Soluções para Problemas Difíceis

Caixeiro Viajante

Carlos Henrique B. M. Leão

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

Resumo. Este trabalho apresenta uma análise abrangente do desempenho de algoritmos para o Problema do Caixeiro Viajante (TSP), com foco em implementações exatas e aproximativas. Utilizando instâncias do TSP provenientes da TSPLIB, os algoritmos foram testados no ambiente do Google Colab, destacando a eficácia de abordagens como Branch-and-Bound, Twice-Around-the-Tree e Christofides. Os resultados evidenciam a superioridade dos métodos aproximativos em termos de eficiência computacional, especialmente em conjuntos de dados extensos. A análise prática reforça a relevância dessas abordagens em contextos nos quais a busca pela solução ótima é inviável devido a restrições de recursos computacionais.

1. Introdução

O Problema do Caixeiro Viajante (TSP) é um dos desafios fundamentais na área da otimização combinatória. Ele consiste em encontrar a rota mais curta que visita um conjunto de cidades exatamente uma vez e retorna à cidade de origem. Apesar de sua simplicidade aparente, o TSP é conhecido por ser NP-difícil, o que significa que não existe um algoritmo eficiente conhecido para resolver todas as instâncias em tempo polinomial.

Este trabalho aborda os aspectos práticos dos algoritmos para solucionar o TSP, com foco em implementações específicas para o TSP euclidiano. Serão exploradas soluções exatas, baseadas em branch-and-bound, assim como soluções aproximadas, incluindo o algoritmo Twice-Around-the-Tree e o algoritmo de Christofides. O objetivo é, não apenas implementar esses algoritmos, mas também avaliar seu desempenho em termos de tempo, espaço e qualidade da solução.

Nesse sentido, optamos por implementar os algoritmos em *Python3*, aproveitando as bibliotecas padrão da linguagem, incluindo *NumPy* para operações matriciais eficientes. A escolha da representação de dados foi orientada pela eficiência na manipulação de grafos e pela

facilidade de implementação. Detalhes específicos de cada algoritmo, como a estimativa de custo utilizada e as estruturas de dados empregadas, são discutidos nas seções subsequentes. Conforme supracitado, foi realizada a implementação dos seguintes algoritmos:

- **Branch-and-Bound:** Desenvolvemos uma solução eficiente baseada na técnica de branch-and-bound. Essa abordagem permite explorar o espaço de soluções de maneira inteligente, reduzindo o número de soluções candidatas a serem avaliadas.
- **Twice-Around-the-Tree:** Implementamos o algoritmo Twice-Around-the-Tree, uma heurística que busca construir uma solução aproximada percorrendo uma árvore geradora mínima duas vezes. Este método é conhecido por sua simplicidade e eficácia em muitos casos práticos, com fator de aproximação 2.
- **Algoritmo de Christofides:** Desenvolvemos o algoritmo de Christofides, uma solução que combina a construção de uma árvore geradora mínima com um procedimento de emparelhamento perfeito. Essa abordagem visa fornecer soluções aproximadas com garantias com fator de aproximação 1,5.

2. Algoritmos Implementados

Esta seção detalha as escolhas de implementação feitas para cada um dos algoritmos abordados no contexto do Problema do Caixeiro Viajante (TSP). Cada algoritmo possui características distintas, e as decisões tomadas durante a implementação têm impacto direto no desempenho, na eficácia e na precisão das soluções obtidas.

No âmbito das estruturas de dados, cada algoritmo apresentou uma diferente escolha de estruturas que foi norteada pelo melhor desempenho. Porém, para todos os algoritmos foi utilizada uma matriz de adjacência para representar o grafo e as arestas entre cada par de vértices. Nesse sentido, cada posição (i, j) na matriz, representa o custo de deslocamento do vértice i ao vértice j.

2.1. Algoritmo Branch-and-Bound

O algoritmo de Branch-and-Bound é uma abordagem exata para resolver problemas de otimização combinatória, como o Problema do Caixeiro Viajante (TSP). Sua estratégia principal envolve a exploração sistemática do espaço de soluções, utilizando técnicas de poda para evitar a análise de soluções que já se mostraram subótimas.

A ideia geral para esse algoritmo é simples, começamos com uma solução parcial, frequentemente uma solução vazia. A cada passo, o algoritmo expande essa solução parcial, considerando todas as cidades não visitadas como candidatas a serem adicionadas à rota.

No contexto do caminhamento na árvore, optamos por utilizar uma busca em profundidade (Depth First Search - DFS) ao invés de um algoritmo de busca melhor primeiro (Best First Search - BFS). A BFS sempre escolhe iterar pelos caminhos mais promissores, no nosso caso, os caminhos que apresentam o menor custo estimado.

Porém, para o algoritmo encontrar pelo menos uma solução, mesmo que não seja ótima, é preciso que o caminhamento encontre pelo menos uma folha. Em contrapartida, a estratégia BFS faz com que o algoritmo precise de muito mais iterações até chegar em uma folha, já que a tendência do caminho é aumentar o seu custo a cada nível da árvore, o que pode fazer que o algoritmo não encontre nenhuma possível solução em instâncias com um grande número de pontos. Portanto, optamos pelo algoritmo DFS, que realiza uma busca em profundidade, na árvore, garantindo que sempre será retornado pelo menos uma solução, mesmo que com um custo não ótimo.

Durante a exploração, são calculados limites para o custo da solução completa. Esses limites são fundamentais para a poda de ramos que não levariam a soluções melhores do que as já encontradas. Se, em algum ponto, o limite de um nó é maior ou igual ao melhor custo encontrado até o momento, esse ramo pode ser podado.

Nesse sentido, para calcular o limite, como devemos sempre chegar e sair de um vértice, utilizamos como estimativa a soma das duas arestas de menor peso incidentes em cada vértice ainda não visitado, como cada aresta seria contada duas vezes, dividimos esse valor por 2. Ademais, somamos esse valor ao peso atual do caminho, dessa forma, temos um limite para aquele nó.

A cada folha encontrada representa uma possível solução para o problema, se esta tiver custo menor que a melhor solução encontrada até o momento, o algoritmo a atualiza. A busca continua até que todo o espaço de soluções seja explorado.

Em resumo, o algoritmo opera iterativamente sobre os nós de uma árvore de caminhamento, utilizando a heurística de remover os nós não promissores. Em um cenário de pior caso, no qual a poda de ramos não seja possível, a iteração abrange toda a extensão da

árvore. Dessa forma, temos um problema permutativo, em que a complexidade assintótica é $O(V!)$, em que V representa o número total de vértices.

Por fim, além da matriz adjacência que armazena o grafo em custo de espaço $O(n^2)$, também é preciso armazenar os nós da árvore de caminhamento. Ingenuamente, podemos pensar que o custo desse armazenamento é $O(V!)$, assim como a complexidade assintótica, porém, é necessário fazer uma análise mais profunda. Em cada iteração do algoritmo, são armazenados não apenas os nós atuais, mas também todos os nós precedentes. Além disso, para cada nó, é necessário armazenar também todos os possíveis caminhos possíveis a partir daquele nó, dessa forma temos custo espacial de $O(V^2)$. É crucial notar que, à medida que o algoritmo retorna de um nó para um nível superior da árvore, o espaço associado a esse nó é liberado. Essa liberação possibilita utilizar o mesmo espaço para armazenar outros caminhos. Com esse reaproveitamento de espaço, concluímos que o algoritmo apresenta custo de espaço $O(V^2)$.

2.2. Twice-Around-the-Tree

O algoritmo Twice-Around-the-Tree é uma heurística construtiva que visa encontrar uma solução 2-aproximada para o Problema do Caixeiro Viajante (TSP). Sua abordagem é baseada na construção de uma árvore geradora mínima (MST) e na repetição dessa árvore para formar uma rota fechada.

A ideia geral do algoritmo consiste em alguns passos sequenciais.Primeiramente, é necessário gerar a árvore geradora mínima a partir do grafo original, escolhemos realizar esse processo utilizando o algoritmo Prim, que apresenta complexidade assintótica $O(V^2)$, em que V representa a quantidade de vértices do grafo. Nesse sentido, é preciso escolher arbitrariamente um vértice como ponto de partida. Em cada iteração, seleciona-se a aresta de menor peso que conecta um vértice já incluído na MST a um vértice fora da MST. Este processo continua até que todos os vértices estejam incluídos na MST.

Depois que a MST já está gerada, precisamos realizar um caminhamento pré-ordem na árvore, a partir de um vértice de início escolhido. Isso determina a ordem em que os vértices serão visitados na rota final. Por fim, a rota final é formada pela repetição do caminhamento pré-ordem. Essa repetição da MST garante uma solução fechada que visita cada cidade exatamente uma vez. Ademais, o custo da rota é calculado somando as distâncias entre as

cidades na ordem determinada pelo caminhamento pré-ordem. Esses passos apresentam custo total de $O(V + E)$, onde E é o número de arestas na MST, já que cada aresta é visitada duas vezes. Por fim, encontramos que o custo assintótico total do algoritmo é $O(V^2)$.

Nesse algoritmo, além da matriz de adjacência do grafo original, é utilizada outra para armazenar a MST. Além disso, também é utilizado um arranjo de tamanho $O(V)$ para armazenar o caminhamento. Dessa forma, o algoritmo apresente um custo de espaço assintótico de $O(V^2)$, em que V representa a quantidade de vértices.

2.3. Christofides

O algoritmo Christofides apresenta uma solução 1.5-aproximada eficiente para o Problema do Caixeiro Viajante (TSP). Sua abordagem envolve a construção de uma árvore geradora mínima (MST), a criação de um emparelhamento perfeito mínimo para vértices de grau ímpar e a formação de um circuito euleriano seguido por um caminho hamiltoniano.

O processo começa gerando a MST utilizando o algoritmo de Prim, o mesmo utilizado para o outro algoritmo aproximativo supracitado, com uma complexidade assintótica de $O(V^2)$, onde V é o número de vértices no grafo. A escolha arbitrária de um ponto de partida continua até que todos os vértices estejam incluídos na MST.

Em seguida, são identificados vértices de grau ímpar na MST, para os quais é criado um emparelhamento perfeito de peso mínimo no subgrafo induzido pelos vértices ímpares, com complexidade $O(V^3)$. Considerando o multigrafo formado com os vértices do grafo original e as arestas MST do emparelhamento perfeito, é computado um circuito euleriano. A obtenção do circuito euleriano e a subsequente remoção de arestas duplicadas têm uma complexidade total de $O(V^2)$. O cálculo do custo do tour, somando as distâncias entre cidades na ordem do caminho hamiltoniano, é uma operação com custo $O(V)$. Por fim, concluímos que o algoritmo possui complexidade assintótica derivada do emparelhamento perfeito, $O(V^3)$.

Em relação ao espaço, o algoritmo utiliza matrizes de adjacência para o grafo original e para a MST, resultando em um custo total de O(V²). Além disso, arrays de tamanho O(V) são utilizados para armazenar informações sobre o emparelhamento perfeito e o caminho hamiltoniano.

Em resumo, o algoritmo Christofides apresenta uma solução aproximada eficiente para o TSP, balanceando qualidade da solução e tempo de execução, sendo adequado para instâncias em que uma solução rápida é preferível à busca exata.

3. Experimentos e Resultados

Para avaliar o desempenho dos algoritmos implementados, realizamos testes utilizando instâncias do Problema do Caixeiro Viajante (TSP) provenientes da biblioteca TSPLIB. Especificamente, limitamos nossa análise a instâncias cuja função de custo é a distância euclidiana em 2D, possibilitando a aplicação eficiente dos algoritmos aproximativos desenvolvidos para o problema do Caixeiro Viajante Euclidiano.

A execução dos testes ocorreu no ambiente Google Colab, que oferece suporte ao Python 3 e recursos substanciais, incluindo 12GB de RAM e 100GB de espaço em disco para armazenar os arquivos de teste. Essa escolha proporcionou a capacidade de processamento necessária para lidar com instâncias desafiadoras do TSP.

O processo de teste consistiu na execução de cada algoritmo para cada instância do conjunto de teste, registrando variáveis de desempenho, como tempo de processamento e qualidade da solução. Importante mencionar que o algoritmo Branch-and-Bound foi configurado para um tempo máximo de trinta minutos. Caso esse tempo fosse excedido, o algoritmo retornava a melhor solução encontrada até aquele momento.

Devido a limitações de recursos disponíveis, não foi possível executar o algoritmo Branch-and-Bound para instâncias com mais de 2000 pontos. Além do longo tempo de execução, conjuntos maiores demandaram uma quantidade significativa de memória RAM para armazenar todos os nós necessários para o caminhamento na árvore de busca. Sendo assim, conjuntos de dados extensos foram analisados apenas pelos algoritmos aproximativos.

Em relação às instâncias menores, não ocorreram problemas de recursos. Contudo, em diversos testes com o algoritmo Branch-and-Bound, utilizando o Best First Search, a execução não era concluída, e a solução ótima não era encontrada. Optamos por utilizar o caminhamento em profundidade, conforme explicado na Seção 2.1, para obter pelo menos uma solução dentro do limite de 30 minutos de execução, mesmo que esta não fosse ótima.

Surpreendentemente, devido ao longo tempo de execução, os algoritmos aproximativos apresentaram custos de caminhos menores do que o próprio algoritmo Branch-and-Bound em

diversos casos. Além disso, esses resultados foram obtidos pelos algoritmos aproximativos em tempos de execução e uso de memória significativamente menores. A eficiência na utilização de memória RAM decorre do fato de que, como identificado na Seção 2 durante a análise de complexidade de espaço, os algoritmos aproximativos utilizam no máximo duas matrizes de adjacência de inteiros.

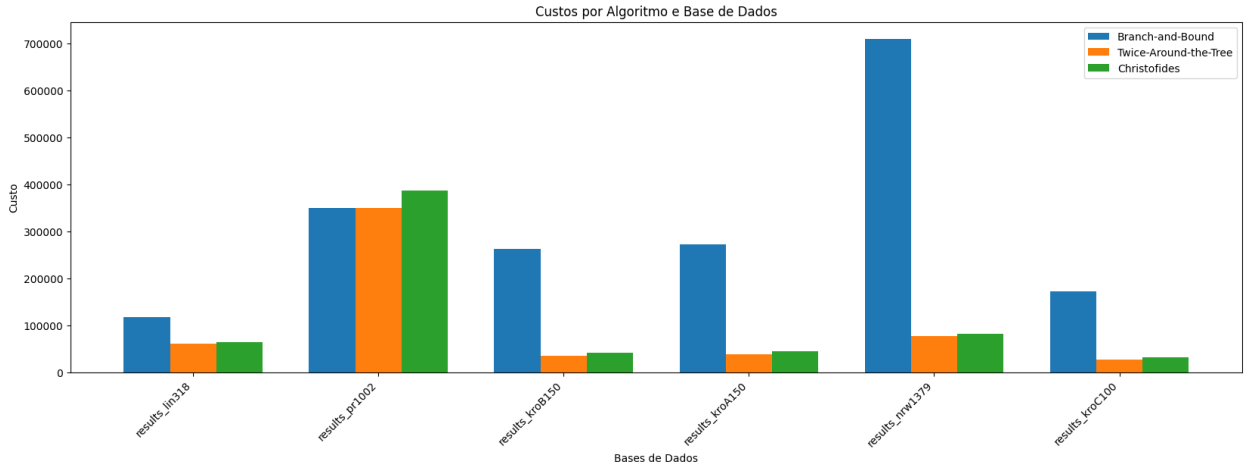

Figura 1. Custos Por Algoritmo e Base de Dados

Na Figura 1, é evidente que os algoritmos aproximativos superaram o desempenho do algoritmo exato dentro do limite de 30 minutos de execução. Além disso, destaca-se a notável semelhança e, em muitos casos, a superioridade do algoritmo aproximativo Twice-Around-the-Tree em relação ao algoritmo Christofides, apesar de o primeiro possuir um fator de aproximação de 2, enquanto o segundo ostenta um fator de 1,5. A análise detalhada dessas instâncias indica que ambos os algoritmos respeitaram efetivamente seus fatores aproximativos.

Nesse sentido, a comparação entre os algoritmos aproximativos destaca a importância de considerar não apenas os fatores de aproximação, mas também a eficácia prática em conjuntos de dados reais. A superioridade do Twice-Around-the-Tree sobre o Christofides em diversas instâncias destaca a influência de outros fatores que podem alterar os resultados médios.

Um achado interessante é observado nos resultados para a base de dados *pr1002*, onde o custo encontrado para o algoritmo exato parece coincidir com o algoritmo aproximativo. Contudo, ao analisar especificamente esse caso, notamos que embora os caminhos encontrados pelos dois algoritmos sejam semelhantes, não são exatamente iguais. Portanto, os custos

resultantes também diferem, com o algoritmo exato alcançando um custo de 349019, enquanto o algoritmo aproximativo registrou um valor ligeiramente superior, totalizando 349155.

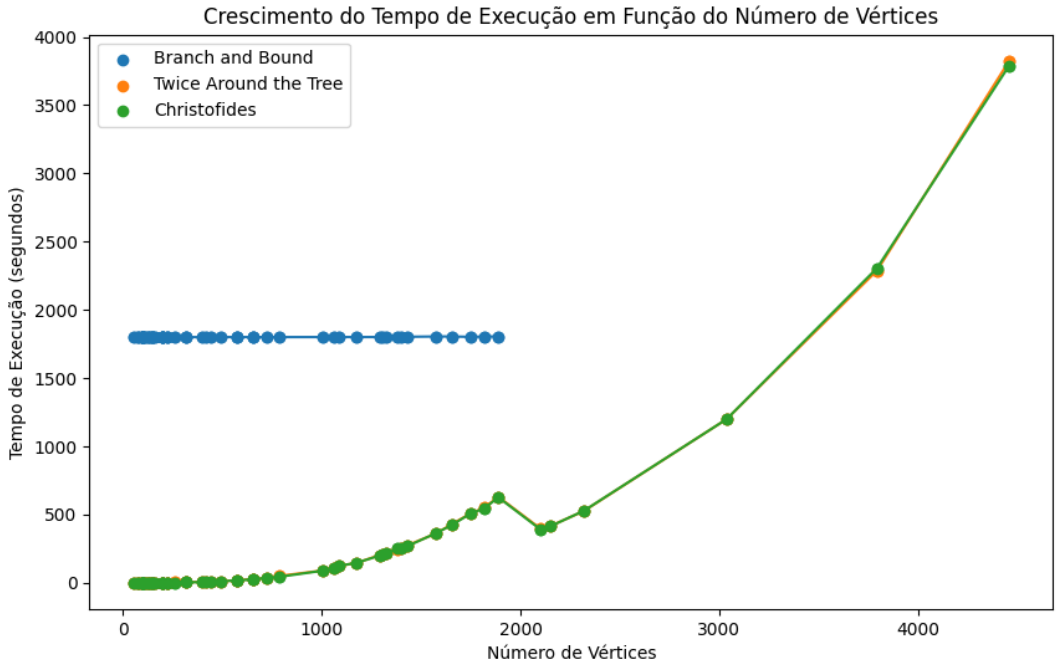

Figura 2. Crescimento do Tempo de Execução em Função do Número de Vértices

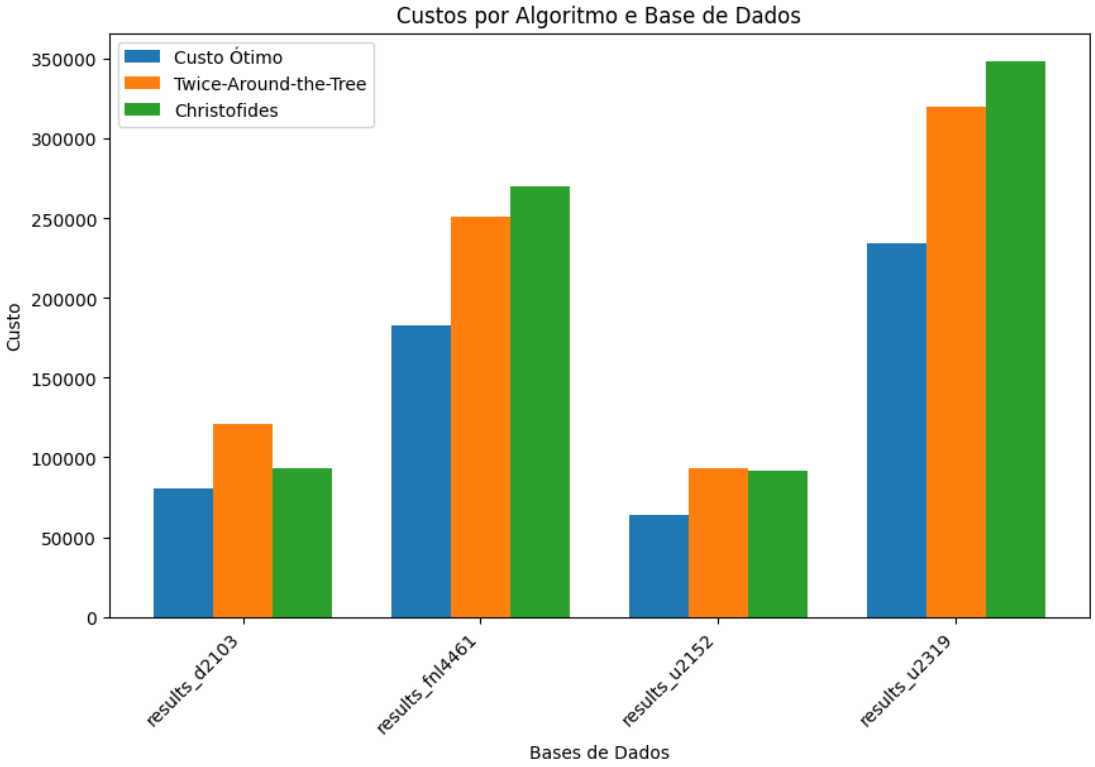

Figura 3. Custo por Algoritmos e Custo Ótimo em Função da Base de Dados

Ao avaliar o tempo de execução dos diferentes algoritmos na Figura 2, fica evidente a natureza exponencial dos algoritmos aproximativos, que exibiram tempos de execução muito próximos. Além disso, observamos que todas as execuções do algoritmo exato consumiram os 30 minutos disponíveis para a execução.

A Figura 3 destaca um cenário específico em que a aplicação prática dos algoritmos aproximativos se destaca. Ao considerar conjuntos de dados com mais de 2000 cidades, o algoritmo Branch-and-Bound demanda um grande número de recursos, os quais não temos disponíveis, impossibilitando a análise exata dessas instâncias. No entanto, ao observar os resultados dos algoritmos aproximativos, percebemos que tanto o Twice-Around-the-Tree quanto o Christofides produzem soluções próximas ao custo ótimo, com todos os valores abaixo do fator de aproximação de 1,5.

É interessante notar os casos em que o Twice-Around-the-Tree supera o desempenho do Christofides, enfatizando a importância de avaliar não apenas os fatores de aproximação teóricos, mas também a eficácia prática. Contudo, em sua aplicação em dados do mundo real, não devemos esquecer dos reais limiares de aproximação, para não tomar decisões equivocadas baseadas em soluções aproximadas encontradas.

Com esses testes, podemos verificar que os algoritmos aproximativos, mesmo que não encontrem a solução ótima, produzem soluções muito próximas. Nesse sentido, na prática, essa abordagem pode se mostrar muito relevante, pois frequentemente é mais vantajoso ter uma boa solução em tempo hábil do que a melhor em um tempo consideravelmente maior. Além disso, problemas reais frequentemente lidam com imprecisões na coleta de dados, como, por exemplo, no cálculo de distâncias. Essas imprecisões diminuem ainda mais a necessidade de encontrar uma solução ótima, o que valoriza ainda mais os algoritmos aproximativos na prática.

Essa análise adiciona uma perspectiva prática à avaliação dos algoritmos, ressaltando a relevância dos métodos aproximativos em contextos nos quais encontrar a solução ótima pode ser inviável em termos de recursos computacionais.

4. Conclusões

Diante dos experimentos e análises realizados, algumas conclusões importantes podem ser extraídas. Primeiramente, observamos que os algoritmos aproximativos, nomeadamente o Twice-Around-the-Tree e o Christofides, demonstraram ser alternativas viáveis e eficientes para

instâncias do Problema do Caixeiro Viajante (TSP) Euclidiano em que a solução exata utilizando a técnica Branch-and-Bound é computacionalmente inviável.

Os resultados destacam a notável eficácia dos algoritmos aproximativos em termos de tempo de execução e uso de recursos, superando o desempenho do algoritmo exato dentro de um tempo de execução limitado, especialmente em instâncias com um grande número de cidades. A Figura 1 ilustra claramente essa tendência, revelando que, dentro do limite de 30 minutos, os algoritmos aproximativos conseguiram produzir soluções com custos inferiores ao algoritmo exato.

A análise comparativa entre o Twice-Around-the-Tree e o Christofides também destaca a importância de avaliar não apenas os fatores de aproximação teóricos, mas também a eficácia prática em conjuntos de dados reais. Em alguns casos, o Twice-Around-the-Tree demonstrou um desempenho igual ou superior ao Christofides, apesar do fator de aproximação 2 versus 1,5. Essa observação ressalta a necessidade de considerar diversos fatores ao escolher uma heurística para resolver o TSP, levando em conta as características específicas das instâncias em questão. Contudo, como já citado anteriormente, o real fator aproximativo de um algoritmo não deve ser esquecido ao tomar uma decisão.

Por fim, a Figura 3 destaca a aplicabilidade prática dos algoritmos aproximativos para conjuntos de dados extensos, onde o algoritmo Branch-and-Bound se mostra impraticável devido aos requisitos significativos de recursos e de tempo. Nesse cenário, tanto o Twice-Around-the-Tree quanto o Christofides conseguiram produzir soluções próximas ao ótimo, demonstrando sua utilidade em situações em que encontrar a solução exata é impraticável em termos de recursos computacionais e também desnecessária, devido às possíveis inconsistências e imprecisões nas coletas dos dados reais.

Em síntese, este estudo oferece percepções valiosas sobre a escolha e desempenho de algoritmos para o Problema do Caixeiro Viajante Euclidiano. A aplicação prática dos algoritmos aproximativos destaca sua relevância em contextos nos quais a busca pela solução ótima é limitada por restrições computacionais. Essa pesquisa contribui para a compreensão e seleção informada de estratégias para abordar o TSP em cenários do mundo real, onde a eficiência e a eficácia são fundamentais.

5. Referências Bibliográficas

PYTHON. **Python 3.12.1 documentation**. Disponível em: https://docs.python.org/3/ . Acesso em: 9 dez. 2023.

NUMPY. **NumPy documentation**. Disponível em: https://numpy.org/doc/stable/ . Acesso em: 9 dez. 2023.

MATPLOTLIB. **Matplotlib 3.8.2 documentation**. Disponível em: https://matplotlib.org/stable/index.html . Acesso em: 9 dez. 2023.

WIKIPEDIA. **Traveling salesman problem**. Disponível em: https://en.wikipedia.org/wiki/Travelling_salesman_problem . Acesso em: 9 dez. 2023.

VIMIEIRO, Renato. **Aula 12 - Soluções exatas para problemas difíceis (branch-and-bound).** Belo Horizonte, Minas Gerais. out. 2023. Apresentação de PDF. color.

VIMIEIRO, Renato. **Aula 13 - Soluções aproximadas para problemas difíceis**. Belo Horizonte, Minas Gerais. out. 2023. Apresentação de PDF. color.

GEEKS FOR GEEKS. **Eulerian path and circuit for undirected graph.** Disponível em: https://www.geeksforgeeks.org/eulerian-path-and-circuit/. Acesso em: 9 dez. 2023.

GEEKS FOR GEEKS. **Prim's Algorithm for Minimum Spanning Tree (MST).** Disponível em: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/. Acesso em: 9 dez. 2023.

BRILLIANT. **Blossom Algorithm.** Disponível em: https://brilliant.org/wiki/blossom-algorithm/. Acesso em: 9 dez. 2023.