Uma Avaliação Abrangente dos Algoritmos Branch-and-Bound, Twice-Around-the-Tree e Christofides para o Problema do Caixeiro Viajante

Erik Roberto Reis Neves, Gabriel Campos Prudente

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

Abstract. *This paper investigates the feasibility of the branch-and-bound, twice-around-the-tree, and Christofides algorithms for solving the Traveling Salesman Problem (TSP), comparing their effectiveness in terms of execution time, memory usage, and solution quality. The analysis includes the implementation of the algorithms and an experimental evaluation of their performance on different problem instances, highlighting the advantages and limitations of each technique in solving computationally challenging problems.*

Resumo. *Este trabalho investiga a viabilidade dos algoritmos branch-and-bound, twice-around-the-tree e Christofides para resolver o problema do Caixeiro Viajante (TSP), comparando sua eficácia em termos de tempo de execução, uso de memória e qualidade das soluções. A análise inclui a implementação dos algoritmos e uma avaliação experimental de seu desempenho em diferentes instâncias do problema, destacando as vantagens e limitações de cada técnica na resolução de problemas computacionalmente difíceis.*

1. Introdução

Problemas presentes na vida cotidiana, como os que envolvem decisões complexas ou otimização, frequentemente apresentam grandes desafios computacionais, principalmente aqueles em que achar a melhor solução possível envolve potencialmente testar todas as possíveis soluções. Tais problemas são classificados como NP-Difíceis, uma categoria que engloba aqueles para os quais não existem algoritmos polinomiais determinísticos que garantam a obtenção de uma solução ótima em todas as instâncias possíveis.

A resolução eficiente de problemas NP-Difíceis motivou o desenvolvimento de duas abordagens principais. A primeira envolve algoritmos exatos, que buscam explorar todo o conjunto de soluções possíveis, priorizando as mais promissoras para alcançar a solução ótima, embora com alto custo computacional. A segunda abordagem, mais comum em situações práticas, são os algoritmos aproximativos, que visam gerar soluções próximas da ótima, mas com um custo computacional significativamente reduzido. Essas técnicas representam a essência da pesquisa em otimização e são cruciais para a aplicação em diversas áreas, desde a logística até a inteligência artificial.

Um exemplo clássico e amplamente estudado dessa classe é o problema do Caixeiro Viajante (TSP), que será abordado na Seção 2. Para esse problema, serão implementados três algoritmos, um com solução exata e dois com soluções aproximadas. Além disso, será realizada uma análise experimental visando comparar os resultados das diferentes implementações frente a diversas instâncias do TSP.

2. Problema do Caixeiro Viajante

O problema do Caixeiro Viajante (do inglês, *Traveling Salesman Problem* – TSP) é um dos exemplos mais emblemáticos de problemas NP-Difíceis. Ele consiste em determinar o trajeto de menor custo que percorra todos pontos, visitando cada local apenas uma vez e, por fim, retornando a origem. Este custo pode representar distâncias, tempos ou outros critérios que variam conforme o cenário de aplicação do problema.

Um exemplo prático é a entrega de encomendas por uma agência onde deve-se visitar n cidades para realizar todas as entregas, passando por cada uma apenas uma vez. O objetivo é encontrar a rota de menor custo que garanta que todas as encomendas sejam entregues.

Para análise, o TSP é frequentemente modelado como um grafo não direcionado, em que os pontos são vértices e as conexões, arestas ponderadas pelos custos associados. Apesar de sua simplicidade aparente, não há algoritmo conhecido que resolva o TSP de forma exata em tempo polinomial, destacando a importância de abordagens que priorizem soluções aproximadas ou heurísticas que reduzam o conjunto de possíveis soluções. Além disso, para atender as especificações dos algoritmos, a análise será baseada no TSP euclidiano, onde as distâncias entre os vértices respeitam a *desigualdade triangular*.

3. Algoritmos

Nesta seção, serão apresentados três algoritmos amplamente utilizados no estudo do TSP: *Branch and Bound*, *Twice Around The Tree* e *Christofides*. Cada um desses métodos possui características próprias que os tornam adequados a diferentes contextos e requisitos, variando desde estratégias exatas, que exploram o espaço completo de soluções, até técnicas aproximativas, que equilibram precisão e desempenho.

3.1. Branch And Bound

Dentre os algoritmos análisados, o Branch and Bound que calcula a solução exata. Para isso, em teoria, ele deve explorar todas as soluções possíveis, abordagem chamada de *ingênua* ou *força bruta*. Porém, pequenas mudanças o tornam mais eficientes, sendo elas:

(a) **Descarte de solução inválida**: Essa abordagem explora a própria característica do problema, que é visitar os nós do grafo apenas uma vez, logo, possíveis soluções que voltam para um nó já visitado são descartadas.

```
for i in range(self.numNodes):
    if i in currentSolution:
        continue
```

***BranchAndBound.py* - Verificação de viabilidade da solução atual**

(b) **Descarte de solução que será futuramente pior que a atual**: O algoritmo simula, para cada solução parcial um $Bound$ dela. O $Bound$ é um limite usado para restringir a busca de soluções. Para isso, ele faz uma estimativa da melhor escolha possível, a partir da solução atual, mesmo que inválida. Se mesmo assim, o $Bound$ calculado - o melhor resultado obtenível, considerando opções inválidas - for maior que uma solução viável atual, então toda a solução pode ser descartada.

```
bound = self.__bound(newSolution)
if bound < self.cost:
    heapq.heappush(queue, (bound, newSolution, newCost))
else:
    self.prunes += 1
```

***BranchAndBound.py* - Verificação de bound *versus* custo da melhor solução**

O conjunto das possíveis soluções pode ser modelado como uma árvore, em que cada nó representa uma solução parcial e nós folhas são soluções completas. Os filhos de cada nó correspondem a todos os possíveis caminhos que a solução parcial atual pode seguir. Em (a), reduzimos a complexidade para fatorial, visto temos $n!$ possíveis soluções. Em (b), as soluções infrutíferas são cortadas. Desse modo, sempre que um nó é cortado, todas as possíveis ramificações desse nó são descartadas, visto que todas elas irão conter, ou uma solução inválida, ou uma solução pior (ou igual) à solução atual. Esse processo de descarte de solução é chamado de **poda**.

3.1.1. Calculo do Bound

O cálculo do bound é realizado escolhendo as duas menores arestas de menor custo ligadas a cada vértice, salvo se alguma outra aresta tiver sido escolhida anteriormente. No caso em que uma já foi escolhida, escolhe-se a aresta de menor custo, enquanto, para o caso em que ambas arestas já tiverem sido escolhidas, nenhuma aresta nova é selecionada. Após isso, é calculado a soma de todos os custos (de arestas selecionadas) e dividido por 2, visto que estamos incluindo cada aresta duas vezes. O resultado obtido é o bound.

Algorithm 1: Bound

Input: Solução parcial s (nó atual)
Output: Bound da solução parcial (do nó atual)

1 **1. Crie uma lista vazia de custos mínimos C.**
2 **2. Encontre os custos das arestas usadas e os armazene na lista de custos mínimos C.**
3 **3. Se para o vértice i, há duas arestas usadas:**
4 Ignore essa linha da matriz de adjacência.
5 **4. Se para o vértice i, há uma aresta usada:**
6 Encontre a aresta de custo mínimo, excluindo a aresta usada. Inclua ela em C.
7 **5. Se para o vértice i não há aresta usada**
8 Encontre duas arestas de custo mínimo. Inclua elas em C
9 **6. Calcule a soma da lista de custos mínimos C e divida por 2.**
10 Retorne o bound

Note que, como sempre que possível, são escolhidas as arestas de custo mínimo no trajeto, é provável de resultar em um trajeto que retorne a um vértice já visitado, ou seja, uma solução inválida. O $bound$ é, portanto, uma estimativa do melhor que se pode obter a partir da solução parcial, mesmo se for inválido. Além disso, o bound não é calculado para quando a solução parcial possui $n - 1$ elementos, uma vez que, nesse estado, basta apenas ir para a única opção possível viável, que não viole (a). Desse modo, é calculado uma possível solução.

3.1.2. Best-First-Search

No algoritmo, usa-se uma Pilha de prioridade, que armazena soluções parciais e seus bounds. Desse modo, por decisão de projeto, foi escolhido explorar a soluções por meio da pilha de prioridades: primeiro expandindo todos os filhos de um nó, avaliando possíveis soluções parciais e seus bounds e inserindo esses dados na pilha. Após isso, escolhe-se a solução parcial na pilha com o menor bound e continua o processo. A escolha por essa implementação é aumentar a chance de encontrar a solução ótima rapidamente, visto que, com a pilha, soluções promissoras são verificadas primeiro e com isso, possivelmente, podar mais ramos do espaço de busca cedo. Porém, isso é uma heurística, uma decisão do projeto, que não garante uma maior rapidez para achar a solução.

3.1.3. Algoritmo completo

A seguir, está a explicação do algoritmo:

Algorithm 2: Branch And Bound

Input: Grafo G com distâncias euclidianas entre os nós
Output: Valor da solução ótima exata

1 **1. Inicie o algoritmo com um nó arbitrário inicial, de custo 0**
2 **2. Calcule o bound para essa solução parcial e armazene na pilha**
3 **3. Repita: Enquanto a pilha estiver cheia, faça:**
4 Retire da pilha o primeiro valor. Ele é a solução parcial atual
5 **4. Se a solução atual possui tamanho $n-1$:**
6 Calcule o custo dessa solução completa.
7 **5. Se ele for menor que o melhor custo atual:** Atualize o custo atual
8 **6. Se não:** Prossiga
9 **7. Repita para $i = 0$ até $i =$ número de nós**
10 Se i está na solução atual: itere novamente
11 **8. Calcule o bound da solução atual e verifique se ele é pior que a melhor solução:**
12 Se sim: descarte essa solução
13 Se não: Coloque a solução na pilha

A implementação atual visa podar o máximo de nós possíveis, descartando, desse modo, soluções inviáveis e piores. Apesar disso, o algoritmo ainda apresenta complexidade fatorial, que é o custo para achar a melhor solução, visto que, no pior caso, o algoritmo irá explorar todas as possíveis soluções.

A observação acima demonstra que, apesar de todas as otimizações, ele ainda continua sendo exponencial, o que requer muito tempo para achar uma solução exata. Portanto, é possível afirmar que a complexidade de tempo do Branch and Bound, mesmo com todas as podas é fatorial: $O(n!)$, para o n o número de pontos. Isso reforça o que foi falado anteriormente: **não há solução conhecida que resolva o TSP com exatidão em tempo polinomial, salvo se** $P = NP$. Esse é o maior desafio para os problemas NP-Completos, visto que as soluções exatas demoram um tempo extensivamente grande.

Outro fator a ser destacado é o gasto excessivo de memória computacional do TSP,

visto que o input dele é um grafo completo que requer memória $O(n^2)$ para o tamanho da entrada. Além disso, dentro da própria execução do algoritmo, há muitas possíveis soluções a serem armazenadas na pilha, o que pode gerar um estouro de memória.

3.2. Twice Around The Tree

O algoritmo Twice Around The Tree acha uma solução aproximada para o TSP com um fator de aproximação 2, isto é, ele é um algoritmo 2-aproximado. Desse modo, a solução é no máximo duas vezes pior que a solução ótima.

Para encontrar essa solução apoximada, o código segue duas etapas:

(a) **Encontrar a arvore geradora mínima (MST) do grafo atual**: A arvore geradora mínima é encontra por meio do Algoritmo de Prim, gerenciado pela biblioteca *network*

```
return nx.minimum_spanning_tree(self.graph)
```

***TwiceAroundTheTree.py* - Construção da MST**

(b) **Achar o caminho hamiltoniano a partir de um caminhamento em *pré-ordem* na MST**: Ao fazer um caminhamento em *pré-ordem* na MST, visita-se primeiro os todos os filhos à esquerda e depois todos à direita. Como a arvore geradora mínima não possui ciclos, ao chegar em uma folha, ela para o pai, e continua visitando os outros filhos. Se grafo original é completo, para sair do vértice A e ir para o B, não é necessário passar por nós intermediários (*desigualgade triângular*). Ou seja, constrói-se um ciclo a medida que os vértices são visitados primeiro nesse caminhamento pré-ordem na MST, ignorando os vértices que são visitados mais de uma vez. Após isso, soma-se o custo desse caminho, que será no máximo duas vezes pior que o ótimo:

```
path = list(nx.dfs_preorder_nodes(mst_prim, source=0))
h_cycle = path + [path[0]]
solution_aprox = 0
for i in range(len(h_cycle) - 1):
    [...]
    if self.graph.has_edge(u, v): solution_aprox += self.graph[u][v]['weight']
```

***TwiceAroundTheTree.py* - Construção do ciclo hamiltoniano e o custo dele**

A seguir, apresenta-se o trecho principal do código implementado:

Algorithm 3: Twice Around the Tree

Input: Grafo G com distâncias euclidianas entre os nós – Matriz de adjacências

Output: Valor da solução aproximada

1 **1. Construir a Árvore Geradora Mínima (MST):**
2 Utilize o algoritmo de Prim para gerar a MST de G.
3 **2. Fazer uma visita pré-ordem na MST gerada**
4 Visite todos os filhos a esquerda, depois todos as direita. Guarde os vértices visitados na ordem em que são visitados
5 **3. Construa o Circuito Hamiltoniano a partir da lista anterior**
6 Realize atalhos para evitar repetição de vértices.
7 **4. Calcule o valor do caminho construído**
8 Retorne esse valor.

O algoritmo anterior, embora ache uma solução aproximada com fator 2 pode não ser o mais adequado, visto que existem algoritmos com fator de aproximação melhores, como o próximo a ser analisado.

Porém, o que faz desse algoritmo útil, mesmo com um fator de aproximação pior que outros, é a sua complexidade menor que a maioria deles, sendo $O(|E|log|V|$ a complexidade dominante no código - o tempo necessário para construir a MST. Como o grafo é completo, o número de arestas $E \simeq V^2$. Logo a complexidade total do Twice Around the Tree é $O(|V|^2 log|V|)$.

Como notado no Branch and Bound, essa implementação também requer um espaço consideravelmente grande. Começando pela construção do grafo, que vai demandar uma matriz quadrática. Não só isso, a construção da MST e do caminho hamiltoniano tem custo linear no tamanho da entrada, sendo esses fatores acima, possíveis gargalos de memória, principalmente a construção do grafo.

3.3. Christofides

O algoritmo de Christofides foi implementado com base em três etapas principais: (i) construção da Árvore Geradora Mínima (MST), (ii) cálculo do Emparelhamento Perfeito Mínimo para os vértices de grau ímpar e (iii) conversão do circuito Euleriano em um circuito Hamiltoniano. Para a construção da MST, optou-se pelo algoritmo de Prim devido à sua eficiência $O(E + V \log V)$, o que o torna particularmente adequado para grafos densos, com muitos vértices. Essa escolha assegura uma construção rápida da árvore, que é fundamental para o desempenho do algoritmo como um todo. O emparelhamento perfeito, que conecta os vértices de grau ímpar na MST, foi realizado de forma eficiente, minimizando o custo adicional.

O fator de aproximação do algoritmo de Christofides garante que a distância da solução encontrada seja no máximo 1.5 vezes maior que a distância da solução ótima. Essa propriedade faz do algoritmo uma das heurísticas mais eficazes para o problema do Caixeiro Viajante, especialmente para instâncias de grande escala, onde métodos exatos enfrentam dificuldades de desempenho. A vantagem do fator de aproximação é que, mesmo em casos de grande porte, a solução obtida é de alta qualidade, estando dentro de uma margem de erro relativamente pequena em relação à solução ótima.

A análise de complexidade do algoritmo de Christofides leva em conta as operações de construção da MST, emparelhamento e conversão do circuito. A construção da MST utilizando Prim tem uma complexidade de $O(E + V \log V)$, enquanto o cálculo do emparelhamento perfeito mínimo é feito por meio do algoritmo de Edmonds-Blossom, com custo $O(V^3)$. A conversão do circuito Euleriano em Hamiltoniano tem complexidade linear em relação ao número de vértices, ou seja, $O(V)$. Portanto, a complexidade total do algoritmo é dominada pelo cálculo do emparelhamento perfeito, resultando em uma complexidade assintótica de $O(V^3)$, o que é consideravelmente mais eficiente do que abordagens exatas para o TSP.

Em termos de estrutura de dados, a implementação utilizou a biblioteca `NetworkX` para representar o grafo e realizar operações como o cálculo da MST e a manipulação do circuito Euleriano. `NetworkX` é uma ferramenta poderosa para análise e manipulação de grafos, o que facilita a implementação e torna o código mais conciso.

A conversão do circuito Euleriano para Hamiltoniano foi realizada com um algoritmo de atalho eficiente, que evita a visita repetida a vértices, mantendo a solução válida.

Além disso, a implementação inclui uma estimativa do espaço de memória necessário para armazenar o grafo e seus componentes. Essa estimativa considera a quantidade de nós, arestas e a representação interna do grafo, fornecendo uma visão abrangente dos requisitos computacionais do algoritmo.

A seguir, apresenta-se os passos implementados pelo algoritmo de Christofides:

Algorithm 4: Algoritmo de Christofides

Input: Grafo G com distâncias euclidianas entre os nós – Matriz de adjacências

Output: Valor da solução aproximada

1 **1. Construir a Árvore Geradora Mínima (MST):**
2 Utilize o algoritmo de Prim para gerar a MST de G.
3 **2. Construir o Emparelhamento Perfeito Mínimo:**
4 Crie um emparelhamento perfeito para os vértices de grau ímpar.
5 **3. Gerar o Grafo Euleriano:**
6 Adicione as arestas do emparelhamento à MST para gerar um grafo Euleriano.
7 **4. Encontrar o Circuito Euleriano:**
8 Execute o algoritmo de circuito Euleriano para obter um caminho que passe por todas as arestas de forma contínua.
9 **5. Converter para Circuito Hamiltoniano:**
10 Construa o circuito Hamiltoniano a partir do circuito Euleriano, realizando atalhos para evitar repetição de vértices.
11 **6. Calcular o Preço da Solução:**
12 Utilize o cálculo do preço baseado nas distâncias euclidianas entre os nós visitados no circuito Hamiltoniano.

Essa implementação oferece um equilíbrio entre clareza e eficiência, utilizando algoritmos determinísticos para aproximar a solução ótima do TSP em tempo polinomial. Mesmo sendo consideravelmente mais lento que o *Twice Around The Tree*, o *Christofides* mostra-se fundamental nos casos onde é necessário uma solução de mais qualidade.

4. Experimentação

Para avaliar o desempenho dos algoritmos, serão executadas diversas instâncias do problema do Caixeiro Viajante (TSP). Essa diversidade de instâncias busca evidenciar os limites de funcionamento de cada implementação, destacando tanto seus pontos fortes quanto suas limitações. Além disso, será realizada uma comparação abrangente entre os algoritmos, considerando métricas como tempo de execução, consumo de memória e qualidade das soluções obtidas. Essa análise conjunta permitirá identificar as condições em que cada abordagem se mostra mais eficiente e eficaz. Por fim, um limite de 30 minutos foi estipulado para a execução de cada algoritmo para cada instância do problema que será avaliada.

4.1. Origem dos dados

As instâncias utilizadas para testar os algoritmos foram extraídas da biblioteca TSPLIB[1], uma coleção amplamente reconhecida de problemas padrão para o estudo do TSP. Para este trabalho, foram selecionadas apenas as instâncias cujo cálculo de custo é baseado na distância euclidiana em duas dimensões. Além das informações quanto a estrutura do grafo, essa coleção contém as soluções para cada problema, possibilitando uma análise da qualidade das resposta dos algoritmos.

4.2. Desempenho dos algoritmos

O desempenho dos algoritmos foi avaliado considerando três métricas principais: tempo de execução, consumo de memória e qualidade das soluções obtidas. Para cada instância, o tempo de execução foi medido em segundos, enquanto o consumo de memória foi monitorado para identificar a viabilidade em diferentes contextos computacionais. A qualidade das soluções foi avaliada comparando o custo obtido com o custo ótimo fornecido pela TSPLIB.

Os testes foram realizados em um ambiente controlado e as instâncias foram agrupadas por tamanho e densidade para evidenciar padrões de desempenho. Algoritmos exatos, como o *Branch and Bound*, apresentaram maior tempo de execução em instâncias grandes, enquanto heurísticas como *Twice Around the Tree* e *Christofides* demonstraram maior escalabilidade, mantendo boa proximidade em relação às soluções ótimas.

4.2.1. Ambiente de testes

Os testes foram realizados em uma máquina com as seguintes especificações: processador Intel Core i7-14700K, 128 GB de memória RAM, SSD de 512 GB e sistema operacional Linux Ubuntu 22.04 LTS. Essa configuração garante recursos computacionais adequados para avaliar o desempenho dos algoritmos, minimizando impactos de limitações de hardware. Além disso, todos os experimentos foram executados em ambiente controlado, com processos secundários minimizados para evitar interferências nos resultados.

4.2.2. Branch And Bound

Durante os testes do algoritmo *Branch and Bound* para o TSP, observou-se que todos os casos ultrapassaram o limite de tempo de 30 minutos, o que impediu uma avaliação detalhada da eficácia do algoritmo. Esse resultado destaca uma limitação crítica da abordagem exata em problemas de grande escala, onde a complexidade do algoritmo se torna um obstáculo significativo.

Embora esse algoritmo seja eficaz em podar soluções inviáveis, ele ainda enfrenta a complexidade exponencial do TSP, onde o número de possibilidades cresce rapidamente **(O(n!))** com o aumento do número de nós. Mesmo com técnicas de otimização, como a poda de caminhos não promissores, o tempo necessário para explorar todas as soluções possíveis em instâncias maiores excede o limite de tempo estipulado, levando a um *time-out*. Isso evidencia a dificuldade de aplicar métodos exatos em problemas NP-completos,

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95

onde a escalabilidade e a rapidez são essenciais, tornando essa abordagem inadequada para instâncias de maior porte.

4.2.3. Twice Around The Tree

Nesta seção, será avaliado o algoritmo Twice Around de Tree (TATT) com base em sua complexidade, isto é, desempenho computacional, e em seu desempenho como algoritmo aproximativo, ou seja, o quão perto o algoritmo consegue chegar da solução ótima.

1. **Complexidade**: A complexidade computacional de tempo do algoritmo é dominada pela construção da Árvore Geradora Mínima, sendo $O(V^2)$. Logo, o algoritmo se apresenta como uma boa solução para respostas rápidas, mesmo que relativamente imprecisas. Esses fatos serão elucidados em seções posteriores, onde será apresentados gráficos que tomam a forma de uma função quadrática. A complexidade espacial, por sua vez, é dominada pela construção do grafo, que é, quadrática - $O(V^2)$, uma vez que o grafo é representado por uma matriz de. Os demais processos tem custo linear.
2. **Desempenho aproximativo**: O algoritmo possui fator de aproximação de no máximo 2. Logo, ele se mostra viável, pois possui uma aproximação relativamente boa, mesmo que ainda existam outras implementações com fatores de aproximação menores. Por meio de análise dos resultados obtidos, foi observado que o algoritmo possui um **desvio médio relação à solução ótima de 38.36%**. Essa estimativa é relativamente satisfatória, embora isso seja apenas a média de todos os resultados. A Figura 1 apresenta o desvio percentual de todas as soluções geradas para suas respectivas soluções ótimas:

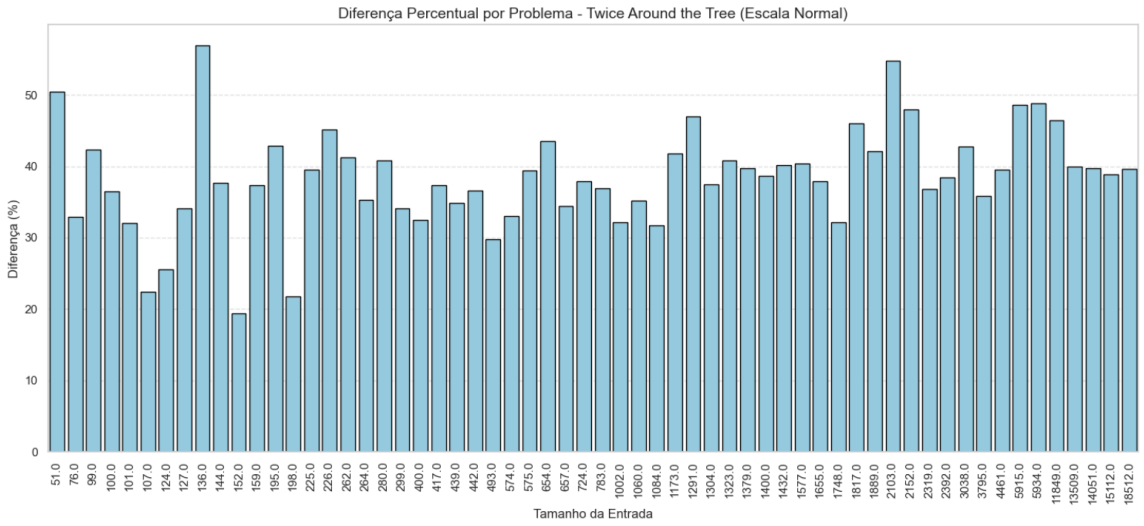

Figure 1. Twice Around the Tree: Diferença entre a solução do algoritmo e a solução ótima

Em geral, os desvios percentuais se concentram na faixa dos 40%, o que indica que o desvio médio reflete a realidade do desempenho do algoritmo. Além disso a organização do grafo pode ser impactante, como é possível ver na instância com 152 vértices, que tem uma solução aproximada melhor que outras instâncias menores. Portanto, o algoritmo

apresenta rapidez para achar a solução aproximada boa, por conta de seu baixo custo, tanto para pequenas instâncias, quanto para grandes.

4.2.4. Christofides

Nesta seção, analisamos o algoritmo de Christofides, abordando sua complexidade computacional e desempenho aproximativo. Avaliamos sua eficiência em diferentes instâncias do problema do caixeiro-viajante (TSP), destacando suas vantagens e limitações para cenários de tamanhos variados.

1. **Complexidade**: O algoritmo de Christofides apresenta complexidade polinomial, o que o torna eficiente para instâncias de tamanho pequeno a médio. No entanto, devido ao custo das etapas de construção de emparelhamentos perfeitos mínimos e árvores geradoras mínimas, o crescimento da complexidade é significativo à medida que o número de nós aumenta. Logo, a complexidade computacional de tempo do algoritmo é dominada pela tarefa de encontrar o emparelhamento mínimo perfeito, sendo $O(V^3)$. Essa característica impacta diretamente a escalabilidade do algoritmo para instâncias maiores, onde tanto o tempo de execução quanto o consumo de memória se tornam fatores limitantes.
2. **Desempenho aproximativo**: O desempenho aproximativo do algoritmo de Christofides foi avaliado por sua capacidade de fornecer soluções com custo de até 1.5 vezes o ótimo. Essa garantia teórica foi corroborada pelos experimentos, onde ele frequentemente produziu resultados próximos aos ótimos.
   Conforme ilustrado na Figura 2, a diferença entre o custo da solução gerada pelo algoritmo e o ótimo foi mantida dentro do limite teórico de 50%. Em problemas com até 4800 nós, o algoritmo apresentou bom desempenho, equilibrando qualidade da solução e eficiência. Contudo, em instâncias maiores, o tempo de execução e o uso de memória cresceram de forma significativa, apontando a necessidade de heurísticas ou melhorias adicionais para lidar com problemas de maior escala.

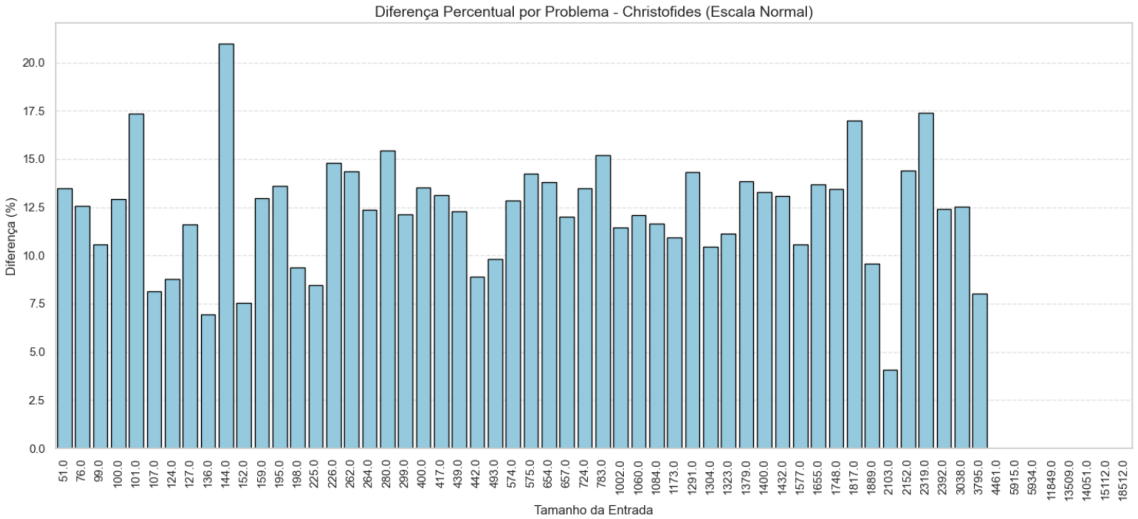

Figure 2. Christofides: Diferença entre a solução do algoritmo e a solução ótima

Concluímos que o algoritmo de Christofides apresenta um desempenho robusto para instâncias pequenas e moderadas do problema do caixeiro-viajante, oferecendo soluções com qualidade garantida devido ao seu fator de aproximação. Durante os testes, o **desvio médio em relação à solução ótima foi de aproximadamente 12,22%**, confirmando a eficiência do algoritmo em alcançar soluções próximas ao ótimo teórico. No entanto, para instâncias maiores, seu uso pode ser limitado devido ao aumento significativo no custo do algoritmo.

4.3. Comparação dos resultados

Esta seção tem como objetivo comparar os algoritmos *Twice Around The Tree* e *Christofides*, uma vez que o algoritmo de *Branch and Bound* não produziu resultados suficientes para ser incluído nesta análise. A comparação será conduzida com base na qualidade das soluções e nos recursos computacionais necessários para sua execução, com o intuito de destacar as vantagens e limitações de cada abordagem, aprofundando o que foi abordado nas seções anteriores.

4.3.1. Tempo de Execução

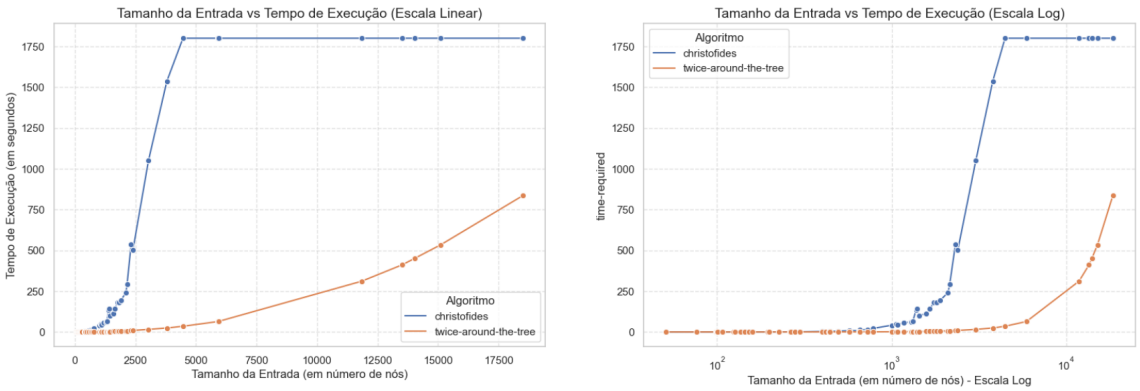

Figure 3. Tempo de Execução para os 2 algoritmos

Como abordado nas seções anteriores, o algoritmo Twice Around the Tree é $O(V^2)$, enquanto a do Christofides é $O(V^3)$. Portanto, o TATT se mostrou notavelmente mais rápido, como é possível ver na Figura 3. Além disso, por conta do limite de tempo (30 minutos = 1800 segundos), o algoritmo de Christofides não foi capaz de produzir uma solução, para instâncias com mais de 4800 nós. Logo, embora o TATT apresente uma qualidade ”pior”, ele se mostra mais eficiente para grandes instâncias. Além disso, as curvas são, de fato, representante da complexidade do algoritmo, como é possível a curva laranja se assemelha a função quadrática $f(x) = x^2$, para o Twice Around the Tree, e a curva azul, à função cúbica $f(x) = x^3$, para Christofides.

4.3.2. Espaço gasto

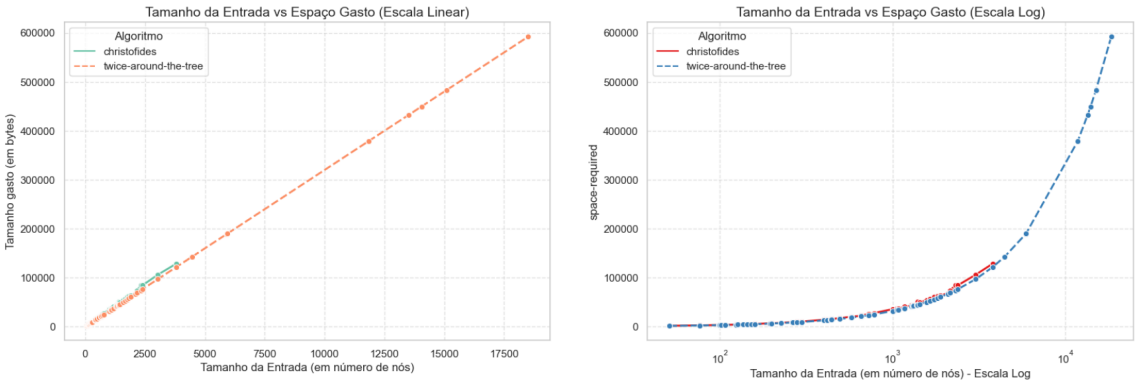

Figure 4. Espaço gasto para os 2 algoritmos

Ambos os algoritmos possuem complexidade espacial quadrática, o que é demostrado na Figura 4. Em termos de espaço, os dois se comportam da mesma maneira, tornando a memória um fator de pouca importância para a avaliação da escolha dos algoritmos.

4.3.3. Erro

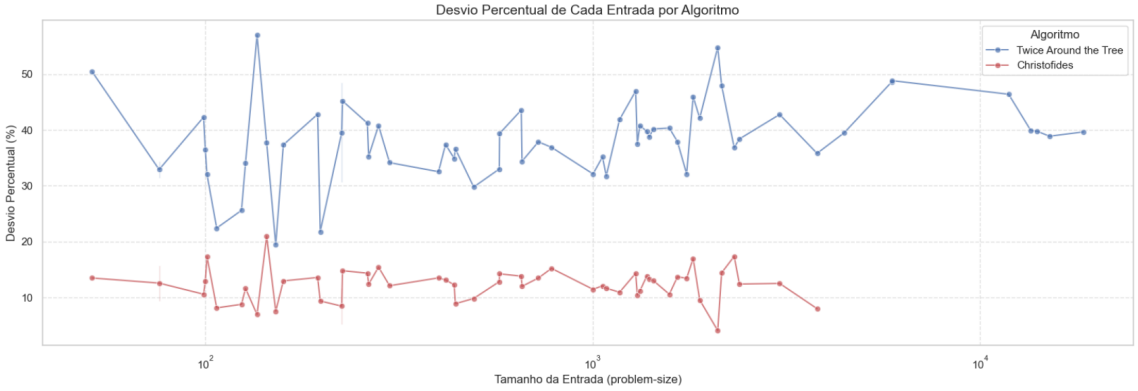

Figure 5. Comparação entre os erros de cada algoritmo para cada instância

Com relação ao erro de cada solução, o algoritmo de Christofides é a melhor escolha entre os dois. Sua faixa de erro se concentra nos 12%, enquanto as saídas produzidas pelo TATT estão na faixa de 40%. Porém, um fator a ser considerado, é a velocidade de produção de resposta: se a demanda for rapidez, talvez Twice Around the Tree seja melhor, mesmo com precisão inferior. Se a demanda for precisão, mesmo que através de um tempo maior, certamente Christofides será melhor. Portanto, esse é o *trade-off* dos algoritmos - eficiência *versus* qualidade.

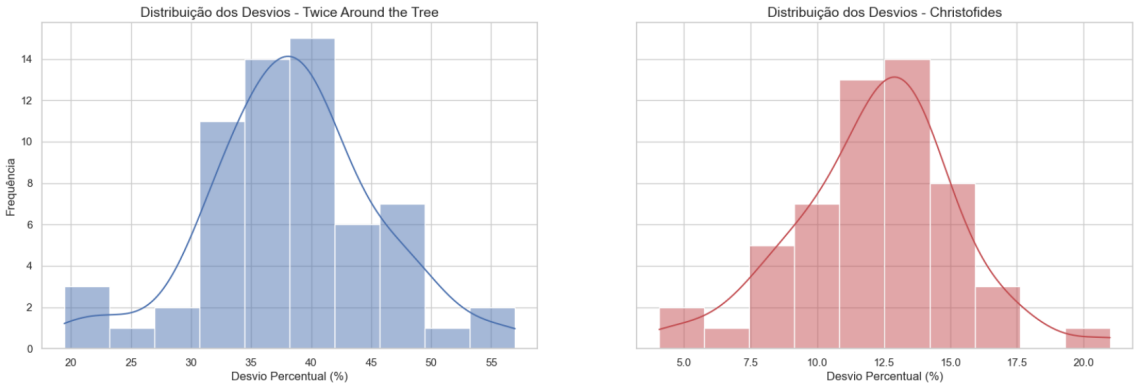

Figure 6. Erro médio de cada algoritmo

A Figura 6 revela que há grande chance do algoritmo Twice Around the Tree produzir uma saída que seja 22,5% a 52,5% pior que a solução ótima, com grandes chances de estar próxima de 36%. Para Christofides, essa estimava se dá no intervalo de 7,5% a 17,5% pior que a ótima, com maior frequência em 12%.

5. Conclusão

Neste trabalho, analisamos o desempenho de diferentes algoritmos aplicados ao problema do Caixeiro Viajante (TSP) euclidiano, abordando as características e limitações de métodos exatos e aproximativos. O algoritmo *Branch and Bound* demonstrou-se ineficiente para as instâncias avaliadas, excedendo consistentemente o limite de tempo estipulado, o que reforça a dificuldade em aplicar métodos exatos para problemas NP-completos.

Em contraste, os algoritmos aproximativos *Twice Around the Tree* (TATT) e *Christofides* mostraram-se viáveis para diversas instâncias, embora apresentem diferentes *trade-offs* entre eficiência e qualidade da solução. O TATT destacou-se pela simplicidade e rapidez, mesmo em instâncias maiores, graças à sua complexidade $O(V^2)$. No entanto, isso ocorre à custa de uma precisão menor, com desvios médios de 36% em relação à solução ótima.

O algoritmo de *Christofides*, por sua vez, apresentou soluções com qualidade significativamente melhor, com desvios médios de apenas 12%. No entanto, sua complexidade $O(V^3)$ impôs limitações para instâncias grandes, resultando em tempos de execução elevados e inviáveis dentro do limite estipulado.

A comparação evidenciou que a escolha do algoritmo ideal depende das restrições do problema e dos requisitos da aplicação. Para cenários em que a rapidez é primordial, o TATT é uma alternativa satisfatória. Por outro lado, em situações onde a precisão é o fator mais importante, o algoritmo de *Christofides* se destaca como a melhor escolha.

Em síntese, a análise dos algoritmos permitiu compreender melhor as limitações e potencialidades de diferentes abordagens para o TSP, destacando a importância de considerar o equilíbrio entre eficiência e qualidade na escolha de soluções para problemas de grande escala.

References

[1] **CHRISTOFIDES, Nicos**. Worst-case analysis of a new heuristic for the travelling salesman problem. Operations Research Forum, v. 3, 1976. Disponível em: https://api.semanticscholar.org/CorpusID:123194397. Acesso em: 03 jan. 2025.

[2] GeeksforGeeks. *Traveling Salesman Problem using Branch and Bound*. Disponível em: `https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/`. Acesso em: 28 dec. 2024.

[3] Wikipedia. *Algoritmo de Christofides*. Disponível em: `https://pt.wikipedia.org/wiki/Algoritmo_de_Christofides`. Acesso em: 28 dec. 2024.

[4] Wikipedia. *Travelling salesman problem*. Disponível em: `https://en.wikipedia.org/wiki/Travelling_salesman_problem`. Acesso em: 27 dec. 2024.

[5] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. *Slides virtuais da disciplina de Algoritmos II*. Disponibilizado via Moodle. Belo Horizonte, Brasil.