Análise de Algoritmos para o Problema do Caixeiro Viajante

Brisa do Nascimento Barbosa[1]

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

brisabn@ufmg.br

Abstract. *This work addresses the implementation and analysis of algorithms for solving the classic Traveling Salesman Problem (TSP). For the search of exact solutions, we investigate the Branch and Bound algorithm, while for approximate approaches, we apply the Christofides and Twice Around the Tree algorithms.*

Resumo. *Este trabalho aborda a implementação e análise de algoritmos para a resolução do clássico problema do caixeiro viajante (TSP). Para a busca de soluções exatas, investigamos o algoritmo Branch and Bound, enquanto para abordagens aproximativas, utilizamos os algoritmos de Christofides e Twice Around the Tree.*

1. Introdução

O Problema do Caixeiro Viajante (TSP) é um clássico exemplo de problema NP-completo de otimização combinatória em grafos. A formulação matemática do TSP pode ser descrita da seguinte maneira:

Dado um grafo não direcionado $G = (V, E)$ com vértices V representando cidades e arestas ponderadas E representando as distâncias entre as cidades, o objetivo é encontrar um ciclo hamiltoniano de custo mínimo. Isso implica minimizar a seguinte expressão:

$$\min \left(\sum_{i=1}^{n-1} w(v_i, v_{i+1}) + w(v_n, v_1) \right) \tag{1}$$

onde n é o número de vértices no grafo, $w(v_i, v_{i+1})$ é o peso da aresta entre a cidade v_i e a cidade v_{i+1}, e $w(v_n, v_1)$ é o peso da aresta que conecta a última cidade à primeira no ciclo. [Cormen et al. 2023]

Além disso, consideramos que estamos tratando de instâncias euclidianas, dados pelas intâncias da biblioteca TSPLIB95, onde as distâncias entre as cidades são medidas na métrica euclidiana.

Perceba que a complexidade do TSP em força-bruta é fatorial, pois para n cidades, existem (n - 1)! rotas possíveis e, para cada rota, teríamos calcular o comprimento total das viagens e ver qual delas tem o menor comprimento. Portanto, no presente trabalho, buscamos abordagens que reduzam essa complexidade. Para tal, empregamos o algoritmo exato *Branch and Bound*, que visa encontrar a solução ótima, bem como os algoritmos aproximativos Christofides e *Twice Around the Tree*.

2. Utilitários

As implementações foram desenvolvidas no ambiente Python 3.11, utilizando principalmente as bibliotecas *NetworkX* e tsplib95 para manipulação e análise de grafos. Para dar uma base de comparação do desempenho do sistema, o ambiente operacional utilizado é o Windows 11. A configuração de hardware envolvida no desenvolvimento inclui 8GB de memória RAM e um processador Intel Core i7.

3. Implementação dos Algoritmos

3.1. Algoritmos exatos

Os algoritmos exatos são uma classe de algoritmos que fornecem a solução ótima para um problema. No contexto do Problema do Caixeiro Viajante, esses algoritmos exploram o espaço de soluções de maneira sistemática e garantem a obtenção da rota mais curta possível. Assim, eles se baseiam em procedimentos determinísticos para garantir precisão e confiabilidade.

3.1.1. Branch-and-Bound

O método de *Branch-and-Bound* é uma técnica que constrói candidatos à solução de forma incremental, e descarta conjuntos de candidatos que não podem produzir uma solução melhor do que a melhor já encontrada. Esse processo é feito por meio de uma estratégia de ramificação (*branch*) divide subproblemas e uma estratégia de limitação (*bound*) que poda candidatos. O cerne do método consiste em uma enumeração sistemática de soluções candidatas por meio de uma busca no espaço de estados: o conjunto de soluções candidatas é pensado como formando uma árvore enraizada com o conjunto completo na raiz. No entanto, pode exigir a exploração de todas as permutações possíveis no pior caso.

No Problema do Caixeiro Viajante, o método *Branch-and-Bound* começa com uma rota inicial e gera rotas possíveis (*branching*). Rotas com custo maior que a melhor rota encontrada são descartadas (*bounding*). Esse processo continua até que todas as rotas possíveis sejam exploradas ou descartadas, resultando na rota de menor custo. Implementamos por uma abordagem recursiva, explorando um nível abaixo da árvore de soluções a cada iteração. Durante cada iteração, o caminho atual e o nível da árvore atual são armazenados e prosseguimos com verificações para podar ramos da árvore de recursão que não podem levar a uma solução melhor do que a melhor solução atual. Para tal, empregamos a seguinte estimativa:

Limite inferior (*Lower bound*): calcula-se a soma dos pesos das duas arestas de menor peso incidentes em cada vértice, dividida por dois para evitar a contagem duplicada de cada aresta.

Assim, para cada nó no grafo, as duas arestas mais leves conectadas a ele são identificadas e os pesos dessas arestas são armazenados em uma lista. Adicionalmente, a técnica de memoização é utilizada para criar um dicionário que memoriza soluções,

evitando recálculos desnecessários e garantindo que subproblemas idênticos não sejam resolvidos novamente, otimizando o tempo. A função recursiva é encerrada quando todos os nós foram visitados, e a solução final, que é custo e o melhor circuito, é retornada.

Destacamos aqui que usamos a estratégia de *best-first*, o que significa que o algoritmo seleciona, a cada passo, o nó mais promissor para explorar primeiro [Russell and Norvig 2009]. No código desenvolvido, a abordagem *best-first* é adotada ao ordenar os vizinhos de cada nó com base no peso de suas arestas. Essa ordenação permite que o algoritmo priorize a exploração dos vizinhos mais promissores, ou seja, aqueles com as arestas mais leves. A escolha dessa abordagem visa acelerar a convergência do algoritmo, focando nas regiões do espaço de solução mais propensas.

3.2. Algoritmos Aproximativos

Algoritmos aproximativos são projetados para encontrar soluções aproximadas, especialmente em situações em que encontrar a solução exata é computacionalmente inviável ou consome muito tempo. O fator de aproximação é a medida que avalia o desempenho de um algoritmo aproximativo em relação à solução ótima. Ele é representado pelo menor valor de c tal que $r(I) \leq c \cdot OPT(I)$ permanece válida para todas as instâncias I, sendo r o resultado do algoritmo e OPT a solução ótima. Quanto mais próximo de 1 for esse fator, melhor é o desempenho do algoritmo aproximativo. [Goodrich and Tamassia 2015]

3.2.1. Twice-around-the-tree

O *Twice-around-the-tree* é uma abordagem gulosa para o problema do caixeiro viajante que garante uma solução 2-aproximada [Cormen et al. 2023]. Isso significa que a rota encontrada por essa heurística será, no máximo, duas vezes mais longa do que a rota ótima. A heurística utilizada é a *Nearest Neighboour*, em que o algoritmo começa em um ponto arbitrário e, em cada etapa, escolhe o ponto mais próximo que ainda não foi visitado, formando assim uma rota.

No código implementado, inicialmente é calculada a árvore geradora mínima do grafo (MST), com isso temos o peso total da MST como o limite inferior do percurso ótimo. A partir do primeiro vértice da MST, o algoritmo realiza uma busca em profundidade (DFS) para gerar as arestas na ordem em que foram visitadas durante a busca. Essas arestas são então utilizadas para formar um circuito hamiltoniano, que representa a rota que o caixeiro viajante seguirá. Considerando instâncias euclidianas, o circuito final não será mais que duas vezes o peso da MST.

Na implementação realizada, foram baseadas e utilizadas funções da biblioteca *NetworkX*. Para a construção da árvore geradora mínima, desenvolvemos um algoritmo simplificado baseado no *source* do algoritmo de Prim da biblioteca. Este algoritmo encontra uma subestrutura de árvore que conecta todos os vértices de um grafo, minimizando a soma dos pesos das arestas. Mais especificamente, o método de Prim inicia com um vértice arbitrário e, de maneira iterativa, adiciona a aresta de menor peso que conecta um vértice da árvore a um vértice fora dela, expandindo assim a árvore até que todos os vértices estejam inclusos. Para a busca em profundidade (DFS), optamos por utilizar a

função *dfs preorder nodes*. Este método executa um caminho em profundidade no grafo, visitando os nós em uma ordem específica denominada "*preorder*", que indica que um nó é visitado antes de seus descendentes.

3.2.2. Algoritmo de Christofides

O algoritmo de Christofides foi proposto em 1976 por Nicos Christofides e é notável por sua garantia de fornecer uma solução dentro de um fator de 3/2 da solução ótima [Cormen et al. 2023].

A implementação que desenvolvemos do algoritmo começa calculando a MST do grafo, utilizando o método de Prim explicitado anteriormente. Pela MST, são identificados os vértices de grau ímpar na MST, ou seja, aqueles que têm um número ímpar de arestas incidentes. Com isso, usando a função *min weight matching* da biblioteca *NetworkX*, executamos emparelhamento perfeito mínimo induzido pelos vértices de grau ímpar. é então encontrado entre esses vértices, formando um grafo euleriano.

Um emparelhamento perfeito mínimo é um conjunto de arestas em um grafo que não compartilham vértices e cuja soma total dos pesos das arestas é a menor possível. A função utilizada é baseado no algoritmo de Edmonds. Este algoritmo começa com um emparelhamento vazio e busca caminhos aumentadores, identificando ciclos ímpares conhecidos como *blossoms*. Dessa forma, tenta melhorar iterativamente o emparelhamento escolhendo caminhos aumentadores até que não seja mais possível aumentar o emparelhamento.

Em seguida, o algoritmo combina as arestas da árvore geradora mínima e do acoplamento perfeito para formar um multigrafo. Um multigrafo é um grafo que permite múltiplas arestas entre quaisquer dois vértices. Neste multigrafo, cada vértice tem grau par, o que é uma propriedade necessária para a existência de um circuito Euleriano.

Para encontrar o circuito Euleriano neste multigrafo, utilizamos a função *eulerian circuit* da biblioteca *NetworkX*, que é uma adaptação de um algoritmo linear descrito por J. Edmonds e E. L. Johnson. Por fim, o algoritmo transforma o circuito Euleriano em um circuito Hamiltoniano. Isso é feito ignorando vértices repetidos no circuito Euleriano.

4. Experimentos

Propomos uma abordagem em dois estágios para a avaliação de tempo dos algoritmos, onde primeiro comparamos os algoritmos aproximativos e, em seguida, avaliamos o desempenho do algoritmo exato em comparação com eles. Subsequentemente, analisaremos as complexidades espaciais associadas a eles.

4.1. Comparação entre os algoritmos aproximados

A **Tabela 1** apresenta uma análise comparativa dos tempos de execução (em segundos) de dois algoritmos, Christofides e *Twice Around the Tree*, aplicados a diferentes instâncias compiladas pela TSPLIB95, uma biblioteca específica em Python para lidar com instâncias do TSP.

Tabela 1. Comparação de tempo (em segundos)

Instância	Nós	Christofides	Twice Around
berlin52	52	0.014299	0.005899
rat99	99	0.049826	0.007862
ch130	130	0.098630	0.036961
pr264	264	0.276245	0.110012
d493	493	3.982193	0.284765
rat575	575	5.169682	0.453924
rat783	783	13.690768	0.954042
u1060	1060	33.213636	3.476558
fl1400	1400	35.610183	2.729183
u1432	1432	48.423620	2.617007
fl1577	1577	39.610183	4.922160
vm1748	1748	55.492058	8.959156
u2152	2152	61.782740	15.235313
pcb2038	3038	NA	28.868293

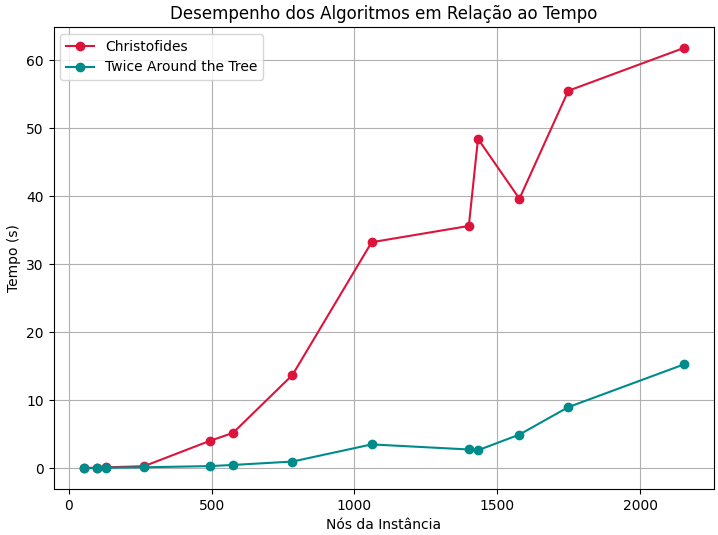
Ao examinar os resultados, é evidente que o desempenho dos algoritmos varia significativamente com base no tamanho da instância. Em instâncias menores, como “berlin52”e “rat99”, as diferenças nos tempos entre os dois métodos são marginais. Isso sugere que, para problemas de menor escala, ambas as abordagens apresentam desempenho comparável, e a escolha entre elas pode depender de outros critérios, como precisão da solução ou facilidade de implementação.

À medida que a quantidade de nós aumenta, algoritmo de Christofides, embora eficaz para instâncias menores, mostra tempos de execução substancialmente superiores. Em contraste, o método *Twice Around the Tree* apresenta uma escalabilidade mais favorável, tornando-se particularmente eficiente para instâncias maiores, como evidenciado nas instâncias “u1060”, “fl1400”e “vm1748”. Percebe-se também que, para a instância “pcb2038”, não temos resultado para o algorimo de Christofides, indicando que ultrapassou o limite máximo de execução de 30 minutos. Isso sugere a possível dificuldade desse algoritmo em instâncias de grande porte.

É importante ressaltar também que o tempo de execução de um algoritmo não depende apenas do número de nós, mas também das características específicas de cada instância. Por exemplo, observou-se que a instância “fl1577”, apesar de ter um número um pouco menor de nós, foi resolvida mais rapidamente do que a instância “u1432”. Isso sugere que a estrutura específica da instância pode influenciar o desempenho do algoritmo, possivelmente devido a fatores como a distribuição de pesos nas arestas. Além disso, em outros experimentos, não apresentados aqui, foram observadas discrepâncias em tempos de execução entre instâncias com um número similar de nós. Isso reforça que o desempenho do algoritmo é dependente tanto do tamanho quanto características da instância.

Como podemos observar pelo **Gráfico 1**, o algoritmo de Christofides apresenta um crescimento significativamente mais elevado em comparação com o *Twice Around the Tree* . De fato, a complexidade de tempo do Christofides implementado é dominada pelo algoritmo de Edmonds, que tem complexidade $\mathcal{O}(n^3)$. Já o *Twice Around the Tree* possui complexidade de tempo $\mathcal{O}(E + VlogV)$, já que é dominada pelo algoritmo de Prim.

Figura 1. Gráfico de desempenho de tempo



Apresentamos agora o **Gráfico 2**, cujos resultados foram extraídos da **Tabela 2**, que compara os fatores de aproximação. Observou-se que as aproximações ficaram bem abaixo do fator de aproximação definido, demonstrando um comportamento bastante controlado. Para Christofides, a média entre os valores analisados foi 1.12, sendo limite estabelecido de 1.5. Para *Twice Around de Tree*, constatamos que a média foi de 1.40, sendo limite 2. Essa consistência reforça a eficácia da métrica em ambos algoritmos.

Figura 2. Gráfico de Fatores de Aproximação

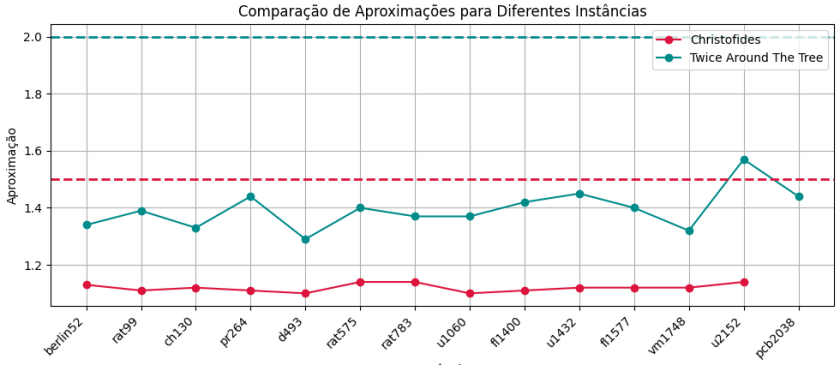


Tabela 2. Comparação de aproximação

Instância	Nós	Christofides		Twice Around	
		Custo	Aprox.	Custo	Aprox.
berlin52	52	8560	1.13	10114	1.34
rat99	99	1350	1.11	1679	1.39
ch130	130	6841	1.12	8129	1.33
pr264	264	54748	1.11	70619	1.44
d493	493	38511	1.10	45087	1.29
rat575	575	7737	1.14	9497	1.40
rat783	783	10068	1.14	12065	1.37
u1060	1060	247182	1.10	306855	1.37
fl1400	1400	22327	1.11	28632	1.42
u1432	1432	171579	1.12	222322	1.45
fl1577	1577	24922	1.12	31112	1.40
vm1748	1748	377237	1.12	443969	1.32
u2152	2152	73407	1.14	100564	1.57
pcb2038	3038	NA	NA	198744	1.44

4.2. Comparação entre o algoritmo exato e os aproximativos

O *Branch and Bound*, em comparação com os algoritmos aproximativos, tem a maior complexidade: no pior caso, o código ainda tem que explorar todas as possíveis permutações de nós, logo, $\mathcal{O}(n!)$.

Diante disso, analisaremos para grafos que geramos aleatoriamente (semente = 128 para geração aleatória de pesos). Percebemos, pela **Tabela 3**, que com 24 vértices, o *Branch and Bound* já estagnou. Isso é justificável, uma vez que o pior caso para $n = 20$ implica $(n-1)!$ possíveis circuitos hamiltonianos, ou seja, $19! = 121645100408832000$.

Tabela 3. Comparação de tempo (em segundos) com semente = 128

Nós	Branch and Bound	Christofides	Twice Around
8	0.001513	0.000998	0.000000
12	0.070174	0.002074	0..000000
16	13.800819	0.002387	0.000222
20	30.417816	0.003640	0.001001
24	NA	3.982193	0.284765

Na **Tabela 4**, realizamos uma comparação entre os algoritmos para duas pequenas instâncias do TSPLIB95: “burma14” e “ulysses16”. No caso de “burma14”, o *Branch and Bound* encontrou a solução ótima, mas em tempo bem superior comparado aos aproximados. Para “ulysses16”, mesmo sendo uma instância pequena, o algoritmo de *Branch and Bound* não encontrou a solução dentro do tempo limite, enquanto os métodos aproximativos apresentaram resultados satisfatórios, com destaque ao Christofides, que obteve uma aproximação de 1.02.

Tabela 4. Comparação dos aproximativos com Branch and Bound

Instância	Nós	Branch and Bound		Twice Around		Christofides	
		Tempo(s)	Aprox.	Tempo(s)	Aprox.	Tempo(s)	Aprox.
burma14	14	11.745442	1	0.000929	1.15	0.002000	1.09
ulysses16	16	NA	1	0.003471	1.15	0.004499	1.02

Considerando que a maioria instâncias do TSPLIB95 são dados de ordem igual ou superior a 10^2, evidencia-se que o algoritmo *Branch and Bound* torna-se ineficiente para resolvê-las, sendo útil apenas para conjuntos de dados bastante reduzidos.

Portanto, embora esse método garanta encontrar a solução ótima, sua eficiência é questionável, podendo não dar resultados em tempo hábil para instâncias mínimas.

4.3. Comparação geral entre custo espacial

A complexidade temporal dos algoritmos aproximativos é dominada pela representação de grafos, que é $\mathcal{O}(E + V)$. Já o *Branch and Bound*, no pior caso, pode ter uma complexidade exponencial $\mathcal{O}(2^v)$, em vista que a função recursiva gera uma chamada para cada possível caminho no grafo.

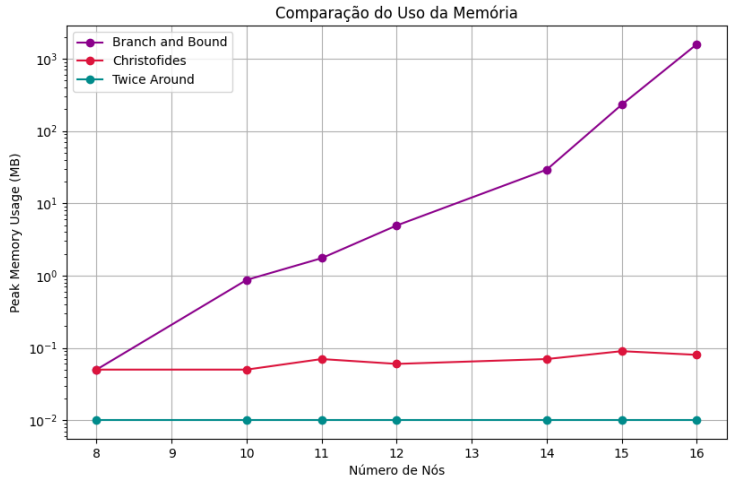
Tabela 5. Peak memory usage (em MB) com semente = 128

Nós	Branch and Bound	Christofides	Twice Around
8	0.05	0.05	0.01
10	0.87	0.05	0.01
11	1.74	0.07	0.01
12	4.93	0.06	0.01
14	29.21	0.07	0.01
15	231.24	0.09	0.01
16	1585.29	0.08	0.01

No **Gráfico 3**, expomos num gráfico com escala logarítmica as comparações de memória da **Tabela 5** . Com isso, podemos observar claramente que o *Branch and Bound*, com sua complexidade exponencial no pior caso, mostra crescimento pronunciado de memória com o aumento de nós, enquanto os algoritmos Christofides e *Twice Around the Tree* demonstram eficiência significativa, mantendo um consumo de memória notavelmente inferior.

Assim, a escolha entre essas abordagens não se resume apenas à precisão da solução, mas também à consideração prática da eficiência computacional, o que torna os métodos aproximativos mais interessantes para o TSP em termos de uso eficiente de recursos.

Figura 3. Gráfico de desempenho de Espaço



5. Conclusão

Após a implementação e análise de resultados, notamos que o *Branch and Bound* é capaz de encontrar a solução ótima, mas tem uma complexidade exponencial e se torna inviável para instâncias maiores. Os algoritmos aproximativos, por outro lado, fornecem soluções suficientemente boas em tempo polinomial, sendo o *Twice Around the Tree* o mais rápido e o Christofides mais preciso.

Portanto, concluímos que a escolha do algoritmo depende do equilíbrio desejado entre a precisão da solução e o tempo de execução, além do tamanho e das características da instância. No caso deste trabalho, para as instâncias do TSPLIB95, o algorimo a mais destacável foi o *Twice Around The Tree*, dado o tempo de execução em menos de um minuto para todos os casos avaliados, eficiente uso da memória, e implementação simples.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2023). *Introduction to Algorithms*. The MIT Press, 4th edition.

Goodrich, M. T. and Tamassia, R. (2015). *Algoritmos - Estruturas de Dados e Programação Orientada a Objetos*. Wiley, 5th edition.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition.