Relatório do Trabalho Prático

Caroline C. Carvalho[1], **Déborah B. Yamamoto**[1]

[1]Universidade Federal de Minas Gerais - UFMG

carvalho-campos@ufmg.br, dyamamoto@ufmg.br

Abstract. *This work presents the implementation and analysis of three strategies to solve the Euclidean Traveling Salesman Problem, focusing on one exact method and two heuristic algorithms. The research explores the Branch-And-Bound algorithm, an exact approach that seeks the optimal solution but with high time complexity and significant memory consumption, especially for large instances. In addition, two heuristic algorithms are discussed: Twice-Around-The-Tree and Christofides' algorithm, both offering approximate solutions with approximation factors of 2 and 1.5, respectively. The experimental results show that while Branch-And-Bound provides an exact solution, its feasibility is limited in terms of time for large instances. On the other hand, the heuristic algorithms stand out for their time efficiency. The choice of the most suitable algorithm depends on the problem's priorities, such as accuracy or temporal efficiency.*

Resumo. *Este trabalho apresenta a implementação e análise de três estratégias para resolver o Problema do Caixeiro Viajante Euclidiano, com foco em um método exato e dois algoritmos heurísticos. A pesquisa explora o algoritmo Branch-And-Bound, uma abordagem exata que busca a solução ótima, mas com alta complexidade no tempo de execução e elevado consumo de memória, especialmente para instâncias grandes. Além disso, são abordados dois algoritmos heurísticos: Twice-Around-The-Tree e o algoritmo de Christofides, ambos oferecendo soluções aproximadas com fatores de aproximação de 2 e 1,5, respectivamente. Os resultados experimentais evidenciam que, enquanto o Branch-And-Bound oferece uma solução exata, sua viabilidade é limitada em termos de tempo para instâncias grandes, enquanto os algoritmos heurísticos, se destacam pela eficiência em tempo. A escolha do algoritmo mais adequado depende das prioridades do problema, como a precisão ou a eficiência temporal.*

1. Introdução

Robert et al. (2019) apresenta um estudo que destaca o uso do Problema do Caixeiro Viajante (TSP, do inglês Traveling Salesperson Problem) na otimização de rotas para drones, evidenciando sua importância na resolução de desafios logísticos. Esse trabalho reforça a relevância do TSP não apenas no campo teórico da ciência da computação, mas também em aplicações práticas. Essa importância torna-se ainda mais evidente diante de avanços tecnológicos, como os planejados pela China, que prevê a circulação de 100.000 veículos voadores, incluindo táxis, vans de entrega e carros familiares, em suas cidades nos próximos seis anos[The Sun 2024]. Para viabilizar esse cenário futurista, algoritmos eficientes baseados no TSP desempenham um papel fundamental ao otimizar rotas e garantir operações sustentáveis e eficazes desses novos meios de transporte.

O TSP consiste em encontrar o menor percurso que permite a um viajante visitar um conjunto de cidades exatamente uma vez, retornando à cidade de origem [Pop et al. 2024]. Apesar de sua formulação aparentemente simples, ele pertence à classe de problemas NP-difíceis, o que significa que não existe um algoritmo conhecido que resolva todas as instâncias de forma eficiente no pior caso. Essa característica o torna um dos problemas mais desafiadores e estudados na ciência da computação, sendo amplamente utilizado como referência para o desenvolvimento de novos métodos de resolução.

Historicamente, a resolução do TSP foi abordada por métodos exatos, como a força bruta, que avalia todas as possíveis combinações de rotas, e a implementação com *Branch-And-Bound*, que utiliza estratégias mais eficientes, embora ainda limitadas a instâncias pequenas. Com o avanço da tecnologia, surgiram técnicas heurísticas, como o algoritmo twice-around-the-tree e o algoritmo de Christofides, que, apesar de não garantirem soluções ótimas, oferecem respostas 2-aproximadas e 1.5-aproximadas, respectivamente.

O impacto do TSP vai além de problemas acadêmicos. Ele encontra aplicações práticas em diversas áreas, como planejamento de rotas para transporte [Muriyatmoko et al. 2024], logística de armazéns, design de circuitos eletrônicos e até mesmo na análise de sequências genômicas. Sua relevância em campos tão variados demonstra a importância de buscar soluções cada vez mais eficientes para esse problema, especialmente em um mundo onde a otimização de recursos é fundamental para o desenvolvimento sustentável.

Neste trabalho, é apresentada a implementação de diferentes estratégias para resolver o Problema do Caixeiro Viajante Euclidiano, abrangendo um método exato e dois heurísticos. Além de descrever os algoritmos, há a realização de uma análise comparativa de desempenho, considerando aspectos como precisão, espaço e tempo de execução. Assim, pretende-se não apenas implementar soluções para o problema, mas realizar as análises para as instancias de teste disponibilizadas .

2. Descrição dos Algorítmos

A apresentação de decisões tomadas, bem como complexidade de cada implementação é explicada a seguir. Entender esses pontos permite realizar uma melhor análise posterior dos resultados obtidos nos experimentos.

2.1. Branch-and-Bound

As decisões tomadas para a implementação do algoritmo *Branch-And-Bound* aplicado ao Problema do Caixeiro Viajante, bem como a sua análise complexidade são apresentadas a seguir. A estimativa de custo escolhida segue o mesmo modelo abordado em aula. Quando há arestas ainda não selecionadas, opta-se pelas duas menores conexões disponíveis. Se o vértice atual formar uma aresta com o anterior, essa aresta será fixada, juntamente com a menor aresta restante. Todas essas arestas são somadas e o resultado é dividido por dois, arredondando para cima. Esse cálculo garante uma estimativa viável para o custo mínimo restante, ajudando a podar ramos menos promissores do espaço de busca.

O algoritmo utiliza duas principais estruturas de dados: matriz de adjacência e listas auxiliares. A matriz de adjacência armazena as distâncias entre os vértices, per-

mitindo acesso rápido e evitando recalcular as mesmas distâncias repetidamente, o que reduz a complexidade computacional. As listas, por sua vez, são empregadas para marcar os vértices visitados e armazenar as duas menores conexões de cada vértice. Dessa forma, evita-se a necessidade de realizar múltiplos cálculos para encontrar os menores valores em cada iteração.

No que diz respeito à estratégia de busca, houve dificuldades na implementação do Best-First Search. Durante os testes, ao inserir os elementos no heap, a memória disponível era insuficiente para armazenar todas as combinações possíveis, levando ao estouro da memória e impossibilitando a continuidade da execução. Em contrapartida, ao utilizar a Depth-First Search, o desempenho do algoritmo foi prejudicado, uma vez que essa abordagem sempre explora os caminhos até a folha da árvore antes de retroceder, enquanto poderia ter seguido por um melhor, resultando em um tempo de execução significativamente maior.

Embora seja uma abordagem mais eficiente do que a força bruta, o algoritmo ainda apresenta uma complexidade de tempo desfavorável no pior caso. Isso ocorre porque, se nenhum ramo for podado, ele precisará explorar todas as permutações possíveis, resultando em uma complexidade de tempo de $O(n!)$. Por outro lado, a complexidade de espaço é dominada pela matriz de adjacência, que exige $O(n^2)$. Além disso, há um custo adicional de $O(n)$ para o vetor de visitado e de $O(n)$ para a pilha de chamadas recursivas. Assim, a complexidade total de espaço é $O(n^2)$, com a matriz de adjacência sendo o fator dominante.

2.2. Twice-Around-The-Tree

Nesta seção, será apresentado um panorama geral do funcionamento do algoritmo Twice Around The Tree (TATT), seguido pelas escolhas de implementação feitas durante sua construção. O algoritmo TATT é um algoritmo 2-aproximado para o problema do caixeiro viajante, essencialmente, o algoritmo Twice Around The Tree pode ser dividido em três etapas principais.

A primeira etapa do algoritmo consiste na construção de uma Árvore Geradora Mínima (AGM). Para isso, o algoritmo recebe uma instância do problema do caixeiro viajante e constrói a AGM correspondente ao grafo fornecido. Em seguida, na segunda etapa, é gerado um caminho euleriano a partir do caminhamento na AGM. Para esse fim, assume-se que todas as arestas da árvore foram duplicadas, permitindo que, a partir de um vértice arbitrário, seja possível percorrê-las exatamente uma vez, configurando assim um trajeto de ida e volta. Por fim, na terceira etapa, o circuito euleriano é transformado em um circuito hamiltoniano. Esse processo consiste em filtrar a lista de vértices obtida, removendo ocorrências repetidas e preservando apenas o primeiro e o último vértice, dado que o ponto de partida deve coincidir com o ponto de término. Os vértices restantes, após a filtragem, formam o circuito hamiltoniano, que representa a saída final do algoritmo.

Para a construção da Árvore Geradora Mínima, os dois algoritmos mais utilizados são o algoritmo de Kruskal e o algoritmo de Prim. Neste projeto, optou-se pela utilização do algoritmo de Prim devido à sua melhor eficiência em grafos densos, enquanto o algoritmo de Kruskal apresenta vantagens em grafos esparsos. Como as instâncias utilizadas nos testes correspondem a grafos completos, que são intrinsecamente densos, o algoritmo de Prim mostrou-se mais adequado para esta aplicação.

Na segunda etapa do algoritmo, foi utilizado o algoritmo de Busca em Profundidade (DFS), que percorre a árvore geradora mínima e retorna um caminho euleriano. A principal vantagem do uso da DFS é que não é necessário duplicar explicitamente as arestas, pois sua lógica intrínseca já simula esse comportamento. O caminho é, então, utilizado na parte final do algoritmo, onde os vértices repetidos são eliminados. Esse processo é realizado ao percorrer a lista do caminho euleriano e filtrar os vértices já visitados com o auxílio de uma estrutura de conjunto (set). O algoritmo de Prim implementado na função `prim_algorithm` possui complexidade de tempo $O(V^2)$. Isso ocorre porque o algoritmo percorre todos os V vértices e, para cada vértice, verifica os pesos de todas as arestas adjacentes, o que, em grafos completos, envolve V iterações adicionais. Dessa forma, para grafos densos, essa implementação pode ser mais eficiente do que a abordagem tradicional baseada em filas de prioridade, cuja complexidade é $O(E \log V)$. Já a complexidade de espaço é $O(V + A)$, dominada pelo grafo f, que contém os V vértices e as A arestas selecionadas para formar a árvore geradora mínima.

O algoritmo de busca em profundidade, implementado na função `dfs`, possui complexidade de tempo $O(V + A)$, onde V é o número de vértices e A é o número de arestas do grafo f. Isso ocorre porque o algoritmo percorre cada vértice e cada aresta exatamente uma vez. A complexidade de espaço é $O(V)$, referente ao armazenamento da lista de vértices visitados e à pilha utilizada pela recursão durante a execução.

A função para encontrar o caminho hamiltoniano, implementada na função `hamiltoniano`, possui complexidade de tempo $O(V)$, pois itera sobre os vértices do caminho gerado pela DFS. A complexidade de espaço também é $O(V)$, devido ao armazenamento do conjunto de vértices visitados e da lista que armazena o caminho hamiltoniano.

Assim, a complexidade final da função Twice Around The Tree, implementada pela função `tatt`, é dominada pelo algoritmo de Prim. A complexidade de tempo é $O(V^2)$, devido à execução do algoritmo de Prim, enquanto a complexidade de espaço é $O(V + A)$, também dominada pelas estruturas utilizadas no algoritmo de Prim.

2.3. Algoritmo de Christofides

Nesta seção, será apresentado inicialmente um panorama geral sobre o funcionamento do algoritmo de Christofides. Em seguida, serão detalhadas as escolhas de implementação realizadas durante o seu desenvolvimento. O algoritmo de Christofides é um algoritmo de aproximação com fator 1.5 para o problema do caixeiro viajante em instâncias métricas, ou seja, quando as distâncias satisfazem as propriedades de não-negatividade, simetria e desigualdade triangular. Essencialmente, o algoritmo é composto por seis passos principais mostrados no Algoritmo 1.

Algorithm 1 Construção de um Circuito Hamiltoniano a partir da AGM

Entrada: Grafo $G = (V, E)$ com custos nas arestas
Saída: Circuito Hamiltoniano aproximado
Passo 1: Construção da Árvore Geradora Mínima (AGM)
Construir a AGM T do grafo G usando, por exemplo, os algoritmos de Kruskal ou Prim.
Passo 2: Identificação de Vértices de Grau Ímpar
Identificar o conjunto I, contendo todos os vértices de T com grau ímpar.
Passo 3: Cálculo do Matching Perfeito de Custo Mínimo
Construir o subgrafo induzido pelos vértices de I.
Encontrar o matching perfeito de custo mínimo M nesse subgrafo.
Passo 4: Combinação das Arestas
Combinar as arestas de M com as arestas de T, formando o multigrafo H.
Garantir que todos os vértices de H tenham grau par.
Passo 5: Construção de um Circuito Euleriano
Construir um circuito euleriano C_E a partir do multigrafo H.
Passo 6: Transformação em um Circuito Hamiltoniano
Transformar C_E em um circuito hamiltoniano C_H removendo vértices repetidos e mantendo o ponto de partida como também o ponto de término.
Retornar C_H

Para a construção deste algoritmo, optou-se pela utilização da biblioteca NetworkX. Essa escolha se justifica especialmente pela terceira etapa do algoritmo, na qual é necessário calcular o matching perfeito de custo mínimo para os vértices do conjunto I. Esse cálculo é realizado utilizando o algoritmo de Blossom, que apresenta uma implementação complexa e não trivial. Dado que o foco principal deste projeto é o estudo e análise dos algoritmos relacionados ao problema do caixeiro viajante (TSP), utilizar uma biblioteca que já oferece suporte a essa operação por meio de funções próprias revelou-se uma escolha mais eficiente e prática.

Além disso, a biblioteca NetworkX facilita significativamente a manipulação e criação de grafos, o que contribui para agilizar o desenvolvimento do projeto. Essa vantagem é evidenciada pelo uso da biblioteca na criação do multigrafo, por meio da função `.MultiGraph()`, e na construção do circuito euleriano, utilizando a função `eulerian_circuit()`.

Para o passo 1, optamos por utilizar o algoritmo de Prim implementado pela biblioteca `networkx`, devido à sua praticidade e ao fato de já estarmos utilizando essa biblioteca pelos motivos citados anteriormente. O algoritmo de Prim implementado pela `networkx` possui complexidade de tempo $O(E \log V)$ e complexidade de espaço $O(V + E)$.

No passo 2, o laço `for v in agm.nodes` verifica o grau de cada vértice na AGM, que possui V vértices. Como o cálculo do grau de um vértice é realizado em $O(1)$, a complexidade de tempo total deste passo é $O(V)$.

No passo 3, a construção do subgrafo completo entre os vértices de grau ímpar, denotados por k, utiliza a função `nx.shortest_path_length`, cuja complexidade de

tempo é $O(E + V \log V)$, onde V é o número de vértices e E é o número de arestas do grafo original. Para calcular os pesos de todas as arestas no subgrafo completo, que possui $O(k^2)$ arestas, a complexidade de tempo total é $O(k^2 \cdot (E + V \log V))$. A complexidade de espaço deste passo é $O(k^2)$, correspondente ao espaço necessário para armazenar as arestas do subgrafo completo.

No passo 4, ocorre o emparelhamento mínimo, que utiliza o algoritmo de Blossom. Para isso, utilizamos a função `nx.algorithms.matching.min_weight_matching`, que possui complexidade de tempo $O(k^3)$ e complexidade de espaço $O(k)$, já que armazena $O(k)$ pares de vértices no emparelhamento.

No passo 5, ocorre a criação do multigrafo euleriano. A complexidade de tempo deste passo é $O(V + E + k)$, pois envolve copiar a Árvore Geradora Mínima (AGM), o que requer $O(V + E)$, e adicionar as arestas do emparelhamento mínimo, o que demanda $O(k)$. Já a complexidade de espaço também é $O(V+E+k)$, correspondente ao armazenamento dos vértices e arestas da AGM, além das arestas do emparelhamento mínimo.Por fim, no passo 6, ocorre a conversão para o caminho hamiltoniano, que consiste em percorrer cada aresta do circuito euleriano, adicionando vértices ao caminho hamiltoniano e calculando o custo total. Esse processo tem complexidade de tempo $O(V)$, pois percorre os vértices do caminho, e complexidade de espaço $O(V)$, para armazenar o caminho.

Assim, a complexidade total do algoritmo é $O(V^3)$ para o tempo, dominada pelo algoritmo de Blossom, e $O(k^2)$ para o espaço. Em grafos completos, onde $k \approx V$, a complexidade no pior caso é $O(V^2)$.

3. Experimentos

Os experimentos realizados têm como objetivo analisar três aspectos principais: tempo de execução, consumo do pico espaço e qualidade da solução apresentada. Para medir o tempo de execução, utilizou-se a biblioteca *time*, registrando os instantes de início e término da execução dos algoritmos. O consumo de espaço foi avaliado com a biblioteca *tracemalloc*, que permite identificar o pico de consumo, ou seja, o maior valor identificado em memória durante a execução. Já a qualidade da solução foi avaliada comparando o custo da solução obtida com o custo ótimo.

As tabelas seguintes apresentam tabelas o resultado para cada algoritmo analisado. Essas tabelas incluem informações como o nome do arquivo, tempo de execução, memória utilizada, custo encontrado, custo ótimo e a proximidade da solução obtida em relação à solução ótima. É importante ressaltar, que os testes acima de 10.000 vértices estouraram a memória e não conseguiram ser executados, portanto, os testes foram efetuados com os 72 arquivos restantes.

As Tabelas 1 e 2 mostram os resultados dos testes para o *Branch-And-Bound*. Por sua vez, as Tabelas 3 e 3 apresentam os resultados para o Twice Around The Tree. E por ultimo, através das Tabelas 5 e 6 é possível observar os resultados para o Christofides.

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
a280.tsp	a280	1800.001	0.799495	nan	2579	nan
berlin52.tsp	berlin52	1800.0006	0.108375	nan	7542	nan
bier127.tsp	bier127	1800.002	0.676239	nan	118282	nan
ch130.tsp	ch130	1800.0015	0.558887	nan	6110	nan
ch150.tsp	ch150	1800.0019	0.744391	nan	6528	nan
d1291.tsp	d1291	1800.1845	66.418911	nan	50801	nan
d1655.tsp	d1655	1800.2757	108.407036	nan	62128	nan
d198.tsp	d198	1800.0036	1.382671	nan	15780	nan
d2103.tsp	d2103	1800.46	176.545623	nan	80450	nan
d493.tsp	d493	1800.0223	9.049343	nan	35002	nan
d657.tsp	d657	1819.0908	17.060351	nan	48912	nan
eil101.tsp	eil101	1800.0027	0.116175	nan	629	nan
eil51.tsp	eil51	1800.0003	0.037447	nan	426	nan
eil76.tsp	eil76	1800.0004	0.068735	nan	538	nan
fl1400.tsp	fl1400	1800.1948	69.404559	nan	20127	nan
fl1577.tsp	fl1577	1800.2571	92.286519	nan	22249	nan
fl3795.tsp	fl3795	1801.5522	546.747967	nan	28772	nan
fl417.tsp	fl417	1800.0164	6.463631	nan	11861	nan
fnl4461.tsp	fnl4461	2011.8354	792.926487	nan	182566	nan
gil262.tsp	gil262	2201.0019	0.629655	nan	2378	nan
kroA100.tsp	kroA100	1800.0014	0.424631	nan	21282	nan
kroA150.tsp	kroA150	1800.0026	0.946695	nan	26524	nan
kroA200.tsp	kroA200	1800.0044	1.615983	nan	29368	nan
kroB100.tsp	kroB100	1800.0014	0.425031	nan	22141	nan
kroB150.tsp	kroB150	1800.0028	0.944991	nan	26130	nan
kroB200.tsp	kroB200	1800.0045	1.615503	nan	29437	nan
kroC100.tsp	kroC100	1800.0013	0.424415	nan	20749	nan
kroD100.tsp	kroD100	1800.0029	0.423751	nan	21294	nan
kroE100.tsp	kroE100	1800.0013	0.424775	nan	22068	nan
lin105.tsp	lin105	1800.0014	0.448055	nan	14379	nan
lin318.tsp	lin318	1800.0105	4.134671	nan	42029	nan
linhp318.tsp	linhp318	1800.0105	4.135175	nan	41345	nan
nrw1379.tsp	nrw1379	1800.1882	73.348703	nan	56638	nan
p654.tsp	p654	1800.0413	16.051999	nan	34643	nan
pcb1173.tsp	pcb1173	1800.1472	54.386143	nan	56892	nan
pcb3038.tsp	pcb3038	1801.0457	369.813991	nan	137694	nan
pcb442.tsp	pcb442	1800.0221	7.855343	nan	50778	nan
pr1002.tsp	pr1002	1800.1097	41.137207	nan	259045	nan
pr107.tsp	pr107	1800.0014	0.480567	nan	44303	nan
pr124.tsp	pr124	1800.0019	0.646479	nan	59030	nan
pr136.tsp	pr136	1800.0022	0.785303	nan	96772	nan
pr144.tsp	pr144	1800.0038	0.857775	nan	58537	nan
pr152.tsp	pr152	1800.0026	0.977279	nan	73682	nan

Table 1. Resultados dos testes Branch-And-Bound

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
pr226.tsp	pr226	1800.0121	2.079191	nan	80369	nan
pr2392.tsp	pr2392	1800.7456	232.367728	nan	378032	nan
pr264.tsp	pr264	1800.0072	2.814391	nan	49135	nan
pr299.tsp	pr299	1800.0091	3.623887	nan	48191	nan
pr439.tsp	pr439	1800.0301	7.964167	nan	107217	nan
pr76.tsp	pr76	1800.0009	0.251799	nan	108159	nan
rat195.tsp	rat195	1800.0006	0.394191	nan	2323	nan
rat575.tsp	rat575	1800.0107	5.835519	nan	6773	nan
rat783.tsp	rat783	1800.0265	12.889199	nan	8806	nan
rat99.tsp	rat99	1800.0004	0.113775	nan	1211	nan
rd100.tsp	rd100	1800.0012	0.383951	nan	7910	nan
rd400.tsp	rd400	1800.015	5.673111	nan	15281	nan
rl1323.tsp	rl1323	1800.181	71.108247	nan	270199	nan
rl1889.tsp	rl1889	1801.1153	144.915743	nan	316536	nan
rl5915.tsp	rl5915	1805.4385	1433.302055	nan	565530	nan
rl5934.tsp	rl5934	1805.2694	1441.410791	nan	556045	nan
st70.tsp	st70	1800.0015	0.064047	nan	675	nan
ts225.tsp	ts225	1800.0054	2.089671	nan	126643	nan
tsp225.tsp	tsp225	1800.0016	0.813063	nan	3916	nan
u1060.tsp	u1060	1800.1232	45.416599	nan	224094	nan
u1432.tsp	u1432	1800.2214	83.613663	nan	152970	nan
u159.tsp	u159	1800.0029	1.055911	nan	42080	nan
u1817.tsp	u1817	1800.3709	130.902535	nan	57201	nan
u2152.tsp	u2152	1800.4525	181.619815	nan	64253	nan
u2319.tsp	u2319	1800.5434	218.868687	nan	234256	nan
u574.tsp	u574	1800.0347	12.955263	nan	36905	nan
u724.tsp	u724	1800.0509	20.668895	nan	41910	nan
vm1084.tsp	vm1084	1800.1341	47.281343	nan	239297	nan
vm1748.tsp	vm1748	1800.3283	122.934775	nan	336556	nan

Table 2. Resultados dos testes Branch-And-Bound

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
a280.tsp	a280	0.0462	0.024359	4884	2579	1.8938
berlin52.tsp	berlin52	0.0026	0.006469	12156	7542	1.6118
bier127.tsp	bier127	0.0107	0.017275	189412	118282	1.6014
ch130.tsp	ch130	0.0127	0.017384	10332	6110	1.691
ch150.tsp	ch150	0.0142	0.018127	11756	6528	1.8009
d1291.tsp	d1291	1.3561	0.26489	93862	50801	1.8476
d1655.tsp	d1655	2.315	0.293316	113092	62128	1.8203
d198.tsp	d198	0.0247	0.021864	23476	15780	1.4877
d2103.tsp	d2103	3.616	0.59497	152662	80450	1.8976
d493.tsp	d493	0.1691	0.074854	58538	35002	1.6724
d657.tsp	d657	0.3252	0.085056	84976	48912	1.7373
eil101.tsp	eil101	0.0067	0.016605	1102	629	1.752
eil51.tsp	eil51	0.028	0.006347	750	426	1.7606
eil76.tsp	eil76	0.0075	0.007649	926	538	1.7212
fl1400.tsp	fl1400	1.6246	0.271712	33662	20127	1.6725
fl1577.tsp	fl1577	2.0711	0.374274	38688	22249	1.7389
fl3795.tsp	fl3795	13.961	0.808171	50224	28772	1.7456
fl417.tsp	fl417	0.1188	0.066193	20302	11861	1.7117
fnl4461.tsp	fnl4461	20.1612	0.549986	336924	182566	1.8455
gil262.tsp	gil262	0.0584	0.023678	4178	2378	1.7569
kroA100.tsp	kroA100	0.0079	0.016056	37544	21282	1.7641
kroA150.tsp	kroA150	0.0152	0.018469	47114	26524	1.7763
kroA200.tsp	kroA200	0.0258	0.019999	51860	29368	1.7659
kroB100.tsp	kroB100	0.0074	0.016683	38516	22141	1.7396
kroB150.tsp	kroB150	0.0161	0.01807	45602	26130	1.7452
kroB200.tsp	kroB200	0.0249	0.019999	52394	29437	1.7799
kroC100.tsp	kroC100	0.007	0.015999	36804	20749	1.7738
kroD100.tsp	kroD100	0.0068	0.016512	37192	21294	1.7466
kroE100.tsp	kroE100	0.0079	0.015942	38446	22068	1.7422
lin105.tsp	lin105	0.0076	0.016722	26110	14379	1.8158
linhp318.tsp	lin318	0.0707	0.059735	75812	42029	1.8038
lin318.tsp	lin318	0.066	0.060305	75812	42029	1.8038
nrw1379.tsp	nrw1379	1.6158	0.268286	103978	56638	1.8358
p654.tsp	p654	0.3254	0.088024	58912	34643	1.7005
pcb1173.tsp	pcb1173	1.1111	0.125204	102830	56892	1.8075
pcb3038.tsp	pcb3038	8.4395	0.439532	254604	137694	1.8491
pcb442.tsp	pcb442	0.1477	0.071108	92716	50778	1.8259
pr1002.tsp	pr1002	0.8382	0.1108	448358	259045	1.7308
pr107.tsp	pr107	0.009	0.016567	69514	44303	1.5691
pr124.tsp	pr124	0.0098	0.017108	101070	59030	1.7122
pr136.tsp	pr136	0.0549	0.018585	177928	96772	1.8386
pr144.tsp	pr144	0.0134	0.018364	98932	58537	1.6901
pr152.tsp	pr152	0.0149	0.018539	118342	73682	1.6061

Table 3. Resultados Testes Twice Around The Tree

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
pr226.tsp	pr226	0.032	0.022414	137286	80369	1.7082
pr2392.tsp	pr2392	4.9715	0.406876	684538	378032	1.8108
pr264.tsp	pr264	0.0437	0.023807	82284	49135	1.6747
pr299.tsp	pr299	0.0569	0.028686	84976	48191	1.7633
pr439.tsp	pr439	0.1438	0.073429	184386	107217	1.7197
pr76.tsp	pr76	0.0046	0.007193	174434	108159	1.6128
rat195.tsp	rat195	0.0222	0.020244	4310	2323	1.8554
rat575.tsp	rat575	0.2318	0.079483	12496	6773	1.845
rat783.tsp	rat783	0.5265	0.09504	16250	8806	1.8453
rat99.tsp	rat99	0.0073	0.015877	2214	1211	1.8282
rd100.tsp	rd100	0.0085	0.017139	13924	7910	1.7603
rd400.tsp	rd400	0.2013	0.065467	27276	15281	1.785
rl1323.tsp	rl1323	1.5741	0.320566	479972	270199	1.7764
rl1889.tsp	rl1889	3.4109	0.361653	556532	316536	1.7582
rl5915.tsp	rl5915	37.2223	1.19371	1043742	565530	1.8456
rl5934.tsp	rl5934	37.9387	1.194168	1027904	556045	1.8486
st70.tsp	st70	0.0042	0.007103	1126	675	1.6681
ts225.tsp	ts225	0.0309	0.021209	224000	126643	1.7688
tsp225.tsp	tsp225	0.029	0.053945	7028	3916	1.7947
u1060.tsp	u1060	0.8791	0.117342	390926	224094	1.7445
u1432.tsp	u1432	1.6442	0.269624	291954	152970	1.9086
u159.tsp	u159	0.0154	0.018903	74322	42080	1.7662
u1817.tsp	u1817	2.753	0.319434	108572	57201	1.8981
u2152.tsp	u2152	4.0205	0.358097	122984	64253	1.9141
u2319.tsp	u2319	4.4105	0.330058	464400	234256	1.9824
u574.tsp	u574	0.2396	0.081292	64156	36905	1.7384
u724.tsp	u724	0.4318	0.092356	75918	41910	1.8115
vm1084.tsp	vm1084	1.0579	0.117164	418494	239297	1.7488
vm1748.tsp	vm1748	3.0245	0.483886	589254	336556	1.7508

Table 4. Resultados Testes Twice Around The Tree

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
a280.tsp	a280	97.0636	0.244938	2930	2579	1.1361
berlin52.tsp	berlin52	0.2152	0.238391	8639	7542	1.1455
bier127.tsp	bier127	10.852	0.240342	131046	118282	1.1079
ch130.tsp	ch130	5.8352	0.240159	6773	6110	1.1085
ch150.tsp	ch150	9.8812	0.240891	7137	6528	1.0933
d1291.tsp	d1291	1800.1657	0.229501	0	50801	nan
d1655.tsp	d1655	1800.3131	0.229556	0	62128	nan
d198.tsp	d198	41.1098	0.242151	16753	15780	1.0617
d2103.tsp	d2103	1800.4972	0.229736	0	80450	nan
d493.tsp	d493	1668.9867	0.256718	38489	35002	1.0996
d657.tsp	d657	1800.1188	0.229736	0	48912	nan
eil101.tsp	eil101	2.5137	0.239517	721	629	1.1463
eil51.tsp	eil51	0.43	0.238176	472	426	1.108
eil76.tsp	eil76	0.855	0.238994	609	538	1.132
fl1400.tsp	fl1400	1800.5885	0.229683	0	20127	nan
fl1577.tsp	fl1577	1800.3939	0.229681	0	22249	nan
fl3795.tsp	fl3795	1962.0063	0.229597	0	28772	nan
fl417.tsp	fl417	863.5852	0.252605	13151	11861	1.1088
fnl4461.tsp	fnl4461	1941.9818	0.229683	0	182566	nan
gil262.tsp	gil262	96.6247	0.244152	2668	2378	1.122
kroA100.tsp	kroA100	3.0512	0.239562	23293	21282	1.0945
kroA150.tsp	kroA150	17.6872	0.241011	29688	26524	1.1193
kroA200.tsp	kroA200	52.0649	0.24233	32806	29368	1.1171
kroB100.tsp	kroB100	1.7148	0.239562	24012	22141	1.0845
kroB150.tsp	kroB150	17.3328	0.240903	30053	26130	1.1501
kroB200.tsp	kroB200	43.3757	0.24233	33119	29437	1.1251
kroC100.tsp	kroC100	2.9161	0.239505	23470	20749	1.1311
kroD100.tsp	kroD100	2.5424	0.239452	23583	21294	1.1075
kroE100.tsp	kroE100	3.2531	0.239444	23783	22068	1.0777
lin105.tsp	lin105	2.3448	0.239741	17093	14379	1.1887
lin318.tsp	lin318	257.5175	0.247001	48205	42029	1.1469
linhp318.tsp	lin318	256.4433	0.247054	48205	42029	1.1469
nrw1379.tsp	nrw1379	1800.1309	0.229683	0	56638	nan
p654.tsp	p654	1224.949	0.265959	39307	34643	1.1346
pcb1173.tsp	pcb1173	1800.3919	0.229736	0	56892	nan
pcb3038.tsp	pcb3038	1805.9748	0.229683	0	137694	nan
pcb442.tsp	pcb442	1190.5697	0.253772	55811	50778	1.0991
pr1002.tsp	pr1002	7561.952	0.229736	0	259045	nan
pr107.tsp	pr107	3.6034	0.239816	47891	44303	1.081
pr124.tsp	pr124	2.7516	0.240314	65306	59030	1.1063
pr136.tsp	pr136	3.6134	0.240583	103771	96772	1.0723
pr144.tsp	pr144	1.8751	0.240617	70404	58537	1.2027
pr152.tsp	pr152	2.5147	0.241008	79713	73682	1.0819

Table 5. Tabela de Resultados Christofides

Arquivo	Nome	Tempo(s)	Memória (MB)	Custo T	Custo O	Taxa
pr226.tsp	pr226	33.081	0.242999	91990	80369	1.1446
pr2392.tsp	pr2392	1801.8028	0.229736	0	378032	0.0
pr264.tsp	pr264	75.3641	0.244207	54748	49135	1.1142
pr299.tsp	pr299	209.5601	0.246064	52964	48191	1.099
pr439.tsp	pr439	692.6114	0.253494	118198	107217	1.1024
pr76.tsp	pr76	0.7641	0.238992	116684	108159	1.0788
rat195.tsp	rat195	22.7663	0.241866	2611	2323	1.124
rat575.tsp	rat575	1800.2796	0.229791	0	6773	0.0
rat783.tsp	rat783	1800.3684	0.229609	0	8806	0.0
rat99.tsp	rat99	1.5076	0.239446	1350	1211	1.1148
rd100.tsp	rd100	3.9721	0.239452	8906	7910	1.1259
rd400.tsp	rd400	958.0343	0.251694	17112	15281	1.1198
rl1323.tsp	rl1323	1800.3391	0.229618	0	270199	0.0
rl1889.tsp	rl1889	1800.5717	0.229685	0	316536	0.0
rl5915.tsp	rl5915	1808.131	0.243074	0	565530	0.0
rl5934.tsp	rl5934	1812.6289	0.243019	0	556045	0.0
st70.tsp	st70	0.7208	0.238695	760	675	1.1259
ts225.tsp	ts225	7.6936	0.242717	132587	126643	1.0469
tsp225.tsp	tsp225	46.8718	0.242825	4442	3916	1.1343
u1060.tsp	u1060	1800.1223	0.229662	0	224094	0.0
u1432.tsp	u1432	2205.631	0.229736	0	152970	0.0
u159.tsp	u159	15.6485	0.240983	47639	42080	1.1321
u1817.tsp	u1817	1800.7483	0.229717	0	57201	0.0
u2152.tsp	u2152	2756.2864	0.229683	0	64253	0.0
u2319.tsp	u2319	1800.2848	0.229736	0	234256	0.0
u574.tsp	u574	1800.3268	0.229736	0	36905	0.0
u724.tsp	u724	1800.1666	0.229791	0	41910	0.0
vm1084.tsp	vm1084	9696.4893	0.229683	0	239297	0.0
vm1748.tsp	vm1748	1800.2149	0.229791	0	336556	0.0

Table 6. Tabela de Resultados Christofides

As Figuras 1 e 2 apresentam a dispersão dos intervalos e a quantidade de testes executados que alcançaram resultados próximos do ótimo entre essas taxas.

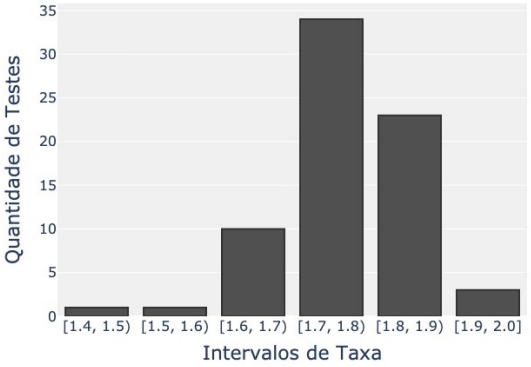

Figure 1. Gráfico de intervalos das taxas do twice around the tree

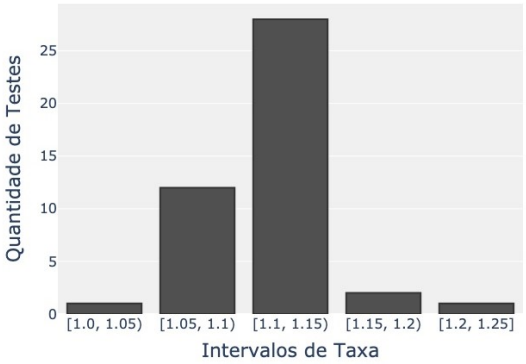

Figure 2. Gráfico de intervalos das taxas do algoritmo de Christofides

4. Discussão

Ao analisar as tabelas, observa-se que o algoritmo *Branch-And-Bound* não conseguiu calcular os resultados para nenhuma das instâncias testadas, pois todas ultrapassaram o limite de tempo de 30 minutos. Além disso, o uso de memória foi significativamente elevado para instâncias maiores. Como não foram obtidos valores de solução, não foi possível calcular a taxa de aproximação.

Em relação ao algoritmo Twice Around The Tree, verificou-se que todas as taxas de aproximação são, como esperado, menores ou iguais a 2. Esse comportamento é consistente com a natureza 2-aproximada do algoritmo. Por outro lado, o algoritmo de Christofides apresentou todas as taxas de aproximação menores ou iguais a 1,5, ilustrando claramente sua característica 1,5-aproximada.

No que diz respeito ao tempo de execução, o algoritmo mais eficiente foi o Twice Around The Tree, que não ultrapassou o limite de 30 minutos em nenhuma instância. Contudo, suas soluções são menos precisas em comparação com as do algoritmo de Christofides, que oferece uma aproximação mais próxima do custo ótimo. Em contrapartida, o *Branch-And-Bound* excedeu o tempo limite em todas as instâncias; porém, caso alcançasse uma solução, ela seria exata.

Esses resultados destacam que o *Branch-And-Bound* não é adequado para instâncias muito grandes ou situações em que o tempo de resolução é um fator crítico. Por outro lado, para problemas em que não é necessária uma solução exata, os algoritmos aproximativos são boas alternativas. O algoritmo de Christofides é recomendado quando a prioridade é obter uma solução mais próxima do ótimo, enquanto o Twice Around The Tree é mais indicado quando a eficiência no tempo de execução é o fator mais relevante.

5. Conclusão

Este trabalho abordou a implementação e análise de três estratégias para resolver o Problema do Caixeiro Viajante Euclidiano, considerando um método exato e dois algoritmos

heurísticos. Os resultados experimentais permitiram analisar e avaliar as características, vantagens e limitações de cada abordagem.

O algoritmo *Branch-And-Bound* demonstrou ser ineficaz para instâncias maiores devido ao elevado tempo de execução e alto consumo de memória, reforçando sua aplicabilidade restrita a problemas de menor escala ou situações em que a solução exata seja necessária. Caso contrário, os algoritmos aproximativos se mostraram alternativas viáveis para instâncias maiores, apresentando tempos de execução significativamente menores. Entre os algoritmos heurísticos, o Twice Around The Tree foi o mais eficiente em termos de tempo de execução, mas apresentou soluções menos precisas, com taxas de aproximação menores ou iguais a 2. Já o algoritmo de Christofides se destacou pela maior proximidade das soluções ao custo ótimo, com taxas de aproximação inferiores ou iguais a 1,5, confirmando sua superioridade em termos de qualidade da solução.

Esses resultados evidenciam que a escolha do algoritmo depende diretamente da necessidade do problema. Assim, este estudo reforça a importância de selecionar algoritmos adequados às necessidades específicas de cada aplicação, contribuindo para avanços na resolução de problemas complexos como o TSP em contextos práticos e acadêmicos

References

Muriyatmoko, D., Djunaidy, A., and Muklason, A. (2024). Heuristics and metaheuristics for solving capacitated vehicle routing problem: An algorithm comparison. *Procedia Computer Science*, 234:494–501. Seventh Information Systems International Conference (ISICO 2023).

Pop, P. C., Cosma, O., Sabo, C., and Sitar, C. P. (2024). A comprehensive survey on the generalized traveling salesman problem. *European Journal of Operational Research*, 314(3):819–835.

Roberti, R. and Ruthmair, M. (2019). Exact methods for the traveling salesman problem with drone.

The Sun (2024). China plans to have 100,000 flying cars buzzing around its cities as taxis, delivery vans and family motors in just 6 years. Accessed: 2025-01-11.