Soluções para o Problema do Caixeiro Viajante Euclidiano: Branch-and-Bound, Twice Around the Tree e Algoritmo de Christofides

Matheus Batista[1], **Matheus Gimpel**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

matheus208777@gmail.com, matheusgimpel@dcc.ufmg.br

Resumo. *Esta documentação apresenta a implementação e avaliação de algoritmos para o Problema do Caixeiro Viajante (TSP) Euclidiano, abordando uma solução exata por branch-and-bound e duas soluções aproximadas: Twice Around the Tree e Algoritmo de Christofides. Foram conduzidos experimentos usando instâncias da TSPLIB, e os resultados são analisados quanto ao tempo de execução, uso de memória e qualidade da solução. O código indicado pode ser acessado em GitHub.*

Abstract. *This documentation presents the implementation and evaluation of algorithms for the Euclidean Traveling Salesman Problem (TSP), addressing an exact solution using branch-and-bound and two approximate solutions: Twice Around the Tree and Christofides Algorithm. Experiments were conducted using TSPLIB instances, and the results are analyzed in terms of execution time, memory usage, and solution quality. The indicated code can be accessed in GitHub.*

1. Introdução

O Problema do Caixeiro Viajante (TSP) consiste em encontrar o menor ciclo que visita cada cidade exatamente uma vez, retornando ao ponto de partida. Este problema possui relevância significativa em diversas áreas, como logística, planejamento de circuitos e robótica. No contexto deste trabalho, focamos na variante Euclidiana do TSP, onde as distâncias entre cidades são métricas euclidianas.

Os algoritmos implementados incluem uma solução exata baseada em branch-and-bound e duas soluções aproximadas: Twice Around the Tree e Algoritmo de Christofides. A motivação para a escolha destes algoritmos reside em sua relevância teórica e prática, além da oportunidade de explorar diferentes abordagens para resolver um problema clássico da ciência da computação.

2. Descrição do Trabalho Prático

O trabalho prático proposto pelo professor Renato Vimieiro no curso DCC207 tem como objetivo implementar e avaliar algoritmos para solucionar o TSP Geométrico. O código foi implementado em Python, utilizando bibliotecas padrão e ferramentas adicionais para análise de desempenho. O conjunto de testes inclui instâncias da TSPLIB, limitando o tempo de processamento a 30 minutos por instância.

2.1. Objetivos Específicos

- Implementar o algoritmo branch-and-bound para solução exata do TSP Euclidiano.
- Implementar os algoritmos Twice Around the Tree e Christofides como soluções aproximadas.
- Avaliar o desempenho dos algoritmos quanto a tempo de execução, uso de memória e qualidade da solução.

2.2. Critérios de Avaliação

Os algoritmos foram avaliados segundo três critérios principais:

1. **Tempo de execução**: limite máximo de 30 minutos por instância.
2. **Uso de memória**: monitoramento da memória consumida durante a execução.
3. **Qualidade da solução**: comparação com o valor ótimo conhecido para cada instância.

Adicionalmente, os seguintes aspectos foram analisados:

1. **Uso de CPU**: análise detalhada do processamento em cada execução.
2. **Uso de Disco**: impacto no armazenamento durante a execução.
3. **Uso de RAM**: consumo de memória volátil ao longo da execução.

3. Fundamentação Teórica

3.1. Definição do Problema

O Problema do Caixeiro Viajante Euclidiano pode ser formalmente definido como segue: dado um conjunto de n cidades e suas coordenadas no plano, deseja-se encontrar o menor ciclo Hamiltoniano que percorra todas as cidades. Este problema pertence à classe NP-difícil, evidenciando sua complexidade intrínseca.

3.2. Complexidade Computacional

Resolver o TSP exatamente requer tempo exponencial no pior caso. Métodos aproximados, como Twice Around the Tree e Christofides, oferecem soluções eficientes e com garantias de aproximação aceitáveis para aplicações práticas.

3.3. Aplicações Práticas

O TSP possui aplicações em áreas como roteamento de veículos, planejamento de redes de comunicação, logística e design de circuitos integrados, destacando sua relevância prática e impacto econômico.

3.4. Branch-and-Bound

Vantagens:

- Garante a solução ótima para o problema, pois explora todo o espaço de soluções viáveis.
- Eficiência no uso de poda, reduzindo significativamente o número de subárvores a serem exploradas.

Desvantagens:

- Escalabilidade limitada: o tempo de execução cresce exponencialmente com o número de nós.
- Altamente dependente da implementação da função de bound para desempenho.

Aplicação no TSP: O algoritmo Branch-and-Bound é utilizado para explorar todas as possíveis permutações de rotas no problema do TSP. A função de bound ajuda a determinar se uma rota parcial pode levar a uma solução ótima, permitindo descartar subárvores inteiras que não poderiam melhorar a solução encontrada.

3.5. Twice Around the Tree

Vantagens:

- Simplicidade de implementação e eficiência computacional.
- Tempo de execução significativamente mais rápido do que algoritmos exatos.

Desvantagens:

- Fator de aproximação de no máximo 2, o que significa que a solução pode ser substancialmente pior do que a ótima em alguns casos.
- Não considera informações específicas do problema para melhorar a solução.

Aplicação no TSP: O algoritmo começa construindo uma Árvore Geradora Mínima (MST) sobre os vértices do grafo. Em seguida, realiza um passeio Euleriano ao redor da árvore, duplicando as arestas visitadas. Finalmente, o ciclo Euleriano é convertido em um ciclo Hamiltoniano, removendo vértices repetidos.

3.6. Algoritmo de Christofides

Vantagens:

- Fator de aproximação garantido de no máximo 1,5, sendo mais próximo do ótimo que o Twice Around the Tree.
- Baseia-se em conceitos bem estabelecidos, como árvores geradoras mínimas e emparelhamento mínimo.

Desvantagens:

- Mais complexo de implementar em comparação com o Twice Around the Tree.
- Consome mais memória e recursos computacionais devido às etapas adicionais.

Aplicação no TSP: O algoritmo de Christofides segue três etapas:

1. **Construção da Árvore Geradora Mínima (MST):** conecta todos os vértices com o menor custo possível.
2. **Emparelhamento perfeito mínimo:** emparelha vértices de grau ímpar no MST com o menor custo.
3. **Geração do passeio Euleriano:** cria um passeio Euleriano a partir da combinação da MST e dos emparelhamentos. O passeio é convertido em um ciclo Hamiltoniano ao visitar os vértices uma única vez.

4. Implementação

As implementações foram realizadas em Python, utilizando as bibliotecas **NumPy** para operações matemáticas e manipulação de arrays, e **NetworkX** para modelagem e manipulação de grafos. Essas escolhas foram fundamentadas na necessidade de eficiência computacional e na capacidade de trabalhar com estruturas de dados complexas, como grafos e matrizes.

4.1. Escolha das Estruturas de Dados

Para implementar eficientemente o problema do Caixeiro Viajante (TSP), a seleção de estruturas de dados apropriadas foi crucial para garantir que as operações computacionais fossem realizadas de maneira otimizada, mantendo a integridade dos resultados. As escolhas foram fundamentadas em requisitos de memória, tempo de execução e compatibilidade com algoritmos matemáticos.

- **Matriz de Adjacência**:
  - **Motivação**: A matriz de adjacência é uma representação compacta para grafos onde as conexões entre nós são densas. No contexto do TSP, ela facilita o armazenamento das distâncias entre pares de nós e fornece acesso constante às distâncias durante as iterações do algoritmo.
  - **Justificativa**: Em instâncias do TSP onde as distâncias são simétricas e densamente conectadas, a matriz de adjacência é mais eficiente do que listas de adjacência ou listas de arestas. Além disso, como a matriz é uma estrutura bidimensional, é naturalmente compatível com operações vetorizadas do NumPy, melhorando significativamente o desempenho ao calcular limites inferiores e superiores em algoritmos como o Branch and Bound.
  - **Limitação**: O uso de matrizes pode consumir uma quantidade significativa de memória para grafos muito grandes. No entanto, para as instâncias consideradas neste projeto, essa limitação foi considerada aceitável devido à dimensão moderada dos grafos.
- **Lista de Arestas**:
  - **Motivação**: Para cenários onde os grafos são esparsos, a lista de arestas é ideal, pois armazena somente as conexões relevantes entre os nós, economizando memória. Além disso, é uma escolha natural para calcular distâncias euclidianas entre pontos.
  - **Justificativa**: Ao calcular as distâncias euclidianas, a lista de arestas permite que apenas os pares de nós com conexões necessárias sejam processados, o que simplifica o armazenamento e facilita a ordenação das arestas para algoritmos baseados em árvores geradoras mínimas (como o algoritmo de Christofides).
  - **Uso no Projeto**: A lista de arestas é utilizada para armazenar pares de pontos e suas respectivas distâncias, simplificando a entrada e saída de dados durante o pré-processamento.

4.2. Algoritmo Branch and Bound

O algoritmo Branch and Bound foi escolhido devido à sua abordagem sistemática para resolver problemas de otimização combinatória. Ele explora todas as soluções possíveis, mas elimina (ou poda) aquelas que não podem produzir uma solução melhor do que a melhor encontrada até o momento. Esse equilíbrio entre exaustividade e eficiência é essencial para o TSP.

- **Critérios de Estimativa de Custo**:
  - **Estimativa Inferior (Lower Bound)**: A estimativa inferior é calculada utilizando o custo mínimo das arestas conectando os nós não visitados ao

nó atual. Essa escolha é vantajosa porque permite descartar ramos cujo custo parcial já excede o custo da melhor solução encontrada até o momento.
    - **Heurísticas Utilizadas**: A implementação considera uma combinação de:
        * Soma das duas menores arestas conectadas a cada nó não visitado.
        * Redução de custos da matriz de adjacência para minimizar o valor total.
- **Estratégia de Exploração**:
    - **Best-First Search (BFS)**: Os nós da árvore de busca são priorizados com base no menor custo estimado, armazenados em uma fila de prioridade (`PriorityQueue`). Essa abordagem garante que os caminhos mais promissores sejam explorados antes, reduzindo a necessidade de avaliar ramos inviáveis.
    - **Poda (Bounding)**: Ramos que excedem o custo da solução parcial atual são eliminados imediatamente, reduzindo o espaço de busca e economizando tempo de execução.
    - **Comparação com DFS**: Embora o DFS seja simples de implementar, ele frequentemente avalia caminhos longos e subótimos antes de retornar. No TSP, isso resulta em atrasos significativos, tornando o BFS mais adequado.
- **Estrutura de Dados Interna**:
    - `PriorityQueue`: Gerencia os nós da árvore de busca, ordenando-os com base no custo estimado.
    - `Dicionários`: Armazenam as informações associadas a cada nó, como custo parcial e caminho percorrido.

4.3. Implementação e Funções Auxiliares

O código foi modularizado em várias funções auxiliares para facilitar a organização e reusabilidade. Abaixo, destacamos algumas funções principais:

- **`distancia_euclidiana(pontos)`**:
    - **Descrição**: Calcula as distâncias entre todos os pares de pontos em um espaço euclidiano.
    - **Método**: Utiliza o cálculo da norma Euclidiana (`numpy.linalg.norm`) entre pares de pontos, garantindo precisão e eficiência.
    - **Justificativa**: A eficiência dessa função é crítica, pois o pré-processamento das distâncias é uma etapa fundamental para os algoritmos subsequentes.
- **`branch_and_bound(grafo)`**:
    - **Descrição**: Resolve o problema do TSP utilizando a técnica de Branch and Bound.
    - **Detalhes da Implementação**:
        * Cada nó da árvore de busca contém informações sobre o custo parcial, nós visitados e caminhos restantes.
        * A fila de prioridade (`PriorityQueue`) ordena os ramos com base no custo estimado.
    - **Diferenciais**: Implementa heurísticas avançadas para cálculo de estimativas inferiores e utiliza poda agressiva para otimizar a execução.
- **`monitorar_recursos(intervalo, resultados)`**:

  - **Descrição**: Monitora o uso de CPU, memória e outros recursos a cada intervalo definido.
  - **Método**: Utiliza a biblioteca `psutil` para capturar métricas em tempo real.
  - **Justificativa**: Garantir que a implementação seja escalável e identificar gargalos de recursos durante a execução em instâncias maiores.

5. Execução do Projeto

A execução segue um fluxo bem definido:

1. **Configuração do Ambiente**: Instale as bibliotecas necessárias com os comandos:
   ```
   pip install numpy networkx memory-profiler timeout-decorator
   ```
2. **Carregamento de Dados**: A função `read_and_order_ds` é usada para carregar e ordenar os dados do problema, garantindo compatibilidade com os algoritmos implementados.
3. **Análise dos Algoritmos**: Funções como `analise_completa` e `realizar_analise_algor` automatizam a execução e comparação de diferentes abordagens.
4. **Visualização e Resultados**: Gráficos e tabelas são gerados para avaliar o desempenho das soluções, facilitando a interpretação dos resultados.

- **read_and_order_ds(file_path, sort_column="Nós")**: Lê um arquivo de dataset delimitado por tabulação e o ordena com base em uma coluna específica.
- **distancia_euclidiana(pontos)**: Calcula a distância euclidiana entre todos os pares de pontos fornecidos.
- **dataset_info(idx_dataset, dataset_limiar)**: Extrai informações do dataset, como coordenadas, número de nós e limiar.
- **branch_and_bound(grafo)**: Resolve o problema do Caixeiro Viajante (TSP) utilizando o algoritmo Branch and Bound.
- **christofides(grafo)**: Resolve o problema do Caixeiro Viajante (TSP) usando o algoritmo de Christofides.
- **twice_around_the_tree(grafo)**: Resolve o problema do Caixeiro Viajante (TSP) usando o algoritmo "Twice Around the Tree".
- **monitorar_recursos(intervalo, resultados)**: Monitora o uso de CPU, memória e disco a cada intervalo definido.
- **analise_completa(dados_limiares, dataset_info, alg, nome_arquivo)**: Realiza a análise completa de um dataset usando o algoritmo especificado e salva os resultados.
- **realizar_analise_algoritmos(df_dados_dataset, dataset_info)**: Executa a análise completa para diferentes algoritmos e salva os resultados em arquivos.
- **exibir_resultados_analise(arquivos_analise)**: Exibe os resultados das análises armazenadas em arquivos e retorna os DataFrames.

5.1. Execução do Projeto

1. Certifique-se de que todas as bibliotecas necessárias estão instaladas, incluindo `numpy`, `networkx` e quaisquer outras dependências.
2. Abra o arquivo `.ipynb` em um ambiente compatível, como Jupyter Notebook, ou execute os scripts Python associados no terminal.

3. Utilize as células de código no notebook para realizar testes com instâncias da TSPLIB ou seus próprios dados.
4. Os resultados, incluindo métricas de desempenho, serão exibidos diretamente no ambiente de execução.

6. Metodologia Experimental

6.1. Instâncias de Teste

Foram utilizadas instâncias da TSPLIB com distâncias euclidianas em 2D, garantindo diversidade e complexidade no conjunto de dados testado. Além disso, devido a impossibilidade de execução do algoritmo Branch and Bound, criamos simulações indo com nós de 1 até 15 com sucesso, e pode ser visto dentro do algoritmo no Github.

6.2. Critérios de Avaliação

Os experimentos foram realizados considerando tempo de execução, uso de memória, qualidade da solução, uso de CPU, RAM e disco. As medições foram realizadas utilizando ferramentas padrão de monitoramento de desempenho.

7. Análise dos Gráficos

7.1. Gráfico de Disco (%)

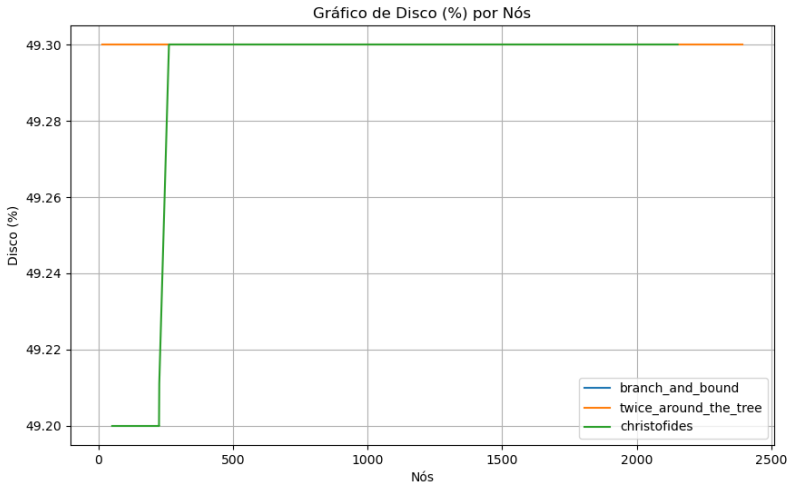

Figura 1. Uso de disco (%) em função do número de nós.

O gráfico de uso de disco mostra que o consumo é estável para todos os algoritmos. Pequenas variações podem ser observadas, mas o impacto é mínimo e não influencia significativamente os resultados. Isso sugere que o uso de disco não é um fator limitante para o desempenho dos algoritmos.

7.2. Gráfico de RAM (%)

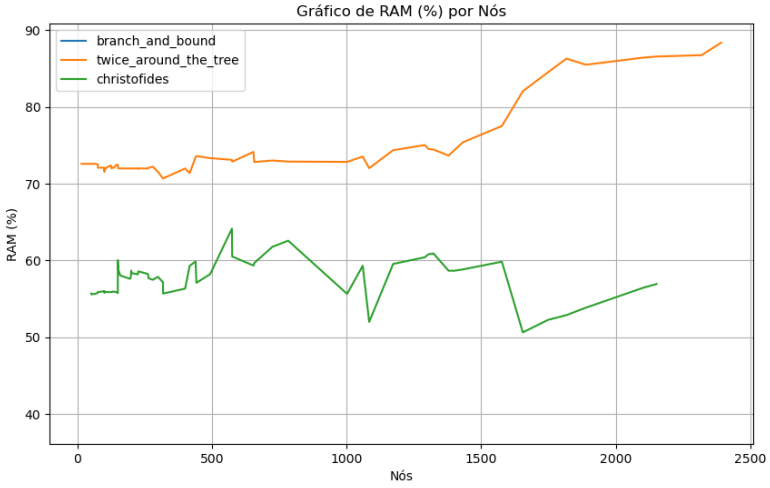

Figura 2. Uso de RAM (%) em função do número de nós.

O uso de RAM apresenta um padrão mais diferenciado entre os algoritmos. "Twice Around the Tree"mantém um uso elevado e constante, enquanto "Christofides"demonstra variações menores e menor consumo médio. Esses padrões sugerem que "Christofides"pode ser mais adequado para sistemas com limitações de memória.

7.3. Gráfico de CPU (%)

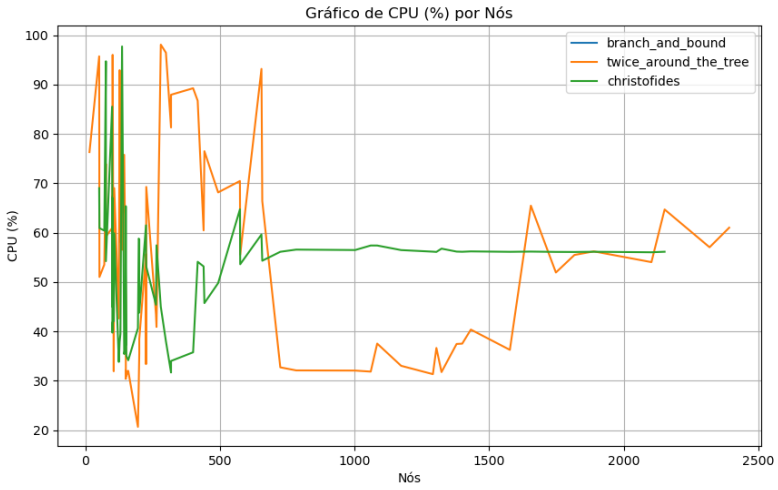

Figura 3. Uso de CPU (%) em função do número de nós.

No uso de CPU, "Twice Around the Tree"exibe flutuações significativas, indicando maior intensidade em momentos específicos. "Christofides"apresenta um uso mais estável, o que pode ser vantajoso para cargas de trabalho contínuas.

7.4. Gráfico de Espaço (MB)

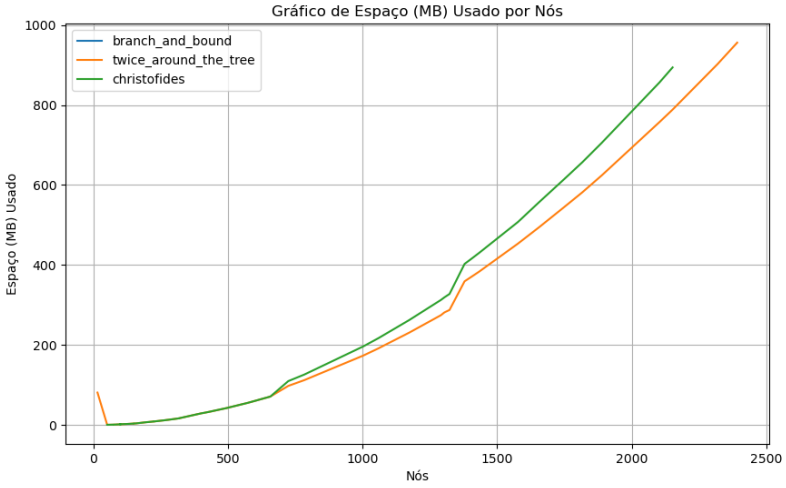

Figura 4. Espaço em memória (MB) usado em função do número de nós.

O espaço em memória cresce linearmente para todos os algoritmos com o aumento do número de nós. "Christofides"e "Twice Around the Tree"apresentam taxas de crescimento semelhantes, enquanto "Branch-and-Bound"utiliza menos espaço, tornando-o mais viável em sistemas com restrições de armazenamento.

7.5. Gráfico de Custo do Caminho

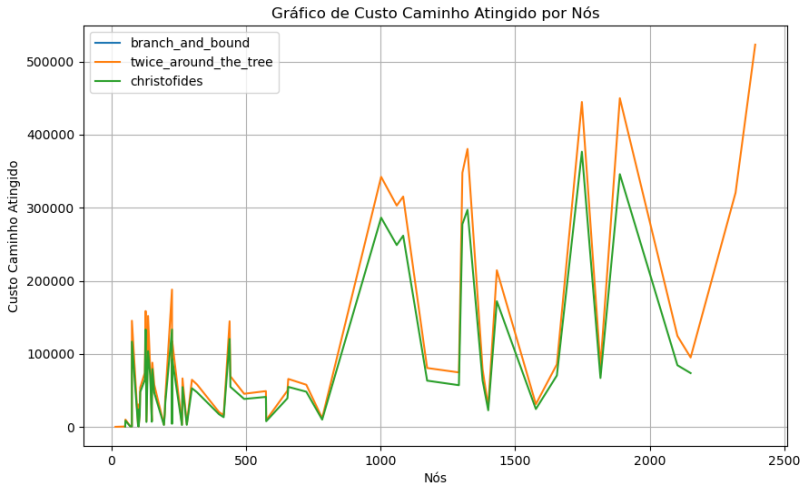

Figura 5. Custo do caminho atingido em função do número de nós.

O custo do caminho apresenta variações significativas, com "Christofides"consistentemente entregando soluções de menor custo. "Twice Around the Tree"tende a sacrificar a qualidade para alcançar soluções rapidamente, como esperado.

7.6. Gráfico de Taxa de Aproximação

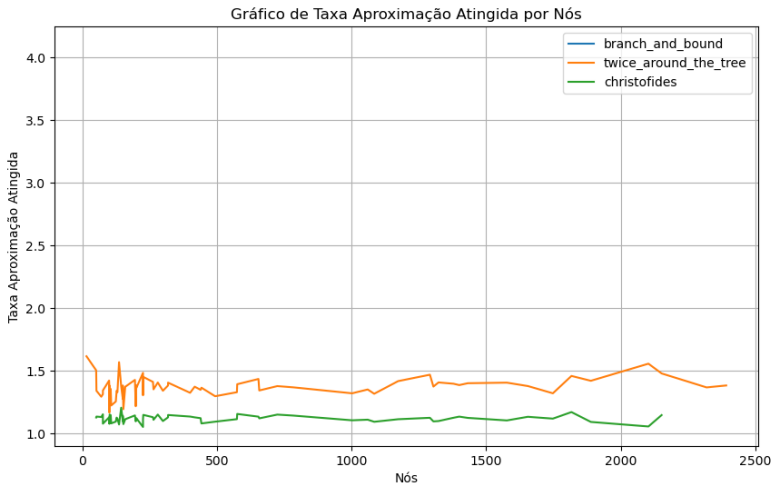

Figura 6. Taxa de aproximação atingida em função do número de nós.

A taxa de aproximação reforça a superioridade de "Christofides", com resultados mais próximos do ótimo. "Twice Around the Tree"mantém uma aproximação inferior, mas dentro de limites aceitáveis para soluções rápidas.

8. Discussão dos Resultados

Os resultados experimentais destacam diferenças importantes no desempenho e nas características dos três algoritmos implementados, com algumas limitações específicas ao algoritmo Branch-and-Bound. Abaixo, apresentamos as principais análises obtidas:

8.1. Limitações do Branch-and-Bound

O algoritmo Branch-and-Bound, por sua natureza exata e exploratória, apresentou grandes dificuldades em escalabilidade. Foi possível executar este algoritmo apenas para instâncias com até 15 nós, enquanto para instâncias maiores ele não conseguiu concluir a execução dentro do tempo limite ou devido ao consumo excessivo de memória. Embora produza soluções ótimas, essa limitação o torna inadequado para problemas maiores, reforçando a importância de algoritmos aproximados para tais casos.

8.2. Desempenho em Recursos Computacionais

Os algoritmos "Twice Around the Tree"e "Christofides"apresentaram um comportamento mais robusto em termos de escalabilidade e uso de recursos. O consumo de RAM pelo "Twice Around the Tree"manteve-se consistentemente alto (em torno de 70-80%), indicando estabilidade e previsibilidade. Já o algoritmo "Christofides"utilizou menos RAM em média, mas apresentou maior variação. Em relação ao uso de CPU, "Twice Around the Tree"mostrou picos de consumo, enquanto "Christofides"manteve um padrão mais uniforme, tornando-o mais eficiente em sistemas com restrições de processamento.

8.3. Qualidade das Soluções

A análise da qualidade das soluções confirmou as expectativas teóricas. O algoritmo "Christofides"obteve consistentemente uma taxa de aproximação próxima de 1,0, demonstrando sua eficiência em gerar soluções de alta qualidade. O "Twice Around the Tree", com uma taxa de aproximação em torno de 1,5, mostrou-se uma alternativa aceitável para cenários onde a precisão é menos crítica. Em contrapartida, o "Branch-and-Bound", quando executado em instâncias menores, garantiu a solução ótima, embora com custos computacionais muito superiores.

8.4. Espaço e Escalabilidade

Os gráficos de espaço utilizado evidenciam um crescimento linear do consumo de memória para os três algoritmos à medida que o número de nós aumenta. Contudo, "Christofides"e "Twice Around the Tree"apresentaram taxas de crescimento quase idênticas, enquanto "Branch-and-Bound"utilizou significativamente menos espaço, limitado pela impossibilidade de execução em instâncias maiores.

9. Conclusão

O algoritmo Branch-and-Bound é capaz de encontrar a solução ótima para o problema, mas apresenta uma alta demanda computacional. Durante os experimentos, ele só conseguiu ser executado em instâncias com até 15 nós; para instâncias maiores, não foi possível concluí-lo devido ao consumo excessivo de tempo e memória. Apesar de garantir a melhor qualidade de solução, sua aplicação prática é limitada a problemas de pequena escala.

O algoritmo Christofides, por outro lado, mostrou-se significativamente mais rápido que o Branch-and-Bound. Embora a qualidade da solução obtida seja inferior, com uma taxa de aproximação de até 1,5, ele se posiciona como a segunda melhor opção em termos de precisão entre os três algoritmos avaliados. Além disso, quando comparado ao Twice Around the Tree, o Christofides demanda um pouco mais de tempo para ser executado, mas entrega soluções de qualidade superior.

Por fim, o Twice Around the Tree destacou-se como o mais rápido dos três algoritmos avaliados, sendo ideal para cenários onde a velocidade de execução é uma prioridade. No entanto, ele apresenta a pior qualidade de solução, ficando atrás tanto do Branch-and-Bound quanto do Christofides. Em termos de espaço utilizado durante a execução, os experimentos indicaram que o Twice Around the Tree empata com o Christofides, confirmando um comportamento semelhante no uso de memória.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein, *Algoritmos: Teoria e Prática*. Elsevier, 3ª edição, 2012.

[2] R. Vimieiro, *Notas de aula de Algoritmos 2*, Departamento de Ciência da Computação, UFMG, 2024.

[3] Python Software Foundation, *Python Standard Library Documentation*. Disponível em: `https://docs.python.org/3/library/index.html`. Acesso em: 28 dez. 2024.