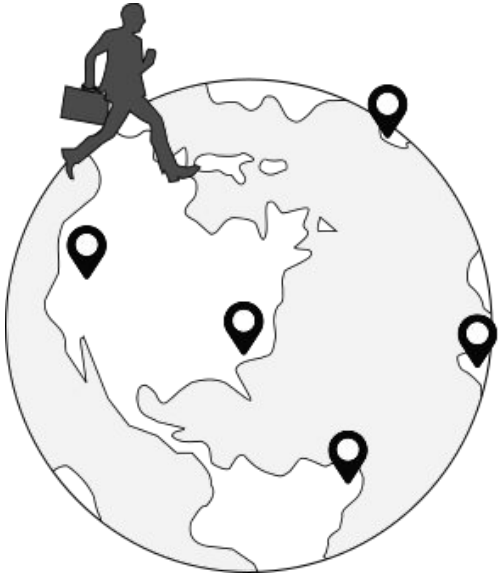
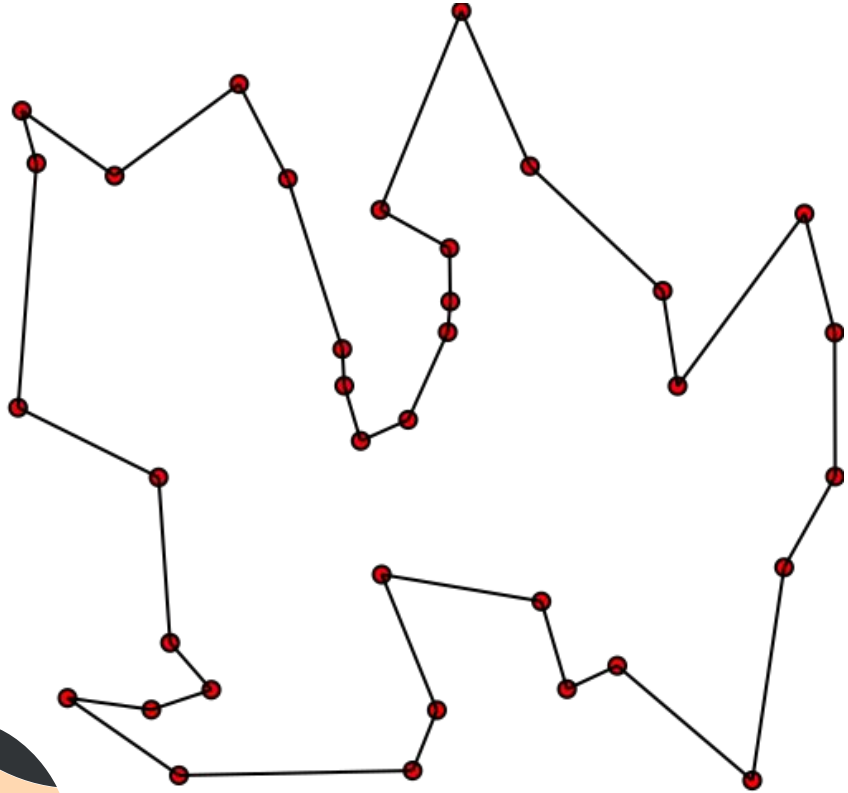
Implementando um AG para solucionar o problema do caixeiro viajante

Arquimedes Vinicius Pereira de França Moura
Audrey Emmely Rodrigues Vasconcelos



O problema



“Dada uma lista de cidades e as distâncias entre cada par de cidades, qual é o trajeto mais curto possível que visita cada cidade e retorna à cidade de origem?”

A abordagem

1. Gene: uma cidade (representada como coordenadas (x, y))
2. Indivíduo (também conhecido como "cromossomo"): uma única rota que satisfaça às condições citadas anteriormente
3. População: uma coleção de rotas possíveis (ou seja, coleção de indivíduos)
4. Pais: duas rotas que são combinadas para criar uma nova rota
5. Pool de acasalamento (mating): uma coleção de pais que são usados para criar nossa próxima população (criando assim a próxima geração de rotas)
6. Fitness: uma função que nos diz o quão bom é cada trajeto (no nosso caso, quão curta é a distância)
7. Mutação: uma forma de introduzir variação em nossa população trocando aleatoriamente duas cidades em uma rota
8. Elitismo: uma forma de transportar os melhores indivíduos para a próxima geração

Nosso AG consiste nas seguintes etapas:

01 Crie a população

02 Determine o fitness

03 Selecione o pool de acasalamento

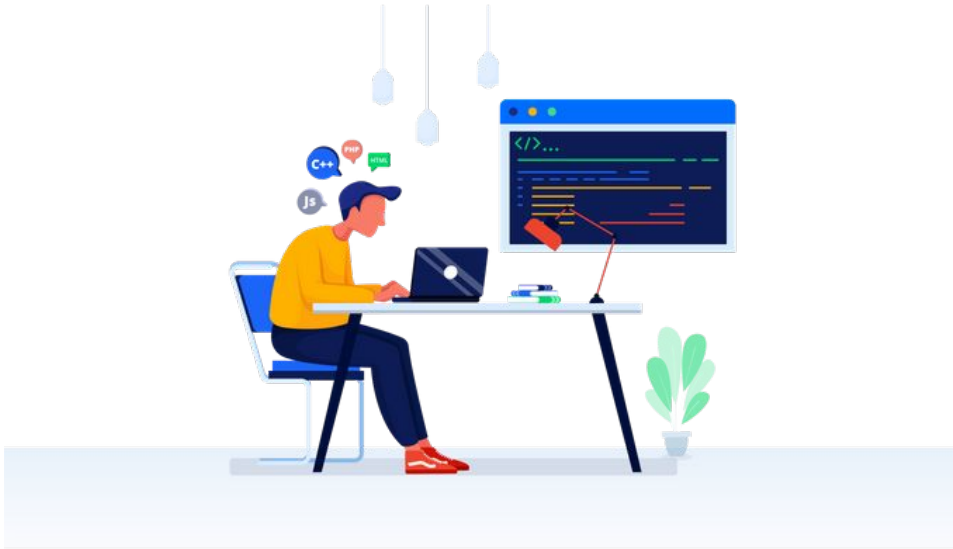
04 Faça o crossover (cruzamento)

05 Faça a mutação

06 Repita o processo

Construindo o algoritmo genético

https://tinyurl.com/ag-tsp-github



01

Criação das classes Cidade e Fitness

```
import numpy as np
import pandas as pd
import random
import operator
import matplotlib.pyplot as plt
```

```
class Cidade:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distancia(self, cidade):
        xDis = abs(self.x - cidade.x)
        yDis = abs(self.y - cidade.y)
        distancia = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distancia

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

```python
class Fitness:
    def __init__(self, rota):
        self.rota = rota
        self.distancia = 0
        self.fitness = 0.0

    def rotaDistancia(self):
        if self.distancia ==0:
            caminhoDistancia = 0
            for i in range(0, len(self.rota)):
                cidadeDePartida = self.rota[i]
                cidadeDeChegada = None
                if i + 1 < len(self.rota):
                    cidadeDeChegada = self.rota[i + 1]
                else:
                    cidadeDeChegada = self.rota[0]
                caminhoDistancia += cidadeDePartida.distancia(cidadeDeChegada)
            self.distancia = caminhoDistancia
        return self.distancia

    def rotaFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.rotaDistancia())
        return self.fitness
```

02

Criando a população e determinando o fitness

```python
def criarRota(listaDeCidades):
  rota = random.sample(listaDeCidades, len(listaDeCidades))
  return rota
```

```python
def populacaoInicial(tamPop, listaDeCidades):
    populacao = []

    for i in range(0, tamPop):
        populacao.append(criarRota(listaDeCidades))
    return populacao
```

```python
def rankRota(populacao):
    fitnessResultado = {}
    for i in range(0,len(populacao)):
        fitnessResultado[i] = Fitness(populacao[i]).rotaFitness()
    return sorted(fitnessResultado.items(), key = operator.itemgetter(1), reverse = True)
```

03

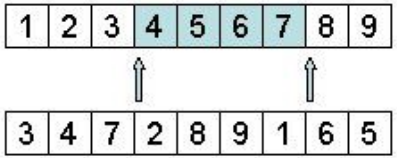
Selecionando o pool de acasalamento

```python
def selecao(popRankeada, tamElite):
    selecaoResultado = []
    df = pd.DataFrame(np.array(popRankeada), columns=["Index","Fitness"])
    df['cum_soma'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_soma/df.Fitness.sum()

    for i in range(0, tamElite):
        selecaoResultado.append(popRankeada[i][0])
    for i in range(0, len(popRankeada) - tamElite):
        escolha = 100*random.random()
        for i in range(0, len(popRankeada)):
            if escolha <= df.iat[i,3]:
                selecaoResultado.append(popRankeada[i][0])
                break
    return selecaoResultado
```

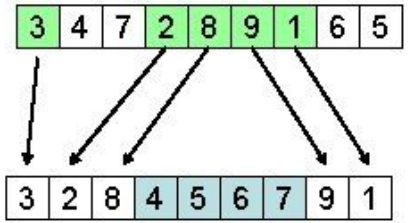
```python
def poolAcasalamento(populacao, selecaoResultado):
    poolacasalamento = []
    for i in range(0, len(selecaoResultado)):
        index = selecaoResultado[i]
        poolacasalamento.append(populacao[index])
    return poolacasalamento
```

04

Crossover



The remaining alleles are 1 2 3 8 9.
Their order in the other parent is 3 2 8 9 1



```python
def crossover(pai1, pai2):
    filho = []
    filhoP1 = []
    gilhoP2 = []

    geneA = int(random.random() * len(pai1))
    geneB = int(random.random() * len(pai1))

    inicioGene = min(geneA, geneB)
    fimGene = max(geneA, geneB)

    for i in range(inicioGene, fimGene):
        filhoP1.append(pai1[i])

    filhoP2 = [item for item in pai2 if item not in filhoP1]

    filho = filhoP1 + filhoP2
    return filho
```

```python
def crossoverPopulacao(poolAcasalamento, tamElite):
    filhos = []
    comp = len(poolAcasalamento) - tamElite
    pool = random.sample(poolAcasalamento, len(poolAcasalamento))

    for i in range(0, tamElite):
        filhos.append(poolAcasalamento[i])

    for i in range(0, comp):
        filho = crossover(pool[i], pool[len(poolAcasalamento)-i-1])
        filhos.append(filho)
    return filhos
```

05

Mutação

```python
def mutacao(individuo, taxaMutacao):
    for trocado in range(len(individuo)):
        if(random.random() < taxaMutacao):
            trocaCom = int(random.random() * len(individuo))

            cidade1 = individuo[trocado]
            cidade2 = individuo[trocaCom]

            individuo[trocado] = cidade2
            individuo[trocaCom] = cidade1
    return individuo
```

```python
def mutacaoPopulacao(populacao, taxaMutacao):
    popPosMutacao = []

    for ind in range(0, len(populacao)):
        indPosMutacao = mutacao(populacao[ind], taxaMutacao)
        popPosMutacao.append(indPosMutacao)
    return popPosMutacao
```

06

Repetição e função AG

```python
def proxGeracao(atualGeracao, tamElite, taxaMutacao):
    popRankeada = rankRota(atualGeracao)
    selecaoResultado = selecao(popRankeada, tamElite)
    poolacasalamento = poolAcasalamento(atualGeracao, selecaoResultado)
    filhos = crossoverPopulacao(poolacasalamento, tamElite)
    proxGeracao = mutacaoPopulacao(filhos, taxaMutacao)
    return proxGeracao
```

```python
def algoritmoGenetico(populacao, tamPop, tamElite, taxaMutacao, geracoes):
    pop = populacaoInicial(tamPop, populacao)
    print("Distância inicial: " + str(1 / rankRota(pop)[0][1]))

    for i in range(0, geracoes):
        pop = proxGeracao(pop, tamElite, taxaMutacao)

    print("Distância final: " + str(1 / rankRota(pop)[0][1]))
    melhorRotaIndex = rankRota(pop)[0][0]
    melhorRota = pop[melhorRotaIndex]
    return melhorRota
```

Rodando o algoritmo genético

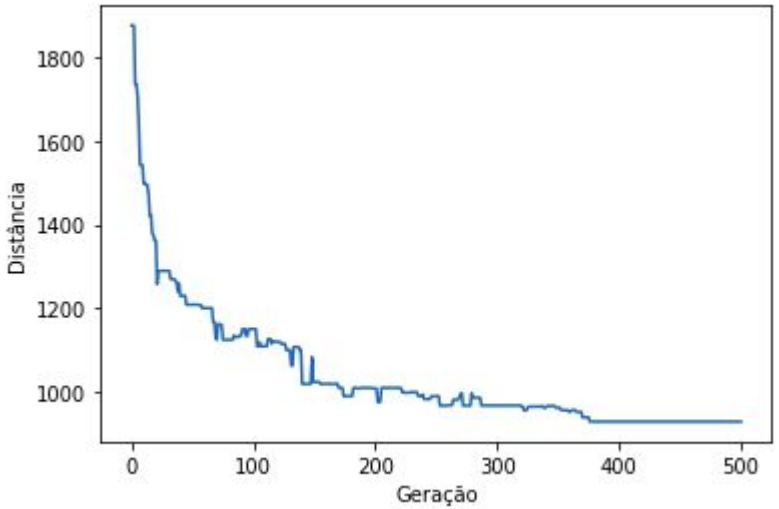
```
listaDeCidades = []

for i in range(0,25):
    listaDeCidades.append(Cidade(x=int(random.random() * 200), y=int(random.random() * 200)))
```

```
algoritmoGenetico(populacao=listaDeCidades, tamPop=100, tamElite=20, taxaMutacao=0.01, geracoes=500)
```

```
Distância inicial: 1939.4274682514626
Distância final: 966.5307749602428
[(154,66),
 (137,55),
 (64,41),
 (61,65),
 (0,66),
 (27,127),
 (70,181),
 (106,194),
 (160,136),
 (173,149),
 (187,128),
 (197,144),
 (193,199),
 (172,192),
 (162,177),
 (148,194),
 (146,172),
 (133,151),
 (105,115),
 (119,79),
 (149,45),
 (156,41),
 (185,15),
 (182,75),
 (181,101)]
```

Obs.: A cada execução é retornado um resultado diferente.

Melhorando a visualização dos dados:
Plotando a melhoria da distância de cada geração



Ao ver que a distância melhorou com o tempo, com um simples ajuste na função “algoritmoGenetico”, podemos armazenar a distância mais curta de cada geração em uma lista de progresso e, em seguida, plotar os resultados.

Obs.: A cada execução é retornado um resultado diferente.

Referências

http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5

https://gist.github.com/turbofart/3428880

https://gist.github.com/NicolleLouis/d4f88d5bd566298d4279bcb69934f51d

https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante