TP2 - Algoritmos II
Soluções para problemas difíceis

Lucas Momede Barreto Rezende[1], **Luiza Sodré Salgado**[2]

[1]Instituto de Ciências Exatas — Universidade Federal de Minas Gerais (UFMG)

[2]Departamento de Ciência da Computação – Universidade de Minas Gerais
Belo Horizonte, Brasil.

lucasmbr@ufmg.br, luiza-salgado@ufmg.br

Abstract. *The Traveling Salesman Problem (TSP) is recognized as one of the most studied optimization problems in the area of computer science. In this vein, this article presents an analysis of the functioning and results of three algorithms that offer solutions, whether exact or approximate, for the TSP. The algorithms covered are: the method based on the Branch and Bound technique, the Twice Around the Tree algorithm and the Christofides method.*

Resumo. *O Problema do Caixeiro-Viajante (Travelling Salesman Problem - TSP) é reconhecido como um dos problemas de otimização mais estudados na área de ciência da computação. Neste viés, esse artigo apresenta uma análise sobre o funcionamento e os resultados de três algoritmos que oferecem soluções, sejam elas exatas ou aproximadas, para o TSP. Os algoritmos abordados são: o método baseado na técnica de Branch and Bound, o algoritmo Twice Around the Tree e o método de Christofides.*

1. Introdução

O estudo sobre o problema do Caixeiro-Viajante é motivado pela aplicação prática do mesmo no contexto de transporte, logística urbana e muito mais. Visando compreender melhor esse problema, esse trabalho apresenta algoritmos que permitem encontrar soluções aproximadas e exatas para esse problema. O algoritmo que visa solucionar o problema com solução exata é uma abordagem feita com o algoritmo de Branch and Bound, que, no pior caso, testa todas as possibilidades, mas usa métodos para cortar soluções não promissoras. Os demais algoritmos são aproximativos, ou seja, mesmo não encontrando uma solução ótima, encontram uma solução que, no pior caso, nunca é 2 ou 1.5 vezes pior que a ótima, isto é, são 2 e 1.5-aproximativos, respectivamente. Esses algoritmos são Twice Around the Tree e Christofides.

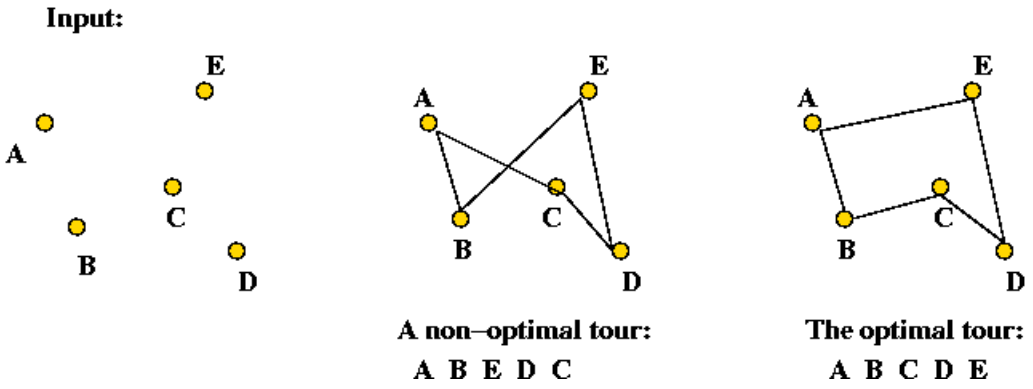

Figure 1. TSP

2. Implementações

A ideia dos algoritmos supracitados é encontrar soluções para o TSP, seja uma solução aproximativa ou exata. Neste viés, foi utilizada a linguagem Python em conjunto com a biblioteca NetworkX, a fim de representar os grafos e proporcionar implementações eficientes de algoritmos como a computação da árvore geradora mínima, DFS e outros que foram usados na implementação. A implementação de cada algoritmo é explicitada detalhadamente a seguir.

2.1. Twice Around The Tree

O algoritmo Twice Around the Tree foi implementado da seguinte maneira: O algoritmo aproximativo recebe como parâmetros um grafo (G) e o peso das arestas (c). Um vértice é selecionado arbitrariamente como raiz r, a partir da qual é computada a árvore geradora mínima (MST) de G, utilizando o algoritmo de Prim. Em seguida, calcula-se, com auxílio da DFS, uma lista de vértices (H) ordenados de acordo com o momento em que são visitados pela primeira vez em um passeio preorder na MST gerada previamente. Por fim, fazendo uso da lista de vértices H, removem-se duplicatas da mesma, gerando assim um circuito hamiltoniano, servirá como o caminho do caixeiro no contexto do TSP euclidiano. Por ser um algoritmo 2-aproximativo, o custo de percorrer esse circuito hamiltoniano não excederá o dobro da solução exata para o problema. Além disso, essa solução é válida, uma vez que o circuito hamiltoniano visita todos os vértices do grafo exatamente uma vez antes de retornar ao ponto de partida.

2.2. Christophies

Semelhantemente ao algoritmo anterior, o algoritmo de Christophies também recebe como parâmetro um grafo (G) e também computa uma árvore geradora mínima (MST) para este grafo. Em seguida, busca-se na MST os vértices que possuem grau ímpar, isto é, os vértices do grafo que tem um número ímpar de arestas ligadas nele. Com esses vértices armazenados (I), calcula-se M, um matching perfeito de peso mínimo no subgrafo induzido por I. Pelo lema do aperto de mão, I possui um número par de arestas, assim é feita essa verificação antes de prosseguir para a próxima etapa, o qual é a construção de G', um multigrafo formado com os vértices do grafo original e arestas de M e T. Em seguida, é computado um circuito euleriano em G' utilizando a busca em profundidade (DFS). Por fim, um circuito hamiltoniano é derivado do circuito euleriano por meio da

eliminação de vértices duplicados, substituindo subcaminhos da forma u-w-v por arestas diretas u-v (shortcutting).

2.3. Branch and Bound

Diferentemente dos anteriores, o algoritmo de Branch and Bound entrega a solução ótima exata. Ele explora o espaço de soluções de forma sistemática, um vértice por vez. Essas várias soluções estão no que é chamado árvore de busca. O algoritmo começa ordenando os pesos de cada aresta. Ele cria também um caminho qualquer inicial para possivelmente conseguir cortar mais ramos. Depois ele começa a explorar a árvore, calculando um limite inferior para cada caminho. Nessa implementação, a árvore é explorada usando uma fila de prioridade, então os caminhos mais promissores são verificados primeiro. O limite, também chamado de bound é usado para evitar pesquisas desnecessárias. Se o bound é maior que a melhor solução que já temos, esse caminho e todos que o utilizam podem ser ignorados. Esse limite é calculado pensando que se deve sempre chegar e sair de um vértice, então, a estimativa é o teto das somas das duas arestas de menor peso incidentes em cada vértice divido por 2, pois são contadas duas vezes.

3. Experimentos e Resultados

Para testar o desempenho e corretude dos algoritmos, foram utilizados todos os testes da categoria euclideano bidimensional disponibilizados na TSPLIB. Os resultados obtidos para cada algoritmo, incluindo proximidade da solução aproximativa e desempenho de tempo em relação ao número de nós, são apresentados a seguir.

3.1. Algoritmos Aproximativos

Apesar da dimensão do problema TSP (número de nós) não ser o único fator que influencia no tempo de execução, é possível notar que o aumento do mesmo está correlacionado com o aumento do tempo de execução.

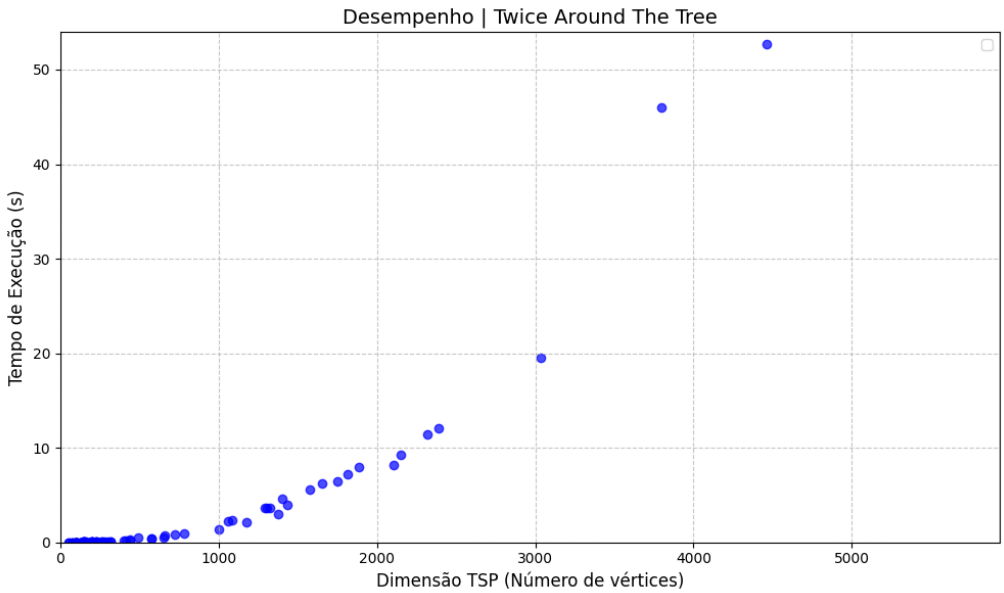

Figure 2. Desempenho Twice Around The Tree

Todavia, conforme é perceptível na Figura 3, que mostra o desempenho do algoritmo de Christofides, é evidente que existem casos em que, apesar de haver um número menor de nós, o tempo necessário para resolver o problema é maior.

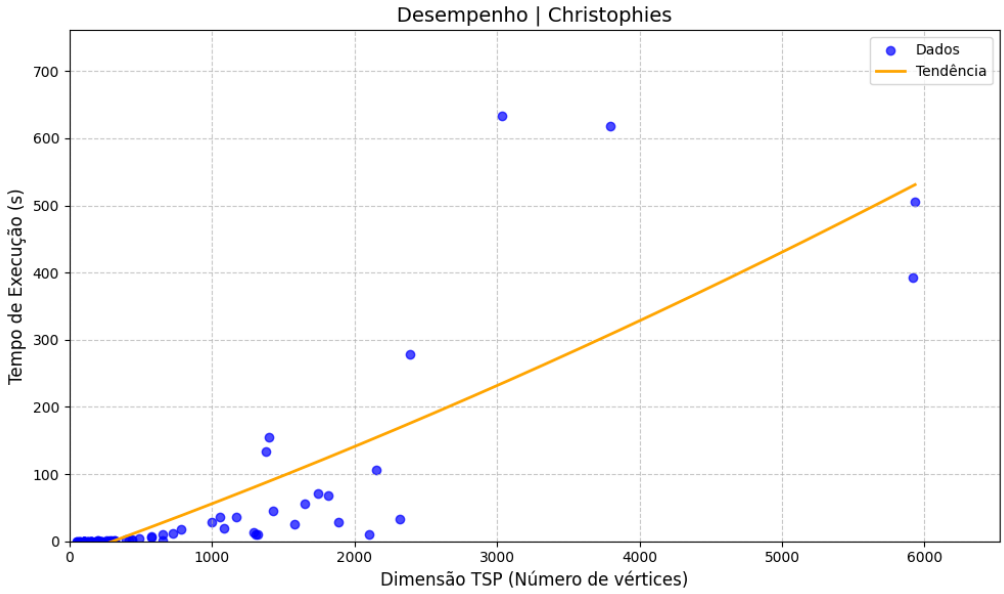

Figure 3. Desempenho Christophies

Outrossim, mesmo nesses casos, é notável que o algoritmo de Christofides precisa de mais tempo de execução do que o algoritmo Twice Around The Tree. Isso está conforme as suas complexidades, que são, respectivamente, $O(n^3 \log n)$ e $O(E + VlogV)$.

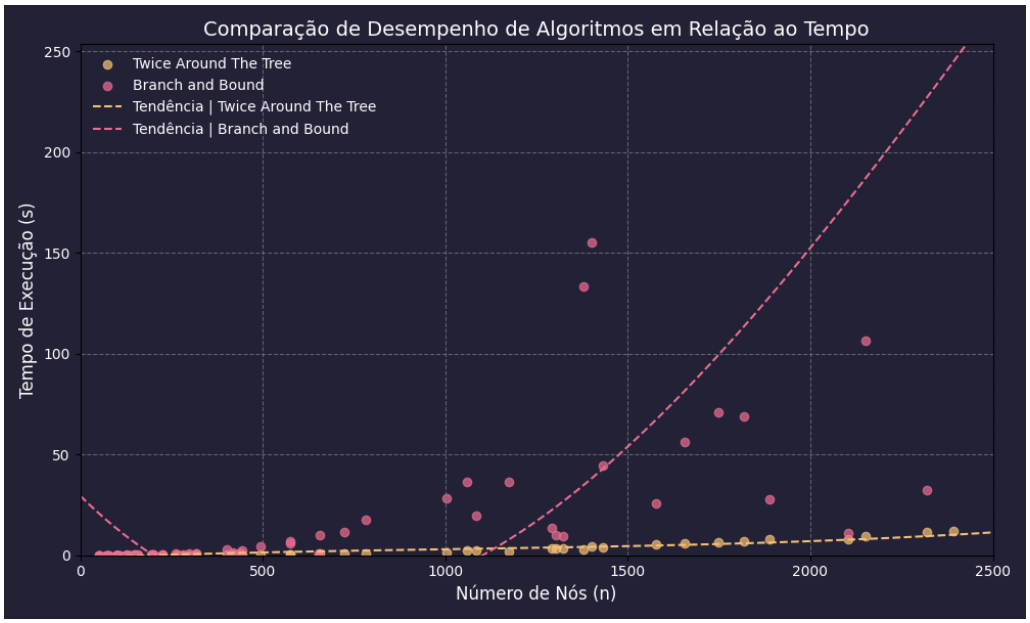

Figure 4. Comparação entre algoritmos aproximativos

3.2. Branch and Bound

Esse algoritmo possui uma complexidade fatorial em relação ao número de nós, isto é, $O(n! \log n)$. Ele possui essa complexidade, no pior caso, devido ao número das permutações de rotas que cresce fatorialmente conforme a exploração. O desempenho variado para intâncias de tamanho próximo é devido aos métodos e heurísticas para calcular os limites (bound), que podem funcionar muito bem para certos grafos, cortando várias soluções da busca ou não ser tão eficiente e precisar explorar mais do espaço. Nesse sentido, não foi possível executar nenhuma instância da TSPLIB em tempo hábil (menos que 30 minutos) através desse algoritmo.

3.3. Análise de memória

Visando analisar o uso de memória em cada algoritmo, foram realizados testes com instâncias do TSP modificadas, variando de 4 a 14 nós. Observou-se que o algoritmo de Branch and Bound consome significativamente mais memória, devido à necessidade de permutar várias soluções e armazenar estados intermediários. Em contrapartida, o algoritmo Twice Around the Tree apresenta o menor consumo de memória entre os algoritmos analisados, pois ele calcula a árvore geradora mínima e realiza um passeio preorder simples para construir o ciclo hamiltoniano, que será o caminho utilizado pelo caixeiro viajante. Ademais, o algoritmo de Christophies entrega uma performance em memória intermediária, haja vista que, além da construção da árvore geradora mínima, é necessária uma análise nos vértices de grau ímpar.

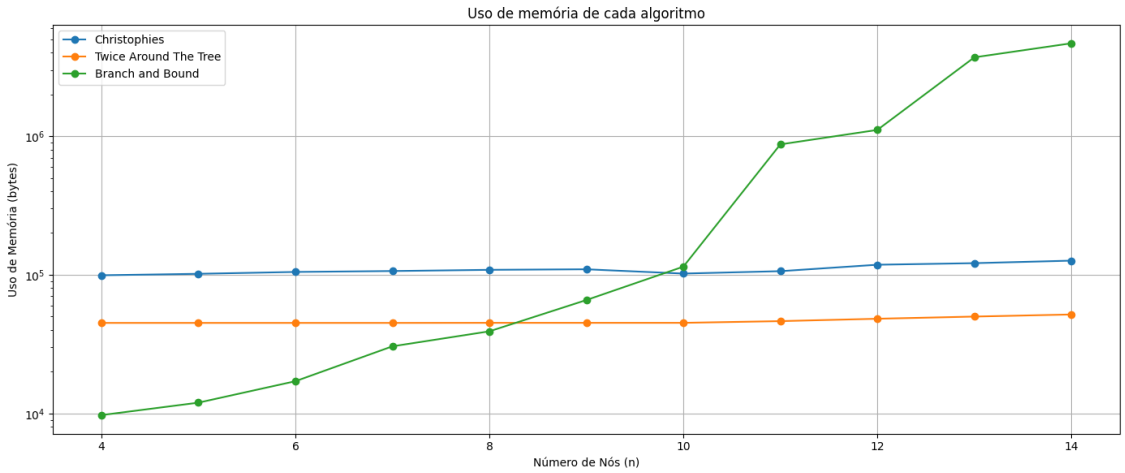

Figure 5. Uso de memória

3.4. Verificação de aproximação

Para ambos os algoritmos, as soluções aproximativas apresentadas geralmente não se aproximaram do limite superior teórico de aproximação, ou seja, mesmo que no pior apresentem valores muito discrepantes da solução ótima, dependendo da natureza do problema em questão, os algoritmos aproximativos oferecem um equilíbrio entre a eficiência computacional e a qualidade da solução, representando um trade-off relevante entre poupar recursos e tempo de execução e a busca por uma solução exata.

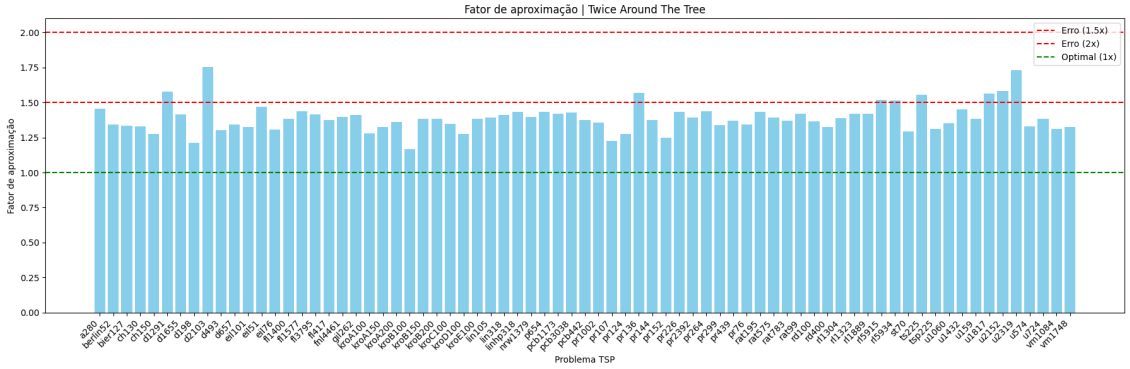

Figure 6. Aproximação do Twice Around The Tree

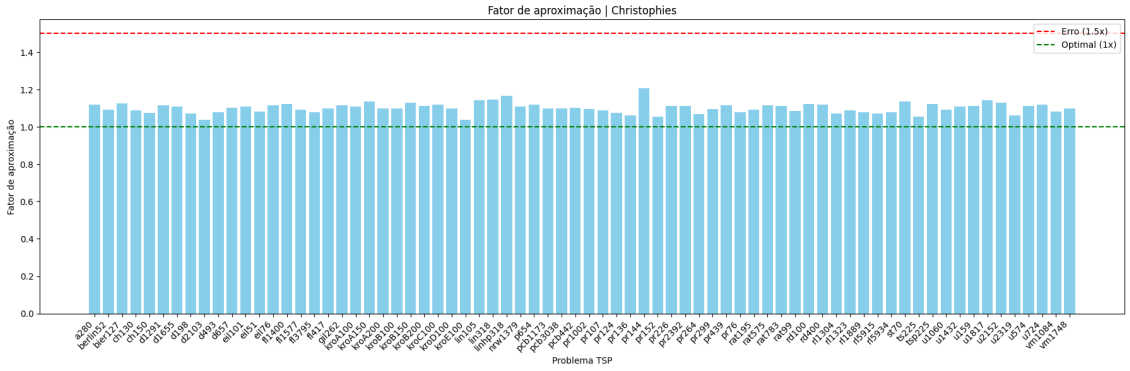

Figure 7. Aproximação do Christophies

4. Conclusão

Neste trabalho, foram analisadas e implementadas três abordagens distintas para o problema do Caixeiro Viajante (TSP): o algoritmo exato Branch and Bound e os algoritmos aproximativos Christofides e Twice Around The Tree. Cada um apresenta vantagens e limitações que os tornam adequados para diferentes contextos e aplicações. Em particular, o algoritmo Branch and Bound, sendo uma abordagem exata, é indicado para situações onde a obtenção da solução ótima é indispensável, especialmente em cenários em que o tempo de execução não é uma restrição crítica. Além disso, diferentemente dos algoritmos aproximativos, ele pode ser aplicado em grafos que não satisfazem a desigualdade triangular. Por outro lado, os algoritmos aproximativos oferecem soluções eficientes em termos de tempo, mesmo para instâncias maiores. Como demonstrado na figura 7, o algoritmo de Christofides é uma boa escolha para problemas de grande escala e garante uma solução no máximo 1.5 vezes o ótimo. Já o algoritmo Twice Around the Tree, embora menos preciso, com uma garantia de aproximação de até 2 vezes o custo ótimo, apresenta-se como uma alternativa mais simples e rápida de implementar, sendo ideal para aplicações onde simplicidade e velocidade são prioridades.

References

[1] *Approximation Algorithm*. Disponível em: https://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2028%20-%20Approximation%20Algorithm.htm. Acesso em: 13 jan. 2025.

[2] *Christofides Algorithm*. Disponível em: https://en.wikipedia.org/wiki/Christofides_algorithm. Acesso em: 13 jan. 2025.