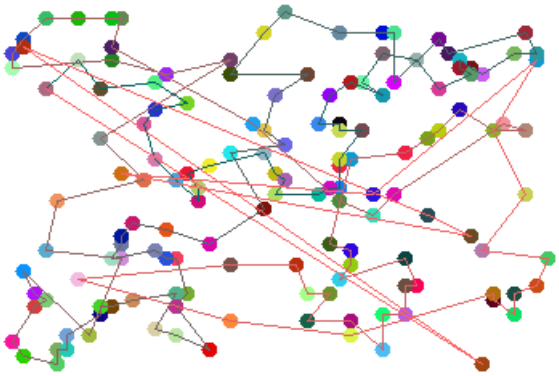
Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-FOZ IGUACU          -59  55
2-LONDRINA            291 297
3-ORTIGUEIRA          310 202
4-PALMAS              200 -54
5-PARANAVAI           158 323
6-PIRAI SUL           397 163
7-PONTA GROSSA        377 102
```

Começando e terminando na cidade de FOZ IGUACU .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	1							1

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

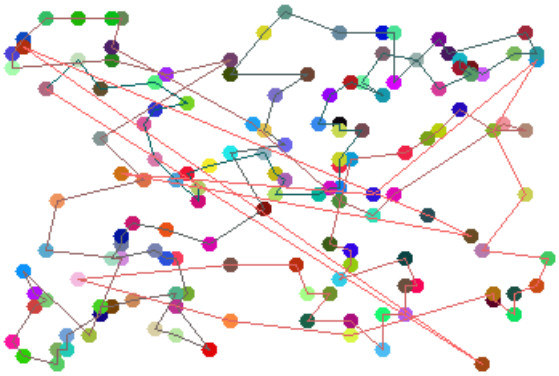
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75565 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CAMPO MOURAO          164 216
2-CASCAVEL               53 118
3-GUARAPUAVA            254  67
4-LARANJ SUL            159  68
5-LONDRINA              291 297
6-PALMAS                200 -54
7-UMUARAMA               72 246
```

Começando e terminando na cidade de UMUARAMA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	7							7

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

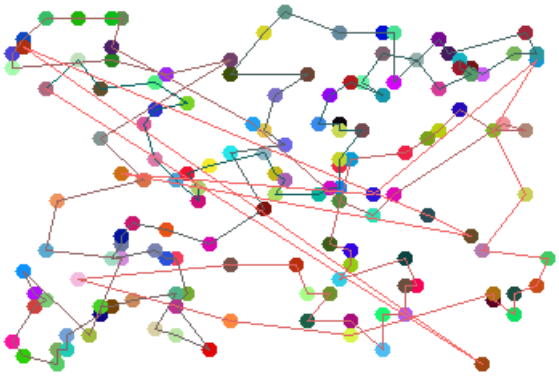
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75572 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CANDIDO ABREU         269 157
2-GUAIRA                -25 213
3-IRATI                 338  62
4-JACAREZINHO           399 315
5-MARINGA               211 287
6-ORTIGUEIRA            310 202
7-PARANAGUA             543  51
```

Começando e terminando na cidade de CANDIDO ABREU .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	1							1

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

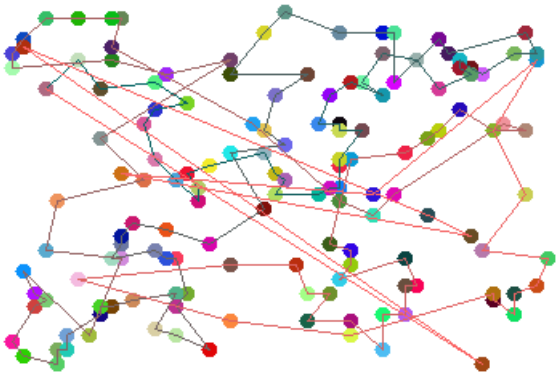
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75589 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-CANDIDO ABREU       269 157
3-CURITIBA            467  63
4-GEN CARNEIRO        268 -49
5-ORTIGUEIRA          310 202
6-PARANAGUA           543  51
7-PONTA GROSSA        377 102
```

Começando e terminando na cidade de PONTA GROSSA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	7							7

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

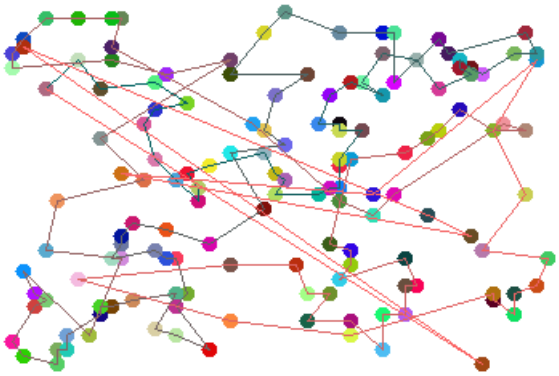
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75596 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS         495 146
2-CAMPO MOURAO         164 216
3-FOZ IGUACU           -59  55
4-GEN CARNEIRO         268 -49
5-IRATI                338  62
6-LONDRINA             291 297
7-UMUARAMA              72 246
```

Começando e terminando na cidade de IRATI .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	5							5

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

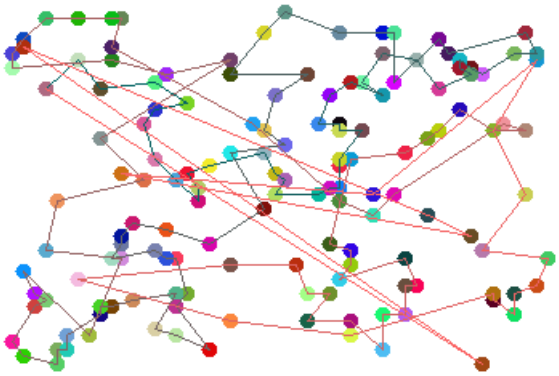
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75608 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CAMPO MOURAO          164 216
2-CANDIDO ABREU         269 157
3-GEN CARNEIRO          268 -49
4-GUAIRA                -25 213
5-JACAREZINHO           399 315
6-LARANJ SUL            159  68
7-PONTA GROSSA          377 102
```

Começando e terminando na cidade de LARANJ SUL .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	6							6

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

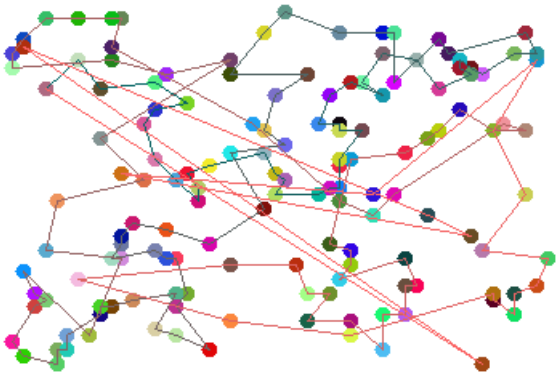
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75615 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-LARANJ SUL          159  68
3-LONDRINA            291 297
4-MARINGA             211 287
5-ORTIGUEIRA          310 202
6-PIRAI SUL           397 163
7-UMUARAMA             72 246
```

Começando e terminando na cidade de PIRAI SUL .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	6							6

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

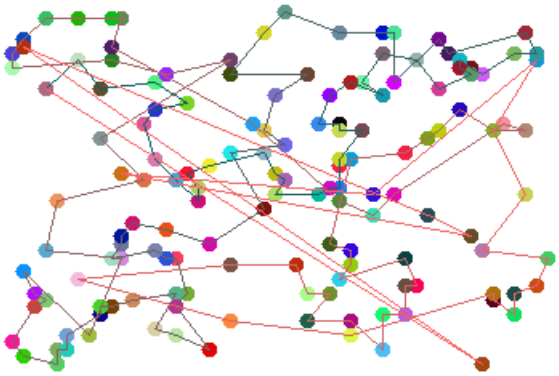
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75622 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CANDIDO ABREU        269 157
2-JACAREZINHO          399 315
3-LONDRINA             291 297
4-ORTIGUEIRA           310 202
5-PARANAVAI            158 323
6-PIRAI SUL            397 163
7-UMUARAMA              72 246
```

Começando e terminando na cidade de PIRAI SUL .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	6							6

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

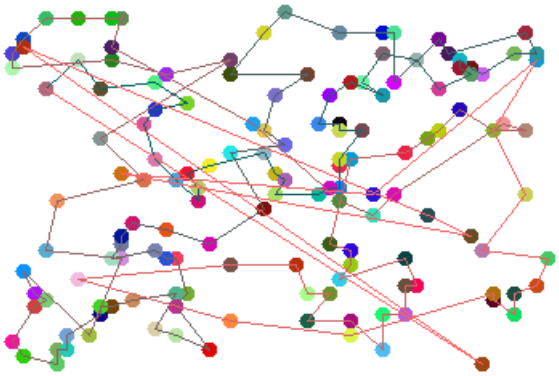
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75639 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-FOZ IGUACU          -59  55
3-IRATI               338  62
4-MARINGA             211 287
5-PARANAGUA           543  51
6-PONTA GROSSA        377 102
7-UMUARAMA             72 246
```

Começando e terminando na cidade de UMUARAMA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	7							7

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

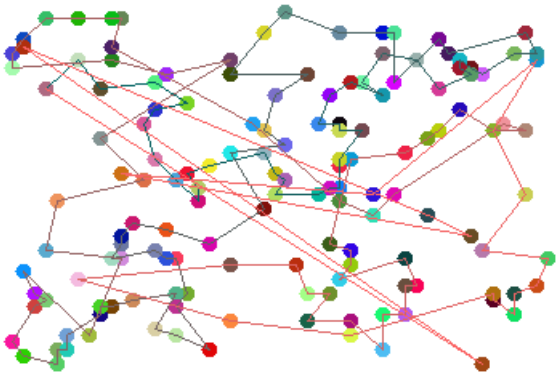
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75646 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS          495 146
2-CAMPO MOURAO          164 216
3-CANDIDO ABREU         269 157
4-GUARAPUAVA            254  67
5-LARANJ SUL            159  68
6-TELEM BORBA           341 185
7-UMUARAMA               72 246
```

Começando e terminando na cidade de TELEM BORBA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	6							6

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

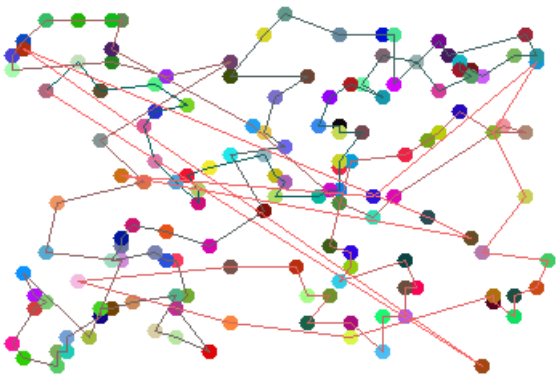
7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75653 - /

CEP - UFPR - UP - UTFPR - PUCPr – 12/08/2024 - 11:58:54.3
Prof Dr P Kantek (pkantek@gmail.com)
Caixeiro Viajante VIVXq83a, V: 2.20 75660 JOAO VICTOR GOMES VILELA GER 24JOG402 - 11 _____ / _____ / _____

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CASCAVEL              53 118
2-JACAREZINHO          399 315
3-LARANJ SUL           159  68
4-PALMAS               200 -54
5-PARANAGUA            543  51
6-PARANAVAI            158 323
7-PIRAI SUL            397 163
```

Começando e terminando na cidade de PALMAS .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	4							4

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

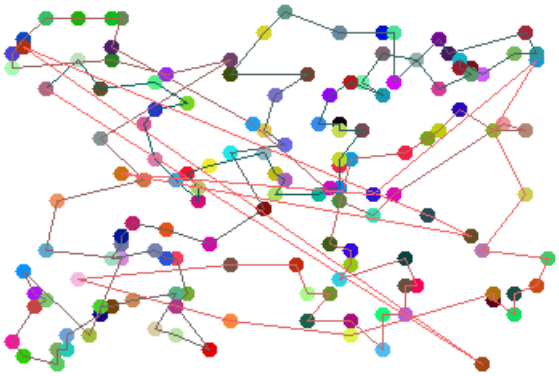
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

CEP - UFPR - UP - UTFPR - PUCPr – 12/08/2024 - 11:58:54.3
Prof Dr P Kantek (pkantek@gmail.com)
Caixeiro Viajante VIVXq83a, V: 2.20 75765 LUCAS ANTOSZCZYSZEN
24JOG402 - 12 _____ / _____ / _____

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-CANDIDO ABREU       269 157
3-CASCAVEL             53 118
4-CURITIBA            467  63
5-FOZ IGUACU          -59  55
6-PARANAGUA           543  51
7-PIRAI SUL           397 163
```

Começando e terminando na cidade de PARANAGUA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	6							6

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

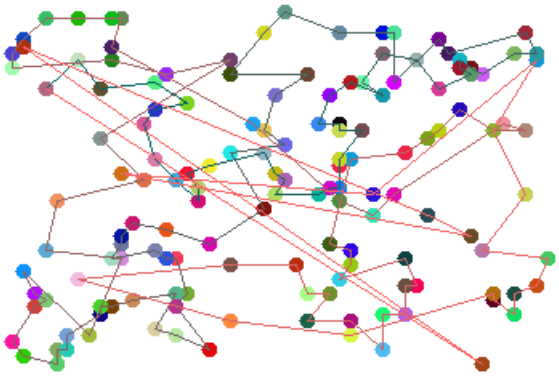
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75765 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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 10198941040598721679320140869951448678400000 000 trajetos.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CANDIDO ABREU        269 157
2-CASCAVEL              53 118
3-CURITIBA             467  63
4-GEN CARNEIRO         268 -49
5-GUAIRA               -25 213
6-PARANAGUA            543  51
7-UMUARAMA              72 246
```

Começando e terminando na cidade de CASCAVEL .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	2							2

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

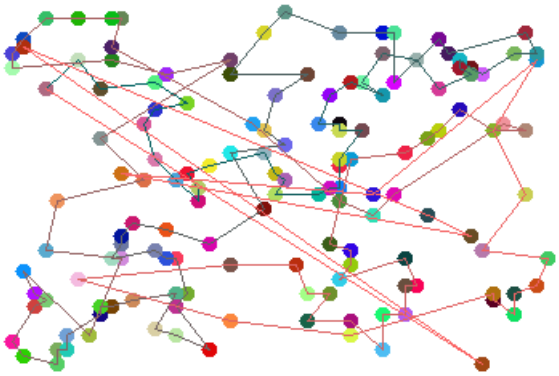
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75677 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CAMPO MOURAO          164 216
2-GEN CARNEIRO          268 -49
3-GUARAPUAVA            254  67
4-ORTIGUEIRA            310 202
5-PIRAI SUL             397 163
6-PONTA GROSSA          377 102
7-TELEM BORBA           341 185
```

Começando e terminando na cidade de PIRAI SUL .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	5							5

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

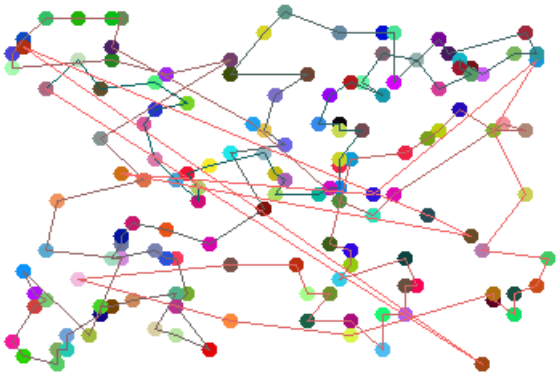
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75684 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS          495 146
2-CAMPO MOURAO          164 216
3-CANDIDO ABREU         269 157
4-GEN CARNEIRO          268 -49
5-IRATI                 338  62
6-LARANJ SUL            159  68
7-PALMAS                200 -54
```

Começando e terminando na cidade de CAMPO MOURAO .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	2							2

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

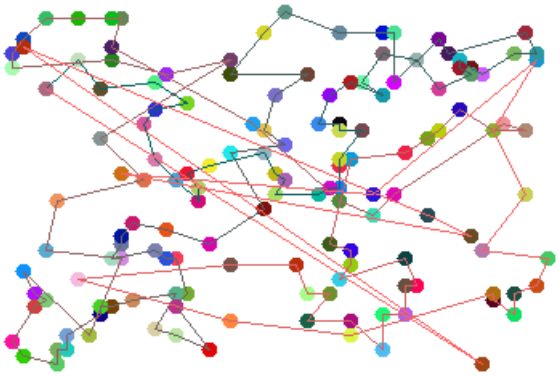
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75691 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CANDIDO ABREU        269 157
2-CASCAVEL              53 118
3-FOZ IGUACU           -59  55
4-GUARAPUAVA           254  67
5-JACAREZINHO          399 315
6-LARANJ SUL           159  68
7-PALMAS               200 -54
```

Começando e terminando na cidade de FOZ IGUACU .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	3							3

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

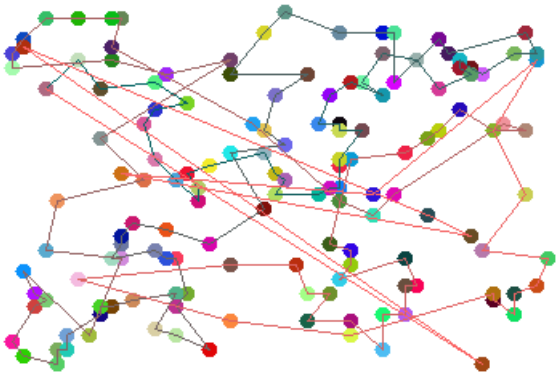
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75703 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS          495 146
2-FCO BELTRAO            91  -8
3-GUARAPUAVA            254  67
4-IRATI                 338  62
5-LARANJ SUL            159  68
6-LONDRINA              291 297
7-TELEM BORBA           341 185
```

Começando e terminando na cidade de GUARAPUAVA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	3							3

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

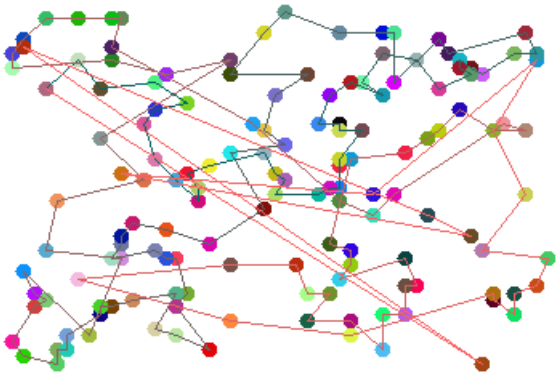
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75710 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-FOZ IGUACU          -59  55
2-GUARAPUAVA          254  67
3-LARANJ SUL          159  68
4-MARINGA             211 287
5-ORTIGUEIRA          310 202
6-PIRAI SUL           397 163
7-PONTA GROSSA        377 102
```

Começando e terminando na cidade de MARINGA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	4							4

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

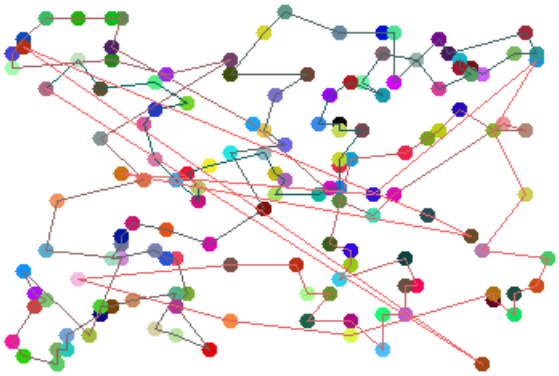
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75727 - /

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
39432893368239525177618160696609253114756798884358663164737126662217972498170167146015214200599231195208860606945981941512882139512131855253096331247641496555673142863538165861869849447196122281072583212012701664593206561371414742663876212120378695162016062870278978433011301595208516203117585042939808946111139481185194868736000000000000000000000000000000000000000000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-CURITIBA            467  63
3-GEN CARNEIRO        268 -49
4-GUAIRA              -25 213
5-IRATI               338  62
6-PARANAGUA           543  51
7-TELEM BORBA         341 185
```

Começando e terminando na cidade de CURITIBA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	2							2

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

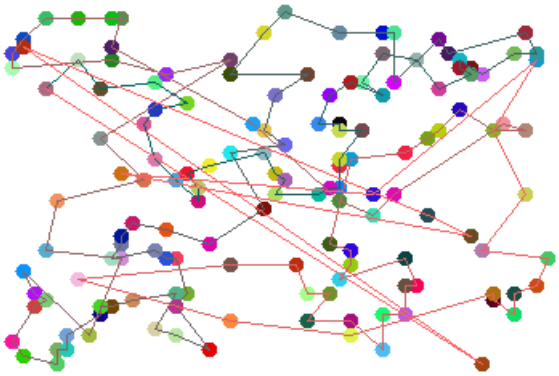
5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-CURITIBA            467  63
2-FCO BELTRAO          91  -8
3-LONDRINA            291 297
4-PALMAS              200 -54
5-PARANAGUA           543  51
6-PARANAVAI           158 323
7-TELEM BORBA         341 185
```

Começando e terminando na cidade de LONDRINA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	3							3

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

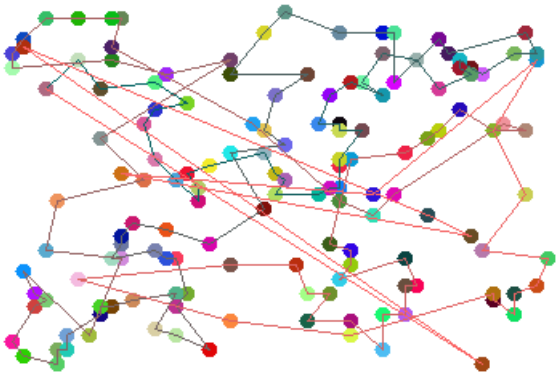
6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

CEP - UFPR - UP - UTFPR - PUCPr – 12/08/2024 - 11:58:54.3
Prof Dr P Kantek (pkantek@gmail.com)
Caixeiro Viajante VIVXq83a, V: 2.20 75758 REBECA CABRAL DOS SANTOS 24JOG402 - 21 _____ / _____ / _____

Caixeiro Viajante

Também é conhecido pelo seu nome em inglês: *Traveling salesman problem - TSP*. 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.

Um problema grande Só para ter uma idéia eis o tamanho do universo de rotas em um problema do caixeiro viajante com 200 cidades: (já dividido por 2)

```
394328933682395251776181606966092531147567988843586631647371266622179
724981701671460152142005992311952088606069459819415128821395121318552
530963312476414965556731428635381658618698494471961222810725832120127
016645932065613714147426638762121203786951620160628702789784330113015
952085162031175850429398089461111394811851948687360000000000000000000
000000000000000000000000000000000
```

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[:]
      alfa.remove(des)
      alja=lja[:]
      alja.append(des)
      akm=km+mat[ori][des]
      cv(alja,alfa,akm,mat)
  else:
    km=km+mat[lja[len(lja)-1]][lja[0]]
    lja.append(lja[0])
    if km<minimo:
      minimo=km
      caminho=lja
minimo=99999999
caminho=[]
cv([1],[0,2,3,4,5,6,7,8,9],0,mat)
print(minimo,' Km')
print('cam=',caminho)
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.

Embora tenham sido desenvolvidos bons algoritmos de aproximação para o PCV, o problema continua a oferecer uma grande atração para a aplicação de novos algoritmos. Isto deve-se, essencialmente, às seguintes razões:

- A problemática do PCV pode ser entendida facilmente, uma vez que se aproxima dos problemas populares do mundo real;
- O PCV demonstra o caso mais simples dos problemas de requisição que são de enorme relevância para a programação de processos industriais;
- Existem vários conjuntos de dados sobre o PCV *standard* que estão disponíveis em literatura, de tal forma que os resultados são comparáveis mesmo que o ótimo global não seja ainda definitivamente conhecido;
- Relativamente à complexidade computacional, o PCV, como um problema NP-completo, é conhecido por representar uma larga classe de problemas para os quais não existem algoritmos polinomiais em séries temporais determinísticos.

Métodos de construção de circuitos estratégia do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas;

uma outra estratégia é a da cobertura mínima, onde é elaborada um árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível;

Para além destes métodos, existem outros de melhorias baseados em metaheurísticas do tipo *simulated annealing* e busca tabu. Estes, para além de se basearem no desenvolvimento de estratégias consolidadas, procuram uma solução que não a dada pelos métodos anteriores. No *simulated annealing* é utilizado um controle de possibilidades de solução melhores partindo de piores, no início. Na busca tabu os movimentos considerados tabu, isto é, que não se podem efetuar, mesmo que melhorem a solução são temporariamente interditos com o objetivo de se alcançar soluções piores no início que no final poderão ser consideradas melhores.

Redes Neurais A rede neural é uma das mais promissoras estratégias da inteligência artificial na busca de resolvedores de problemas. Ela simula uma rede de neurônios e estes simulam o funcionamento de um neurônio animal, tal como descrito por Ramon y Cajal no início do século XX. Embora não tenhamos ideia de como funciona o raciocínio humano, sabemos muito sobre o funcionamento de um neurônio isolado. Colocando-os em rede, tem-se um arranjo capaz de aprender. Diferentemente de outros algoritmos, este exige ser treinado antes de começar a atuar. No caso do TSP, a ordem das cidades pode ser ensinada (segundo qualquer um dos algoritmos acima) e a rede treinada com diversas instâncias do problema. Depois de muitos ciclos a rede pode apresentar uma boa solução.

Algoritmos genéticos Esta é outra possibilidade de tentar resolver o TSP. Os algoritmos genéticos advém do trabalho pioneiro de John Holland (meados dos anos 70). Esta abordagem é excelente para problemas complexos e tem sido objeto de enorme pesquisa em todo o mundo. Este método tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema. Esta técnica simula o processo de evolução dos seres vivos demonstrada por Darwin. Da mesma forma que os sistemas biológicos, ao longo da sua evolução, tiveram que se "moldar" às alterações ambientais para a sua sobrevivência, os AGs acumulam a informação sobre o ambiente com o intuito de se adptarem

ao novo meio. Tal informação funciona como um sistema de triagem para a obtenção de novas soluções exequíveis. O método dos algoritmos genéticos é muito utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático ou onde o mesmo se torna impreciso em funções lineares e não-lineares.

colônia de formigas (Algoritmo ACO) O PCV tem um papel importante na optimização das colônias de formigas, *ant colony optimization (ACO)* (em inglês), desde o primeiro algoritmo ACO, chamado "Sistema de Formigas", do inglês Ant System, até aos mais recentes.

Nos algoritmos ACO, as formigas são simples agentes que, no caso do PCV, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construida pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona (uma informação numérica que é modificada durante o algoritmo) e após um número grande de iterações, a trilha a seguir fica delimitada pelos ferormônios.

Um caso real Em 1998 uma equipe de matemáticos encontrou o caminho mais curto para visitar as 13.509 cidades americanas que tinham, naquele ano, mais de 500 habitantes. Foram necessários 3,5 meses de processamento de três multiprocessadores (32 pentium cada) ligados em rede. Aqui, eliminaram-se as rotas obviamente ineficientes logo de cara. O problema: a estratégia só vale para este problema e para estas cidades.

Outro caso real Os algoritmos mostrados nesta folha e rodados em um micro bem lerdinho (um pentium 4 com 2.8GHz, com 632 MB de RAM, mas rodando Linux e neste sob WMware um Windows XP, e neste rodando APL2 da IBM) e em um micro rápido (2 CPUS de 2.80GHz, 3.3GB de memória e com APL2 sob Windows XP nativo)

qtd de cidades	CPU lerda	CPU rápida
6 cidades	31 milisseg	desprezível
7 cidades	156 milisseg	78 milisseg
8 cidades	1.1 seg	500 milisseg
9 cidades	9 seg	4 seg
10 cidades	81 seg	37 seg
11 cidades	836 seg	367 seg
12 cidades	153 min	67 min
13 cidades	30.6 horas	13.4 horas
14 cidades	16.5 dias	7.2 dias

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=["Campo Mourao","Curitiba","Fco Beltrao", "Guaira","Irati",
    "Londrina","Ortigueira","Palmas","Paranagua","Paranavai"]
  t=len(ci)
  dfinal=0
  inicio=1
  onde=[inicio]
  lx=[164,467,91,-25,338,291,310,200,543,158]
  ly=[216,63,-8,213,62,297,202,-54,51,323]
  dx=np.zeros((10,2))
  for i in range(10):
    dx[i,0]=lx[i]
    dx[i,1]=ly[i]
  di=np.zeros((t,t),float)
  for i in range(t):
    for j 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
  print(di)
  while min(di[qual,])!=99999:
    jj=np.argmin(di[qual,])
    dfinal=dfinal+min(di[qual,])
    for k in range(t):
      di[k,jj]=99999
    onde.append(jj)
    qual=jj
  dfinal=dfinal+dz[jj,inicio]
  onde.append(inicio)
  print(dfinal,onde)
gulosotsp()
```

☞ Para você fazer

Aplique o algoritmo guloso do caixeiro viajante para o seguinte problema.

```
1-ADRIANOPOLIS        495 146
2-GUAIRA              -25 213
3-JACAREZINHO         399 315
4-ORTIGUEIRA          310 202
5-PIRAI SUL           397 163
6-PONTA GROSSA        377 102
7-TELEM BORBA         341 185
```

Começando e terminando na cidade de GUAIRA .

Responda aqui: Informe a soma das distâncias obtidas no algoritmo guloso (deve-se visitar a cidade mais próxima ainda não visitada) e depois a sequência de cidades visitadas. Note que o primeiro e o último números devem ser o número da cidade citada como inicial, já que o circuito começa e termina nela. A distância deve ser calculada com 2 casas decimais.

$\sum$ distância	2							2

Como fazer

1. A primeira coisa é construir uma tabela de distâncias entre cidades. Desenhe em um papel uma matriz colocando na primeira linha e na primeira coluna a lista de cidades de seu exercício.

2. Anule a diagonal principal, já que nela as distâncias são 0.

3. Preencha as células acima da diagonal principal usando a fórmula do teorema de pitágoras:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

   onde i = linha da cidade e j = coluna da cidade.

4. Copie (já que a distância de $i \to j$ é igual à distância de $j \to i$) as distâncias para debaixo da diagonal principal nas posições equivalentes.

5. Agora, saindo da origem localize a cidade mais próxima, e vá para ela. Risque na tabela a linha e a coluna dessa cidade.

6. A partir da nova origem, refaça o passo anterior, até visitar todas as cidades.

7. Retorne à origem inicial e está cálculado o caminho do caixeiro viajante.

402-75758 - /