# Trabalho Prático 2
# Soluções para Problemas Difíceis

**Mateus Krause**[1]

[1]Universidade Federal de Minas Gerais

mateuskrause@ufmg.br

***Abstract.*** *This document shows the description and analysis of an implementation of three algorithms for the Geometric Traveler Salesman Problem (TSP), two of them using aproximating algorithms - Twice Around the Tree and Christofides - and one exact solution using the Branch and Bound technique.*

***Resumo.*** *Este documento apresenta a descrição e análise da implementação de três algoritmos para o Problema do Caixeiro Viajante Geométrico (TSP), duas baseadas em algoritmos aproximados - Twice Around the Tree e Christofides - além de uma solução exata utilizando a técnica de Branch and Bound.*

## 1. Introdução

De uma maneira geral, o Problema do Caixeiro Viajante (TSP, abreviação para *Traveler Salesman Problem*) pode ser definido como encontrar a menor rota em um conjunto de $n$ cidades tal que cada uma seja visitada apenas uma vez, tendo como fim o retorno da jovem pessoa à cidade de partida. Podemos modelar esse problema como sendo um grafo ponderado com valores positivos, onde os vértices são as cidades e arestas as estradas, e o objetivo de um algoritmo sendo encontrar o menor caminho Hamiltoniano deste grafo.

Este é um dos problemas mais conhecidos na literatura pela sua característica de ser *NP-difícil*. Não há algoritmo de tempo polinomial conhecido até então capaz de solucionar o problema, e até existem conjecturas não provadas de que jamais existirá [Levitin 2012]. Entretanto, temos técnicas de resolver algumas instâncias desse problema, além de abordagens não exatas que nos retornam soluções aproximadas com certa proporção de erro, porém em um tempo realístico.

Neste trabalho, iremos considerar o problema de forma que cada cidade possua coordenadas em um espaço bidimensional $(x, y)$, além de existir uma aresta entre qualquer par de cidades, com seu custo sendo calculado como a simples distância euclidiana entre elas. Serão apresentados testes realizados em *datasets* públicos disponíveis na biblioteca TSPLIB.

## 2. Implementação

Os algoritmos foram implementados com Python3 (testados na versão 3.12 porém compatíveis com +3.9) utilizando, além das biliotecas padrão, *numpy*, *networkx* e *pandas*.

### 2.1. Branch and Bound

A ideia dessa abordagem é usar uma árvore de possibilidades para evitar cálculo de caminhos que seriam descartados por já existir outro mais promissor. Em um pior caso, seria o mesmo que testar todas as possibilidades com complexidade exponencial, mas quando isso não acontece diminui os gastos do algoritmo. Note que não se trata de uma aproximação, chega à solução ótima com atalhos, mas ainda não se trata de algo polinomial.

Partimos de uma estrutura Grafo da biblioteca *networkx*. Começando do vértice inicial, vemos as possibilidades de caminho e calculamos um custo, abordagem que é repetida ao longo das iterações por meio do cálculo de limite inferior. Usamos como estimativa desse *bound* o custo mínimo de sair do vértice atual para outro não visitado de forma que seja mínimo, além de um outro custo também mínimo para completar um ciclo no caminho parcial. Isto é, vemos qual caminho será mais eficiente dadas as configurações atuais do nó.

De maneira geral, removemos caminhos que levariam a distâncias maiores que as já vistas. Na exploração, utilizamos uma estrutura de fila de prioridade, ordenando em relação aos seus limites inferiores. Com isso, por uma abordagem *best-first*, priorizamos caminhos mais promissores. Continuamos a análise até não haver mais nós a serem visitados retornando assim o custo total.

Essa técnica pode ajudar a resolver algumas instâncias desse problema, porém, não podemos prever se determinado exemplo será calculado em um tempo realístico, ainda mais considerando a quantidade de nós a serem computados.

### 2.2. Twice Around the Tree

Esta abordagem é uma versão aproximativa para o TSP com distâncias euclidianas que utiliza de poucos passos para obter uma solução. Consideramos que ele busca o caminho mais curto dado determinado vértice do grafo e com isso completa o caminho.

Primeiro é calculada a árvore geradora mínima dado o grafo que representa o problema especifico; Começando do vértice inicial, computamos um caminho utilizando uma abordagem de DFS; por fim, adicionamos a volta e retornamos o caminho seguido do respectivo custo. Assim como a estrutura de grafo não direcionado da biblioteca *networkx*, utilizamos os métodos incluídos para o cálculo da árvore geradora mínima e computar o DFS. A complexidade do algoritmo é limitada pela própria do DFS.

Temos que Twice Around the Tree é uma abordagem com fator de aproximação 2, isto é, nossa solução (o custo do circuito) estará no máximo o dobro do resultado ótimo [Levitin 2012]. Uma das principais vantagens são os poucos passos, podendo ser resolvido em tempo polinomial.

### 2.3. Christofides

Podemos obter um algoritmo com fator de aproximação melhor que o visto anteriormente, este é o algoritmo de Christofides. De forma semelhante, utiliza uma árvore geradora mínima, porém com mais passos intermediários para obter uma solução mais próxima da ideal.

Utilizando da estrutura Grafo da biblioteca *networkx*, além de métodos incluídos como o para cálculo da árvore geradora mínima (*mst*), DFS e emparelhamento de peso mínimo, é feito o seguinte: Dado o grafo entrada do problema, é criado uma árvore geradora mínima e então removidas as arestas pares; a seguir encontramos o *matching* perfeito deste novo grafo; criamos um multigrafo com as arestas da nossa *mst* com as do emparelhamento; assim, calculamos o circuito euleriano utilizando DFS e ao fim removemos arestas repetidas e adicionamos a volta.

Este algoritmo possui fator de aproximação $3/2$, o que significa que as suas respostas serão no máximo 1.5 vezes a ótima. Também é um algoritmo que pode ser resolvido em tempo polinomial, mas por possuir mais passos pode ser mais demorado que o visto anteriormente, por exemplo.

## 3. Experimentos e Resultados

Após a implementação dos algoritmos, foram realizados diversos experimentos com os *datasets* disponibilizados, além de um caso simples criado. Para isso, foi preparado um algoritmo que carrega cada *dataset*, cria seu grafo e aplica cada estratégia, computando métricas como tempo de execução e memória utilizada para cada caso.

Como os exemplos apresentados chegam a um número de nós muito grandes, os experimentos foram limitados a tempo e recursos computacionais. Deixam então de serem mencionados casos em que foi necessário mais de 30 minutos para sua execução, estes marcados como $N/A$. Além disso, em algumas quantidades de nós não foi possível computar soluções devido ao limite de memória.

### 3.1. Algoritmos Aproximativos

Na Tabela 1 e Tabela 2, vemos a comparação dos algoritmos Twice Around the Tree e Christofides. A qualidade dos nossos resultados, definida pela razão entre resultado obtido e resultado ótimo, mostra que os algoritmos respeitam o limite máximo de erro apresentado anteriormente (o que pode ser demonstrado).

Como Twice Around the Tree possui menos passos em relação a Christofides, é perceptível sua diferença no tempo de execução, em que se mostrou muito mais eficiente mesmo com resultados muito próximos. Em relação à memoria, vemos que compartilham a necessária para a criação do grafo, mas devido aos passos extrar, o algoritmo de Christofides acaba consumindo mais.

Não puderam ser computados *datasets* com um número maior de nós por falta de recursos computacionais. Criar o grafo correspondente consome uma quantidade considerável de memória, e por isso não foi possível computar soluções a partir de 10000 nós, por exemplo. Especificamente para o algoritmo de Christofides, a partir dos 3000 nós o tempo de execução foi maior que 30 minutos.

Para problemas em que é possível ter uma margem de erro maior, o algoritmo Twice Around the Tree se mostra uma ótima escolha, já que consegue computar casos com rapidez mesmo não sendo os melhores mas com grande chances de serem bons o suficiente. Mas caso seja necessário garantia de um resultado até 1.5 vezes maior que o ótimo, é interessante avaliar o uso do algoritmo de Christofides.

| Dataset | Nós | Christofides | | | Twice Around the Tree | | |
|---|---|---|---|---|---|---|---|
| | | Qualidade | Tempo (s) | Mem. (MB) | Qualidade | Tempo (s) | Mem. (MB) |
| graph | 6 | 1 | 0 | 0.09 | 1 | 0 | 0.02 |
| eil51 | 51 | 1.31 | 0.07 | 0.86 | 1.5 | 0.02 | 0.45 |
| st70 | 70 | 1.31 | 0.14 | 1.37 | 1.29 | 0.03 | 0.79 |
| eil76 | 76 | 1.28 | 0.18 | 1.54 | 1.32 | 0.04 | 0.93 |
| pr76 | 76 | 1.2 | 0.1 | 1.53 | 1.34 | 0.04 | 0.93 |
| rat99 | 99 | 1.3 | 0.22 | 2.9 | 1.42 | 0.06 | 1.83 |
| kroA100 | 100 | 1.42 | 0.27 | 2.97 | 1.28 | 0.06 | 1.87 |
| kroB100 | 100 | 1.22 | 0.28 | 2.98 | 1.17 | 0.06 | 1.87 |
| kroC100 | 100 | 1.34 | 0.26 | 2.98 | 1.35 | 0.07 | 1.87 |
| kroD100 | 100 | 1.33 | 0.26 | 2.97 | 1.27 | 0.07 | 1.87 |
| kroE100 | 100 | 1.37 | 0.34 | 2.97 | 1.38 | 0.06 | 1.87 |
| rd100 | 100 | 1.41 | 0.39 | 2.98 | 1.36 | 0.06 | 1.87 |
| eil101 | 101 | 1.31 | 0.47 | 3.03 | 1.32 | 0.07 | 2.05 |
| lin105 | 105 | 1.41 | 0.26 | 3.2 | 1.36 | 0.07 | 2.05 |
| pr107 | 107 | 1.3 | 0.21 | 3.32 | 1.22 | 0.07 | 2.12 |
| pr124 | 124 | 1.35 | 0.23 | 4.22 | 1.26 | 0.1 | 2.79 |
| bier127 | 127 | 1.19 | 0.88 | 4.32 | 1.34 | 0.1 | 3.05 |
| ch130 | 130 | 1.29 | 0.39 | 4.58 | 1.33 | 0.11 | 3.04 |
| pr136 | 136 | 1.31 | 0.22 | 4.93 | 1.57 | 0.12 | 3.31 |
| pr144 | 144 | 1.14 | 0.3 | 5.43 | 1.38 | 0.14 | 3.68 |
| ch150 | 150 | 1.29 | 0.49 | 5.83 | 1.28 | 0.15 | 3.98 |
| kroA150 | 150 | 1.42 | 1.21 | 5.73 | 1.32 | 0.15 | 4.12 |
| kroB150 | 150 | 1.35 | 0.94 | 5.84 | 1.38 | 0.14 | 3.98 |
| pr152 | 152 | 1.31 | 0.33 | 5.87 | 1.19 | 0.15 | 4.21 |
| u159 | 159 | 1.32 | 0.71 | 6.43 | 1.37 | 0.16 | 4.44 |
| rat195 | 195 | 1.25 | 1.41 | 10.93 | 1.43 | 0.25 | 7.39 |
| d198 | 198 | 1.26 | 1.56 | 11.22 | 1.22 | 0.26 | 7.61 |
| kroA200 | 200 | 1.42 | 2.47 | 11.29 | 1.36 | 0.26 | 7.88 |
| kroB200 | 200 | 1.33 | 1.92 | 11.29 | 1.38 | 0.27 | 7.88 |
| ts225 | 225 | 1.33 | 0.68 | 13.75 | 1.48 | 0.33 | 9.7 |
| tsp225 | 225 | 1.41 | 1.89 | 13.75 | 1.31 | 0.33 | 9.7 |
| pr226 | 226 | 1.5 | 1.47 | 13.96 | 1.45 | 0.33 | 9.63 |
| gil262 | 262 | 1.38 | 3.99 | 17.94 | 1.41 | 0.55 | 12.57 |
| pr264 | 264 | 1.44 | 1.42 | 18.05 | 1.35 | 0.45 | 12.93 |
| a280 | 280 | 1.41 | 3.2 | 19.96 | 1.41 | 0.57 | 14.36 |
| pr299 | 299 | 1.34 | 4.23 | 22.37 | 1.34 | 0.67 | 16.17 |

**Tabela 1. Comparação entre algoritmos aproximativos para TSP**

| **Dataset** | **Nós** | **Christofides** | | | **Twice Around the Tree** | | |
|---|---|---|---|---|---|---|---|
| | | Qualidade | Tempo (s) | Mem. (MB) | Qualidade | Tempo (s) | Mem. (MB) |
| lin318 | 318 | 1.4 | 5.54 | 24.92 | 1.38 | 0.75 | 18.1 |
| linhp318 | 318 | 1.42 | 5.49 | 25.06 | 1.41 | 0.8 | 17.96 |
| rd400 | 400 | 1.36 | 15.91 | 44.58 | 1.33 | 1.19 | 31.24 |
| fl417 | 417 | 1.53 | 7.07 | 47.78 | 1.37 | 1.34 | 33.52 |
| pr439 | 439 | 1.34 | 7.49 | 52.1 | 1.35 | 1.52 | 36.77 |
| pcb442 | 442 | 1.35 | 12.69 | 52.7 | 1.37 | 1.54 | 37.2 |
| d493 | 493 | 1.33 | 20.21 | 63.61 | 1.3 | 1.9 | 45.12 |
| u574 | 574 | 1.35 | 32.01 | 82.7 | 1.33 | 2.62 | 59.53 |
| rat575 | 575 | 1.38 | 36.99 | 82.82 | 1.39 | 2.63 | 59.85 |
| p654 | 654 | 1.46 | 8.2 | 103.98 | 1.44 | 3.38 | 75.59 |
| d657 | 657 | 1.31 | 47.16 | 104.79 | 1.34 | 3.27 | 76.22 |
| u724 | 724 | 1.39 | 61.73 | 150.14 | 1.38 | 4.24 | 104.13 |
| rat783 | 783 | 1.39 | 91.02 | 170.98 | 1.37 | 4.76 | 119.42 |
| pr1002 | 1002 | 1.36 | 136.91 | 259.7 | 1.32 | 7.74 | 185.42 |
| u1060 | 1060 | 1.33 | 177.6 | 286.22 | 1.35 | 8.54 | 205.33 |
| vm1084 | 1084 | 1.39 | 96.43 | 297.42 | 1.32 | 9.14 | 214.34 |
| pcb1173 | 1173 | 1.44 | 169.93 | 341.47 | 1.42 | 10.61 | 247.14 |
| d1291 | 1291 | 1.35 | 53.66 | 404.23 | 1.47 | 12.83 | 294.22 |
| rl1304 | 1304 | 1.45 | 53.62 | 411.5 | 1.37 | 13.27 | 300.38 |
| rl1323 | 1323 | 1.37 | 55.09 | 422.18 | 1.41 | 13.62 | 308.29 |
| nrw1379 | 1379 | 1.4 | 583.38 | 551.68 | 1.4 | 15.41 | 381.78 |
| fl1400 | 1400 | 1.62 | 536.25 | 565.52 | 1.39 | 15.6 | 391.72 |
| u1432 | 1432 | 1.36 | 184.09 | 587.02 | 1.4 | 16.02 | 407.11 |
| fl1577 | 1577 | 1.39 | 125.95 | 689.15 | 1.41 | 19.37 | 482.62 |
| d1655 | 1655 | 1.36 | 234.18 | 747.25 | 1.38 | 21.65 | 526.15 |
| vm1748 | 1748 | 1.41 | 343.27 | 819.67 | 1.32 | 24.14 | 580.07 |
| u1817 | 1817 | 1.38 | 265.39 | 875.41 | 1.46 | 26.17 | 620.59 |
| rl1889 | 1889 | 1.38 | 156.98 | 935.52 | 1.42 | 28.84 | 666.02 |
| d2103 | 2103 | 1.17 | 63.11 | 1125.52 | 1.56 | 34.77 | 808.89 |
| u2152 | 2152 | 1.38 | 385.55 | 1171.45 | 1.48 | 36.61 | 842.71 |
| u2319 | 2319 | 1.35 | 1351.74 | 1334.73 | 1.37 | 43.09 | 964.96 |
| pr2392 | 2392 | 1.4 | 1271.92 | 1409.36 | 1.38 | 46.13 | 1022.6 |
| pcb3038 | 3038 | N/A | N/A | N/A | 1.43 | 75.18 | 1804.9 |
| fl3795 | 3795 | N/A | N/A | N/A | 1.36 | 117.39 | 2684.08 |
| fnl4461 | 4461 | N/A | N/A | N/A | 1.39 | 171.98 | 3600.69 |
| rl5915 | 5915 | N/A | N/A | N/A | 1.49 | 304.8 | 6876.48 |
| rl5934 | 5934 | N/A | N/A | N/A | 1.49 | 314.15 | 6914.46 |

**Tabela 2. Comparação entre algoritmos aproximativos para TSP**

### 3.2. Branch and Bound

Por se tratar de um problema combinatório, não há muitas abordagens que solucionam este problema difícil. Vemos pela Tabela 3 que só foi possível computar o caso mais simples de forma exata, com 6 nós. Todos os *datasets* apresentados possuem uma quantidade de nós muito elevada deixando assim o tempo de execução extremamente alto.

Temos que por ser uma forma mais inteligente de força bruta, é preferível sua utilização no lugar de outro modo que testa todas as possibilidades. Ainda sim deve ser utilizado em casos muito específicos, onde precisa se saber com certeza o resultado ótimo. Quanto maior o tamanho da entrada, mais chances de ter um tempo total de execução inviável.

| **Dataset** | **Nós** | **Branch and Bound** | | |
|---|---|---|---|---|
| | | Qualidade | Tempo (s) | Mem. (MB) |
| graph | 6 | 1 | 0.01 | 0.04 |

**Tabela 3. Resultados de experimentos com a técnica de Branch and Bound**

## 4. Conclusões

Após a implementação de alguns algoritmos para o Problema do Caixeiro Viajante, é possível perceber como é complicado lidar com problemas difíceis. O custo computacional para uma solução perfeita em grande parte das vezes não justifica a sua busca. Heurísticas se tornam então uma ferramenta muito poderosa para obter soluções com um nível muito bom de erro, e com isso a busca por algoritmos aproximativos cada vez mais eficientes obtém foco.

Nos testes realizados, abordagens como Twice Around the Tree se mostraram possuir um ótimo custo-benefício, já que conseguem obter uma solução com um grau bom de erro mesmo sendo computadas rapidamente. No entanto, isto vale para as configurações específicas tratadas neste trabalho, já que em algumas aplicações podem existir características próprias para cada cenário.

Ainda vale lembrar que, mesmo não se mostrando eficientes nos casos de testes apresentados, formas de otimizar soluções ótimas como Branch and Bound podem poupar grande tempo de processamento com suas técnicas para cálculos mais eficientes.

## Referências

Levitin, A. (2012). *Introduction to The Design and Analysis of Algorithms*. Pearson, 3ed edition.

Vimieiro, R. (2023). Slides virtuais da disciplina algoritmos 2.