import numpy as np, json, random, solver, operator, pandas as pd from flask import \* import pandas as pd import numpy as np import matplotlib.pyplot as plt from sklearn.cluster import KMeans import csv app = Flask(\_\_name\_\_)

city\_name\_data = pd.read\_csv('namelist.csv',header=None) city\_dist\_data = pd.read\_csv('distlist.csv',header=None) city\_weight\_data = pd.read\_csv('poplist.csv',header=None) city\_coord\_data = pd.read\_csv('Latlong.csv',header=None) print(city\_name\_data) print(city\_dist\_data) print(city\_coord\_data) class City:

def \_\_init\_\_(self,name,population,coord):

self.name=name self.population=population self.coord=coord

def distance(self, city): distance=city\_dist\_data.iloc[self.name,city.name] return distance def CityName(self):

return str(city\_name\_data.iloc[self.name,0]) def CityCoord(self): return self.coord def \_\_repr\_\_(self): return "\""+str(city\_name\_data.iloc[self.name,0]) + " "+str(self.coord[0])+", "+str(self.coord[1])+"" +"\"" class Fitness: def \_\_init\_\_(self, route): self.chromosome = route self.distance = 0 self.fitness= 0.0 self.total\_population= 0

def routeDistance(self): pathDistance = 0 for i in range(0, len(self.chromosome)): for j in range(0,len(self.chromosome[0])):

fromCity = self.chromosome[i][j] toCity = None if j + 1 < len(self.chromosome[0]):

toCity = self.chromosome[i][j +1] #doubtfull else: break

if(type(toCity) == list): break pathDistance += fromCity.distance(toCity) self.distance = pathDistance return self.distance

def routePopulation(self): path\_population = 0 for i in range(0, len(self.chromosome)): for j in range(0,len(self.chromosome[0])): City = self.chromosome[i][j] if type(City) != list: path\_population += int(City.population.replace(',', '')) else:

path\_population = 0 self.total\_population = path\_population return self.total\_population

def routeFitness(self): if self.fitness == 0: if(self.routeDistance()==0): return self.routePopulation() self.fitness = self.routePopulation() / float(self.routeDistance()) return self.fitness def Diff(l1, l2): li\_dif = [i for i in l1 if i not in l2] return li\_dif

def GenerateTiming():

res = []

res.append( str(random.randint(6,9)) + ":"+str(random.randint(0,11)\*5) ) res.append( str(random.randint(9,12)) +":"+ str(random.randint(0,12)\*5) ) res.append( str(random.randint(12,15)) +

":"+str(random.randint(0,12)\*5) )

res.append( str(random.randint(15,18)) +

":"+str(random.randint(0,12)\*5) )

res.append( str(random.randint(18,21)) +

":"+str(random.randint(0,12)\*5) )

res.append( str(random.randint(21,23)) +

":"+str(random.randint(0,12)\*5) ) return res cityRoute = solver.solve() def createRoute(cityList): tempcityList = cityList.copy() chromosome = [] for i in range(5):

route = random.sample(tempcityList, 7) chromosome.append(route)

tempcityList = Diff(tempcityList,route) return chromosome

def initialPopulation(popSize, cityList): population = [] for i in range(0, popSize):

population.append(createRoute(cityList)) return population def rankRoutes(population): fitnessResults = {}

for i in range(0,len(population)): fitnessResults[i] = Fitness(population[i]).routeFitness() return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True) def selection(popRanked, eliteSize): selectionResults = []

df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"]) df['cum\_sum'] = df.Fitness.cumsum()

df['cum\_perc'] = 100\*df.cum\_sum/df.Fitness.sum()

for i in range(0, eliteSize):

selectionResults.append(popRanked[i][0]) for i in range(0, len(popRanked) - eliteSize): pick = 100\*random.random() for i in range(0, len(popRanked)): if pick <= df.iat[i,3]:

selectionResults.append(popRanked[i][0]) break return selectionResults def matingPool(population, selectionResults):

matingpool = [] for i in range(0, len(selectionResults)): index = selectionResults[i] matingpool.append(population[index]) return matingpool def breed(parent1, parent2): copyParent1 = parent1.copy()

copyParent1 = [ j for i in parent1 for j in i ] copyParent2 = parent2.copy()

copyParent2 = [ j for i in parent2 for j in i ] child = [] childP1 = [] childP2 = []

geneA = int(random.random() \* len(copyParent1)) geneB = int(random.random() \* len(copyParent1))

startGene = min(geneA, geneB) endGene = max(geneA, geneB) for i in range(startGene, endGene): childP1.append(copyParent1[i]) for item in copyParent2: if item not in childP1 and len(childP1)<35:

childP1.append(item) child = childP1 offspring = [] temp=[] for i in range(len(child)): temp.append(child[i]) if(len(temp)==7):

offspring.append(temp) temp=[] return offspring def breedPopulation(matingpool, eliteSize): children = []

length = len(matingpool) - eliteSize

By pool = random.sample(matingpool, len(matingpool)) for i in range(0,eliteSize):

children.append(matingpool[i])

for i in range(0, length): child = breed(pool[i], pool[len(matingpool)-i-1]) children.append(child) return children

def mutate(individual, mutationRate): # this can be improved for swapped in range(len(individual)): if(random.random() < mutationRate): swapWith = int(random.random() \* len(individual))

city1 = individual[swapped] city2 = individual[swapWith]

individual[swapped] = city2 individual[swapWith] = city1 return individual

def mutatePopulation(population, mutationRate):

mutatedPop = []

for ind in range(0, len(population)): mutatedInd = mutate(population[ind], mutationRate) mutatedPop.append(mutatedInd) return mutatedPop def nextGeneration(currentGen, eliteSize, mutationRate): popRanked = rankRoutes(currentGen)

selectionResults = selection(popRanked, eliteSize) matingpool = matingPool(currentGen, selectionResults) children = breedPopulation(matingpool, eliteSize) nextGeneration = mutatePopulation(children, mutationRate) return nextGeneration

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):

pop = initialPopulation(popSize, population) print("Initial distance: " +

str(Fitness(pop[rankRoutes(pop)[0][0]]).routeDistance())) print("Initial population: " + str(Fitness(pop[rankRoutes(pop)[0][0]]).routePopulation())) for i in range(0, generations):

pop = nextGeneration(pop, eliteSize, mutationRate) total\_distance=0 total\_population = 0 for i in range (0,1): bestRouteIndex = rankRoutes(pop)[i][0] bestRoute = pop[bestRouteIndex] fitness = Fitness(bestRoute) fitness.routeFitness() for route in bestRoute: print(route,"\n")

total\_distance += fitness.distance total\_population += fitness.total\_population print("Total distance= "+ str(total\_distance))

FirstbestRouteIndex = rankRoutes(pop)[0][0]

FirstbestRoute = pop[FirstbestRouteIndex]

return FirstbestRoute def toCity(city):

ind = 0 for i in range(len(city\_name\_data)): if city == city\_name\_data[0][i]:

ind = i break

return City( name=ind,population=city\_weight\_data.iloc[ind,0],coord =

LatLongDict[ city\_name\_data.iloc[ind,0] ]) LatLongDict = {} for i in range(len(city\_coord\_data)):

LatLongDict[city\_coord\_data.iloc[i,0]] = [ city\_coord\_data.iloc[i,1]

, city\_coord\_data.iloc[i,2]] cityList = [] cities=[] for i in range(0,len(city\_name\_data)): cityList.append(City(name=i,population=city\_weight\_data.iloc[i,0], coord = LatLongDict[ city\_name\_data.iloc[i,0] ] )) cities.append(city\_name\_data.iloc[i,0])

X=city\_coord\_data

X.columns =["Name","latitude","longitude","demand"] kmeans = KMeans(n\_clusters = 5, init ='k-means++') kmeans.fit(np.array(X.iloc[:,1:3]))

X['cluster\_label'] = kmeans.fit\_predict(np.array(X.iloc[:,1:3])) centers = kmeans.cluster\_centers\_

labels = kmeans.predict(np.array(X.iloc[:,1:3]))

X.plot.scatter(x = 'latitude', y = 'longitude', c=labels, s=50, cmap='viridis')

plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5)

# plt.show() # uncomment for graph

FirstbestRoute = geneticAlgorithm(population=cityList, popSize=60, eliteSize=20, mutationRate=0.15, generations=10) for i in cityRoute: for j in range(len(i)):

i[j] = toCity(str(i[j])) AllRoutes = FirstbestRoute + cityRoute

index = 0 MapRouteToCity={} for route in AllRoutes: if route[0].CityName() not in MapRouteToCity: MapRouteToCity[ route[0].CityName() ] = []

MapRouteToCity[ route[0].CityName() ].append(route) TotalRoute = [] for FromCity in MapRouteToCity.keys():

timing = GenerateTiming() itr = 0 for route in MapRouteToCity[ FromCity ]: TotalRoute.append( [ timing[itr] , route] ) itr+=1 if(itr==5) : itr =0 @app.route('/getAllCities') def CityList():

d = {} for i in range(len(cityList)):

d[i] = [cityList[i].CityName() , cityList[i].CityCoord()] return json.dumps(d) @app.route('/getBusRouteByID') def getBusRouteByID():

ID = request.args.get('ID', default = 0, type = int) if(ID<0 or ID>int(len(TotalRoute))):

return "Invalid ID" else:

res = {} for i in range(len(TotalRoute[ID][1])):

res[i] = [ TotalRoute[ID][1][i].CityName() , TotalRoute[ID][1][i].CityCoord() ] res[-1] = TotalRoute[ID][0] return json.dumps(res)

@app.route('/getBusesBySrcDest') def getBusesBySrcDest():

src = request.args.get('src', default = 0, type = str).lower() dest = request.args.get('dest', default = 0, type = str).lower() res = {} for routeInd in range(len(TotalRoute)): l = [i.CityName().lower() for i in TotalRoute[routeInd][1]] if src in l and dest in l: if(l.index(src) < l.index(dest)):

res[routeInd] = [TotalRoute[routeInd][0]]+[ [i.CityName()

, i.CityCoord()] for i in TotalRoute[routeInd][1] ] if res == {}:

return "Invalid" return json.dumps(res)

if \_\_name\_\_ == '\_\_main\_\_': app.run(host="0.0.0.0",port=4000).

**Output:**



Problem statement:

The current bus transportation system relies on experience-based manual decisions for covering stops and timings. And longer distances travelled which increases cost as well as carbon emission, and use of more resources than required. Timetables are often outdated and created based on static information of traffic resulting in suboptimal results and also waiting time of passengers increases due to unreliable routing of buses.

**Solution :**

Route Optimization:

Identifying the most effective route connections and traffic and population

Application (Passengers):

Real-time information and recommendation about buses Automatic personalized notifications about new stops and timings on modification of routes/timetables

First, the user must determine his location by activating the location feature in a smartphone. To get the information entered the application will provide the details about buses, bus location, bus speed, bus arrival time, nearest bus from a user by offering the distance between user location and bus. This information will assist the passenger to select their suitable bus. In the flowchart that describes the proposed system.

Second A GPS is connected to an ESP32 Microcontroller with a built-in Wi-Fi is placed inside each bus. When the power supply is on, the GPS is communicates continuously with the satellite to get coordinates. The GPS will initialize itself, then the module will get the co-ordinate

Once the GPS obtains the coordinates, it sends the data, including latitude and longitude, and speed to the IOT Blynk server through the ESP32. At the Blynk server, the latitude and longitude are extracted and used on the visual map in the Blynk application. The live location of the bus can be seen on the Google map. Continuous data digital updates such as speed, distance, and the arrival time of the bus are displayed on the mobile application.

Finally the prototype has been installed (GPS unit and ESP32) inside a vehicle with supplied internet to use the possibilities offered by the Internet of Things.

This information will be transmitted via a Wi-Fi internet connection to the Blynk server and then to the Android mobile application(latitude, longitude, speed, distance, and time of arrival).