that starts around day 60.

Part 2.2 Agent-Based Modeling and Simulation

In [40]:
```
# Run this to install pygame
import sys
!{sys.executable} -m pip install pygame
```

Import Libraries

In [11]:
```
import plotly.express as px
import plotly.graph_objects as go
from IPython.display import clear_output
import numpy as np
import pygame
import math
import random as rnd
import time
```

```
pygame 2.1.2 (SDL 2.0.18, Python 3.9.2)
Hello from the pygame community. https://www.pygame.org/contribute.html (https://www.pygame.org/contribute.html)
```

Define Functions

In [14]:
```
# Update the state of dot to become newState
def UpdateState(dot, newState):
    currentDistribution[state[dot]] -= 1 # Decrement the category dot is leaving
    state[dot] = newState # Change the state of dot
    currentDistribution[newState] += 1 # Increment the category dot entered

# Draw the population distribution
def UpdateGraph():
    # Add the current distribution as a new datapoint
    for i in range(len(statusHistory)):
        statusHistory[i].append(currentDistribution[i])

    # Labeling and coloring
    titles = ["Susceptible", "Infectious", "Recovered", "Dead", "Quarantined"]
    colors = ["blue", "red", "green", "black", "purple"]

    fig = go.Figure()

    # Draw a line for each category
    for i in range(len(statusHistory)):
        fig.add_trace(go.Scatter(y=statusHistory[i], mode='lines',
                                 name=titles[i],
                                 line_color=colors[i]))

    fig.update_layout(title_text='''<b>Amount of people per each SIRDQ category in the populatic
                                 over time''')
    fig.update_yaxes(title_text="<b>Amount of people</b>")
    fig.update_xaxes(title_text="<b>Elapsed time</b> in weeks")
    # Clear output before drawing new figure
    clear_output(wait=True)
    fig.show()
```

Simulation

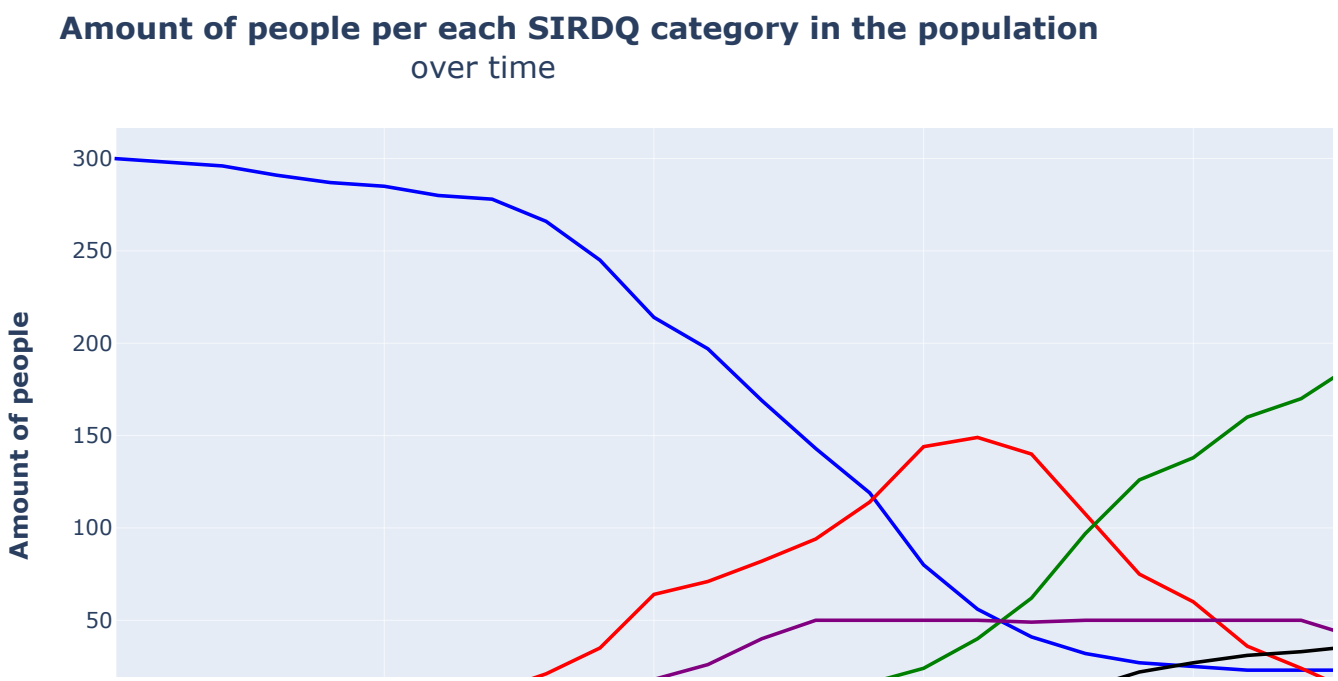
In [15]:

```
# Run pygame
pygame.init()

# Set size for screen
width, height = 800, 600
size = 1000, 800
screen = pygame.display.set_mode(size)
# Amount of seconds between each graph update
graphUpdateFrequency = 1
# frames per seconds
fps = 100

# Specify populationsize
numDots = 300

# All metadata for each dot
dotPos = [] # Position of each dot
targetDir = [] # The target direction vector to travel in
currentDir = [] # The current direction vector
dotTimers = [] # An internal timer, purpose depends on state
passedQTime = [] # COMMENT ON THIS
# The state of the dots represented as a number
# S = 0, I = 1, R = 2, D = 3, Q = 4
# this number is equal to the index of which the categories
# are stored in 'currentDistribution' [S, I, R, D, Q]
state = []

# Initialize the population
for i in range(numDots):

    # randomly position dots
    dotPos.append(np.array([rnd.random()*width,rnd.random()*height]))

    # Set a random target direction vector
    angle = rnd.random()*math.pi*2
    targetDir.append([math.cos(angle), math.sin(angle)])
    currentDir.append([0, 0])

    # Set all to susceptible
    state.append(0)
    dotTimers.append(0)
    passedQTime.append(False)

# Initial condition for population
statusHistory = [[numDots], [0], [0], [0], [0]]
currentDistribution = [numDots, 0, 0, 0, 0]
# Make one random individual infected
UpdateState(rnd.randint(0, len(state) - 1), 1)

# White background color
backgroundClr=[255,255,255]
# The color for the dots depending on their state
# Order is [S, I, R, D, Q]
dotColors = [[0,0,255], [255,0,0], [0,255,0], [0, 0, 0], [191, 64, 191]]

# The size of the dots
dotSize = 5
# The average amount of seconds each dot takes before they select a new target direction to tra
avgTimeTurn = 1
# How quickly/sharply the dots adapt to the new direction vector
turningSpeed = 1
# The maximum speed of the dots
maxSpeed = 2

# The distance within infected people can spread the disease
infectDist = 20
# Time in seconds within proximity that counts as one interaction
suffInteractionTime = .5
# Probability of one interaction spreading the disease
infProb = .5
# The simulation does not wait for an interaction to occur,
```

```
# instead the probability of an infection happening each iteration is infProb/(fps*suffInteract
# This effectively means that a susceptible dot will on average become infected after
# spending 'infProb/suffInteractionTime' seconds within the 'infectDist' if an infected dot

# Set amount of seconds each dot spends as infected before becoming recovered or dead
recoveryTime = 6
# Fraction of how many of infected die
mortalityRate = .2
# The fraction of recovered dots that will become susceptible
probLoseImmunity = .05
# The set amount of seconds spent as recovered, if dot will become susceptible again
timeFreqLoseImmunity = 20

# What fraction of infected we can detect the disease, and then quarantine
# Set this to zero to test without quarantining
quarantineRate = .9
# How long time it takes to quarantine after start of disease
quarantineDelay = 3
# How many people the quarantine facility can take
quarantineCapacity = 50
# I have a smaller mortality rate for people quarantining, which represents increased quality o
qMortalityRate = .05

# iteration counter
iteration = 0

running = True
while running:

    # Terminate pygame if exit button is pressed
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Terminate if disease disappeared
    if currentDistribution[1] == 0:
        running = False

    # fill background
    screen.fill(backgroundClr)

    # For each dot
    for i in range(len(dotPos)):

        # Draw dots with updated positions and colors
        pygame.draw.circle(screen,dotColors[state[i]],dotPos[i],dotSize)

        # If dot is infected
        if state[i] == 1:
            # Draw a radius circle, visualizing 'infectDist'
            pygame.draw.circle(screen,dotColors[state[i]],dotPos[i],infectDist, 1)

            # Increment timer
            dotTimers[i] += 1/fps

            # If 'quarantineDelay' time has passed, and is first time we check for quarantining
            if dotTimers[i] > quarantineDelay and not passedQTime[i]:
                passedQTime[i] = True
                # If disease is detectable, so we can quarantine, and if there's capacity
                if rnd.random() < quarantineRate and currentDistribution[4] < quarantineCapacit
                    # Convert to quarantined
                    UpdateState(i, 4)
                    dotTimers[i] = 0
                    # Move to quarantine box
                    dotPos[i] = [800 + rnd.random() * 100 - 50, 700 + rnd.random() * 100 - 50]

            # If 'recoveryTime' spent as infected
            if dotTimers[i] > recoveryTime:
                # Reset timer
                dotTimers[i] = 0

```

```
                # move dot to deceased according to 'mortalityRate'
                if rnd.random() < mortalityRate:
                    UpdateState(i, 3)
                # Else, become recovered
                else:
                    UpdateState(i, 2)

        # If quarantined
        if state[i] == 4:
            # When 'recoveryTime' time has passed
            dotTimers[i] += 1/fps
            if dotTimers[i] > recoveryTime:
                # move dot to deceased according to quarantine 'qMortalityRate'
                if rnd.random() < qMortalityRate:
                    UpdateState(i, 3)
                # Else, become recovered
                else:
                    UpdateState(i, 2)

        # If is recovered and timer has not been paused
        if state[i] == 2 and dotTimers[i] != -1:
            # Update timer
            dotTimers[i] += 1/fps

            # If 'timeFreqLoseImmunity' has been spent as recoverd
            if dotTimers[i] > timeFreqLoseImmunity:
                # Make dot susceptible according to 'probLoseImmunity'
                if rnd.random() < probLoseImmunity:
                    UpdateState(i, 0)
                # Else, pause timer
                # I'm not increasing the probability of becoming immune over time,
                # instead I just calculate this probability once
                else:
                    dotTimers[i] = -1

        # If dot is not dead or quarantined
        if state[i] != 3 and state[i] != 4:
            # Update positions
            # travel in currentDir with a speed of maxSpeed
            # magnitude of currentDir can be < 1 when dot is still turning from last direction,
            # Therefore maxSpeed is speed only when currentDir == targetDir.
            # I do modulus of the screen dimensions so that the dots can walk out on the sides
            # and appear on the opposite side. Making the traverse, topologically speaking
            dotPos[i][0] = (dotPos[i][0] + maxSpeed*currentDir[i][0]) % width
            dotPos[i][1] = (dotPos[i][1] + maxSpeed*currentDir[i][1]) % height

        # Spread infection
        if state[i] == 1:
            for ind in range(len(dotPos)):
                if state[ind] == 0: # This does not take edge teleportation into account
                    # If distance is within infection distance
                    if math.sqrt((dotPos[i][0]-dotPos[ind][0])**2 + (dotPos[i][1]-dotPos[ind][1
                        # Calculate probability of infection
                        if rnd.random() < infProb/(fps*suffInteractionTime):
                            # Make dot infected
                            UpdateState(ind, 1)
                            passedQTime[ind] = False

        # Set new target direction vector, on average each 'avgTimeTurn' seconds
        if rnd.random() < 1/(fps*avgTimeTurn):
            # Randomize angle
            angle = rnd.random()*math.pi*2

            # Create noramalized direction vector based on angle
            targetDir[i][0] = math.cos(angle)
            targetDir[i][1] = math.sin(angle)

        # Update current direction to move smoothly toward target direction
        # If x component of current direction is not equal target x component
        if currentDir[i][0] != targetDir[i][0]:
            # Move closer to the target value,
```

```
            # change in x component is proportional to 'turningSpeed' and the difference betwee
            # We get smoother turning by having turning be proportional to the difference rathe
            # direction with a set amount each iteration.
            currentDir[i][0] += (turningSpeed/fps)*(targetDir[i][0] - currentDir[i][0])
        if currentDir[i][1] != targetDir[i][1]:
            currentDir[i][1] += (turningSpeed/fps)*(targetDir[i][1] - currentDir[i][1])

    # Draw quarantine zone
    pygame.draw.rect(screen,[0, 0, 0],(740, 640, 120, 120), 2)

    # Update the display
    pygame.display.update()

    # Increment the iteration counter
    iteration += 1

    # Each 'graphUpdateFrequency' seconds
    if iteration % (graphUpdateFrequency*fps) == 0:
        # Update linegraph
        UpdateGraph()

    # Sleep 1/fps seconds
    time.sleep(1/fps)

# Quit pygame if loop is exited
pygame.quit()
```



Notes

When running the simulation above a pygame window should appear, displaying the simulation. The simulation is not adapted to the disease Ebola, as I don¨t have sufficient data with good #sourcequality to support decisions about hyperparameters, this is instead a more abstract representation of a possible disease

Discussion

Using this agent-based simulation we can include stochastic elements to better represent the random events of the real world, this was not taken into account in the deterministic SIR model which used Euler's method to get a numerical solution. This model has many more hyperparameters, which have an intuitive interpretation for the real world, when

Bibliography

Binti Sariff, N., & Buniyamin, N. (2010). GENETIC ALGORITHM VERSUS ANT COLONY OPTIMIZATION ALGORITHM - Comparison of Performances in Robot Path Planning Application. Proceedings of t he 7th International Conference on Informatics in Control, Automation and Robotics. http s://doi.org/10.5220/0002892901250132

Dorigo, M. (1999, April). (PDF) ACO Algorithms for the Traveling Salesman Problem. Research Gate. https://www.researchgate.net/publication/2771967_ACO_Algorithms_for_the_Traveling_Sal esman_Problem

Gómez, O., & Barán, B. (n.d.). Relationship between Genetic Algorithms and Ant Colony Optim ization Algorithms (p. Section 7.1). Retrieved March 7, 2022, from https://www.cnc.una.py/p ublicaciones/1_86.pdf

Kumar, R., & Dey, S. (2015). SIR Model for Ebola Outbreak in Liberia. International Journal of Mathematics Trends and Technology, 28(1), 28–30. https://doi.org/10.14445/22315373/ijmtt -v28p506

Manfrin, M., Birattari, M., Stützle, T., & Dorigo, M. (2006). Parallel Ant Colony Optimizat ion for the Traveling Salesman Problem. Ant Colony Optimization and Swarm Intelligence, 224 –234. https://doi.org/10.1007/11839088_20

Nyenswah, T. G., Kateh, F., Bawo, L., Massaquoi, M., Gbanyan, M., Fallah, M., Nagbe, T. K., Karsor, K. K., Wesseh, C. S., Sieh, S., Gasasira, A., Graaff, P., Hensley, L., Rosling, H., Lo, T., Pillai, S. K., Gupta, N., Montgomery, J. M., Ransom, R. L., & Williams, D. (2016). Ebola and Its Control in Liberia, 2014–2015. Emerging Infectious Diseases, 22(2), 169–177. https://doi.org/10.3201/eid2202.151456

Pan American Heatlh Organization. (n.d.). Ebola Virus Disease - PAHO/WHO | Pan American Hea lth Organization. Www.paho.org. https://www.paho.org/en/topics/ebola-virus-disease#:~:text= The%20virus%20is%20transmitted%20to

worldbank.org. (2014). Population, total - Liberia | Data. Data.worldbank.org. https://dat a.worldbank.org/indicator/SP.POP.TOTL?locations=LR

[#sourcequality]

I use multiple scientific articles which are directly relevant to the context. The sources are of high quality as the authors are of appropriate academic authority, for example Marco Dorigo which has over 90 000 citations. I use multiple sources which can confirm eachother, strengthening the reliability.