Universidade Federal do Estado do Rio de Janeiro - UFRJ
Métodos computacionais em física II

Estevan Augusto Amazonas Mendes

Algoritmos Genéticos aplicados ao Problema do Caixeiro Viajante

Rio de Janeiro - RJ.
Dezembro de 2019.

Introdução

O problema do caixeiro viajante se mostra um desafio de otimização combinatória do tipo NP-difícil de grande interesse para diversas áreas, e um dos mais populares [1]. Consiste em realizar a viagem para diversas cidades otimizando o custo associado a cada uma delas. Pode-se otimizar desde a distância até o tempo associado. A multidisciplinaridade associada ao problema leva a diversas aplicações, como na produção de circuitos impressos [5]. Os objetivos do presente trabalho são: (1) a criação de um código que realize a otimização por meio de algoritmos genéticos, (2) analisar os diferentes tipos de operadores de recombinação, e (3) reproduzir resultados expostos no artigo de Potvin [3].

Metodologia

Os algoritmos genéticos (GA) se amparam no processo evolutivo sob uma perspectiva darwiniana, selecionando os indivíduos mais bem adaptados. Nesse sentido, os GA's tem por objetivo realizar a otimização de determinados parâmetro definidos previamente fazendo com que após gerações de pressão evolutiva, os indivíduos pertencentes a população compartilhe a característica almejada. O algoritmo genético simples idealizado na década de 80 por David Goldberg [6], é composto por uma população fixa, isto é, o número de indivíduos de uma geração.Os genes que definem os indivíduos são binários. A população é classificada em por uma função desempenho. Os indivíduos com os melhores desempenhos são escolhidos para gerarem a próxima população. A nova geração é criada por meio dos melhores desempenhos e de operadores de recombinação dos genes.A cada geração a população é exposta a possibilidades de mutação dos genes.O processo é acontece até que um critério de convergência seja atingido O algoritmo encontra-se ilustrado na figura 1.

```
Algoritmo Genético Simples {
    Definindo {
          função desempenho
          formação do indivíduo e tamanho da população
          probabilidade dos operadores
    }
    Inicializar população aleatória
    Enquanto não alcançar critério de convergência faça {
          avaliar os indivíduos da população
          executar seleção
          executar cruzamento e mutação
    }
}
```

Figura 1: algoritmo associado ao GA simples desenvolvido por Goldberg.

O problema do caixeiro viajante(PCV) consiste em realizar o menor trajeto possível entre N cidades que devem ser visitados. Sendo um problema do Tipo NP completo, mostra-se inviável calcular os caminhos possíveis e escolher o melhor, uma vez que há (N-1)! caminhos possíveis. Na implementação do algoritmo genético associado ao PCV se faz necessário o tratamento de tour, que significa que a posição os genes não mais serão binários, e representarão uma cidade. Nesse tratamento, a posição da cidade na string, ou

no vetor, guarda uma informação relevante, a ordem de visitação da cidade. Além disso, o desempenho de cada, caminho, ou cada indivíduo está relacionado com a distância percorrida pelo caminho. A função de desempenho, ou FIT como é referida na literatura é construída no problema do caixeiro viajante utilizando-se como base a distância entre as cidades. A biblioteca TSPLIB, disponibiliza matrizes com distâncias para problemas assimétricos, quando a distância entre a cidade A e B, é diferente do distância BA, e para problemas simétricos de diversas tamanhos, ou seja, para diferente números de cidades a serem visitadas. Os GA's selecionam os melhores indivíduos, ou seja, com os maiores desempenhos. Por isso, a função desempenho, ou FIT deve igual ao inverso da distância, ou enquanto a o valor máximo da distância entre as cidade menos a distância em tratada. Dessa forma, os maiores FIT estarão associados aos caminhos mais curtos. A escolha dos indivíduos que devem gerar a nova população é dados pelo método de seleção proporcional ao FIT. Para a implementação é necessário somar os valores de todos os FITs da geração, e sortear um número aleatório, entre 0, e a soma dos FIT. De forma ordenada, soma-se o FIT de cada indivíduo, o indivíduo cuja soma do FIT ultrapassa o valor sorteado é escolhido. Uma ferramenta utilizada para manter boas características na população é criar um elitismo, isto é, um grupo com os melhores FITs de cada geração. A elite é passada para a próxima geração sem modificações. Os genes dos pais selecionados são recombinados para criar novos indivíduos segundos dois operadores neste trabalho. O primeiro, o operador de recombinação ordenado (BREED), sorteia dois valores aleatórios entre 0, e o número de cidades visitadas. Copia-se de um dos pais os genes no intervalo dos números sorteados. Os outros genes são copiados dos segundo pai obedecendo a ordem que aparecem, exclui-se as cidades inseridas no caminho previamente pelo primeiro pai. Esse processo encontra-se ilustrado na figura 2. O segundo, o operador de recombinação dos extremos utilizada os vizinhos de cada gene para criar um novo indivíduos. Cria-se uma lista com os genes vizinhos de cada cidade visitada, seguindo a ordem em que aparecem no vetor, ou na string. Por exemplo, se tivermos os pais [1,2,3,4 e [4,3,2,1] as primeiras cidades visitadas respectivamente serão 1 e 4. Seus vizinhos serão 2,4 e 3,1. O novo indivíduo tomará o gene que tiver o menor custo, a menor distância, ou seja, que maximizar o FIT. A nova população é gerada favorecendo os indivíduos com maior FIT, consequentemente, caso haja um FIT muito grande este indivíduo rapidamente será responsável pela criação das novas gerações. E por isso, a diversidade de soluções pode ser pedida, levando somente a uma solução máxima local. A ferramenta de mutação fornece uma alternativa para que haja diversidade na população. Todos os indivíduos têm uma probabilidade que é definida previamente por um parâmetro PM de sofrer uma mutação. A cada geração nova sorteia-se para cada caminho um número aleatório, caso seja menor que PM, os genes sofrem mutação. Novamente, sorteia-se dois valores correspondentes a posição de dois genes do indivíduo que trocarão de lugar.

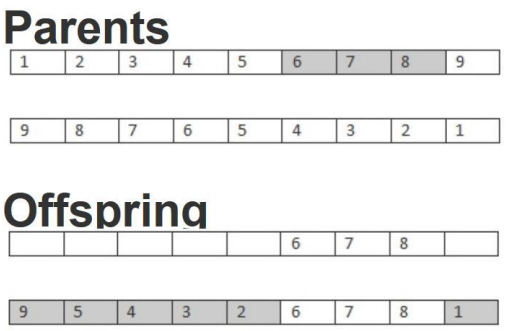

FIgura 2: Exemplo do processo de recombinação por meio de um operador linear(BREED).

Resultados e Discussão

O código foi desenvolvido encontra-se em no anexo 1, utilizou-se a linguagem de programação python. O algorítmo genético teve uma população de 200 indivíduos por geração. Foram utilizadas 3 bibliotecas do Python, numpy, matplotlib, e Random. O parâmetro de mutação escolhido foi de 1%, e o tamanho da elite utilizado foi de 25% do número de cidades visitadas. O conjunto de dados assimétricos analisados para o problema do caixeiro viajante para 171 cidades, FTRV170, foram obtidos da biblioteca TSPLIB. Por meio dos dados padronizados obtidos, foi possível observar a eficiência dos diferente operadores de recombinação implementados. Pode-se concluir que o operador de recombinação dos extremos (ERX) foi mais eficiente que o BREED. Este resultado encontra-se compatível com a literatura, uma vez que os operadores lineares apresentam um desempenho pouco expressivo [3]. As figuras 3 e 4 ilustram a diferenças entre os operadores de recombinação. Além disso, foi possível reproduzir o comportamento associado ao operador ERX exposto pelo artigo de Zakir Ahmed [4] que pode ser observado na figura 4.

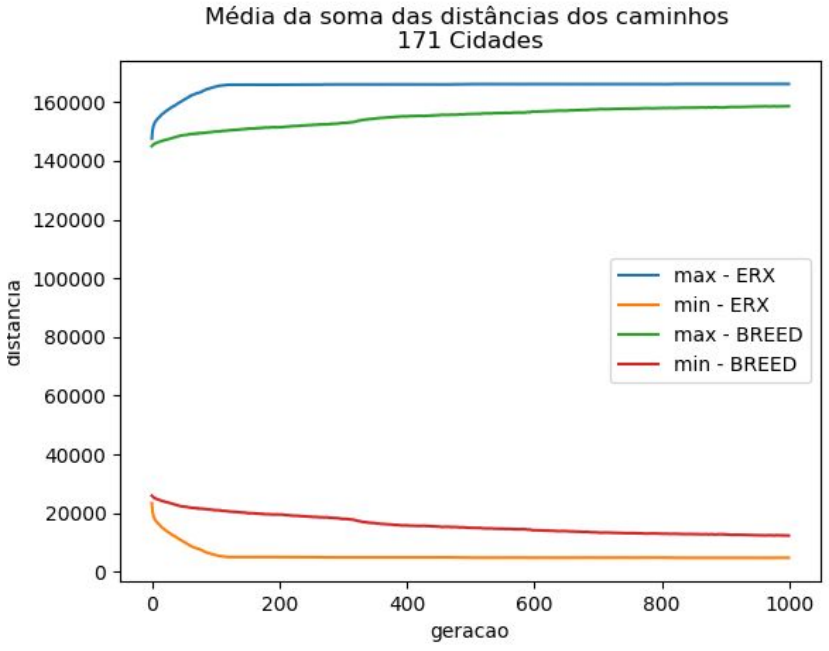

Figura 3: Gráfico da distância média de um caminho em função das gerações.

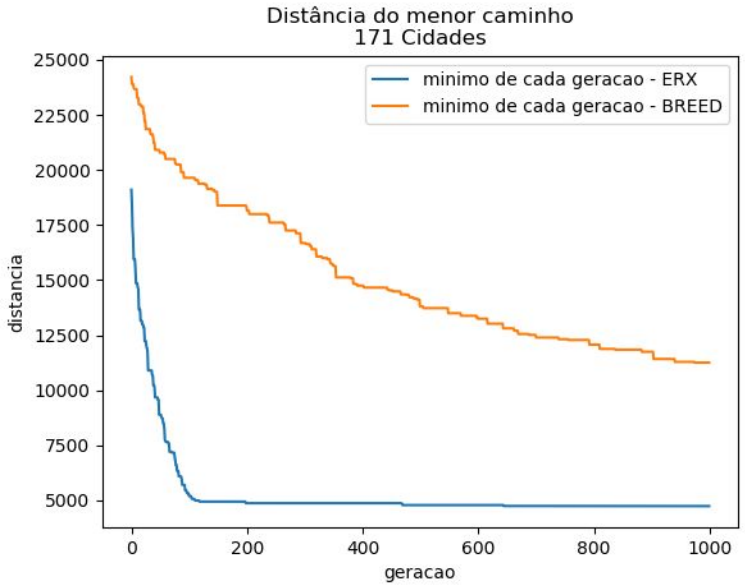

Figura 4: gráfico da distância do melhor caminho a cada geração.

Referências

[1] HOFFMAN, A. J.; WOLFE, P. History. In: LAWLER, E. L (Ed.); LENSTRA, J. K. (Ed.); RINNOOY KAN, A. H. G. (Ed.); SHMOYS, D. B. (Ed.). The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. New York: John Wiley & Sons, 1985. p. 1-16.
[2] HOLLAND, J. H., Adaptation in Natural and Artificial Systems. Ann Arbor: University of Michigan Press, 1975.
[3] Potvin, JY. Ann Oper Res (1996) 63: 337. https://doi.org/10.1007/BF02125403
[4] Ahmed, Zakir. (2010). Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator. International Journal of Biometric and Bioinformatics. 3.
[5] CAMPELLO, R. E.; MGenetic Algorithms in Search, Optimization and Machine Learning, N. Algoritmos e Heurísticas. 1. ed. Niterói. EDUFF, 1994.
[6]David E. Goldberg. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1989,ISBN:0201157675ACULAN.

Anexo 1 - Código desenvolvido

```
#----------le os dados dos caminhos dos arquivos de TSPLIB
def leitorteste():
 data=open("ftv170.dat","r")
 temp=data.read()
 data.close()
 import numpy as np
 S=171
 import re
 numbers = re.findall(r"[-+]?\d*\.\d+|\d+",temp)

 ax1=[None]*S*S
 ax1=numbers
 a=np.zeros((S,S))
 for i in range(0,S):
   a[i,:]=ax1[S*i:S*(i+1)]
 return a
#----- calculando fitness de cada indivíduo ----
```

```
  # ---- A=geracao ---- B=fitfuncao
def fitness(A,B):
 aux=np.zeros(populacao)
 for k in range(0,len(aux)):
   ax1=0
   for s in range(0,Ncidades):
    ax1+=B[A[k,s%Ncidades],A[k,(s+1)%Ncidades]]
   aux[k]=float(ax1+float(k/1000.))
 return aux

def Testefit(IND,B):
   IND=np.array(IND)
   ax1=0.
   for s in range(0,Ncidades):
    ax1+=B[IND[s%Ncidades],IND[(s+1)%Ncidades]]
   return ax1

#---- selecao dos cromossomos -----
def probselec(V):
  var1=sum(V)
  var2=rd.randint(0,int(var1))
  aux=0
  k=0
  for s in range(len(V)):
    aux+=V[s]
    k=s
    if aux>=var2:
      break
  return k

#----selecionar os melhores cromossomos metodos ERX -
edge recombination crossover

def ERX(A,B,p1,p2):
  f1=np.empty(Ncidades)
  p1=int(p1)
  p2=int(p2)
  edge=np.empty((Ncidades,4))
  f1[:]=Ncidades+1000
  for s in range(0,Ncidades):

edge[s,:]=[A[p1%populacao,(Ncidades+s-1)%Ncidades],A[
p1%populacao,(s+1)%Ncidades],A[p2%populacao,(Ncida
des+s-1)%Ncidades],A[p2%populacao,(s+1)%Ncidades]]
  f1[0]=0#rd.choice(edge[0,:])
  C=B
  for t in range(1,Ncidades):
    b=0
    k=0
    ax1=0
    ax2=0
    while(k<1):
      a=0
      b=C[int(f1[t-1]),:]
      for n in range(0,4):
        bx1=b[int(edge[t,n])]
        if bx1>a:  # Maximizando
          a=bx1
          ax1=n
      k=10
      for s in f1:
        if int(s-edge[(t%Ncidades),ax1])==0:
          k=0
          ax2+=1
          b[int(edge[t,ax1])]=0
          if ax2==10:
           ax3=np.setdiff1d(range(Ncidades),f1)
           edge[(t%Ncidades),ax1]=rd.choice(ax3)
           k=10
    f1[t]=int(edge[(t%Ncidades),ax1])
  if sum(f1)!=sum(range(Ncidades)):
    print("ERRRO")
  return f1

# Funcao pra fazer o gráfico da distancia média percorrida
por geracao


def grafico1(x1,y1,char1,x2,y2,char2,char3):
 import matplotlib.pyplot as plt
 plt.title("Média da soma das distâncias dos caminhos
\n"+str(char3)+" Cidades")
 plt.ylabel("distancia")
 plt.xlabel("geracao")
 plt.plot(x1,y1,label=char1)
 plt.plot(x2,y2,label=char2)
 plt.legend()
 #plt.show()

def grafico2(x1,y1,char1,char2):
 import matplotlib.pyplot as plt
 plt.title("Distância do menor caminho\n"+str(char2)+"
Cidades")
 plt.plot(x1,y1,label=char1)
 plt.ylabel("distancia")
 plt.xlabel("geracao")
 plt.legend()
 #plt.show()

#distancia média percorrida por geracao e o minimo

def medDIST(A,B,d):
  ax1=fitness(A,B)
  if d=="media":
   med=sum(ax1)/len(ax1)
   return med
  if d=="minimo":
   minimo=min(ax1)
   return minimo

#Mutações

def mutacao(A):
 PM=0.01#parametro de mutação
 for ax1 in range(ELITESIZE+1,populacao):
   var1=rd.random()
   if var1<=PM:
     ax2=rd.randint(0,Ncidades-1)
     ax3=rd.randint(0,Ncidades-1)
     if ax2==ax3:
       ax3=(ax2+1)%Ncidades
     temp1=A[ax1,ax2]
     temp2=A[ax1,ax3]
     A[ax1,ax3]=temp1
     A[ax1,ax2]=temp2

 return A

#-breed


def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(rd.randint(0,Ncidades-1))
    geneB = int(rd.randint(0,Ncidades-1))

    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



#main()
import numpy as np
import random as rd

NGERACAO=1000
ELITESIZE=1+100
Ncidades=171
populacao=200
cidade= np.arange(Ncidades)
#distancia=np.random.randint(1000,size=(Ncidades,Ncida
des))
#distancia = np.tril(distancia) + np.tril(distancia, -1).T
distancia=np.array(leitorteste())
print(distancia)
fitfuncao=1000-distancia
for k in range(Ncidades):
 fitfuncao[k,k]=0
print(fitfuncao)
print(distancia)
fitfuncao.flags.writeable = False

geracao=np.arange(Ncidades)
pgeracao=geracao
for d in range(0,populacao-1):
 np.random.shuffle(geracao[1:])
 pgeracao=np.vstack([pgeracao,geracao])
geracao=pgeracao
print(geracao)

#-------   loop para obter os melhores caminhos
-----------------

  #k=0
  #if k==1:
   #pais=np.zeros(populacao)
   #for k in range(0,int(populacao/2)):
    #var4=probselec(FIT)
    #var5=probselec(FIT)
    #if var4!=var5 or var4==var5 :#permiti dois pais iguais
     # pais[k*2]=var4
      #pais[k*2+1]=var5
    #if var4==300:
     # pais[k*2]=var4
      #pais[k*2+1]=(var5+1)%populacao



 #>>>>>definindos os pais que criaram uma
prole---proportional selection<<<<

var8=np.zeros(NGERACAO*2)
var9=np.zeros(NGERACAO)
var10=np.zeros(NGERACAO)
var11=np.zeros(NGERACAO)
var12=np.zeros(NGERACAO)
var13=np.zeros(NGERACAO)
FIT1=np.zeros(populacao)
FIT2=np.zeros(populacao)
geracao1=np.array(geracao)
plt.show()
geracao2=np.array(geracao)
for u in range(0,NGERACAO):
  #DIT=fitness(geracao,distancia)
  geracaon1=np.array(geracao1)
  FIT1=fitness(geracao1,fitfuncao) #tem 200 valores que
representam cada linha da matriz geracao, que represeta
o caminho
  tmp1=np.argmax(FIT1)#Passando pra próxima geracao
o melhor caminho desta geracao
  tmp2=sorted(FIT1)
  tmp3=tmp2[-ELITESIZE:-1]
  geracaon1[0,:]=geracao1[int(tmp1),:]
  for t in range(1,ELITESIZE):
    az1=1000.*float(tmp3[t-1]%int(tmp3[t-1]))
    if np.fabs(float(int(az1)-az1))>0.1:
     az1+=1
    geracaon1[t,:] =geracao1[int(az1),:]
  FIT2=fitness(geracao2,fitfuncao) #tem 200 valores que
representam cada linha da matriz geracao, que represeta
o caminho
  tmp1=np.argmax(FIT2)#Passando pra próxima geracao
o melhor caminho desta geracao
  tmp2=sorted(FIT2)
  tmp3=tmp2[-ELITESIZE:-1]
  geracaon2=np.array(geracao2)
  geracaon2[0,:]=geracao2[int(tmp1),:]
  for t in range(1,ELITESIZE):
    az1=1000.*float(tmp3[t-1]%int(tmp3[t-1]))
    if np.fabs(float(int(az1)-az1))>0.1:
     az1+=1
    geracaon2[t,:] =geracao2[int(az1),:]

  for i in range(ELITESIZE-1,int(populacao)):
    matAUX=np.array(fitfuncao)

var6=ERX(geracao1,matAUX,probselec(FIT1),probselec(F
IT1))

var62=breed(geracao2[probselec(FIT2),:],geracao2[probs
elec(FIT2),:])
    geracaon1[i,:]=var6
    geracaon2[i,:]=var62
  geracao1=np.array(mutacao(geracaon1))
  geracao2=np.array(mutacao(geracaon2))
  #geracao=np.array(geracaon)

#---------------GRAFICO-------------------------------------\\\\\
  var8[u*2]=medDIST(geracao1,fitfuncao,"media")
  var9[u]=medDIST(geracao1,distancia,"media")
  var8[u*2+1]=u
  var10[u]=medDIST(geracao1,distancia,"minimo")
  var11[u]=medDIST(geracao2,fitfuncao,"media")
  var12[u]=medDIST(geracao2,distancia,"media")
  var13[u]=medDIST(geracao2,distancia,"minimo")

print("----------------",str(u*100/NGERACAO),"%--Feito-------
-----")

import matplotlib.pyplot as plt
grafico1(var8[1::2],var8[0::2],"max -
ERX",var8[1::2],var9,"min - ERX",str(Ncidades))
grafico1(var8[1::2],var11[0::],"max -
BREED",var8[1::2],var12,"min - BREED",str(Ncidades))
plt.show()
grafico2(var8[1::2],var10,"minimo de cada geracao -
ERX",str(Ncidades))
grafico2(var8[1::2],var13,"minimo de cada geracao -
BREED",str(Ncidades))
```