Algoritmos Genéticos

Problema do caixeiro viajante

Bruno Jaciel de Mello
Leonardo Borck da Silveira
Victor Trindade de Carvalho

Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante é um problema que tenta determinar a menor rota para percorrer uma série de cidades, retornando à cidade de origem. Porém, quando vamos resolver de forma computacional, há um custo de processamento alto, pois o tamanho do espaço de procura aumenta exponencialmente dependendo de *n*, o número de cidades, uma vez que existem.

Equação:

$$(n-1)!/2 \approx \frac{1}{2}\sqrt{2\pi(n-1)}\left(\frac{n-1}{e}\right)^{n-1}$$

Problema do Caixeiro Viajante

Para quatro cidades A, B, C e D, considere que o caixeiro saia de A, visite as demais cidades em qualquer ordem e retorne a A no menor custo

PCV simétrico: $C_{ij} = C_{ji}$ $(n-1)!/2 = 3$

PCV assimétrico: $C_{ij} != C_{ji}$ $(n-1)! = 6$

Rotas Válidas
ABCDA
ADCBA
ACBDA
ADBCA
ABDCA
ACDBA

Problema do Caixeiro Viajante

- Crescimento da complexidade

  Considere um computador capaz de fazer 1 bilhão de adições por segundo e que no caso de 20 cidades, o computador precise apenas de 19 adições para dizer qual o comprimento de uma rota e então será capaz de calcular 10^9 / 19 = **53 milhões de rotas por segundo**

N° DE ROTAS	ROTAS P SEG.(MILHÕES)	(n-1)!/2	TEMPO
5	250	12	inexpressivo
10	110	181.400	inexpressivo
15	71	43.5 bilhões	10 min
20	53	$6*10^{16}$	36.5 anos

O Algoritmo

Para resolver este problema, foi utilizado um algoritmo genético.

O Algoritmo genético possui as seguintes fases:

- Criação de uma população inicial
- Determinação do "fitness"
- Seleção dos pais para procriação
- Procriação
- Mutação
- Repetição do ciclo por x gerações

População inicial

```python
import numpy as np, random, operator, pandas as pd

from domain.Fitness import *

def CriaRota(listaDeCidades):
    rota = random.sample(listaDeCidades, len(listaDeCidades))
    return rota

def PopulacaoInicial(tamanhoDaPopulacao, listaDeCidades):
    populacao = []

    for i in range(0, tamanhoDaPopulacao):
        populacao.append(CriaRota(listaDeCidades))
    return populacao
```

Aqui criamos a primeira geração selecionando aleatoriamente a ordem em que as cidades serão visitadas.

Fitness

Calculamos a função de aptidão (fitness) para encontrar a melhor solução.

A classe Fitness tem um método que calcula o total das distâncias (DistanciaPercorrida) e outro método que retorna o valor da aptidão de acordo com o valor total das distâncias (FitnessDaRota).

Neste trecho de códigos nós queremos minimizar a distância, então quanto maior o fitness melhor o score. Isso porque estamos tratando o fitness como inversamente proporcional ao tamanho da rota.

```python
from domain.CarregaCidades import *

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

    def DistanciaPercorrida(self):
        if self.distancia == 0:
            distanciaPercorrida = 0
            for i in range(0, len(self.rota)):
                cidadeDeOrigem = self.rota[i]
                cidadeDeDestino = None
                if i + 1 < len(self.rota):
                    cidadeDeDestino = self.rota[i + 1]
                else:
                    cidadeDeDestino = self.rota[0]
                distanciaPercorrida += Distancia(cidadeDeOrigem,cidadeDeDestino)
            self.distancia = distanciaPercorrida
        return self.distancia

    def FitnessDaRota(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.DistanciaPercorrida())
        return self.fitness
```

Escolha do melhor indivíduo

Aqui é como acontece a escolha do indivíduo "mais apto" da seleção natural. O resultado será uma lista ordenada com os seus respectivos scores.

A função de seleção escolhe os pais que serão usados na reprodução para gerar a nova população utilizando o fator de aptidão para a escolha.

Outro fator considerado aqui é o **elitismo**, que garante que os indivíduos com melhor desempenho da população sejam transferidos automaticamente para a próxima geração, fazendo com que indivíduos com mais sucesso persistam.

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

def Selecao(populacaoOrdenada, tamanhoDaElite):
    resultadosDaSelecao = []
    dataFrame = pd.DataFrame(np.array(populacaoOrdenada), columns=["Index","Fitness"])
    dataFrame['cum_sum'] = dataFrame.Fitness.cumsum()
    dataFrame['cum_perc'] = 100*dataFrame.cum_sum/dataFrame.Fitness.sum()

    for i in range(0, tamanhoDaElite):
        resultadosDaSelecao.append(populacaoOrdenada[i][0])
    for i in range(0, len(populacaoOrdenada) - tamanhoDaElite):
        escolhe = 100*random.random()
        for i in range(0, len(populacaoOrdenada)):
            if escolhe <= dataFrame.iat[i,3]:
                resultadosDaSelecao.append(populacaoOrdenada[i][0])
                break
    return resultadosDaSelecao

def PaisParaProcriacao(populacao, resultadosDaSelecao):
    paisDaProximaGeracao = []
    for i in range(0, len(resultadosDaSelecao)):
        index = resultadosDaSelecao[i]
        paisDaProximaGeracao.append(populacao[index])
    return paisDaProximaGeracao
```

Procriação

Agora é o momento de criar a próxima geração com o processo de **crossover (procriação)**. Aqui é o momento de definir quais genes pegar de cada pai.

Para este algoritmo a decisão foi escolher aleatoriamente um subconjunto do primeiro pai e preencher o resto com os genes do segundo pai, sem duplicar genes.

Exemplo:

Pai 1: [(6,10),(4,9),(20,15),***(7,5)*,*(5,6)***,(8,6)]

pai 2: [***(6,10)*,*(8,6)***,(5,6),***(20,15)*,*(4,9)***,(7,5)]

Filho: [**(7,5),(5,6),(6,10),(8,6),(20,15),(4,9)**]

```python
def Procriacao(pai1, pai2):
    filho = []
    filhoP1 = []
    filhoP2 = []

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

    geneInicial = min(geneA, geneB)
    geneFinal = max(geneA, geneB)

    for i in range(geneInicial, geneFinal):
        filhoP1.append(pai1[i])

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

    filho = filhoP1 + filhoP2
    return filho


def ProcriacaoDaPopulacao(paisDaProximaGeracao, tamanhoDaElite):
    filhos = []
    quantidade = len(paisDaProximaGeracao) - tamanhoDaElite
    selecionados = random.sample(paisDaProximaGeracao, len(paisDaProximaGeracao))

    for i in range(0,tamanhoDaElite):
        filhos.append(paisDaProximaGeracao[i])

    for i in range(0, quantidade):
        filho = Procriacao(selecionados[i], selecionados[len(paisDaProximaGeracao)-i-1])
        filhos.append(filho)
    return filhos
```

Mutação

A mutação desempenha uma função importante no algoritmo genético, pois ajuda a evitar a convergência local introduzindo novas rotas que nos permitirão explorar outras partes do espaço de solução. Ou seja, uma forma de variar a população trocando de forma aleatória duas cidades em uma rota

Exemplo:

[(7,5),***(5,6)***,(6,10),***(8,6)***,(20,15),(4,9)]

[(7,5),***(8,6)***,(6,10),***(5,6)***,(20,15),(4,9)]

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

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

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

```python
def MutacaoDaPopulacao(populacao, taxaDeMutacao):
    populacaoMutada = []

    for individuo in range(0, len(populacao)):
        mutaIndividuo = Mutacao(populacao[individuo], taxaDeMutacao)
        populacaoMutada.append(mutaIndividuo)
    return populacaoMutada
```

Repetição

Novas gerações, fase de Repetição

Este processo se resume em:

- Escolher o indivíduo mais apto (utilizando a função AvaliaRotas)
- Escolher os potenciais pais (utilizando a função Seleção)
- Pegar os pais da população (utilizando PaisParaProcriacao)
- Procriar(utilizando a função ProcriacaoDaPopulacao)
- Aplicar a mutação (utilizando MutacaoDaPopulacao)

```
def ProximaGeracao(geracaoAtual, tamanhoDaElite, taxaDeMutacao):
    populacaoOrdenada = AvaliaRotas(geracaoAtual)
    resultadosDaSelecao = Selecao(populacaoOrdenada, tamanhoDaElite)
    paisDaProximaGeracao = PaisParaProcriacao(geracaoAtual, resultadosDaSelecao)
    filhos = ProcriacaoDaPopulacao(paisDaProximaGeracao, tamanhoDaElite)
    proximaGeracao = MutacaoDaPopulacao(filhos, taxaDeMutacao)
    return proximaGeracao
```

Solucionar o problema de acordo com os parâmetros de entrada

```
def AlgoritmoGenetico(populacao, tamanhoDaPopulacao, tamanhoDaElite, taxaDeMutacao, quantidadeDeGeracoes,mostraSomenteResultado):
    populacaoAtual = PopulacaoInicial(tamanhoDaPopulacao, populacao)

    for i in range(0, quantidadeDeGeracoes):
        if(not mostraSomenteResultado):
            print("GERAÇÃO: " + str(i+1))
            for individuo in populacaoAtual:
                print("Distancia Da Rota: " + str(Fitness(individuo).DistanciaPercorrida()))
                print(individuo)
            print("")
        populacaoAtual = ProximaGeracao(populacaoAtual, tamanhoDaElite, taxaDeMutacao)

    menorDistancia = 1 / AvaliaRotas(populacaoAtual)[0][1]

    print("Melhor Rota com " + str(quantidadeDeGeracoes) + " gerações")
    print("Distancia Da Rota: " + str(int(menorDistancia)))
    for individuo in populacaoAtual:
        if(menorDistancia == Fitness(individuo).DistanciaPercorrida()):
            print(individuo)
            break
    indiceDaMelhorRota = AvaliaRotas(populacaoAtual)[0][0]
    melhorRota = populacaoAtual[indiceDaMelhorRota]
    return melhorRota
```

Repositório

https://github.com/leonardoborck/traveling-salesman

Referências

http://aprepro.org.br/conbrepro/2019/anais/arquivos/09302019_220914_5d92b20230a58.pdf