UFPR - UP - UTFPR - PUCPr – 11/02/2019 - 12:33:56.4
Prof Dr P Kantek (pkantek@gmail.com)
Completando algoritmos V VIVO149a, V: 1.20 1 - 1 _____ / _____ / _____

Sistemas Lineares - solução ingênua

É muito comum em engenharia e ciências exatas, o surgimento de sistemas (muitas equações) lineares (de grau 1 no máximo), como por exemplo

$$\begin{array}{rcrcrcr} 3x & + & 8y & + & 9z & = & 18 \\ -2x & + & y & - & z & = & -11 \\ 5x & - & y & + & 3z & = & 25 \end{array}$$

Que tem como resposta $x = 2$, $y = -3$ e $z = 4$.

Existem inúmeras estratégias para a solução destes sistemas, aqui vai se ver o algoritmo (ingênuo) de escalonamento de Gauss. A idéia é aplicar operações elementares nas linhas, gerando sistemas equivalentes (sistemas equivalentes=têm a mesma solução) de tal maneira que cada linha comece com zeros de maneira escalonada. Acompanhe

1. Multiplicando a segunda equação por 5 e a terceira por 2 e somando ambos e substituindo a terceira equação pelo resultado, fica
   $3y + z = -45$

2. Multiplicando a primeira equação por -5 e a terceira por 3 e somando ambos e substituindo a terceira equação pelo resultado fica
   $-43y - 36z = -15$

3. Agora soma-se a segunda equação por 43 e a terceira por 3, soma-se e a nova terceira fica:
   $65z = 260$

4. Daqui, o primeiro resultado é $z = 4$. Levando este valor na segunda equação ($3y + z = -5$), fica $y = -3$ e finalmente, levando estes 2 valores na primeira equação ($3x + 8y + 9z = 18$), tem-se $x = 2$, concluindo-se a solução do sistema

O algoritmo que acha esta solução (para qualquer sistema linear onde número de equações = número de incógnitas) é

```
import numpy as np
def fsl(x):
  t=len(x)
  aux=[0.0]*(t+1)
  i=0
----------------------------------------
    j=t-1
    while j>i:
      if x[j-1][i]==0:
        for k in range(t+1):
          aux[k]=x[j][k]
          x[j][k]=x[j-1][k]
          x[j-1][k]=aux[k]
      if x[j-1][i]==0 and x[j][i]==0:
        ola=0.0
      else:
        ola=(-x[j][i])/x[j-1][i]
      for k in range(t+1):
        x[j][k]=x[j][k]+(x[j-1][k]*ola)
----------------------------------------
    i=i+1
  i=t-1
  r=[0.0]*t
  while i>=0:
----------------------------------------
    for k in range(t):
      x[k][i]=x[k][i+1]-(x[k][i]*r[i])
    i=i-1
  print(r)
z=np.array([[3,8,9,18],[-2,1,-1,-11],
   [5,-1,3,25.0]])
fsl(z)
```

Caixeiro Viajante

Este é um problema célebre na ciência da Computação. Talvez seja o mais famoso problema cuja complexidade (demora) seja $O(n!)$ onde n é o número de cidades que devem ser percorridas. Imagine um vendedor que tem um carro e que deve percorrer um determinado conjunto de cidades. Deve sair da cidade onde reside visitar 1 vez cada uma das demais cidades e voltar à cidade original, percorrendo a menor distância possível entre elas. Vamos ver o que acontece À medida em que o conjunto de cidades cresce.

1. Para 1 cidade, não há problema e a resposta é zero km.

2. Para 2 cidades, a solução é o dobro da distância entre elas.

3. Para 3 cidades, a solução é a distância triangular entre elas.

4. Para 4 cidades, digamos: A,B,C e D, começando em A, as opções são ABCDA, ABDCA, ACBDA, ACDBA, ADBCA e ADCBA, num total de 6 trajetos. Olhando a tabela acima, 2 simplificações podem ser notadas:

   (a) A primeira, é que a cidade origem=destino (A no exemplo) pode ser descartada, já que é fixa.

   (b) A segunda, é que a lista apresenta resultados duplicados, já que – por exemplo, o caminho ABCDA é o mesmo que ADCBA, só que em ordem inversa.

   Neste caso foram 3 caminhos a estudar.

5. Para 5 cidades, serão 12 caminhos (Desafio: escreva eles...)

6. Para 6 cidades, serão 60 caminhos.

7. Para 7 cidades, serão 360 e para 8 cidades, 2520.

8. Para n cidades serão $\frac{(n-1)!}{2}$ trajetos.

Para 10 cidades são 181.440 trajetos e para 15 cidades são 43589145600 trajetos. Para 40 cidades serão 10198941040598721679320140869951448678400000000 trajetos.

Antes de executar o algoritmo necessita-se uma tabela de distâncias entre todas as cidades. (Nada muda no algoritmo, se em vez de distâncias, se usar custo de pedágio, horas de vôo, paradas para abastecimento, ..., o que se quiser). Neste caso, como não se tem as distâncias rodoviárias, o que seria o mais correto, e sim as coordenadas X e Y da cidade, há que se construir a tal matriz de distâncias. Isto é feito usando o Teorema de Pitágoras.

Eis o algoritmo

```
# caixeiro viajante
import numpy
no=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
lx=[164,467,91,-25,338,291,310,200,543,158]
ly=[216,63,-8,213,62,297,202,-54,51,323]
mat=numpy.zeros((len(no),len(no)))
for i in range(len(no)):
  for j in range(len(no)):
    mat[i][j]=(((lx[i]-lx[j])**2)+((ly[i]-ly[j])**2))**0.5
def cv(lja,lfa,km,mat):
  global minimo
  global caminho
  if len(lfa)!=0:
    ori=lja[len(lja)-1]
    for i in range(len(lfa)):
      des=lfa[i]
      alfa=lfa[:]
-------------------------------------------------------------
      alja=lja[:]
-------------------------------------------------------------
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
-------------------------------------------------------------
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([9],[0,1,2,3,4,5,6,7,8],0,mat)
print(minimo,' Km')
for i in range(len(caminho)-1):
  print('de ',no[caminho[i]],' para ',no[caminho[i+1]],
        ' com ',mat[caminho[i],caminho[i+1]],' Km.')
```

O algoritmo acima é o que dá a resposta correta de maneira absoluta. Só que ele pode demorar demais. E, se você tiver que resolver o problema para (digamos) 40 cidades ? Obviamente não dá para esperar o tempo do algoritmo exato. Precisa-se fazer um abrandamento nos requisitos, para poder obter alguma resposta. Uma boa resposta, ainda que não haja certeza de que ela é a ótima, é melhor do que nada.

Guloso

Uma possível solução é usar um algoritmo guloso e neste caso ele é bem simples. A regra agora é: **visite a cidade mais próxima ainda não visitada**. Agora o algoritmo é muito rápido não importando quantas cidades são. Eis como ficou o algoritmo

```
import numpy as np
def gulosotsp():
  ci=['cascavel','guaira','londrina','maringa',
      'palmas','tel. borba','umuarama']
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  dx=np.array([[53,118],[-25,213],[291,297],
              [211,287],[200,-54],[341,185],[72,246]])
  di=np.zeros((t,t),float)
  for i in range(t):
-------------------------------------------------------------
      di[i,j]=(((dx[i,0]-dx[j,0])**2)+((dx[i,1]-dx[j,1])**2))**0.5
  dz=di.copy()
  for i in range(t):
    di[i,i]=99999
    di[i,inicio]=99999
  qual=inicio
-------------------------------------------------------------
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
-------------------------------------------------------------
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

1. Aplique o algoritmo de escalonamento de Gauss ao sistema abaixo e descubra o valor de x_1 e de x_{10}, (a primeira e a última raizes)

37	-11	-22	-10	6	24	34	36	4	-8	-270
36	-1	61	-13	46	49	8	3	69	58	1547
-21	22	59	11	9	59	-22	-24	-21	20	880
62	-8	-28	-15	60	-14	50	-9	-4	45	-502
33	60	-23	42	-20	45	27	54	-11	62	145
48	33	-25	-13	22	26	-24	29	-25	-9	-434
25	-12	54	-5	-18	62	69	27	-12	-18	499
-2	-10	10	30	34	28	14	34	52	-22	512
24	40	63	-9	0	19	-22	44	24	26	1052
31	24	-19	28	19	-24	-5	68	52	7	111

2. Aplique o algoritmo do caixeiro viajante para o seguinte problema e depois aplique o algoritmo guloso para o mesmo conjunto de cidades.

1-FOZ IGUACU	-59	55
2-GUARAPUAVA	254	67
3-IRATI	338	62
4-JACAREZINHO	399	315
5-MARINGA	211	287
6-PALMAS	200	-54
7-PONTA GROSSA	377	102

Começando e terminando na cidade de GUARAPUAVA .

Responda aqui:

x_1	x_{10}	cv exato	guloso