



**Universidad Autónoma de Querétaro**  
**Facultad de Contabilidad y Administración**



***Optimizando el Camino***  
***con Algoritmos Genéticos***

**Proyecto final**

***Realizo:***

***Arce Jacintos Isaías***

***Exp: 307925***

***Materia:***

***Temas selectos de estadística***

# INDICE

Planteamiento.....3

Código.....3

Resultados.....12

## Planteamiento:

En la pizzería *Luigi Bella Notte*, la alta demanda de pedidos a domicilio durante la temporada ha generado un incremento significativo en las entregas. El repartidor en turno debe completar las entregas en la menor distancia posible para tener mayores ganancias. Sin embargo, debido a que el maletero de la moto tiene capacidad para un máximo de 20 pedidos, el repartidor necesita realizar entregas de 20 pedidos por viaje y regresar a la pizzería para recoger más.

Por motivos prácticos, el modelo de las ubicaciones de los domicilios se asigna de forma aleatoria. La pregunta clave es: ¿Cuál es la mejor ruta que debe tomar para completar el reparto y recorrer la menor distancia posible?

## Código en python:

```
import numpy as np

import random

import operator

import pandas as pd

import matplotlib.pyplot as plt
```

```
# Definición de la clase Casa para representar las ciudades con coordenadas x, y
```

```
class Casa:
```

```
    def __init__(self, name, x, y):
```

```
        self.name = name
```

```
        self.x = x
```

```
        self.y = y
```

```
# Método para calcular la distancia entre dos casas
```

```
def distance(self, casa):
```

```
    xDis = abs(self.x - casa.x)
```

```
    yDis = abs(self.y - casa.y)
```

```
    distance = np.sqrt((xDis ** 2) + (yDis ** 2))
```

```
    return distance
```

```
def __repr__(self):
```

```
    return "(" + str(self.name) + ")"
```

```
# Clase para evaluar el fitness de una ruta
```

```
class Fitness:
```

```
    def __init__(self, route):
```

```
        self.route = route
```

```
        self.distance = 0
```

```
        self.fitness = 0.0
```

```
# Método para calcular la distancia total de la ruta
```

```
def routeDistance(self):
```

```
    if self.distance == 0:
```

```
        pathDistance = 0
```

```
        for i in range(0, len(self.route)):
```

```
            fromCasa = self.route[i]
```

```
            toCasa = self.route[i + 1] if i + 1 < len(self.route) else self.route[0]
```

```

        pathDistance += fromCasa.distance(toCasa)

    self.distance = pathDistance

    return self.distance

# Método para calcular el fitness de la ruta
def routeFitness(self):
    if self.fitness == 0:
        self.fitness = 1 / float(self.routeDistance())

    return self.fitness

# Generador de una ruta aleatoria
def createRoute(casaList):
    route = random.sample(casaList, len(casaList))

    return route

# Crear la población inicial con rutas aleatorias
def initialPopulation(popSize, casaList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(casaList))

    return population

# Ordenar individuos en función de su fitness
def rankRoutes(population):

```

```

fitnessResults = {}

for i in range(0, len(population)):

    fitnessResults[i] = Fitness(population[i]).routeFitness()

sorted_results = sorted(fitnessResults.items(), key=operator.itemgetter(1), reverse=True)

return sorted_results


# Selección de rutas para la reproducción

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


# Crear un "pool" de apareamiento basado en los resultados de la selección

```

```
def matingPool(population, selectionResults):
```

```
    matingpool = []
```

```
    for i in range(0, len(selectionResults)):
```

```
        index = selectionResults[i]
```

```
        matingpool.append(population[index])
```

```
    return matingpool
```

```
# Cruzar dos rutas para crear un hijo
```

```
def breed(parent1, parent2):
```

```
    child = []
```

```
    childP1 = []
```

```
    childP2 = []
```

```
    geneA = int(random.random() * len(parent1))
```

```
    geneB = int(random.random() * len(parent1))
```

```
    startGene = min(geneA, geneB)
```

```
    endGene = max(geneA, geneB)
```

```
    for i in range(startGene, endGene):
```

```
        childP1.append(parent1[i])
```

```
    childP2 = [item for item in parent2 if item not in childP1]
```

```
    child = childP1 + childP2
```

```
return child
```

```
# Cruzar todo el "pool" de apareamiento para crear la próxima generación
```

```
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
```

```
# Mutar una ruta
```

```
def mutate(individual, mutationRate):
```

```
    for swapped in range(len(individual)):
```

```
        if random.random() < mutationRate:
```

```
            swapWith = int(random.random() * len(individual))
```

```
            casa1 = individual[swapped]
```



```

    casa2 = individual[swapWith]

    individual[swapped] = casa2

    individual[swapWith] = casa1

    return individual

# Mutar toda la población
def mutatePopulation(population, mutationRate):

    mutatedPop = []

    for ind in range(0, len(population)):

        mutatedInd = mutate(population[ind], mutationRate)

        mutatedPop.append(mutatedInd)

    return mutatedPop

# Crear la siguiente generación
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

```

```

# Algoritmo genético completo

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

    pop = initialPopulation(popSize, population)

    progress = [1 / rankRoutes(pop)[0][1]]

    print("Distancia inicial: " + str(progress[0]))

    for i in range(1, generations + 1):

        pop = nextGeneration(pop, eliteSize, mutationRate)

        progress.append(1 / rankRoutes(pop)[0][1])

        if i % 50 == 0:

            print('Generación ' + str(i) + " Distancia: ", progress[i])

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

    bestRoute = pop[bestRouteIndex]

    plt.plot(progress)

    plt.ylabel('Distancia')

    plt.xlabel('Generación')

    plt.title('Mejor Fitness vs Generación')

    plt.tight_layout()

    plt.show()

    return bestRoute

```

```

# Generar lista de casas

casaList = []

for i in range(0, 20):

    casaList.append(Casa(name=i, x=int(random.random() * 200), y=int(random.random() *
200)))

# Ejecutar el algoritmo genético

best_route = geneticAlgorithm(population=casaList, popSize=100, eliteSize=20,
mutationRate=0.01, generations=300)

# Imprimir el orden de recorrido de las casas

print("Orden de recorrido de las casas:")

for casa in best_route:

    print(f"Casa {casa.name} -> ", end="")

print(f"Casa {best_route[0].name} (regreso al inicio)")

# Mostrar la ruta final

x = []

y = []

for i in best_route:

    x.append(i.x)

    y.append(i.y)

x.append(best_route[0].x)

y.append(best_route[0].y)

```

```

plt.plot(x, y, '--o')

plt.xlabel('X')

plt.ylabel('Y')

ax = plt.gca()

plt.title('Diseño de la Ruta Final')

bbox_props = dict(boxstyle="circle,pad=0.3", fc='C0', ec="black", lw=0.5)

for i in range(1, len(casaList) + 1):

    ax.text(casaList[i - 1].x, casaList[i - 1].y, str(i), ha="center", va="center",

            size=8, bbox=bbox_props)

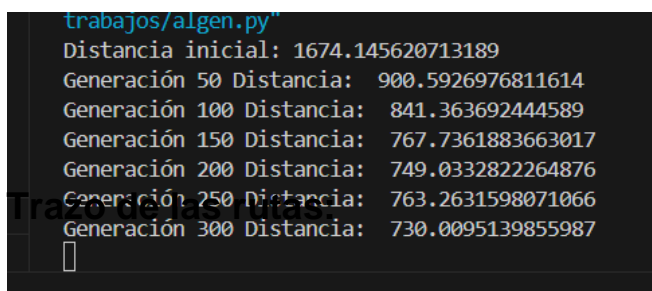
plt.tight_layout()

plt.show()

```

## Resultados:

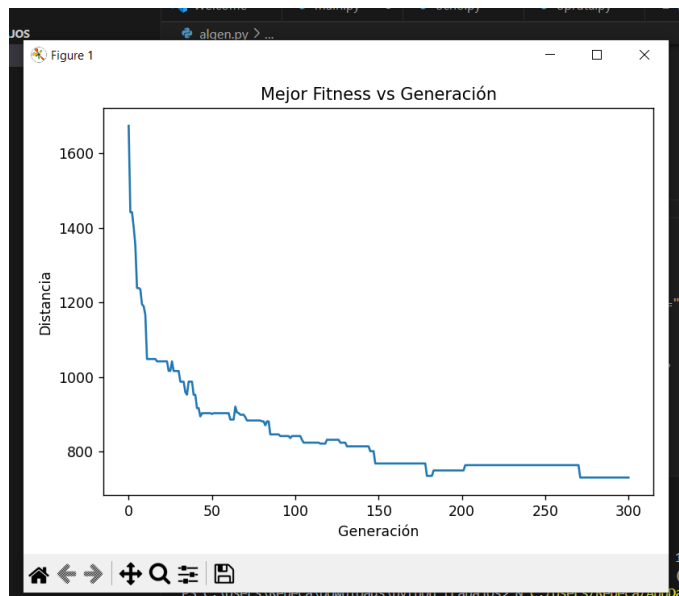
Evolución de optimización por generación:



```

trabajos/algen.py
Distancia inicial: 1674.145620713189
Generación 50 Distancia: 900.5926976811614
Generación 100 Distancia: 841.363692444589
Generación 150 Distancia: 767.7361883663017
Generación 200 Distancia: 749.0332822264876
Generación 250 Distancia: 763.2631598071066
Generación 300 Distancia: 730.0095139855987

```



Ruta:

```

Generación 300 Distancia: 730.0095139855987
Orden de recorrido de las casas:
Casa 16 -> Casa 12 -> Casa 2 -> Casa 9 -> Casa 14 -> Casa 0 -> Casa 17 -> Casa 7 -> Casa
3 -> Casa 15 -> Casa 10 -> Casa 18 -> Casa 8 -> Casa 5 -> Casa 6 -> Casa 11 -> Casa 1 ->
Casa 19 -> Casa 4 -> Casa 13 -> Casa 16 (regreso al inicio)
  
```

