Universidade Federal de São Paulo

Instituto de Ciência e Tecnologia

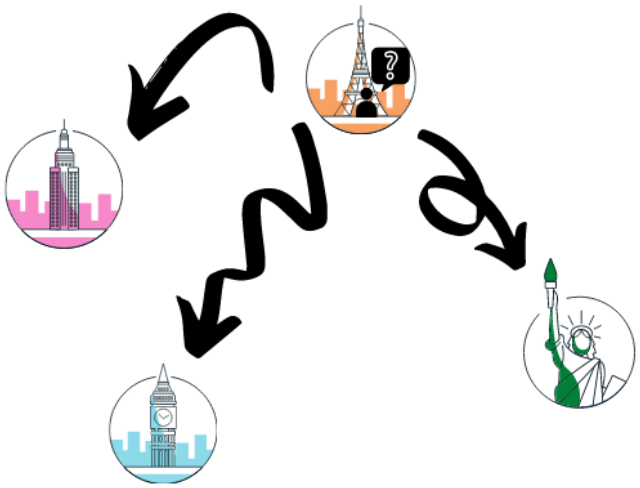
Relatório 2
Problema do Caixeiro Viajante - PCV

Beatriz Viana Ferreira 133548
Davi Juliano Ferreira Alves 133595

São José dos Campos
Janeiro, 2021

OBJETIVO

Devido aos avanços tecnológicos, a velocidade das informações ocorre de uma forma muito rápida e consequentemente cada minuto deve ser muito bem aproveitado. Deste modo, uma ferramenta que vem sendo utilizada para potencializar a economia, sejam financeiros, de materiais ou do tempo gasto é a modelagem matemática na descrição de problemas de otimização combinatória.



O *Problema do Caixeiro Viajante (PCV)*, em inglês chamado de *travelling salesman problem (TSP)*, é um dos mais notórios e tradicionais problemas de programação matemática, cujo o nome que usualmente se dá a uma série de problemas reais importantes que podem ser modelados em termos de ciclos Hamiltonianos em grafos completos.

Podemos utilizar o *Problema do Caixeiro Viajante* em várias aplicações do cotidiano, alguns exemplos são:

- Rotas de locomoção de veículos;
- Redes de distribuição postal;
- Programas de máquinas para realizar várias tarefas em sucessão;
- Construção de placas de circuitos integrados

Definição 1. *Um caminho hamiltoniano é um caminho que permite passar por todos os vértices de um grafo G, não repetindo nenhum, ou seja, passar por todos uma só vez por cada. Caso esse caminho seja possível descrever um ciclo, este é denominado ciclo hamiltoniano (ou circuito hamiltoniano) em G. E, um grafo que possua tal circuito é chamado de grafo hamiltoniano.*

MODELAGEM DO PROBLEMA: De acordo com (PARDINI, 2015), o modelo do Problema do Caixeiro Viajante pode ser dado por:

$$\begin{aligned}
\text{MAX: } & \sum_{i=1}^{n}\sum_{j=1}^{n}(D_{ij}X_{ij}) = Z. \\
\text{Sujeito à: } & \sum_{i=1}^{n}\sum_{j=1}^{n}X_{ij} \leq 1, \forall i \in \{1,...,n\} \\
& \sum_{i=1}^{n}\sum_{j=1}^{n}X_{ij} \leq 1, \forall j \in \{1,...,n\} \\
& \sum_{i,j\in K}X_{ij} \leq |K| - 1, \\
& x_{11},..., x_{nm} \in \{0,1\}.
\end{aligned}$$

Sendo D_{ij} a distância do vértice na rota (i,j), a variável X_{ij} é a variável binária que decide se a rota dada por (i,j) será utilizada ou não, e K é um subconjunto do conjunto de arcos do grafo, mas K exclui o grafo que foi utilizado. O $|K|$ é a quantidade de vértices do sub-grafo.

O PROBLEMA DO CAIXEIRO VIAJANTE (PCV)

IDEIA HEURÍSTICA:

1. Definir cidade origem;
2. Acessar/construir uma matriz $n \times n$, tal que a matriz possua as distâncias de cada cidade dois a dois;
3. Na matriz as colunas representam a cidade em que o caixeiro se encontra e as linhas para qual cidade irá;
4. Identificar o próximo destino, cuja distância seja a menor possível e que ainda não tenha sido visitada;
5. Cada cidade visitada receberá -1;
6. Caso todas as cidades tenham sido alcançados, voltará a cidade origem.

ANÁLISE DAS SOLUÇÕES

Para as instâncias, utilizamos bases de dados utilizando as matrizes de distâncias de cada vértice à cada vértice, como pode ser observado abaixo:

- https://people.sc.fsu.edu/~jburkardt/datasets/tsp/gr17_d.txt (Instância para um conjunto de 17 cidades).
- https://people.sc.fsu.edu/~jburkardt/datasets/tsp/fri26_d.txt (Instância para um conjunto de 26 cidades).
- https://people.sc.fsu.edu/~jburkardt/datasets/tsp/five_d.txt (Instância para um conjunto de 5 cidades).
- https://people.sc.fsu.edu/~jburkardt/datasets/tsp/dantzig42_d.txt (Instância para um conjunto de 42 cidades).
- https://people.sc.fsu.edu/~jburkardt/datasets/tsp/att48_d.txt (Instância para um conjunto de 48 cidades).

Foi extraída cada base de dados para um documento txt e foi tratado os separadores utilizando a normalização do separador (colocamos a string ' ' como separador em vez de vários espaços desordenados). Deixamos nessa pasta compartilhada https://drive.google.com/drive/folders/1XA9Kc_tnssS8-UlBTeVu4quW1c0bhOPk?usp=sharing no drive as tabelas de distâncias para melhor averiguação do código e do resultado da extração e tratamento dos dados. Tendo em vista a otimização das distâncias utilizando um algoritmo de busca dentro das distâncias, a heurística recebe uma complexidade computacional $O(n^2)$, sendo n a quantidade de iterações dentro de um laço de comando.

No código, para ler as instâncias, temos a utilização de dois vetores, um que lê linhas e outro que obtém os sub-vetores e os coloca em ordem na coluna. Como precisamos transformar cada número de string para float, a complexidade da atribuição ficou em $O(n^2)$, sendo n a quantidade de cidades existentes e a atribuição em cada laço de comando. Ou seja, temos:

```
valores = []
results=[]

with open('26 cidades.txt', 'r') as f:
    for line in f.readlines():
      vet = line.split('  ')
      results=[]
      valores.append(results)
```

```
9       for i in range(len(vet)):
10        vet[i]=int(float(vet[i]))
11        results.append(vet[i])
```

O próximo passo deve ser analisar a matriz de distâncias de fato. O ponto da heurística é partir sempre da origem x_0, passar pelos diversos pontos e depois voltar para x_0 utilizando as escolhas de distâncias. Sobre o código, devemos atentar que a análise é feita pelas colunas e a linha é gerada para as cidades que o caixeiro irá passar. As cidades já ocupadas receberão um marcador (No caso, é um valor -1), ou seja, ela não será visitada mais. Quando toda matriz for uma matriz de valores -1 e só restar uma linha de distâncias, o caixeiro voltará para a origem. Para essa atribuição, temos o código dado por:

```
tamanho = len(valores) #Tamanho do vetor de cada linha da
   matriz

"""
for l in range(tamanho): #Caso queira imprimir as matrizes a
   cada visita em uma nova cidade
   print(valores[l])
"""

substitui = [] #Matrizes que serão preenchidas com -1 (marcador
    de que a cidade já foi visitada);
for i in range(tamanho):
    substitui.append(-1)

linha_controle = 0 #Salvará a linha da cidade em que teremos
   que ir, pois está possuirá a menor distância;
c = 0 #Cidade origem, ou seja, neste caso estamos iniciando na
   cidade 0 (coluna 0);
saltos = 0 #Quantidade de cidades que precisaremos visitar,
   partindo da cidade origem, temos n-1 cidades inicialmente.
```

Agora, na parte mais importante da heurística, temos a viagem do caixeiro diante da matriz de distâncias buscando o menor caminho pela análise das colunas. Pelo código, temos:

```
ini1=time.time() #Inicialização do primeiro contador de tempo
while saltos < tamanho-1:
    menor = 100000
    for l in range(tamanho):
        if menor > valores[l][c] and valores[l][c] > 0: #
   Verificando qual é a menor distância na coluna (a cidade cuja
    a distância seja a menor possível)
```

```
            menor = valores[l][c] #salva a distância do percurso
            linha_controle = l #salva a linha, ou seja, a próxima cidade que será visitada
     caminho.append(c) #Adiciona a cidade visita
     valores[c] = substitui.copy() #Marca que a cidade já foi visitada
     c = linha_controle #A próxima cidade que será visitada será a nova coluna c
     saltos += 1 #atualiza o salto
     #print('A menor distância é:', menor)
     soma += menor #Armazenamos a distância percorrida
fim1=time.time() #Finalização do primeiro contador de tempo
tot1=fim1-ini1 #Atribuição do tempo total 1
```

Feita a viagem, precisa-se voltar para a cidade de origem. Para isso, analisaremos a linha restante da viagem (A linha que não possui o marcador -1). Portanto, no código, teremos:

```
ini2=time.time() #Inicialização do segundo contador de tempo
ultimo=100000
for l in range(tamanho):
  """
  Retornaremos a cidade origem, neste momento,n-1 linhas estarão preenchidas com o marcador -1,
  então terá apenas uma posição em que o valor será maior que 0, ou seja, exatamente
  o valor que precisaremos para encontrar a distância.
  Como sabemos que a cidade origem está na coluna 0, então basta percorre-lá até encontrar a distância.
  """
  if ultimo > int(valores[l][0]) and int(valores[l][0]) > 0:
    ultimo=int(valores[l][0])
    c=l

caminho.append(c)
#print('Distância para origem:',ultimo)
soma +=ultimo #Armazeno a distância da cidade em que o caixeiro enconra-se até a cidade origem
fim2=time.time() #Finalização do segundo contador de tempo
tot2=fim2-ini2 #Atribuição do tempo total 2
print(caminho)
print('Resultado final:', soma)
print(f'Tempo de execução: {tot1+tot2} segundos')
```

Temos como resultado o caminho do caixeiro, a soma das distâncias, ou seja, a função objetivo e o tempo de execução da heurística. O resultado também está no drive dado da atribuição das instâncias, que pode ser acessado em https://drive.google.com/drive/folders/1XA9Kc_tnssS8-UlBTeVu4quW1c0bhOPk?usp=sharing.

Já o **link para o código da heurística** é dado por https://colab.research.google.com/drive/1UxRfXb7imSKEvWuAgAohLRO3WGudedCp?usp=sharing, entretanto, se faz necessário possuir as instâncias dadas no drive.

RESULTADOS

- Para a primeira instância **(48 de cidades)**, possuímos o seguinte caminho:

$$Origem = 0 \to 8 \to 37 \to 30 \to 43 \to 17 \to 6 \to 27 \to 35 \to 29 \to 5 \to 36 \to 18 \to 26 \to 42 \to 16 \to 45 \to 32 \to 14 \to 11 \to 10 \to 22 \to 13 \to 24 \to 12 \to 20 \to 46 \to 19 \to 39 \to 2 \to 21 \to 15 \to 40 \to 33 \to 28 \to 4 \to 47 \to 38 \to 31 \to 23 \to 9 \to 41 \to 25 \to 3 \to 34 \to 44 \to 1 \to 7 \to 0 = Origem$$

  Resultado final: 40551

  Tempo de execução: 0.0013782978057861328 segundos

- Para a segunda instância **(42 de cidades)**, possuímos o seguinte caminho:

$$Origem = 0 \to 40 \to 41 \to 1 \to 39 \to 38 \to 37 \to 36 \to 34 \to 33 \to 30 \to 29 \to 31 \to 32 \to 28 \to 27 \to 26 \to 25 \to 24 \to 23 \to 9 \to 8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 35 \to 20 \to 21 \to 22 \to 16 \to 15 \to 17 \to 18 \to 19 \to 12 \to 13 \to 14 \to 11 \to 10 \to 0 = Origem$$

  Resultado final: 956

  Tempo de execução: 0.00135040283203125 segundos

- Para a terceira instância **(5 de cidades)**, possuímos o seguinte caminho:

$$Origem = 0 \to 3 \to 2 \to 1 \to 4 \to 0 = Origem$$

  Resultado final: 21

  Tempo de execução: 0.0007672309875488281 segundos

  Para a quarta instância **(26 de cidades)**, possuímos o seguinte caminho:

$$Origem = 0 \to 14 \to 13 \to 9 \to 10 \to 12 \to 11 \to 8 \to 6 \to 7 \to 15 \to 18 \to 19 \to 17 \to 16 \to 20 \to 21 \to 25 \to 22 \to 23 \to 24 \to 2 \to 1 \to 3 \to 4 \to 5 \to 0 = Origem$$

  Resultado final: 1112

  Tempo de execução: 0.0005943775177001953 segundos

- Para a quinta instância **(17 de cidades)**, possuímos o seguinte caminho:

$$Origem = 0 \to 12 \to 3 \to 6 \to 7 \to 5 \to 16 \to 13 \to 14 \to 2 \to 10 \to 4 \to 9 \to$$
$$1 \to 8 \to 11 \to 15 \to 0 = Origem$$

Resultado final: 2187

Tempo de execução: 0.0004601478576660156 segundos

CONCLUSÃO

A ideia heurística em alguns casos chegou muito próxima da solução ótima, como se pode ver na tabela abaixo:

Quantidade de cidades	48	42	5	26	17
Heurística	40551	956	21	1112	2187
Solução Ótima	33523	699	19	937	2085
Porcentagem (%)	82,66	73,11	90,47	84,26	95,33
Tempo de execução (s)	$1,37 \cdot 10^{-3}$	$1,35 \cdot 10^{-3}$	$7,67 \cdot 10^{-4}$	$5,94 \cdot 10^{-4}$	$4,60 \cdot 10^{-4}$

Sabendo que o *problema do caixeiro viajante* é **NP-Completo**, ou seja, não é resolveu em tempo polinomial, conseguimos uma solução, em alguns casos, boa em tempo quadrático.

Desta forma, para que que possamos ponderar o quão próximo as soluções encontradas estão perto da solução ótima, fizemos o quociente

$$\text{Índice de eficiência} = \frac{\text{Heurística}}{\text{Solução ótima}}$$

e obtivemos que a melhor solução foi para o caso de 17 cidades (com $95,33\%$) e a pior foi para o caso com 42 cidades (com $73,11\%$). Portanto, podemos concluir que a média dos índices de eficiência para esses casos foi de

$$\text{I}_{\text{med}} = 85,16\%$$

o que nos dá uma boa confiabilidade do código para resolver este problema. Em trabalhos futuros, podemos melhorar ainda mais este *índice médio*, variando a cidade origem e salvando os melhores resultados.

REFERÊNCIAS

- MORAIS, José Luiz Machado. Problema do Caixeiro Viajante Aplicado ao Roteamento de Veículos numa Malha Viária. 2010. 54 f. TCC (Graduação) - Curso de Ciência da Computação, Universidade Federal de São Paulo, São José dos Campos, 2010. Disponível em: https://www.ft.unicamp.br/docentes/meira/publicacoes/2010jose.pdf. Acesso em: 11 jan. 2021.
- PARDINI, Dhiego. O Problema do Caixeiro Viajante. 2015. Disponível em: https://otimizacaonapratica.com/2015/11/09/o-problema-do-caixeiro-viajante/. Acesso em: 11 jan. 2021.