Web development for public transport optimization

Abstract:

Internet of Things (IOT) is attracting Increasing attention from various industrial fields. The applications of IoT will be Prevalent in the public transportation system and bring changes to the system in the near future. A new framework for public transport System based on IoT, which integrates the scheduling problems of subway, bus, and shared taxi, is proposed For better-coordinated transfer solutions. Transport flow prediction methods based on periodic patterns. Mining is proposed for road flow analysis and passenger flow analysis. To improve the efficiency of scheduling, and reduce Passengers’ traveling time.

Introduction:

Internet of things (IoT) provides windows of opportunity for public transportation system. As is well-known, IoT is A new paradigm of information technology based on internet And wireless tele-communication. Components in IoT have unique identities, and they can interact and cooperate with Each other to reach common goals. IoT plays an important Role in many industrial fields such as manufacturing, logistics, Transportation, health care, and will bring a revolutionary Change to our daily life . Under the context of IoT, various Types of real time information can be obtained for public Transportation system, which is very useful for reducing the Uncertainty of the system and increasing the system ability of Quick response, and will help to precise control and manage of the public transportation.

Implements of the purposed system:

N The prototype of smart public transportation . It consists of two parts: The first part is the communication unit, which includes a GPS module and an ESP32 micro-controller with a Wi-Fi built-in module. This unit is used in the public transportation system for vehicle monitoring and tracking. With the help of the GPS module, it can determine the current position and calculate the speed of buses. The GPS data is transferred to the Blynk server with the help of the Wi-Fi module for storage and analysis. Then it is displayed on the mobile phone application. The second part is the mobile application. The android application gets data from the Blynk server and provides the required data to the user based on the information provided in the android application.

ESP32 microcontroller:

The ESP32 is a microcontroller with a Wi-Fi module, an open-source IoT platform that is characterized by low-cost and low-power system-on-a-chip (SOC). An ESP32 has a dual-core structure and internal modules such as Wi-Fi, Bluetooth, and many Peripheral Interfaces such as IR, SPI, CAN, Ethernet, and temperature sensors. The specifications of the ESP3

GPS module:

GPS (Global Position System) used for positioning and tracking buses based on satellite communication. GPS satellites cover the entire earth at all times. To get accurate GPS location data, there should be a minimum of three satellites. The NEO-6M GPS module used in the proposed system is small and works on very low power, making it ideal for tracking applications. The GPS module operates at 3.3 V, as a result, powered by connecting the GPS module to the 3.3 V pin of the ESP32

Program:

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 City Name(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

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] ]

LatLong Dict = {}

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 city Route:

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:

