Homework - 2

Task 1

- Radius of the Ring = `100`
- Distance used - as described in the assignment statement
- Related files
    - Node graphs and neighbor lists have been provided for cycles - [1, 5, 10, 15] and final cycle=40.
    - Plot of sum of distances between neighboring nodes after each running cycle.
    - Comparision between plots of sum of node distances between neighbors when a. only receiver list is updated and b. both sender and receiver node neighbor lists are updated.

Task 2

- Radius of the Circle and Semi circles = `100`
- Distance used - `Euclidean`
- Related files
    - Node graphs and neighbor lists have been provided for cycles - [1, 5, 10, 15] and final cycle=40.
    - Plot of sum of distances between neighboring nodes after each running cycle.

Questions

a. For Ring topology

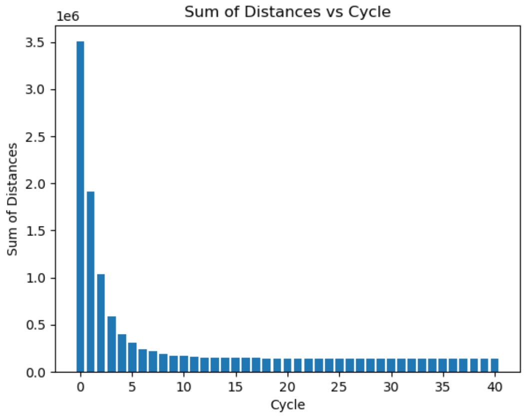
Please note that the code provided in the submission is written as per the question a part ii.

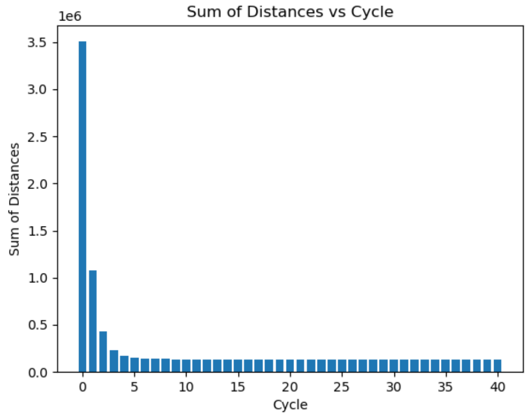
i. Only receiving neighbor update its neighboring-list

Network requires more number of iterations to stablize on the sum of distances compare to the scenario when both (sender and receiver) are sharing the neighboring lists with each other.

ii. both sender and receiver node (neighbor) update the neighboring list

The drop in the sum of distances after the first cycle is more than double compare to the previous scenario. This indicates that allowing both sender and receiver to share their partial views increases the rate of evolution of the network.





b For spectacles topology

- For the lower count of k, I see that network distance converged to a value quickly. However, visually the node graph showed distant connections between two nodes. As I increased value from 1 to 30 (1, 2, 5, 10, 30), the number of connection between distant structures (for e.g. a connection between nodes from both circles) decreased. As shown in figure 1-5.

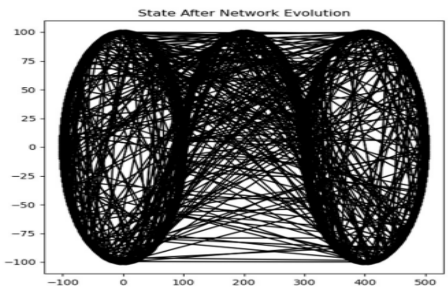


Figure 1: K = 1

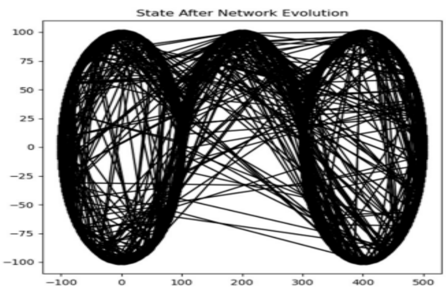


Figure 4: K = 2

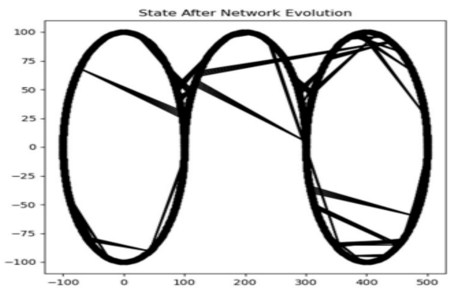


Figure 2: K = 5

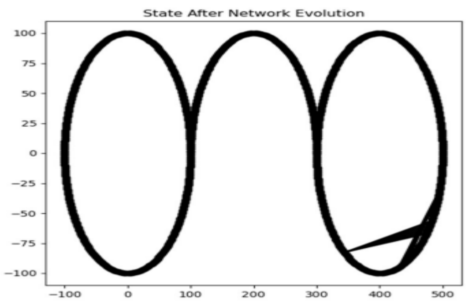


Figure 5: K = 10

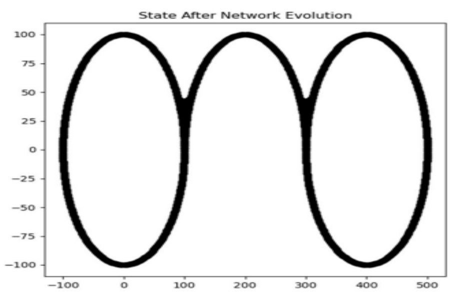


Figure 3: K = 30

- Upon working with various values of k, I found K = 1 but as per my understanding, this could change and is highly dependent on the criteria of distance. I am able to see the signs of connectedness with k = 1 with euclidean distance between nodes.

c

The neighbor's list is initialized randomly with unique neighbors, and then the optimization (evolution) phase sorts the distance between the node and it's neighbors. It is possible that a node to be equidistant from multiple neighbors but it's not possible to have multiple entries of a node in other node's neighbor list.

d

Please note that the code is written in python and I have tested it with `python v3.9` environment.

Additional packages required to run the program -

- pandas
- matplotlib
- sys
- random

Run: `python tman.py <number of nodes> <number of neighbors> <topology required>`

The Program will output the following files -

1. Node graph - `<topology>_N<>_k<>_<cycle>.png`
2. Node v/s neighbors list text document - `<topology>_N<>_k<>_<cycle>.txt`
3. Sum of distances for each cycle - `<topology>_N<>_k<>.txt`
4. Bar plot for sum of distances v/s # cycle - `<topology>_N<>_k<>.png`

Appendix - Code Files

`tman.py`

```
from algo import TMAN
import sys
import pandas as pd
import matplotlib.pyplot as plt

def main():

    # input and file name handling
    topology_dict = {
        'R': 'ring',
        'S': 'spectacles'
    }

    if len(sys.argv) != 4:
        print("Usage: python3 main_class.py <total_nodes> <total_neighbors>
<topology>")
        print("Example: python3 main_class.py 100 5 ring")
        exit(1)

    N = int(sys.argv[1])
    k = int(sys.argv[2])
    topology = topology_dict[sys.argv[3]]

    FILE_NAME = f'{sys.argv[3]}_N{N}_k{k}'

    # create FILE_NAME txt file to store sum of distances
    with open(f'{FILE_NAME}.txt', 'w') as f:
        f.write('cycle,sum_of_distances\n')

    if topology not in ['ring', 'spectacles']:
        print("Topology must be either 'ring' or 'spectacles'")
        exit(1)

    tman = TMAN(N, k, topology)
    nodes = tman.get_nodes()

    # The sum of distances of neighboring nodes during the initialization phase
    initial_sum_of_distances = tman.calculate_total_node_distance(nodes)
    # save the initial sum of distances in the csv file
    with open(f"{FILE_NAME}.txt", "a") as f:
        f.write(f"0,{initial_sum_of_distances}\n")

    # Perform network evolution
    tman.evolve_topology(nodes)
```

```
    def plot_cs_txt():
        df = pd.read_csv(f'{FILE_NAME}.txt', delimiter=',')
        df.columns = ['cycle', 'sum_of_distances']
        plt.bar(df['cycle'], df['sum_of_distances'])
        plt.xlabel('Cycle')
        plt.ylabel('Sum of Distances')
        plt.title('Sum of Distances vs Cycle')
        plt.savefig(f'{FILE_NAME}.png')
        plt.close()

    plot_cs_txt()

if __name__ == "__main__":

   main()
```

`algo.py`

```
import random
from node import Node
from distance import Distance
from topology import Topology
from network import Network
import matplotlib.pyplot as plt

class TMAN:
    """
    Methods:
    - get_nodes(): Get the list of nodes.
    - get_network_directory(): Get the network directory.
    - calculate_total_node_distance(nodes): Calculate the sum of distances of
neighboring nodes.
    - initialize_network(N, k): Initialize the network with nodes and random
connections.
    - select_k_nearest_neighbors(node, neighbor_list, k): Select k nearest
neighbors for a given node.
    - update_new_nearest_neighbors(node, old_neighbors, new_neighbors): Update
nearest neighbors based on new connections.
    - evolve_topology(nodes): Perform the optimization of the network based on
distance over a specified number of cycles.
    """

    # input handling
    topology_dict = {
        'ring': 'R',
        'spectacles': 'S'
    }
```

```
    def __init__(self, total_nodes, total_neighbors, topology='ring'):
            # CONSTANT - given with the homework statement
            self.NUM_OF_CYCLES = 40

            self.N = total_nodes
            self.k = total_neighbors
            self.topology = topology
            self.FILE_NAME = f'{self.topology_dict[topology]}_N{self.N}_k{self.k}'

            self.network = Network(self.N, self.k, self.topology)
            self.nodes = self.network.initialize_network()


    def get_nodes(self):
        return self.nodes

    def get_network_directory(self):
        return self.network.network_directory

    def calculate_total_node_distance(self, nodes):
        """
        Calculate the sum of distances of neighboring nodes during the
initialization phase.

        Args:
        - nodes (list): List of Node objects.

        Returns:
        - sum_of_distances (float): Sum of distances.
        """
        sum_of_distances = 0
        network_nodes_dict = self.get_network_directory()
        for node in nodes:
            for neighbor_id in node.get_neighbor_list():
                neighbor = network_nodes_dict[neighbor_id]
                node_distance = Distance(node, neighbor, self.topology)
                sum_of_distances += node_distance.calculate_distance()
        return sum_of_distances

    def select_k_nearest_neighbors(self, node, neighbor_list):
        """
        Select k nearest neighbors for a given node.

        Args:
        - node (Node): The node for which neighbors are to be selected.
        - neighbor_list (list): List of potential neighbors.

        Returns:
        - k_nearest_neighbor_ids (list): List of IDs of the k nearest neighbors.
        """
```

```
        node_distances = {}
        network_nodes_dict = self.get_network_directory()
        for neighbor_id in neighbor_list:
            neighbor = network_nodes_dict[neighbor_id]
            node_distance = Distance(node, neighbor, self.topology)
            node_distances[neighbor_id] = node_distance.calculate_distance()

        sorted_neighbors = sorted(node_distances.items(), key=lambda x: x[1])
        k_nearest_neighbor_ids = [neighbor[0] for neighbor in
sorted_neighbors[:self.k]]
        return k_nearest_neighbor_ids

    def update_new_nearest_neighbors(self, node, old_neighbors, new_neighbors):
        """
        Update nearest neighbors based on the gossip.

        Args:
        - node (Node): The node whose neighbors need to be updated.
        - old_neighbors (list): List of old neighbors.
        - new_neighbors (list): List of new neighbors.
        """
        unique_neighbors = list(set(old_neighbors + new_neighbors))
        k_nearest_neighbors = self.select_k_nearest_neighbors(node,
unique_neighbors)
        node.update_neighbor_list(k_nearest_neighbors)

    def evolve_topology(self, nodes):
        for _ in range(1, self.NUM_OF_CYCLES+1):
            for node in nodes:
                neighbor_id = node.select_random_neighbor()
                # gossiping
                node_partial_view = [node.id] + [neighbor for neighbor in
node.neighbors if neighbor != neighbor_id]

                network_nodes_dict = self.get_network_directory()
                neighbor = network_nodes_dict[neighbor_id]
                # gossiping
                neighbor_partial_view = [neighbor_id] + [neighbor for neighbor in
neighbor.neighbors if neighbor != node.id]

                # update both node and its neighbor with each other's partial view
                self.update_new_nearest_neighbors(node, node.get_neighbor_list(),
neighbor_partial_view)
                self.update_new_nearest_neighbors(neighbor,
neighbor.get_neighbor_list(), node_partial_view)

            total_distance = self.calculate_total_node_distance(nodes)
```

```
            '''
            Record below parameters -
            1. Sum of distances after each cycle
            2. Each node with their list of neighbors
            3. Node graph of the network
            '''
            # save cycle vs distance in a csv file
            with open(f"{self.FILE_NAME}.txt", "a") as f:
                f.write(f"{_},{total_distance}\n")

            if _ in [1, 5, 10, 15, self.NUM_OF_CYCLES]:
                # save the each node with their list of neighbors in the
self.file_name.txt file
                with open(f"{self.FILE_NAME}_{_}.txt", "w") as f:
                    f.write("node_id,neighbors\n")
                    for node in nodes:
                        f.write(f"{node.id},{node.get_neighbor_list()}\n")
                plt.figure(figsize=(6, 6))
                plt.title(f'State After Network Evolution Cycle {_}')
                final_network_nodes_dict = self.get_network_directory()
                for node in nodes:
                    x, y = node.location
                    plt.scatter(x, y, c=node.color)
                    # print(node.get_neighbor_list())
                    for neighbor_id in node.get_neighbor_list():
                        neighbor = final_network_nodes_dict[neighbor_id]
                        x_neighbor, y_neighbor = neighbor.location
                        plt.plot([x, x_neighbor], [y, y_neighbor], 'k-')
                plt.savefig(f'{self.FILE_NAME}_{_}.png')
                plt.close()
```

`topology.py`

```
import math

class Topology:

    def __init__(self, total_nodes, required_topology):
        self.total_nodes = total_nodes
        self.required_topology = required_topology

    def get_positions(self):
        # stragegy pattern
        if self.required_topology == 'ring':
            return self.ring()
        elif self.required_topology == 'spectacles':
            return self.spectacles()
```

```python
    def ring(self):
        RADIUS = 100

        node_positions = []
        for i in range(self.total_nodes):
            x = RADIUS * math.cos(2 * math.pi * i / self.total_nodes)
            y = RADIUS * math.sin(2 * math.pi * i / self.total_nodes)
            node_positions.append((x, y))
        return node_positions


    def spectacles(self):
        RADIUS = 100
        node_positions = []

        nodes_in_half_circle = self.total_nodes // 5
        # draw a circle
        nodes_in_circle = nodes_in_half_circle * 2

        # for each of the node we will calculate the x and y node_positions
        for i in range(nodes_in_circle):
            angle = 2 * math.pi * i / nodes_in_circle
            x = RADIUS * math.cos(angle)
            y = RADIUS * math.sin(angle)
            node_positions.append((x, y))

        # shift the x axis coordinate to the right by the radius
        for i in range(nodes_in_half_circle):
            angle = math.pi * i / nodes_in_half_circle
            x = RADIUS * math.cos(angle) + 2 * RADIUS
            y = RADIUS * math.sin(angle)
            node_positions.append((x, y))

        # Generate node_positions for the second circle
        for i in range(nodes_in_circle):
            angle = 2 * math.pi * i / nodes_in_circle
            x = RADIUS * math.cos(angle) + RADIUS * 4
            y = RADIUS * math.sin(angle)
            node_positions.append((x, y))

        return node_positions
```

node.py

```
import random

class Node:
    def __init__(self, id, color, location, type='ring'):
        self.id = id
        self.location = location
        self._neighbors = []
        self.color = color
        # spectacles nodes don't need rgb color value
        if type == 'ring':
            self.rgb_color = self.calculate_color_value(color)

    def select_random_neighbor(self):
        return random.choice(self.get_neighbor_list())

    def update_neighbor_list(self, new_neighbors):
        self.neighbors = new_neighbors

    def get_neighbor_list(self):
        return self.neighbors

    # for Ring nodes
    def calculate_rgb_to_xyz(self):
        r1, g1, b1 = self.rgb_color

        r1 = r1 / 255
        g1 = g1 / 255
        b1 = b1 / 255

        if r1 > 0.04045:
            r1 = ((r1 + 0.055) / 1.055) ** 2.4
        else:
            r1 = r1 / 12.92
        if g1 > 0.04045:
            g1 = ((g1 + 0.055) / 1.055) ** 2.4
        else:
            g1 = g1 / 12.92
        if b1 > 0.04045:
            b1 = ((b1 + 0.055) / 1.055) ** 2.4
        else:
            b1 = b1 / 12.92

        r1 = r1 * 100
        g1 = g1 * 100
        b1 = b1 * 100

        x1 = r1 * 0.4124 + g1 * 0.3576 + b1 * 0.1805
        y1 = r1 * 0.2126 + g1 * 0.7152 + b1 * 0.0722
        z1 = r1 * 0.0193 + g1 * 0.1192 + b1 * 0.9505
```

```
        return (x1, y1, z1)

    def calculate_xyz_to_lab(self, x, y, z):
        x = x / 95.047
        y = y / 100.000
        z = z / 108.883

        if x > 0.008856:
            x = x ** (1/3)
        else:
            x = (7.787 * x) + (16 / 116)
        if y > 0.008856:
            y = y ** (1/3)
        else:
            y = (7.787 * y) + (16 / 116)
        if z > 0.008856:
            z = z ** (1/3)
        else:
            z = (7.787 * z) + (16 / 116)

        l = (116 * y) - 16
        a = 500 * (x - y)
        b = 200 * (y - z)

        return (l, a, b)

    def calculate_color_value(self, color):
        major_color_intensity = random.randint(200, 255)
        minor_color_intensity_1 = random.randint(0, 80)
        minor_color_intensity_2 = random.randint(0, 80)

        rgb_value = ()
        # create RGB tuple based on the color value
        if color == 'Red':
            rgb_value = (major_color_intensity, minor_color_intensity_1,
minor_color_intensity_2)
        elif color == 'Green':
            rgb_value = (minor_color_intensity_1, major_color_intensity,
minor_color_intensity_2)
        elif color == 'Blue':
            rgb_value = (minor_color_intensity_1, minor_color_intensity_2,
major_color_intensity)

        return rgb_value
```

`distance.py`

```python
class Distance:

    def __init__(self, node1, node2, topology='ring'):
        self.node1 = node1
        self.node2 = node2
        if topology == 'ring':
            self.strategy = "color"
        else:
            self.strategy = "euclidean"

    def calculate_distance(self):
        if self.strategy == "euclidean":
            return self.calculate_euclidean_distance()
        elif self.strategy == "color":
            return self.calculate_color_distance()

    def calculate_color_distance(self):
        xyz1 = self.node1.calculate_rgb_to_xyz()
        x1, y1, z1 = xyz1
        xyz2 = self.node2.calculate_rgb_to_xyz()
        x2, y2, z2 = xyz2

        lab1 = self.node1.calculate_xyz_to_lab(x1,y1,z1)
        lab2 = self.node2.calculate_xyz_to_lab(x2,y2,z2)

        l1, a1, b1 = lab1
        l2, a2, b2 = lab2

        return ((l1 - l2) ** 2 + (a1 - a2) ** 2 + (b1 - b2) ** 2) ** 0.5

    def calculate_euclidean_distance(self):
        x1, y1 = self.node1.location
        x2, y2 = self.node2.location

        return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
```

`network.py`

```python
from topology import Topology
import random
from node import Node
```

```python
class Network:

    def __init__(self, total_nodes, total_neighbors, topology='ring'):
        self.N = total_nodes
        self.k = total_neighbors
        self.topology = topology
        # NODE_ID: NODE directory
        self.network_directory = {}

    def initialize_network(self):
        if self.topology == 'ring':
            nodes = self.initialize_ring_network()
        elif self.topology == 'spectacles':
            nodes = self.initialize_spectacles_network()

        return self.attach_neighbors(nodes)

    def initialize_ring_network(self):
        nodes = []
        colors = ['Red', 'Green', 'Blue']
        node_positions = Topology(self.N, 'ring').get_positions()
        for i in range(self.N):
            node_id = random.randint(0, 1000000)
            position = node_positions[i]
            color = colors[i % len(colors)]
            node = Node(node_id, color, position)
            self.network_directory[node_id] = node
            nodes.append(node)
        return nodes

    def initialize_spectacles_network(self):
        nodes = []
        node_positions = Topology(self.N, 'spectacles').get_positions()
        for i in range(self.N):
            node_id = random.randint(0, 1000000)
            position = node_positions[i]
            # all nodes are of the same color
            node = Node(node_id, 'black', position)
            self.network_directory[node_id] = node
            nodes.append(node)
        return nodes

    def attach_neighbors(self, nodes):
        # assign neighbors to each node
        for node in nodes:
            potential_neighbors = [n for n in nodes if n != node]
            neighbors = random.sample(potential_neighbors, self.k)
            neighbor_ids = [n.id for n in neighbors]
            node.update_neighbor_list(neighbor_ids)
        return nodes
```