import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.axes\_grid1.inset\_locator import zoomed\_inset\_axes

from mpl\_toolkits.axes\_grid1.inset\_locator import mark\_inset

import random

from matplotlib.ticker import (MultipleLocator, AutoMinorLocator)

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

import networkx as nx

P = 20

L = 20

d\_max = 2

base\_coor = [P/2, L/2, 0]

N = 200

m = 10

kc = 25

kd = 250

kt = kd + N

beta = 0.8

data\_rate = 500 # bps

t\_transmission =  ((kd \* 8)/data\_rate) + ((kc \* 8)/data\_rate) + (((kc + N) \* 8)/data\_rate)

initial\_energy = 5

E\_elec = 50 \* 1e-9

E\_amp =  10 \* 1e-5

E\_fs = 10\*1e-12

E\_mp = 0.0013\*1e-12

p = 0.05

DECKS\_threshold = 3

def ratio\_delay(D, distance):

    v = 3 \* 1e8

    t = distance/v

    a = t/t\_transmission

    # print(f"Ratio delay : {a:.2f} | T\_tranmission : {t\_transmission:.2f} | T\_uw : {t\_uw:.2f}")

    return a

def throughput(a):

    numerator = kc \* np.exp(-a \* kc)

    denominator = kc \* (1 + 2 \* a) + np.exp(-a \* kc)

    rho = numerator / denominator

    # print(f"Throughput : {rho:.2f} | numerator : {numerator:.2f} | denominator : {denominator:.2f}")

    return rho

class Node:

    def \_\_init\_\_(self, x, y, z, id):

        self.x = x

        self.y = y

        self.z = z

        self.energy = initial\_energy

        self.nϵG = False

        self.alive = True

        self.CH = False

        self.which\_cluster = 0

        self.eligible\_round = 0

        self.cluster\_class = 0

        self.id = id

    def distance(self, other\_node):

        return np.sqrt((self.x - other\_node.x)\*\*2 + (self.y - other\_node.y)\*\*2)

    def reset(self):

        self.CH = False

        self.which\_cluster = 0

    def advertisement(self, count\_cluster, eligible\_round):

        self.CH = True

        self.nϵG = False

        self.eligible\_round = eligible\_round

        self.which\_cluster = count\_cluster

    def energySelection(self, d\_CH\_BS, p):

        # If node selected as a CH

        a = ratio\_delay(self.z, d\_CH\_BS)

        rho = throughput(a)

        first\_term = (kc / rho) \* (E\_elec + E\_amp \* (d\_CH\_BS\*\*2))

        second\_term = (p \* (N - 1) / rho) \* (beta \* E\_elec \* kc)

        third\_term = E\_elec \* kt

        E\_Selection = first\_term + second\_term + third\_term

        return E\_Selection

    def energyAdvertisement(self):

        # Broadcasting to all nodes in the range of d\_max, occurs only for CH

        return kc \* (E\_elec + E\_amp \* (d\_max\*\*2))

    def energyJoin(self, p):

        # Node receive the broadcasting message and decide whether want to join as a associated node for CH i-th

        return p \* N \* kc \* E\_elec

    def energy\_contention\_TDMA\_CH(self, Nc):

        return kc \* Nc \* E\_elec + kt \* (E\_elec + E\_amp \* (d\_max\*\*2))

    def energy\_contention\_TDMA\_Node(self, d\_CH\_Node):

        a = ratio\_delay(self.z, d\_CH\_Node)

        rho = throughput(a)

        energy = (kc / rho) \* (E\_elec + E\_amp \* (d\_CH\_Node\*\*2)) + ((N-1)/ rho) \* kc \* beta \* E\_elec + kt \* E\_elec

        return energy

    def energyFrame\_CH(self, Nc, d\_CH\_BS):

        return m \* Nc \* kd \* E\_elec \* beta \* kd \* E\_elec + kd \* (E\_elec +  E\_amp \* (d\_CH\_BS\*\*2))

    def energyFrame\_Node(self, d\_CH\_Node):

        return m \* kd \* E\_elec \* (E\_elec + E\_elec +  E\_amp \* (d\_CH\_Node\*\*2))

def createNetworks():

    areaTotal = P \* L

    #Point process parameters

    lambda0 = N/(P \* L)

    #Simulate a Poisson point process

    numbPoints = np.random.poisson(lambda0 \* areaTotal)

    xx = P \* np.random.uniform(0,1,numbPoints)

    yy = L \* np.random.uniform(0,1,numbPoints)

    nodes = []

    nodes.append(Node(P//2, L//2, 0, 0))

    for i in range(len(xx)):

        nodes.append(

            Node(xx[i], yy[i], 0, i + 1)

        )

    return nodes

class multiHop(object):

    def \_\_init\_\_(self,graph):

        self.graph = graph

        self.adjacent\_mat = nx.adjacency\_matrix(graph).todense()

        self.num\_nodes = len(self.adjacent\_mat)

        self.adjacent\_mat = nx.adjacency\_matrix(graph, nodelist=range(self.num\_nodes)).toarray()#:D

        # print(f"\n Adjacent Matrix \n{self.adjacent\_mat}\n")

    def q\_learning(self,start\_state=0, aim\_state = 10, num\_epoch=200, gamma=0.8, epsilon=0.05, alpha=0.1):

        len\_of\_paths = []

        rewards = self.rewardMapping(aim\_state)

        # print(f"\n Reward \n{rewards}\n")

        q\_table = np.zeros((self.num\_nodes, self.num\_nodes))  # num\_states \* num\_actions

        td = []

        for episode in range(1, num\_epoch + 1):

            #print(f"========================================================================== Episode : {episode} =========================================================================")

            current\_state = start\_state

            path = [current\_state]

            len\_of\_path = 0

            while True:

                next\_state = self.epsilon\_greedy(current\_state, q\_table, start\_state, epsilon=epsilon)

                s\_next\_next = self.epsilon\_greedy(next\_state, q\_table, start\_state, epsilon=epsilon)  # epsilon<0, greedy policy

                # update q\_table

                reward = rewards[current\_state][next\_state]

                delta = reward + gamma \* q\_table[next\_state, s\_next\_next] - q\_table[current\_state, next\_state]

                q\_table[current\_state, next\_state] = q\_table[current\_state, next\_state] + alpha \* delta

                # update current state

                current\_state = next\_state

                len\_of\_path += -reward

                path.append(current\_state)

                # print(f"==========================\n{q\_table}==========================\n")

                # print(f"reward: {reward} | Current state : {current\_state} | Next state : {next\_state} \n\n")

                td.append(delta)

                if current\_state == aim\_state:

                    break

            len\_of\_paths.append(len\_of\_path)

        # print(f"Q learning table : \n{q\_table}")

        return path, td

    def epsilon\_greedy(self,s\_curr, q, start\_state, epsilon):#exploraiton vs exploitation

        try :

            potential\_next\_states = np.where(np.array(self.adjacent\_mat[s\_curr]) > 0)[0]

        except IndexError as e:

            print(e)

            print(f"{self.adjacent\_mat[s\_curr]}")

        potential\_next\_states = potential\_next\_states[potential\_next\_states != start\_state]

        # print(f"potential next state : {potential\_next\_states}")

        if random.random() > epsilon:

            q\_of\_next\_states = q[s\_curr][potential\_next\_states]

            s\_next = potential\_next\_states[np.argmax(q\_of\_next\_states)]

            # print(f"q\_of\_next\_states : {q\_of\_next\_states}   |   s\_next : {s\_next}")

        else:

            s\_next = random.choice(potential\_next\_states)

        return s\_next

    def rewardMapping(self, aim\_state):

        r = self.adjacent\_mat

        r[:, aim\_state] = 10

        r[aim\_state, :] = 10

        for i in range(0, len(r)):

            for j in range(0, len(r)):

                if i == j :

                    r[i][j] = -5

        return r

def createNetworks(N, P, L):

    areaTotal = P \* L

    #Point process parameters

    lambda0 = N/(P \* L)

    #Simulate a Poisson point process

    numbPoints = np.random.poisson(lambda0 \* areaTotal)

    xx = P \* np.random.uniform(0,1,numbPoints)

    yy = L \* np.random.uniform(0,1,numbPoints)

    nodes = []

    nodes.append(Node(P//2, L//2, 0, 0))

    for i in range(len(xx)):

        nodes.append(

            Node(xx[i], yy[i], 0, i + 1)

        )

    return nodes

smallArchitecture = createNetworks(10, P, L)

edges = []

nodes\_distance = [node1.distance(node2) for node1 in smallArchitecture for node2 in smallArchitecture if node1 != node2]

nodes\_energy = [node1.energy for node1 in smallArchitecture]

avg\_distances = sum(nodes\_distance)/len(nodes\_distance)

avg\_energy = sum(nodes\_energy)/len(nodes\_energy)

for p1 in smallArchitecture:

    for p2 in smallArchitecture:

        if p1 != p2 :

            weight\_node = (p1.distance(p2)/avg\_distances)

            edges.append((p1.id, p2.id, weight\_node))

G = nx.Graph()

for start, end, length in edges:

    G.add\_edge(start, end, weight=length)

# Tuning Gamma

rl\_gamma\_80 = multiHop(G)

res, td\_gamma\_80 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.8, epsilon=0.01, alpha=0.1)

rl\_gamma\_50 = multiHop(G)

res, td\_gamma\_50 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.5, epsilon=0.01, alpha=0.1)

rl\_gamma\_20 = multiHop(G)

res, td\_gamma\_20 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.1, epsilon=0.01, alpha=0.1)

# Tuning Epsilon Greedy

rl\_epsilon\_8 = multiHop(G)

res, td\_epsilon\_8 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.08, alpha=0.1)

rl\_epsilon\_5 = multiHop(G)

res, td\_epsilon\_5 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.05, alpha=0.1)

rl\_epsilon\_2 = multiHop(G)

res, td\_epsilon\_2 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.02, alpha=0.1)

# Tuning Alpha

rl\_alpha\_10 = multiHop(G)

res, td\_alpha\_10 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.02, alpha=0.1)

rl\_alpha\_5 = multiHop(G)

res, td\_alpha\_5 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.02, alpha=0.05)

rl\_alpha\_2 = multiHop(G)

res, td\_alpha\_2 = rl\_gamma\_80.q\_learning(start\_state=0, aim\_state = 2, num\_epoch=1000, gamma=0.2, epsilon=0.02, alpha=0.01)

plt.figure(figsize=(12,12))

nx.draw(G, with\_labels=True)

plt.show()

fig, ax = plt.subplots(1,3, figsize=(30,5))

ax[0].plot([i for i in range(len(td\_gamma\_80))], td\_gamma\_80, label=r"$γ = 0.8$")

ax[0].plot([i for i in range(len(td\_gamma\_50))], td\_gamma\_50, label=r"$γ = 0.5$")

ax[0].plot([i for i in range(len(td\_gamma\_20))], td\_gamma\_20, label=r"$γ = 0.2$")

ax[0].set\_title(f"$Variasi \,\, nilai \,\, γ \, pada \,\, Q-Learning \,\, ε=0.01,\, α=0.1$")

ax[0].set\_ylabel("Temporal Difference")

ax[0].legend()

ax[1].plot([i for i in range(len(td\_epsilon\_8))], td\_epsilon\_8, label=r"$ε = 0.08$")

ax[1].plot([i for i in range(len(td\_epsilon\_5))], td\_epsilon\_5, label=r"$ε = 0.05$")

ax[1].plot([i for i in range(len(td\_epsilon\_2))], td\_epsilon\_2, label=r"$ε = 0.02$")

ax[1].set\_title(f"$Variasi \,\, nilai \,\, ε \, pada \,\, Q-Learning \,\, γ=0.02,\, α=0.1$")

ax[1].set\_ylabel("Temporal Difference")

ax[1].legend()

ax[2].plot([i for i in range(len(td\_alpha\_2))], td\_alpha\_2, label=r"$α = 0.01$")

ax[2].plot([i for i in range(len(td\_alpha\_5))], td\_alpha\_5, label=r"$α = 0.05$")

ax[2].plot([i for i in range(len(td\_alpha\_10))], td\_alpha\_10, label=r"$α = 0.1$")

ax[2].set\_title(f"$Variasi \,\, nilai \,\, ε \, pada \,\, Q-Learning \,\, γ=0.02,\, ε=0.02$")

ax[2].set\_ylabel("Temporal Difference")

ax[2].legend()

plt.tight\_layout()

plt.show()

class networkEnvironment:

    def \_\_init\_\_(self, nodes, mode):

        self.nodes = nodes

        self.mode = mode

        self.alive\_data = []

        self.energy\_data = []

        self.centroids = np.array([[P//2, L//2]])

    def showResult(self, hop):

        fig, ax = plt.subplots(1,2, figsize=(20,5))

        rounds = np.array([i for i in range(0, len(self.alive\_data))])

        self.alive\_data = np.array(self.alive\_data)

        self.energy\_data = np.array(self.energy\_data)

        ax[0].plot(rounds, self.alive\_data, color='k')

        ax[0].scatter(rounds[::hop], self.alive\_data[::hop], marker='o', edgecolor='k', color='r')

        ax[0].set\_ylabel("Node Alive")

        ax[0].set\_xlabel("Round")

        ax[1].plot(rounds, self.energy\_data, color='k')

        ax[1].scatter(rounds[::hop], self.energy\_data[::hop], marker='o', edgecolor='k', color='r')

        ax[1].set\_ylabel("Energy Consumed")

        ax[1].set\_xlabel("Round")

        ax[1].yaxis.set\_major\_locator(MultipleLocator(np.max(self.energy\_data)/10))

        ax[1].yaxis.set\_major\_formatter('{x:.0f}')

        ax[1].yaxis.set\_minor\_locator(MultipleLocator(np.max(self.energy\_data)/20))

        ax[1].xaxis.set\_major\_locator(MultipleLocator(np.max(rounds)/10))

        ax[1].xaxis.set\_major\_formatter('{x:.0f}')

        ax[1].xaxis.set\_minor\_locator(MultipleLocator(np.max(rounds)/20))

        ax[0].yaxis.set\_major\_locator(MultipleLocator(np.max(self.alive\_data)/10))

        ax[0].yaxis.set\_major\_formatter('{x:.0f}')

        ax[0].yaxis.set\_minor\_locator(MultipleLocator(np.max(self.alive\_data)/20))

        ax[0].xaxis.set\_major\_locator(MultipleLocator(np.max(rounds)/10))

        ax[0].xaxis.set\_major\_formatter('{x:.0f}')

        ax[0].xaxis.set\_minor\_locator(MultipleLocator(np.max(rounds)/20))

    def showNetwork(self, simulation\_round):

        fig, ax = plt.subplots()

        nodes\_x = [node.x for node in self.nodes if ((node.id != 0) and (node.CH == False) and (node.which\_cluster != 0) and (node.energy > 0))]

        nodes\_y = [node.y for node in self.nodes if ((node.id != 0) and (node.CH == False) and (node.which\_cluster != 0) and (node.energy > 0))]

        nodes\_orphan\_x = [node.x for node in self.nodes if ((node.id != 0) and (node.CH == False) and (node.which\_cluster == 0) and (node.energy > 0))]

        nodes\_orphan\_y = [node.y for node in self.nodes if ((node.id != 0) and (node.CH == False) and (node.which\_cluster == 0) and (node.energy > 0))]

        CH\_x = [node.x for node in self.nodes if ((node.id != 0) and (node.CH == True))]

        CH\_y = [node.y for node in self.nodes if ((node.id != 0) and (node.CH == True))]

        BS\_x, BS\_y = [node.x for node in self.nodes if node.id == 0], [node.y for node in self.nodes if node.id == 0]

        dead\_nodes\_x = [node.x for node in self.nodes if ((node.id != 0) and (node.energy < 0))]

        dead\_nodes\_y = [node.y for node in self.nodes if ((node.id != 0) and (node.energy < 0))]

        ax.scatter(nodes\_x, nodes\_y                       , marker="o",color='purple', edgecolors='k', label = "Node")

        ax.scatter(nodes\_orphan\_x, nodes\_orphan\_y         , marker="o",color='c', edgecolors='k', label = "Orphan Node")

        ax.scatter(CH\_x, CH\_y                             , marker="o",color="g", edgecolors='k', label = "Cluster Head")

        ax.scatter(dead\_nodes\_x, dead\_nodes\_y             , marker="x",color='r', label="Dead Node")

        ax.scatter(self.centroids[:,0], self.centroids[:,1], marker="o",color="y", edgecolors='k', label="Centroid")

        ax.scatter(BS\_x, BS\_y                             , marker="s",color="b", edgecolors='k', label="Base station")

        font = {

                'color':  'black',

                'weight': 'bold'

                }

        ax.set\_title(f'Round : {simulation\_round}', fontdict=font)

        ax.set\_xlabel('Length (m)')

        ax.set\_ylabel('Width (m)')

        ax.legend(loc='upper center', bbox\_to\_anchor=(0.5, -0.15),

              fancybox=True, shadow=True, ncol=5, markerscale=1, fontsize=10)

        plt.tight\_layout()

        plt.show()

    def is\_alive\_and\_eligible(self, node):

        status = ((node.energy > 0) and (node.alive == True) and (node.eligible\_round == 0))

        return status

    def is\_lessEqual\_than\_threshold(self, round\_number):

        if random.uniform(0, 1) <= p/(1-p \* (round\_number % (1/p))):

            return True

        else:

            return False

    def euclidean\_distance(self, nodeA, nodeB):

        return np.sqrt(np.sum((nodeA - nodeB)\*\*2))

    def kmeans\_plusplus(self, X, k):

        centroids = []

        centroids.append(X[np.random.randint(X.shape[0])])

        for \_ in range(1, k):

            distances = np.array([min([np.linalg.norm(x - c) for c in centroids]) for x in X])

            probabilities = distances / distances.sum()

            cumulative\_probabilities = probabilities.cumsum()

            r = np.random.rand()

            for j, p in enumerate(cumulative\_probabilities):

                if r < p:

                    i = j

                    break

            centroids.append(X[i])

        return np.array(centroids)

    def kmeans(self, X, k, iteration=100):

        # Memilih random k centroid sebagai nilai awal

        centroids = self.kmeans\_plusplus(X, k)

        for i in range(iteration):

            # Euclidian Distance

            distances = np.linalg.norm(X[:, None] - centroids, axis=2)

            labels = np.argmin(distances, axis=1)

            # Mengupdate nilai Centroid

            new\_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)])

            centroids = new\_centroids

        return centroids, labels

    def multiHopRouting(self, clusterHead\_id):

        CHs = [node for node in self.nodes if node.id == 0] + [node for node in self.nodes if node.CH == True]

        CHsID = np.array([node.id for node in CHs])

        start\_state = np.where(np.isin(CHsID, clusterHead\_id))[0][0]

        print(f"Start state : {start\_state}")

        edges = []

        X, Y, Z, E, id = [], [], [], [], []

        for ch in CHs:

            X.append(ch.x)

            Y.append(ch.y)

            E.append(ch.energy)

            id.append(ch.id)

        distance = np.array([X, Y]).T

        E = np.array(E).T

        G = nx.Graph()

        for i in range(0, self.k+1):

            for j in range(0, self.k+1):

                if i != j:

                    distance\_normalized = self.euclidean\_distance(distance[i, :], distance[j, :])/np.mean(distance)

                    energy\_normalized = E[j]/np.mean(E)

                    weight = 1/distance\_normalized + energy\_normalized

                    edges.append((i,j, weight))

        for start, end, length in edges:

            G.add\_edge(start, end, weight=length)

        map = multiHop(G)

        paths, \_ = map.q\_learning(start\_state=start\_state, aim\_state = 0, num\_epoch=10000, gamma=0.2, epsilon=0.02, alpha=0.1)#start\_state=0,aim\_state = 10

        ID\_path = []

        print(f"{paths}")

        for path in paths:

            ID\_path.append(CHsID[path])

        print(f"ID ; {ID\_path} | path : {paths}")

    def CHselection(self, simulation\_round):

        # Memilih Cluster Head

        print(f"====================================== Round : {simulation\_round} ======================================")

        for node in self.nodes:

            node.reset()

        if self.mode == "LEACH":

            for node in self.nodes:

                if ((self.is\_lessEqual\_than\_threshold(simulation\_round)) and (self.is\_alive\_and\_eligible(node)) and (node.id != 0)):

                    node.CH = True

                    node.which\_cluster = node.id

                    node.eligible\_round = 1/p

        elif self.mode == "K-Means":

            X, id = [], []

            for node in self.nodes:

                if ((node.id != 0) and (node.alive) and (node.energy > 0)) :

                    X.append([node.x, node.y])

                    id.append(node.id)

            X = np.array(X)

            elbow = []

            max\_k = 30  # Maximum number of clusters to try

            for k in range(1, max\_k + 1):

                centroids, labels = self.kmeans(X, k)

                error = np.sum((X - centroids[labels])\*\*2)

                elbow.append(error)

            # Calculate the change in distortions and find the elbow point

            elbow = np.array(elbow)

            elbow\_diff = np.diff(elbow, prepend=elbow[0])

            acceleration = np.diff(elbow\_diff, prepend=elbow\_diff[0])

            optimal\_k = np.argmax(acceleration)

            self.centroids, labels = self.kmeans(X, optimal\_k)

            print(f"K-opt : {optimal\_k}")

            ch\_id = []

            for ch in self.centroids:

                distances = [[], []]

                for node in self.nodes:

                    if node.id != 0:

                        X = np.array([node.x, node.y])

                        CH = np.array([ch[0], ch[1]])

                        distances[0].append(self.euclidean\_distance(X, CH))

                        distances[1].append(node.id)

                ch\_id.append(distances[1][np.argmin(distances[0])])

            for node in self.nodes:

                if node.id != 0:

                    for id in ch\_id:

                        if node.id == id:

                            node.CH = True

                            node.which\_cluster = id

        elif self.mode == "DECKS":

            X, id = [], []

            for node in self.nodes:

                if ((node.id != 0) and (node.alive) and (node.energy > 0)) :

                    X.append([node.x, node.y])

                    id.append(node.id)

            X = np.array(X)

            elbow = []

            max\_k = 30  # Maximum number of clusters to try

            for k in range(1, max\_k + 1):

                centroids, labels = self.kmeans(X, k)

                error = np.sum((X - centroids[labels])\*\*2)

                elbow.append(error)

            # Calculate the change in distortions and find the elbow point

            elbow = np.array(elbow)

            elbow\_diff = np.diff(elbow, prepend=elbow[0])

            acceleration = np.diff(elbow\_diff, prepend=elbow\_diff[0])

            optimal\_k = np.argmax(acceleration)

            self.centroids, labels = self.kmeans(X, optimal\_k)

            labels = np.array(labels)

            number\_of\_clusters = np.unique(labels)

            for cluster in number\_of\_clusters:

                count = 0

                within\_cluster = []

                for label in labels:

                    if label == cluster:

                        within\_cluster.append(self.nodes[count])

                    count += 1

                nearest\_distance = []

                for node\_i in within\_cluster:

                    sum\_euclidian = 0

                    for node\_j in within\_cluster:

                        if ((node\_i != node\_j) and (node\_i.energy > DECKS\_threshold) and (node\_j.energy > 0)):

                            sum\_euclidian += self.euclidean\_distance(np.array([node\_i.x, node\_i.y]), np.array([node\_j.x, node\_j.y]))

                    nearest\_distance.append(sum\_euclidian)

                nearest\_distance = np.array(nearest\_distance)

                CH\_id = within\_cluster[np.argmin(nearest\_distance)]

                for node in self.nodes:

                  if node.id == CH\_id.id :

                      node.CH = True

                      node.which\_cluster = node.id

        elif self.mode == "Q-Learning":

            X, id = [], []

            for node in self.nodes:

                if ((node.id != 0) and (node.alive) and (node.energy > 0)) :

                    X.append([node.x, node.y])

                    id.append(node.id)

            X = np.array(X)

            optimal\_k = 1

            max\_silhouette\_score = -1  # Initialize with a low value

            for k in range(2, 30):

                self.centroids, labels = self.kmeans(X, k)

                silhouette\_avg = silhouette\_score(X, labels)

                if silhouette\_avg > max\_silhouette\_score:

                    max\_silhouette\_score = silhouette\_avg

                    optimal\_k = k

            self.centroids, labels = self.kmeans(X, optimal\_k)

            labels = np.array(labels)

            number\_of\_clusters = np.unique(labels)

            labels = np.array(labels)

            number\_of\_clusters = np.unique(labels)

            for cluster in number\_of\_clusters:

                count = 0

                within\_cluster = []

                for label in labels:

                    if label == cluster:

                        within\_cluster.append(self.nodes[count])

                    count += 1

                nearest\_distance = []

                for node\_i in within\_cluster:

                    sum\_euclidian = 0

                    for node\_j in within\_cluster:

                        if ((node\_i != node\_j) and (node\_i.energy > DECKS\_threshold) and (node\_j.energy > 0)):

                            sum\_euclidian += self.euclidean\_distance(np.array([node\_i.x, node\_i.y]), np.array([node\_j.x, node\_j.y]))

                    nearest\_distance.append(sum\_euclidian)

                nearest\_distance = np.array(nearest\_distance)

                CH\_id = within\_cluster[np.argmin(nearest\_distance)]

                for node in self.nodes:

                  if node.id == CH\_id.id :

                      node.CH = True

                      node.which\_cluster = node.id

        # Node Bergabung ke Cluster Head

        CHs = [node for node in self.nodes if node.CH]

        count = 0

        if len(CHs) != 0:

            for node in self.nodes:

                if node.id != 0:

                    distances = [[], []]

                    for ch in CHs:

                        X = np.array([node.x, node.y])

                        CH = np.array([ch.x, ch.y])

                        distances[0].append(self.euclidean\_distance(X, CH))

                        distances[1].append(ch.which\_cluster)

                    node.which\_cluster = distances[1][np.argmin(distances[0])]

        else:

            # Jika tidak ada yang terpilih sebagai CH, maka node akan diam

            for node in self.nodes:

                if node.id != 0:

                    node.which\_cluster = 0

        print(f"Idle Node : {count}")

        return len(CHs)

    def Nc(self, which\_cluster):

        count = 0

        for node in self.nodes:

            if node.which\_cluster == which\_cluster:

                count += 1

        return count

    def SetupPhase(self, simulation\_round):

        self.k = self.CHselection(simulation\_round)

        energy\_total = sum([node.energy for node in self.nodes])

        print(f"Setup phase energy initial : {energy\_total}")

        if self.k != 0:

            for node in self.nodes:

                if node.id != 0:

                    if node.CH:

                        d\_CH\_BS = self.euclidean\_distance(np.array([node.x, node.y, node.z]), np.array(base\_coor))

                        energy\_dissipated = node.energyAdvertisement() + node.energySelection(d\_CH\_BS, self.k/N)

                        node.energy = node.energy - energy\_dissipated

                    else:

                        node.energy = node.energy - node.energyJoin(self.k/N)

        else:

            print(f"There is no transmission in {simulation\_round} round")

        energy\_total = sum([node.energy for node in self.nodes])

        print(f"Setup phase energy after : {energy\_total}")

    def SteadyStatePhase(self):

        BS = [node for node in self.nodes if node.id == 0][0]

        CHs = [node for node in self.nodes if node.CH]

        if len(CHs) != 0:

            energy\_total = sum([node.energy for node in self.nodes])

            print(f"Contention phase energy initial : {energy\_total:.2f}")

            for node in self.nodes:

                if node.id != 0 :

                    if node.CH == True:

                        Nc = self.Nc(node.which\_cluster)

                        node.energy = node.energy - node.energy\_contention\_TDMA\_CH(Nc)

                    else:

                        for CH in CHs:

                            if node.which\_cluster == CH.which\_cluster :

                                d\_CH\_Node = self.euclidean\_distance(np.array([node.x, node.y, node.z]), np.array([CH.x, CH.y, CH.z]))

                                node.energy = node.energy - node.energy\_contention\_TDMA\_Node(d\_CH\_Node)

            energy\_total = sum([node.energy for node in self.nodes])

            print(f"Transmission energy begin : {energy\_total:.2f}")

            if self.mode != "Q-Learning":

                for node in self.nodes:

                    if node.id != 0 :

                        if node.CH == True:

                            Nc = self.Nc(node.which\_cluster)

                            d\_CH\_BS = self.euclidean\_distance(np.array([node.x, node.y]), np.array([BS.x, BS.y]))

                            node.energy = node.energy - node.energyFrame\_CH(Nc, d\_CH\_BS)

                        else:

                            for CH in CHs:

                                if node.which\_cluster == CH.which\_cluster :

                                    d\_CH\_Node = self.euclidean\_distance(np.array([node.x, node.y]), np.array([CH.x, CH.y]))

                                    node.energy = node.energy - node.energyFrame\_Node(d\_CH\_Node)

            else:

                for node in self.nodes:

                      if node.id != 0 :

                          if node.CH == True:

                              Nc = self.Nc(node.which\_cluster)

                              d\_CH\_BS = self.euclidean\_distance(np.array([node.x, node.y]), np.array([BS.x, BS.y]))

                              node.energy = node.energy - node.energyFrame\_CH(Nc, d\_CH\_BS)

                              self.multiHopRouting(node.id)

                          else:

                              for CH in CHs:

                                  if node.which\_cluster == CH.which\_cluster :

                                      d\_CH\_Node = self.euclidean\_distance(np.array([node.x, node.y]), np.array([CH.x, CH.y]))

                                      node.energy = node.energy - node.energyFrame\_Node(d\_CH\_Node)

        for node in self.nodes:

            node.eligible\_round -= 1

            if node.eligible\_round < 0:

                node.eligible\_round = 0

            if node.energy < 0:

                node.alive = False

        energy\_total = sum([node.energy for node in self.nodes])

        node\_alive = len([node.energy for node in self.nodes if node.alive])

        print(f"Node Alive : {node\_alive} | CH : {len(CHs)}")

        self.alive\_data.append(node\_alive)

        self.energy\_data.append(energy\_total)

    def startSimulation(self, rounds):

        self.showNetwork(0)

        for simulation\_round in range(1, rounds):

            self.SetupPhase(simulation\_round)

            self.SteadyStatePhase()

            if simulation\_round % 10 == 0:

                self.showNetwork(simulation\_round)

        self.showResult(10)

        return self.alive\_data, self.energy\_data

node\_LEACH    = createNetworks(N, P, L)

node\_K\_Means  = createNetworks(N, P, L)

node\_DECKS    = createNetworks(N, P, L)

node\_proposed = createNetworks(N, P, L)

LEACH = networkEnvironment(node\_LEACH, "LEACH")

LEACH\_aliveNode, LEACH\_EnergyNode = LEACH.startSimulation(150)

DECKS = networkEnvironment(node\_DECKS, "DECKS")

DECKS\_aliveNode, DECKS\_EnergyNode = DECKS.startSimulation(150)

K\_Means = networkEnvironment(node\_K\_Means, "K-Means")

K\_Means\_aliveNode, K\_Means\_EnergyNode = K\_Means.startSimulation(150)

Q\_learning = networkEnvironment(node\_proposed, "Q-Learning")

Q\_learning\_aliveNode, Q\_learning\_EnergyNode = Q\_learning.startSimulation(200)

fig, ax = plt.subplots(1,2, figsize=(20, 5))

rounds = [i for i in range(len(DECKS\_aliveNode))]

# ax[0].scatter(rounds, DECKS\_aliveNode, label="DECKS")

# ax[0].scatter(rounds, LEACH\_aliveNode, label="LEACH")

# ax[0].scatter(rounds, K\_Means\_aliveNode, label="K-Means")

ax[0].plot(rounds, DECKS\_aliveNode, label="DECKS")

ax[0].plot(rounds, LEACH\_aliveNode, label="LEACH")

ax[0].plot(rounds, K\_Means\_aliveNode, label="K-Means")

ax[0].plot(rounds, Q\_learning\_aliveNode, label="Q-Learning")

# ax[1].scatter(rounds, DECKS\_EnergyNode, label="DECKS")

# ax[1].scatter(rounds, LEACH\_EnergyNode, label="LEACH")

# ax[1].scatter(rounds, K\_Means\_EnergyNode, label="K-Means")

ax[1].plot(rounds, DECKS\_EnergyNode, label="DECKS")

ax[1].plot(rounds, LEACH\_EnergyNode, label="LEACH")

ax[1].plot(rounds, K\_Means\_EnergyNode, label="K-Means")

ax[1].plot(rounds, Q\_learning\_EnergyNode, label="Q-Learning")

ax[0].set\_ylabel("Node Alive")

ax[0].set\_xlabel("Rounds")

ax[1].set\_ylabel("Energy Residual")

ax[1].set\_xlabel("Rounds")

ax[0].set\_xlim(0, 200)

ax[1].set\_ylim(ymin=0)

ax[0].legend()

ax[1].legend()