Thomas Courtney

Artificial Intelligence

Professor Blossom

5 September 2021

Metaheuristic Assignment

Summary

The following report dives into two common algorithms within machine learning; simulated annealing and genetic algorithms. The report discusses these two algorithms and gives background in the introduction, and teaches the algorithm through pseudocode and flowcharts.

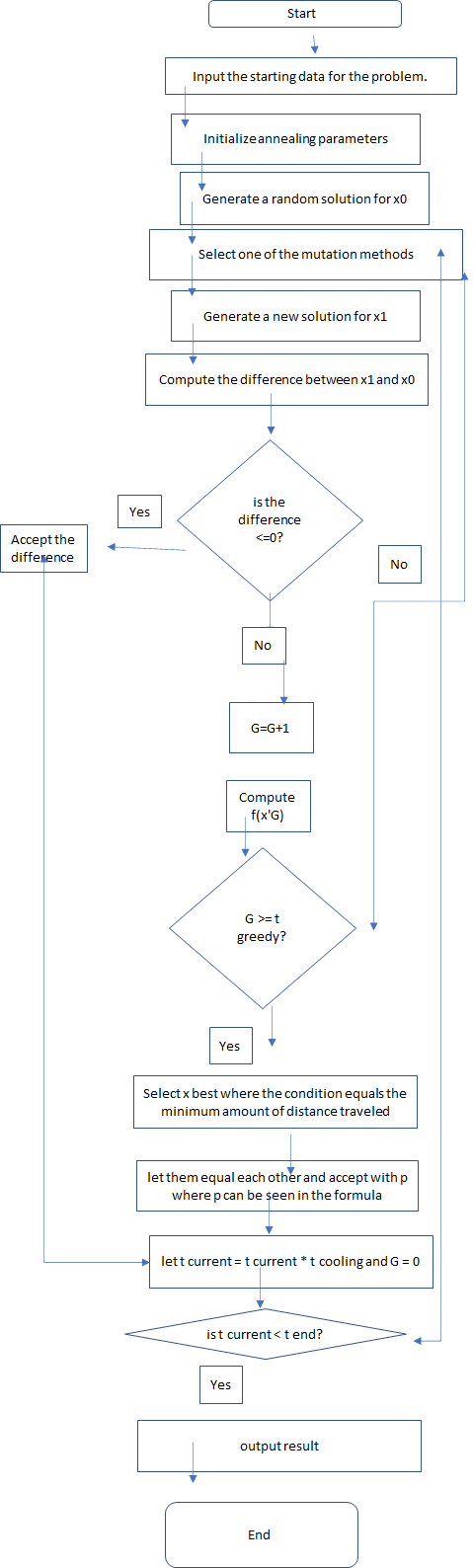
Introduction

Simulated annealing and genetic algorithms are metaheuristics that help approximate for genetic algorithm problems. These algorithms are more commonplace than one would think. When maps services such as Waze or Google Maps, they use these two metaheuristics to determine the fastest way for you to get to your desired location. These algorithms help us determine the most efficient and optimal solutions in everyday life.

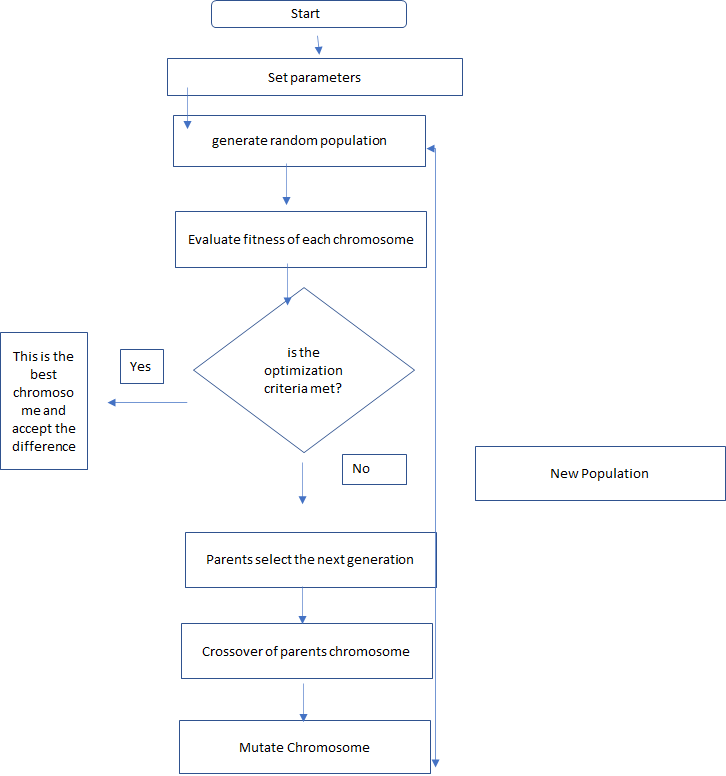
To help explain these algorithms, we will use pseudocode and flowcharts. We will explain this using the traveling salesman problem. The traveling salesman answers the question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? This problem best shows how we can use optimization and artificial intelligence to determine the best path. The flowcharts and pseudocode are a good example of how to break down these problems. After reviewing the two, the conclusion breaks down some of the positives and negatives of both solutions.

Body

Simulated Annealing Flowchart



Genetic Algorithm Flowchart



Simulated Annealing Pseudocode

#!/usr/bin/env python

#import necessary packages

from scipy IMPORT \*

from pylab IMPORT \*

#below the user will define our distince function, which will help calculate the distance of the two cities that are next to each other

FUNCTION Distance(R1, R2):

RETURN sqrt((R1[0]-R2[0])\*\*2+(R1[1]-R2[1])\*\*2)

ENDFUNCTION

#The DistanceTotal function helps us identify the total distance of the trip

FUNCTION DistanceTotal(city, R):

dist=0

FOR i IN range(len(city)-1):

dist += Distance(R[city[i]],R[city[i+1]])

ENDFOR

dist += Distance(R[city[-1]],R[city[0]])

RETURN dist

ENDFUNCTION

FUNCTION mixup(city, n):

nct <- len(city)

nn <- (1+ ((n[1]-n[0]) % nct))/2 # the length of half the segment will be reversed

# we mix up the segment this way n[0]<->n[1], n[0]+1<->n[1]-1, n[0]+2<->n[1]-2,...

# Swap the cities around to determine the best path

FOR j IN range(nn):

k <- (n[0]+j) % nct

l <- (n[1]-j) % nct

(city[k],city[l]) <- (city[l],city[k]) # swap

ENDFUNCTION

FUNCTION new(city, n):

nct <- len(city)

nextcity=[]

# for this range, segment the different pieces n[0]...n[1]

FOR j IN range( (n[1]-n[0])%nct + 1):

newcity.append(city[ (j+n[0])%nct ])

# is followed by segment n[5]...n[2]

ENDFOR

FOR j IN range( (n[2]-n[5])%nct + 1):

nextcity.append(city[ (j+n[5])%nct ])

# is followed by segment n[3]...n[4]

ENDFOR

FOR j IN range( (n[4]-n[3])%nct + 1):

nextcity.append(city[ (j+n[3])%nct ])

ENDFOR

RETURN nextcity

ENDFUNCTION

#plot the path

FUNCTION Plot(city, R, dist):

Pt <- [R[city[i]] FOR i IN range(len(city))]

ENDFOR

Pt += [R[city[0]]]

Pt <- array(Pt)

title('Total distance='+str(dist))

plot(Pt[:,0], Pt[:,1], '-o')

show()

ENDFUNCTION

IF \_\_name\_\_=='\_\_main\_\_':

ncity <- 100 # How many cities the salesman has to visit

maxTsteps <- 100 # Temperature is lowered not more than maxTsteps

Tstart <- 0.2 # Starting temperature

fCool <- 0.9 # At each cooling step, multiple the temperature by this nu,ber

maxSteps <- 100\*ncity # at a constant temperature this is the number of steps

maxAccepted <- 10\*ncity # at a constant temperature, the number of accepted steps

Preverse <- 0.5 # How often to choose reverse/transpose trial move

# Let's choose the coordinates, this will be done randomly

R=[]

FOR i IN range(ncity):

R.append( [rand(),rand()] ) #generates our random coordinates

ENDFOR

R <- array(R)

# Order how our cities are going to be visied

city <- range(ncity)

# Starting amount of travel

dist <- DistanceTotal(city, R)

# Stores points of a move

n <- zeros(6, dtype=int)

nct <- len(R) # number of cities

T <- Tstart # temperature

# plot this on a graph

Plot(city, R, dist)

FOR t IN range(maxTsteps): # Over temperature

accepted <- 0

FOR i IN range(maxSteps): # How many monte Carlo steps at each temperature

WHILE True: # Finds two cities that are close by

# Two cities n[0] AND n[1] are choosen at random

n[0] <- int((nct)\*rand()) # select city 1

n[1] <- int((nct-1)\*rand()) # select city 2

IF (n[1] >= n[0]): n[1] += 1

ENDIF

IF (n[1] < n[0]): (n[0],n[1]) <- (n[1],n[0]) # swap the cities

ENDIF

nn <- (n[0]+nct -n[1]-1) % nct # number of cities not on the segment n[0]..n[1]

IF nn>=3: break

ENDIF

# Create an index before and after the cities

ENDFOR

# So our order is [n2,n0,n1,n3]

ENDWHILE

n[2] <- (n[0]-1) % nct # index before n0

ENDFOR

n[3] <- (n[1]+1) % nct # index after n2

IF Preverse > rand():

# reverse the segment

# See what the difference is if we reversed the path city[n[0]]-city[n[1]]?

de <- Distance(R[city[n[2]]],R[city[n[1]]]) + Distance(R[city[n[3]]],R[city[n[0]]]) - Distance(R[city[n[2]]],R[city[n[0]]]) - Distance(R[city[n[3]]],R[city[n[1]]])

IF de<0 OR exp(-de/T)>rand(): # Metropolis

accepted += 1

dist += de

mixup(city, n)

ENDIF

ELSE:

# Transpose the segment

nc <- (n[1]+1+ int(rand()\*(nn-1)))%nct # Another point outside n[0],n[1] segment. See picture IN lecture nodes!

n[4] <- nc

n[5] <- (nc+1) % nct

# Transposing the segment

de <- -Distance(R[city[n[1]]],R[city[n[3]]]) - Distance(R[city[n[0]]],R[city[n[2]]]) - Distance(R[city[n[4]]],R[city[n[5]]])

de += Distance(R[city[n[0]]],R[city[n[4]]]) + Distance(R[city[n[1]]],R[city[n[5]]]) + Distance(R[city[n[2]]],R[city[n[3]]])

IF de<0 OR exp(-de/T)>rand():

accepted += 1

dist += de

city <- new(city, n)

ENDIF

ENDIF

IF accepted > maxAccepted: break

ENDIF

# Plot the salesman path

ENDFOR

Plot(city, R, dist)

OUTPUT "T=%10.5f , distance= %10.5f , accepted steps= %d" %(T, dist, accepted)

T \*= fCool # The system is cooled down

SET IF accepted TO 0: break # If the path does not want to change any more, we can stop

ENDIF

ENDFOR

Plot(city, R, dist)

Genetic Algorithm Pseudocode

#import necessary packages

IMPORT math

IMPORT random

#will create our x and y coordinates of the cities

DEFINE CLASS City:

#create your random cities

DEFINE FUNCTION \_\_init\_\_(self, x=None, y=None):

SET self.x TO None

SET self.y TO None

IF x is not None:

SET self.x TO x

ELSE:

SET self.x TO int(random.random() \* 200)

IF y is not None:

SET self.y TO y

ELSE:

SET self.y TO int(random.random() \* 200)

DEFINE FUNCTION getX(self):

RETURN self.x

DEFINE FUNCTION getY(self):

RETURN self.y

#determines the distance from each city

DEFINE FUNCTION distanceTo(self, city):

SET xDistance TO abs(self.getX() - city.getX())

SET yDistance TO abs(self.getY() - city.getY())

SET distance TO math.sqrt( (xDistance\*xDistance) + (yDistance\*yDistance) )

RETURN distance

DEFINE FUNCTION \_\_repr\_\_(self):

RETURN str(self.getX()) + ", " + str(self.getY())

#this wil give us our list of cities

DEFINE CLASS TourManager:

SET destinationCities TO []

#function to add the cities to our list

DEFINE FUNCTION addCity(self, city):

self.destinationCities.append(city)

#function to grab a city from the list

DEFINE FUNCTION getCity(self, index):

RETURN self.destinationCities[index]

#function to tell us how many cities there are

DEFINE FUNCTION numberOfCities(self):

RETURN len(self.destinationCities)

#this will create our fitness class, which is the inverse of the route distance

#this should be minimized, so we want to minimize route distance, and a larger fitness score is better

DEFINE CLASS Tour:

DEFINE FUNCTION \_\_init\_\_(self, tourmanager, tour=None):

SET self.tourmanager TO tourmanager

SET self.tour TO [] #a list to put the numbers in

SET self.fitness TO 0.0 # make your fitness equal zero

SET self.distance TO 0 # make your starting point equal to zero

IF tour is not None:

SET self.tour TO tour

#append the list

ELSE:

FOR i IN range(0, self.tourmanager.numberOfCities()):

self.tour.append(None)

DEFINE FUNCTION \_\_len\_\_(self):

RETURN len(self.tour)

DEFINE FUNCTION \_\_getitem\_\_(self, index):

RETURN self.tour[index]

DEFINE FUNCTION \_\_setitem\_\_(self, key, value):

SET self.tour[key] TO value

DEFINE FUNCTION \_\_repr\_\_(self):

SET geneString TO "|"

FOR i IN range(0, self.tourSize()):

geneString += str(self.getCity(i)) + "|"

RETURN geneString

DEFINE FUNCTION generateIndividual(self):

FOR cityIndex IN range(0, self.tourmanager.numberOfCities()):

self.setCity(cityIndex, self.tourmanager.getCity(cityIndex))

random.shuffle(self.tour)

DEFINE FUNCTION getCity(self, tourPosition):

RETURN self.tour[tourPosition]

DEFINE FUNCTION setCity(self, tourPosition, city):

SET self.tour[tourPosition] TO city

SET self.fitness TO 0.0

SET self.distance TO 0

DEFINE FUNCTION getFitness(self):

IF self.fitness EQUALS 0:

SET self.fitness TO 1/float(self.getDistance())

RETURN self.fitness

DEFINE FUNCTION getDistance(self):

IF self.distance EQUALS 0:

SET tourDistance TO 0

FOR cityIndex IN range(0, self.tourSize()):

SET fromCity TO self.getCity(cityIndex)

SET destinationCity TO None

IF cityIndex+1 < self.tourSize():

SET destinationCity TO self.getCity(cityIndex+1)

ELSE:

SET destinationCity TO self.getCity(0)

tourDistance += fromCity.distanceTo(destinationCity)

SET self.distance TO tourDistance

RETURN self.distance

DEFINE FUNCTION tourSize(self):

RETURN len(self.tour)

DEFINE FUNCTION containsCity(self, city):

RETURN city IN self.tour

#we will make our initial population

#this function produces routes that satisty the conditions previously set

DEFINE CLASS Population:

DEFINE FUNCTION \_\_init\_\_(self, tourmanager, populationSize, initialise):

SET self.tours TO []

FOR i IN range(0, populationSize):

self.tours.append(None)

IF initialise:

FOR i IN range(0, populationSize):

SET newTour TO Tour(tourmanager)

newTour.generateIndividual()

self.saveTour(i, newTour)

DEFINE FUNCTION \_\_setitem\_\_(self, key, value):

SET self.tours[key] TO value

DEFINE FUNCTION \_\_getitem\_\_(self, index):

RETURN self.tours[index]

DEFINE FUNCTION saveTour(self, index, tour):

SET self.tours[index] TO tour

DEFINE FUNCTION getTour(self, index):

RETURN self.tours[index]

DEFINE FUNCTION getFittest(self):

SET fittest TO self.tours[0]

FOR i IN range(0, self.populationSize()):

IF fittest.getFitness() <= self.getTour(i).getFitness():

SET fittest TO self.getTour(i)

RETURN fittest

DEFINE FUNCTION populationSize(self):

RETURN len(self.tours)

DEFINE CLASS GA:

DEFINE FUNCTION \_\_init\_\_(self, tourmanager):

SET self.tourmanager TO tourmanager

SET self.mutationRate TO 0.015

SET self.tournamentSize TO 5

SET self.elitism TO True

DEFINE FUNCTION evolvePopulation(self, pop):

SET newPopulation TO Population(self.tourmanager, pop.populationSize(), False)

SET elitismOffset TO 0

IF self.elitism:

newPopulation.saveTour(0, pop.getFittest())

SET elitismOffset TO 1

FOR i IN range(elitismOffset, newPopulation.populationSize()):

SET parent1 TO self.tournamentSelection(pop)

SET parent2 TO self.tournamentSelection(pop)

SET child TO self.crossover(parent1, parent2)

newPopulation.saveTour(i, child)

FOR i IN range(elitismOffset, newPopulation.populationSize()):

self.mutate(newPopulation.getTour(i))

RETURN newPopulation

DEFINE FUNCTION crossover(self, parent1, parent2):

SET child TO Tour(self.tourmanager)

SET startPos TO int(random.random() \* parent1.tourSize())

SET endPos TO int(random.random() \* parent1.tourSize())

FOR i IN range(0, child.tourSize()):

IF startPos < endPos and i > startPos and i < endPos:

child.setCity(i, parent1.getCity(i))

ELSEIF startPos > endPos:

IF not (i < startPos and i > endPos):

child.setCity(i, parent1.getCity(i))

FOR i IN range(0, parent2.tourSize()):

IF not child.containsCity(parent2.getCity(i)):

FOR ii IN range(0, child.tourSize()):

IF child.getCity(ii) EQUALS None:

child.setCity(ii, parent2.getCity(i))

break

RETURN child

DEFINE FUNCTION mutate(self, tour):

FOR tourPos1 IN range(0, tour.tourSize()):

IF random.random() < self.mutationRate:

SET tourPos2 TO int(tour.tourSize() \* random.random())

SET city1 TO tour.getCity(tourPos1)

SET city2 TO tour.getCity(tourPos2)

tour.setCity(tourPos2, city1)

tour.setCity(tourPos1, city2)

DEFINE FUNCTION tournamentSelection(self, pop):

SET tournament TO Population(self.tourmanager, self.tournamentSize, False)

FOR i IN range(0, self.tournamentSize):

SET randomId TO int(random.random() \* pop.populationSize())

tournament.saveTour(i, pop.getTour(randomId))

SET fittest TO tournament.getFittest()

RETURN fittest

IF \_\_name\_\_ EQUALS '\_\_main\_\_':

SET tourmanager TO TourManager()

# Create and add the routes

SET city TO City(60, 200)

tourmanager.addCity(city)

SET city2 TO City(180, 200)

tourmanager.addCity(city2)

SET city3 TO City(80, 180)

tourmanager.addCity(city3)

SET city4 TO City(140, 180)

tourmanager.addCity(city4)

SET city5 TO City(20, 160)

tourmanager.addCity(city5)

SET city6 TO City(100, 160)

tourmanager.addCity(city6)

SET city7 TO City(200, 160)

tourmanager.addCity(city7)

SET city8 TO City(140, 140)

tourmanager.addCity(city8)

SET city9 TO City(40, 120)

tourmanager.addCity(city9)

SET city10 TO City(100, 120)

tourmanager.addCity(city10)

SET city11 TO City(180, 100)

tourmanager.addCity(city11)

SET city12 TO City(60, 80)

tourmanager.addCity(city12)

SET city13 TO City(120, 80)

tourmanager.addCity(city13)

SET city14 TO City(180, 60)

tourmanager.addCity(city14)

SET city15 TO City(20, 40)

tourmanager.addCity(city15)

SET city16 TO City(100, 40)

tourmanager.addCity(city16)

SET city17 TO City(200, 40)

tourmanager.addCity(city17)

SET city18 TO City(20, 20)

tourmanager.addCity(city18)

SET city19 TO City(60, 20)

tourmanager.addCity(city19)

SET city20 TO City(160, 20)

tourmanager.addCity(city20)

# Initialize your population

SET pop TO Population(tourmanager, 50, True);

OUTPUT "Initial distance: " + str(pop.getFittest().getDistance())

# Evolve population FOR 50 generations

SET ga TO GA(tourmanager)

SET pop TO ga.evolvePopulation(pop)

FOR i IN range(0, 100):

SET pop TO ga.evolvePopulation(pop)

# Printresults

OUTPUT "Finished"

OUTPUT "Final distance: " + str(pop.getFittest().getDistance())

OUTPUT "Solution:"

OUTPUT pop.getFittest()

Conclusion

Overall, both simulated annealing and genetic algorithms provide different ways to solve the traveling salesman problem. Simulated annealing’s advantages are that it is guaranteed to find the local minimum or maximum, depending on what the user is shooting for. In our case, this is the minimum. The disadvantage is ultimately it’s speed in processing, which makes sense since you are guaranteed to find the local minimum or maximum. Genetic algorithms' advantages are the opposite. Genetic algorithms are easy to understand, support multi-objective optimization, great for noisy environments and the answer will get better with time. As there are many advantages, there are multiple disadvantages. Disadvantages include having to put in your values at the start (representation, population size) the overall performance also lacks. For example, you will find the right answer, and then have to start the program again. I personally find that simulated annealing is the ideal way to solve the problem, as you can guarantee to get the best results.

Recommendations

Throughout the research process it is important to understand that these problems can be applied to both maximum (highest point) and minimums (lowest point). The traveling salesman problem deals specifically with trying to find the shortest route, but you can use simulated annealing and genetic algorithms to find maximums, specifically in the field of robotics

Appendix

*The flow charts were created in excel, the pseudocode was originally written in a .txt document and copied and pasted into this document.*