

# Algoritmo Aproximativo para TSP

Mathaus C. Huber

<sup>1</sup>Universidade Federal de Pelotas (UFPEL) – Discente do Curso Superior de Ciência da Computação  
R. Gomes Carneiro, 1 - Centro – 96075-630 – Pelotas – RS – Brazil

mchuber@inf.ufpel.edu.br

**Abstract.** *This article describes the report of the practical evaluation of the discipline of Algorithms and Data Structure III, given to the fifth semester of the Computer Science course, at the Federal University of Pelotas, which refers to the solution of each instance of the traveling salesman problem using an approximate algorithm and an exact algorithm and discussion of the execution time and cost of the solution obtained.*

**Resumo.** *Este artigo descreve o relatório da avaliação prática da disciplina de Algoritmos e Estrutura de Dados III, conferida ao quinto semestre do curso de Ciência da computação, da Universidade Federal de Pelotas, no qual se refere a solução de cada instância do problema do Caixeiro Viajante utilizando um algoritmo aproximativo e um algoritmo exato e discussão sobre o tempo de execução e o custo da solução obtida.*

## 1. Informação Geral

O problema do Caixeiro viajante (TSP), do inglês Travelling Salesman Problem, é um problema de otimização NP-difícil inspirado na necessidade dos vendedores em realizar entregas em diversos locais, percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.

O TSP pode ser assim formulado: existem  $n$  pontos e caminhos entre todos eles com comprimentos conhecidos. O objetivo é encontrar o caminho mais curto, que passa por todos os pontos uma única vez, começando e terminando no mesmo ponto. Isso significa que o objetivo é encontrar a viagem de ida e volta mais curta possível.

Neste relatório iremos comparar algoritmos aproximativos e exatos para resolver as instâncias do caixeiro viajante

## 2. Algoritmo Exato

A solução para esse algoritmo exato seria um ordenamento dos vértices, que indica em que ordem estes devem ser visitados, e o custo total deste ordenamento vai ser dado pela soma dos valores das arestas entre vértices consecutivos. Sabemos que, o TSP consiste em encontrar um circuito Hamiltoniano de custo mínimo em um grafo completo. Portanto, utilizaremos de uma estratégia chamada de força bruta para obter o algoritmo exato. A ideia consiste em:

- Fazer uma lista de todos os circuitos possíveis.
- Calcular todos os circuitos possíveis.
- Calcular o peso de cada um deles
- Selecionar o menor deles

**Ele é ótimo?** Sim. Ele encontra a melhor solução

**Ele é eficiente?** De jeito nenhum... Se o número de entradas for muito grande poderá exceder o tempo de execução.

## 2.1. Exemplos de Teste

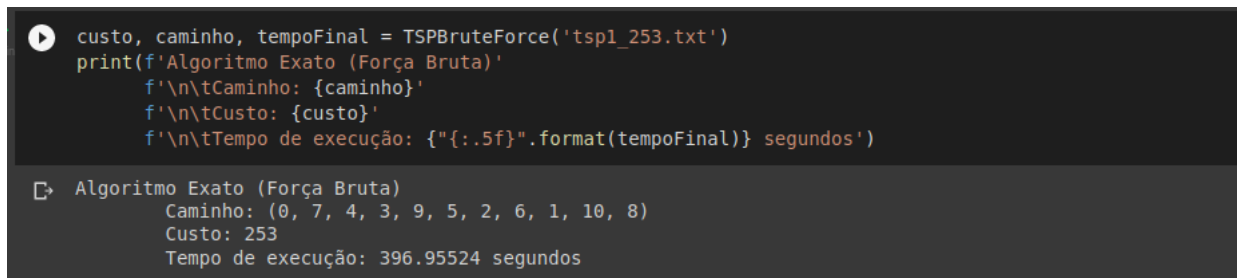
### 2.1.1. Teste 1 - tsp1\_253.txt

Na compilação do teste 1 teremos:

tempo = 0:06:27.301834

custo = 253 (ótimo)

caminho = [0, 7, 4, 3, 9, 5, 2, 6, 1, 10, 8]



```
custo, caminho, tempoFinal = TSPBruteForce('tsp1_253.txt')
print(f'Algoritmo Exato (Força Bruta)'
      f'\n\tCaminho: {caminho}'
      f'\n\tCusto: {custo}'
      f'\n\tTempo de execução: "{:0.5f}".format(tempoFinal)} segundos')

Algoritmo Exato (Força Bruta)
Caminho: (0, 7, 4, 3, 9, 5, 2, 6, 1, 10, 8)
Custo: 253
Tempo de execução: 396.95524 segundos
```

Figura 1. Teste tsp1\_253.txt Algoritmo Exato

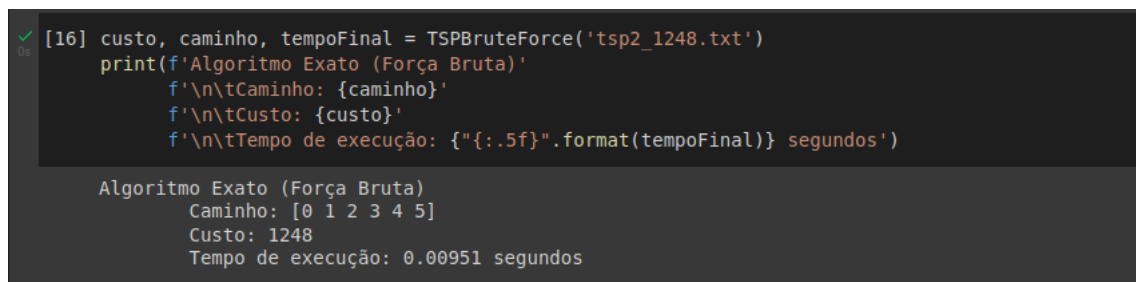
### 2.1.2. Teste 2 - tsp2\_1248.txt

Na compilação do teste 2 teremos:

tempo = 0:00:00.012280

custo = 1248 (ótimo)

caminho = [0, 1, 2, 3, 4, 5]



```
[16] custo, caminho, tempoFinal = TSPBruteForce('tsp2_1248.txt')
print(f'Algoritmo Exato (Força Bruta)'
      f'\n\tCaminho: {caminho}'
      f'\n\tCusto: {custo}'
      f'\n\tTempo de execução: "{:0.5f}".format(tempoFinal)} segundos')

Algoritmo Exato (Força Bruta)
Caminho: [0 1 2 3 4 5]
Custo: 1248
Tempo de execução: 0.00951 segundos
```

Figura 2. Teste tsp2\_1248.txt Algoritmo Exato

### **2.1.3. Teste 3 - tsp3\_1194.txt**

Na compilação do teste 3 teremos:

tempo = null

custo = null (acreditamos que seja ótimo)

caminho = null

Rodei por mais de uma hora e interrompi a compilação.

### **2.1.4. Teste 4 - tsp4\_7013.txt**

Na compilação do teste 4 teremos:

tempo = null

custo = null (acreditamos que seja ótimo)

caminho = null

Rodei por mais de uma hora e interrompi a compilação.

### **2.1.5. Teste 5 - tsp5\_27603.txt**

Na compilação do teste 5 teremos:

tempo = null

custo = null (acreditamos que seja ótimo)

caminho = null

Rodei por mais de uma hora e interrompi a compilação.

## **2.2. Dissertação sobre os testes do Algoritmo exato**

A maioria dos testes não foi possível rodar em tempo plausível, apenas nos dois primeiros (tsp1\_253.txt e tsp2\_1248.txt) o nosso algoritmo de força bruta nos deu a solução esperada, a exata. O máximo de tempo que deixei rodando o algoritmo no Colab foram aproximadamente 60 minutos, contudo, o notebook está disponível nesse relatório, e como o tempo de execução difere entre máquinas, seria interessante testar em outra com mais capacidade de processamento.

Como já era esperado, o nosso algoritmo ingênuo, depois de um longo tempo, para entradas específicas, encontrou a solução exata para o problema do Caixeiro Viajante. Partindo desse princípio, eu consideraria ele um bom algoritmo, pois encontrou a solução. Contudo, não consideraria um algoritmo eficiente, pois para entradas maiores ele pode exceder o tempo de execução.

## **3. Algoritmo aproximativo**

Existem heurísticas para resolver o problema do caixeiro viajante, que são métodos por meio dos quais a solução ótima para o TSP é procurada. Embora as heurísticas não possam garantir que a solução ótima seja encontrada, ou não ao menos

em tempo real, um grande número delas encontrará uma solução próxima à ótima, ou mesmo encontrará uma solução ótima para certos casos do problema do caixeiro viajante. Utilizaremos uma heurística construtiva para a solução do TSP utilizando um algoritmo de Christofides.

### 3.1. Algoritmo de Christofides

Em 1976 Nicos Christofides desenvolveu um algoritmo eficiente que produz um caminho cujo custo excede o ótimo em menos de 50%. Portanto, a heurística de Christofides permite determinar uma solução aceitável para o Problema do Caixeiro Viajante, ainda que possa não ser a solução ótima. O algoritmo segue os seguintes passos:

- Criar um árvore geradora mínima  $T$  de  $G$ .
- Seja  $I$  o conjunto de vértices de grau ímpar em  $T$ . Pelo lema do aperto de mão,  $I$  tem um número par de vértices.
- Encontrar um acoplamento perfeito de peso mínimo  $M$  no subgrafo induzido pelos vértices de  $I$ .
- Combinar as arestas de  $M$  e  $T$  para formar um multigrafo  $H$  em que cada vértice tem grau par.
- Formar um circuito Euleriano em  $H$ .
- Transformar o circuito encontrado na etapa anterior em um circuito Hamiltoniano, ignorando vértices repetidos.

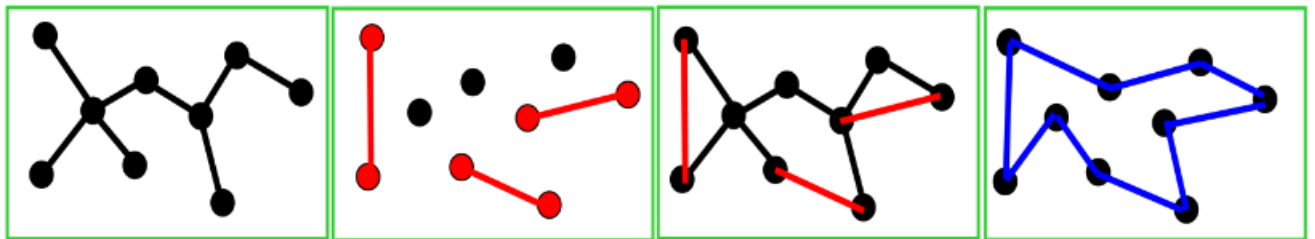


Figura 3. Christofides

### 3.2. Exemplos de Teste

#### 3.2.1. Teste 1 - tsp1\_253.txt

Na compilação do teste 1 teremos:

tempo = 0:00:00.01167

custo = 265

caminho = [1, 9, 8, 3, 7, 2, 11, 5, 4, 6, 10, 1]

#### 3.2.2. Teste 2 - tsp2\_1248.txt

Na compilação do teste 2 teremos:

tempo = 0:00:00.00987

custo = 1272

caminho = [1, 2, 6, 5, 4, 3, 1]

```
[ ] custo, caminho, tempoFinal = christofides('tsp1_253')
print(f'Christofides (Algoritmo Aproximativo)'
      f'\n\tCaminho: {caminho}'
      f'\n\tCusto: {custo}'
      f'\n\tTempo de execução: "{:.5f}".format(tempoFinal)} segundos')

Christofides (Algoritmo Aproximativo)
Caminho: [1, 9, 8, 3, 7, 2, 11, 5, 4, 6, 10, 1]
Custo: 265
Tempo de execução: 0.01807 segundos
```

**Figura 4. Teste tsp1\_253.txt Algoritmo Aproximativo**

```
[ ] custo, caminho, tempoFinal = christofides('tsp2_1248')
print(f'Christofides (Algoritmo Aproximativo)'
      f'\n\tCaminho: {caminho}'
      f'\n\tCusto: {custo}'
      f'\n\tTempo de execução: "{:.5f}".format(tempoFinal)} segundos')

Christofides (Algoritmo Aproximativo)
Caminho: [1, 2, 6, 5, 4, 3, 1]
Custo: 1272
Tempo de execução: 0.01164 segundos
```

**Figura 5. Teste tsp2.1248.txt Algoritmo Aproximativo**

### 3.2.3. Teste 3 - tsp3\_1194.txt

Na compilação do teste 3 teremos:

tempo = 0:00:00.01006

custo = 1360

caminho = [1, 2, 10, 11, 12, 13, 14, 15, 9, 8, 6, 7, 5, 4, 3, 1]

```
[ ] custo, caminho, tempoFinal = christofides('tsp3_1194')
print(f'Christofides (Algoritmo Aproximativo)'
      f'\n\tCaminho: {caminho}'
      f'\n\tCusto: {custo}'
      f'\n\tTempo de execução: "{:.5f}".format(tempoFinal)} segundos')

Christofides (Algoritmo Aproximativo)
Caminho: [1, 2, 10, 11, 12, 13, 14, 15, 9, 8, 6, 7, 5, 4, 3, 1]
Custo: 1360
Tempo de execução: 0.01510 segundos
```

**Figura 6. Teste tsp3.1194.txt Algoritmo Aproximativo**

### 3.2.4. Teste 4 - tsp4\_7013.txt

Na compilação do teste 4 teremos:

tempo = null

custo = null

caminho = null

Rodei por um tempo e o colab indicou que tinha consumido toda a RAM.

A sessão falhou depois de usar toda a RAM disponível. Se você tiver interesse em acessar ambientes de execução com mais memória RAM, confira o [Colab Pro](#).

[Ver registros do ambiente de execução](#)

Figura 7. Teste tsp4\_7013.txt Algoritmo Aproximativo

### 3.2.5. Teste 5 - tsp5\_27603.txt

Na compilação do teste 5 teremos:

tempo = 0:00:00.45811

custo = 38110

caminho = [1, 11, 10, 12, 13, 14, 17, 18, 22, 23, 29, 28, 26, 20, 16, 25, 27, 24, 21, 19, 15, 8, 4, 2, 6, 5, 7, 9, 3, 1]

```
custo, caminho, tempoFinal = christofides('tsp5_27603')
print(f'Christofides (Algoritmo Aproximativo)')
f'\n\tCaminho: {caminho}'
f'\n\tCusto: {custo}'
f'\n\tTempo de execução: {"{:.5f}".format(tempoFinal)} segundos')

Christofides (Algoritmo Aproximativo)
Caminho: [1, 11, 10, 12, 13, 14, 17, 18, 22, 23, 29, 28, 26, 20, 16, 25, 27, 24, 21, 19, 15, 8, 4, 2, 6, 5, 7, 9, 3, 1]
Custo: 38110
Tempo de execução: 0.52984 segundos
```

Figura 8. Teste tsp5\_27603.txt Algoritmo Aproximativo

### 3.3. Dissertação sobre os testes do Algoritmo Aproximativo

Em todos os testes do nosso algoritmo aproximativo, a solução dada se aproxima bastante da ótima, e diferente do último método visto, feito por uma solução força bruta, onde o nosso algoritmo nos dava uma solução exata, o tempo de execução é imensamente inferior, o que nos dá uma vantagem gigantesca. Também deu pra notar que com entradas muito maiores o custo já difere um pouco da solução ótima, o que também era esperado, a exemplo disso temos o teste 5, onde a solução ótima seria dada por um custo de 27603 e o nosso algoritmo aproximativo nos deu um custo de 38110.

A única exceção nos nossos testes foi o teste 4 "tsp4\_7013", onde o colab nos reportou um erro de uso total da capacidade da RAM. Acredito que essa exceção nos dá por causa da versão gratuita do colab, e para fins de comparação seria interessante rodar em uma máquina ou SO, ao invés de rodar pelo colab.

## 4. Conclusão

Considerei um problema muito interessante, já tinha tido um pré-conhecimento sobre o problema de otimização do TSP, e particularmente achei um trabalho muito estimulante de ser desenvolvido, visto que, o desenvolvimento de algoritmos de aproximação é uma das linhas de pesquisa que mais têm crescido na área de otimização combinatória e teoria da computação.

Notamos que para o problema do TSP em dimensões elevadas, a resolução por métodos de força bruta é proibitiva em termos de tempos computacionais. Portanto, a abordagem de solução heurística e de aproximação é mais útil para as aplicações que preferem o tempo de execução do algoritmo em relação à precisão do resultado.

Para a grande maioria dos problemas de otimização interessantes não se conhecem algoritmos "exatos"eficientes, este também é o caso do problema do Caixeiro Viajante. Por isso para esse problema em específico os algoritmos de aproximação são muito úteis.

Conclui também que, o nosso algoritmo aproximativo (Christofides), nos dá uma eficiência maior em termos de tempo computacional e a precisão do resultado não difere tanto em pequenas e médias dimensões, por isso escolheria esse método de heurística se estivesse desenvolvendo uma solução para o problema, ao invés da ótima.

O Link para o código fonte dos algoritmos e testes está disponível no Colab, pelo link:

**[https://colab.research.google.com/drive/1i6lUe0AX1ONM3ZE9anYJ2oMw9p\\_F\\_elO?usp=sharing](https://colab.research.google.com/drive/1i6lUe0AX1ONM3ZE9anYJ2oMw9p_F_elO?usp=sharing)**