Algoritmos II - Trabalho Prático 2
Soluções para Caixeiro Viajante

Lucas Rafael Costa Santos[1], **Lucca Alvarenga de Magalhães Pinto**[2]

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

Abstract. *This work presents the implementation of 3 algorithms, an exact solution, based on the Branch-and-Bound algorithm, and two approximate solutions, the Twice-Around-the-Tree algorithm and the Christofides algorithm, for the Traveling Salesman Problem in its Euclidean variation, where the cost function is based on the Euclidean distance in 2D between the points.*

Resumo. *Esse trabalho apresenta a implementação de 3 algoritmos, uma solução exata, baseada no algoritmo Branch-and-Bound, e duas soluções aproximadas, o algoritmo Twice-Around-the-Tree e o algoritmo de Christofides, para o Problema do Caixeiro Viajante em sua variação euclidiana, onde a função de custo é baseada na distância euclidiana em 2D entre os pontos.*

1. Introdução

O Problema do Caixeiro Viajante (TSP - Traveling Salesman Problem) é um dos problemas mais clássicos e estudados em otimização combinatória e teoria dos grafos. Dado um conjunto de cidades e as distâncias entre cada par de cidades, o objetivo do TSP é encontrar a rota mais curta que visita todas as cidades exatamente uma vez e retornar à cidade de origem. Esse problema é classificado como um problema NP-difícil, o que implica que, não se conhece um algoritmo que resolva todas as instâncias em tempo polinomial em Máquina de Turing Determinística.

Neste trabalho, iremos tratar soluções para uma variante desse problema, o TSP Euclidiano, no qual as cidades são representadas por pontos em um plano cartesiano bidimensional, e as distâncias entre elas são calculadas utilizando a distância euclidiana. Assim, a diferença fundamental entre o TSP geral e o Euclidiano reside na forma como as distâncias entre as cidades são definidas, de forma que, no primeiro as distâncias podem ser arbitrárias, enquanto no segundo as distâncias seguem as regras da geometria euclidiana.

Foram implementados 3 algoritmos para solucionar o problema do caixeiro viajante geométrico: o algoritmo Branch-and-Bound, uma solução exata, e os algoritmos de Christofides e Twice-Around-the-Tree, duas soluções aproximadas ao problema. O objetivo principal deste relatório é analisar essas implementações, detalhando cada uma e justificando as estruturas de dados usadas, e realizando experimentos com base nas instâncias de testes compiladas na biblioteca TSPLIB, fazer uma avaliação de desempenho dos algoritmos, considerando aspectos como tempo de execução, consumo de espaço e qualidade das soluções.

2. Implementações

As implementações foram desenvolvidas em Python e utilizam as bibliotecas padrão, além de pacotes como numpy, networkx e pandas. A biblioteca networkx foi escolhida pela eficiência e clareza ao lidar com estruturas de grafos. As subseções a seguir detalham os algoritmos implementados, destacando suas principais características e abordagens.

2.1. Branch-and-Bound

O algoritmo Branch-and-Bound foi implementado para encontrar a solução ótima do problema utilizando estratégias de poda. A abordagem organiza os nós em uma árvore de busca, explorando-os de maneira best-first com uma fila de prioridade baseada em heap. A principal ideia é evitar calcular caminhos desnecessários ao determinar limites inferiores (bounds) que ajudam a descartar ramos inviáveis.

2.1.1. Estrutura e Funcionamento

Cada nó da árvore armazena informações cruciais para a busca: o custo total até aquele ponto, o nível na árvore (indicando a profundidade da busca), a sequência de vértices já visitados e um limite inferior para o custo total de qualquer solução que possa ser alcançada a partir daquele nó. Esse limite inferior, serve como um guia para a busca, permitindo descartar ramos da árvore que certamente não levarão à solução ótima.

A busca inicia-se a partir do nó raiz, representando o estado inicial e a cada iteração, o algoritmo remove da fila de prioridade o nó com o menor limite inferior e explora seus nós filhos. Ao expandir um nó, são gerados novos nós, cada um correspondendo a uma nova escolha de vértice a ser visitado. Se um nó representar um ciclo completo (isto é, todos os vértices foram visitados), sua solução é comparada com a melhor solução encontrada até então e, se for melhor, passa a ser a nova solução ótima.

2.1.2. Poda

A eficiência do algoritmo está diretamente ligada à capacidade de podar ramos da árvore que não conduzem à solução ótima. A poda é realizada ao comparar o limite inferior de um nó com o custo da melhor solução conhecida. Se o limite for maior que o custo da melhor solução, o ramo é descartado, pois qualquer solução completa a partir desse nó terá um custo maior.

2.1.3. Complexidade

Embora o Branch-and-Bound garanta encontrar a solução ótima, sua complexidade computacional é exponencial em relação ao número de cidades. Isso significa que o tempo de execução pode crescer rapidamente para instâncias maiores, limitando sua aplicação prática em alguns casos.

Em resumo, o Branch-and-Bound é um algoritmo poderoso para resolver o TSP, mas sua eficiência depende da qualidade da heurística utilizada para calcular os limites inferiores e da capacidade de podar eficientemente a árvore de busca.

2.2. Twice-Around-The-Tree

Oe algoritmo Twice-Around-The-Tree aproxima soluções para problema TSP em grafos euclidianos, oferecendo uma solução com custo no máximo duas vezes o custo ótimo. Para isso, ele se baseia na construção de uma Árvore Geradora Mínima (AGM) e em um caminhamento em pré-ordem para formar o ciclo.

2.2.1. Funcionamento

Inicialmente, é construída uma Árvore Geradora Mínima (AGM) a partir do grafo completo que representa o problema, utilizando as ferramentas presentes na biblioteca NetworkX. Em seguida, a partir de um vértice raiz arbitrário na AGM, é realizado um caminhamento em pré-ordem, de modo que, essa travessia visite cada vértice da árvore exatamente uma vez, gerando uma sequência de vértices que forma um caminho. Para obter então, um ciclo Hamiltoniano (que passa por todos os vértices exatamente uma vez), o vértice inicial é adicionado ao final do caminho obtido na etapa anterior. Por fim, o custo total da solução é calculado somando os pesos das arestas que compõem o ciclo.

2.2.2. Vantagens e Complexidade:

Uma das principais vantagens do algoritmo Twice-Around-the-Tree é sua baixa complexidade computacional. A etapa que domina o tempo de execução é a construção da AGM, que pode ser realizada em tempo $O(n^2 \log n)$, onde n é o número de vértices. Essa característica o torna aplicável a problemas de grande porte.

Embora não garanta a solução ótima, o algoritmo oferece uma aproximação razoável para o problema do caixeiro viajante, de forma que, a qualidade da solução obtida depende da estrutura do grafo, mas em geral o algoritmo consegue encontrar soluções com um custo próximo ao ótimo.

Portando, é possível notar que o algoritmo Twice-Around-the-Tree é uma ferramenta simples e eficiente para obter soluções aproximadas para o problema do caixeiro viajante, sendo sua baixa complexidade e facilidade de implementação fatores que o tornam uma opção atraente para diversas aplicações práticas.

2.3. Christofides

O algoritmo de Christofides representa um avanço significativo na busca por soluções aproximadas para o problema do caixeiro viajante. Ele combina a simplicidade do algoritmo Twice-Around-the-Tree com uma técnica mais sofisticada para obter um fator de aproximação menor, garantindo assim soluções de maior qualidade.

2.3.1. Funcionamento

Assim como no algoritmo Twice-Around-the-Tree, inicia-se construindo uma árvore geradora mínima para o grafo completo que representa o problema, a qual conecta todos os vértices com o menor custo total possível. A partir disso então, identificam-se os vértices da AGM que possuem grau ímpar, os quais serão cruciais para a próxima etapa.

Com os vértices de grau ímpar, constrói-se um subgrafo induzido e calcula-se um emparelhamento perfeito (conjunto de arestas que emparelha todos os vértices do subgrafo, sem que nenhuma aresta compartilhe um vértice com outra) de menor peso. Em seguida, as arestas da AGM e as arestas do emparelhamento mínimo são combinadas para formar um multigrafo. Então, utilizando um algoritmo eficiente para encontrar ciclos Eulerianos em multigrafos, obtemos um ciclo que percorre cada aresta do multigrafo exatamente uma vez. O ciclo Euleriano obtido geralmente contém vértices repetidos, dessa forma, para obter um ciclo Hamiltoniano (que passa por cada vértice exatamente uma vez), eliminamos os subcaminhos repetidos e ajustamos o ciclo para incluir apenas os vértices únicos.

2.3.2. Vantagens e Complexidade:

Ao adicionar a etapa do emparelhamento mínimo, o algoritmo de Christofides garante um fator de aproximação de 1,5, o que significa que a solução encontrada é no máximo 50% maior que a solução ótima.

A complexidade do algoritmo é dominada pela construção da AGM e pelo cálculo do emparelhamento mínimo. Embora seja mais complexa que o Twice-Around-the-Tree, ainda é polinomial e eficiente para muitos problemas práticos.

Em resumo, o algoritmo de Christofides oferece um excelente equilíbrio entre a qualidade da solução e a complexidade computacional. Ao combinar técnicas de grafos e otimização combinatória, ele proporciona uma ferramenta poderosa para resolver o problema do caixeiro viajante em diversas aplicações reais.

3. Experimentos e Resultados

Após a implementação dos algoritmos, foram realizados experimentos utilizando um conjunto de 72 instâncias da TSPLIB, considerando a função de custo como a distância euclidiana, sendo avaliados o desempenho de três algoritmos, Branch-and-Bound, Christofides e Twice-Around-the-Tree (TATT), em termos de tempo de execução e qualidade da solução (razão entre o custo da solução encontrada e o custo ótimo). Os testes foram realizados em uma máquina com Windows 11, processador Intel(R) Core(TM) i5 de 10ª geração e 16 GB de memória RAM e a versão do Python utilizada foi a 3.12, com as bibliotecas em suas versões mais recentes disponíveis.

É importante ressaltar que, como os exemplos apresentados chegam a um número de nós muito grande, os experimentos foram limitados a tempo e recursos computacionais. Dessa maneira, não serão mencionados casos em que foi necessario mais de 30 minutos para sua execucução.

3.1. Branch-and-Bound

A técnica Branch-and-Bound demonstrou ser computacionalmente custosa para o problema em questão. Mesmo para instâncias relativamente pequenas, com 52 vértices, o tempo de execução ultrapassou 30 minutos, limitando a coleta de dados mais abrangente. Problemas combinatórios, como este, são notoriamente desafiadores devido à natureza exponencial do espaço de busca e, embora o Branch-and-Bound ofereça uma abordagem

mais inteligente do que a força bruta, sua eficácia diminui significativamente com o aumento do tamanho da instância. Dessa maneira, a obtenção de soluções ótimas em tempo hábil torna-se inviável para problemas de grande escala, restringindo a aplicação do método a casos específicos onde a otimização é essencial.

Buscando um melhor entendimento, foram realizadas análises utilizando instâncias reduzidas, variando entre 5 e 16 nós, derivadas das instâncias da TSPLIB. O gráfico apresentado na Figura 1 revela que, mesmo em menor escala, o tempo de execução aumenta significativamente à medida que o número de nós cresce.

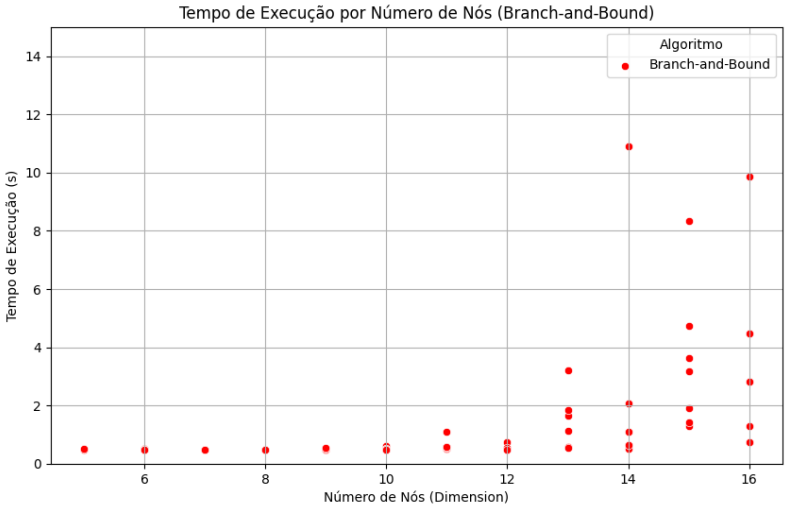

Figure 1. Gráfico Tempo de Execução por Número de Nós (Branch-And-Bound)

3.2. Twice-Around-the-Tree (TATT)

Com os resultados em mãos e gerando um gráfico de Tempo de Execução por Número de Nós (Figure 1), foi possível observar que o algoritmo Twice-Around-the-Tree apresentou desempenho consistente em relação ao tempo de execução, visto que, para instâncias menores, o tempo médio foi de 1.5 segundos. Entretanto, para instâncias maiores, o tempo aumentou de forma considerável, chegando a 109 segundos para o maior grafo testado, gerando uma media final de aproximadamente 7.27 segundos. Já em relação à qualidade, o TATT apresentou um fator de aproximação médio de 1.37, sendo o pior resultado 1.57, o que é esperado devido à garantia teórica do algoritmo.

3.3. Christofides

O algoritmo Christofides (Figure 2), por sua vez, obteve um desempenho geral melhor no quesito qualidade da solução, apresentando um fator de aproximação médio de 1.12, com o pior resultado em 1.20. Entretanto, o tempo de execução para instâncias maiores foi consideravelmente superior ao do TATT, apresentando um tempo médio de 4 segundos para grafos menores, e até 553 segundos para instâncias maiores, gerando uma média geral de aproximadamente 46 segundos.

A análise reforça que, embora o Christofides tenha maior custo computacional, sua qualidade de solução é superior, o que o torna mais adequado para problemas onde a precisão da solução é um fator crítico.

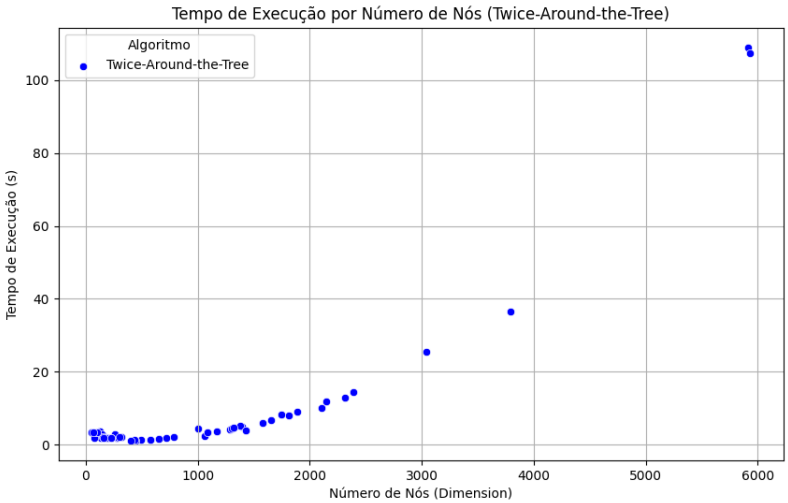

Figure 2. Gráfico Tempo de Execução por Número de Nós (TATT)

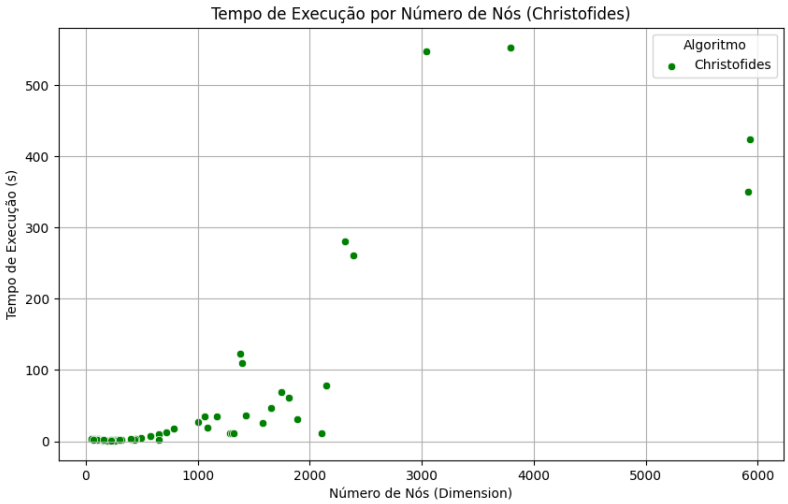

Figure 3. Gráfico Tempo de Execução por Número de Nós (Christofides)

3.4. Comparação dos Algoritmos

Os resultados comparativos entre os algoritmos estão sumarizados na Tabela 1 e Tabela 2. Ao comparar os resultados, é possível observar que o TATT é mais eficiente em termos de tempo, porém sua qualidade da solução é inferior ao Christofides, já em termos de escalabilidade, ambos os algoritmos apresentaram comportamentos semelhantes, mas o Christofides mostrou-se mais limitado em relação ao tempo para instâncias maiores.

A partir dos dados coletados, também foi gerado um gráfico (Figure 4) que relaciona o número de vértices das instâncias com as métricas de tempo e qualidade.

Nesse caso, foi possível notar que o Twice-Around-The-Tree mostrou-se mais eficiente em instâncias com maior número de vértices, apresentando tempos significativamente menores. Já o Christofides, embora mais lento, apresentou crescimento de tempo alinhado com sua complexidade assintótica. Quanto à Qualidade da Solução, o Algoritmo de Christofides apresentou melhores resultados, com soluções mais próximas do ótimo em relação ao TATT. Quanto ao gasto de memória (Figure 5), é possível notar que ambos apresentaram resultados bem semelhantes.

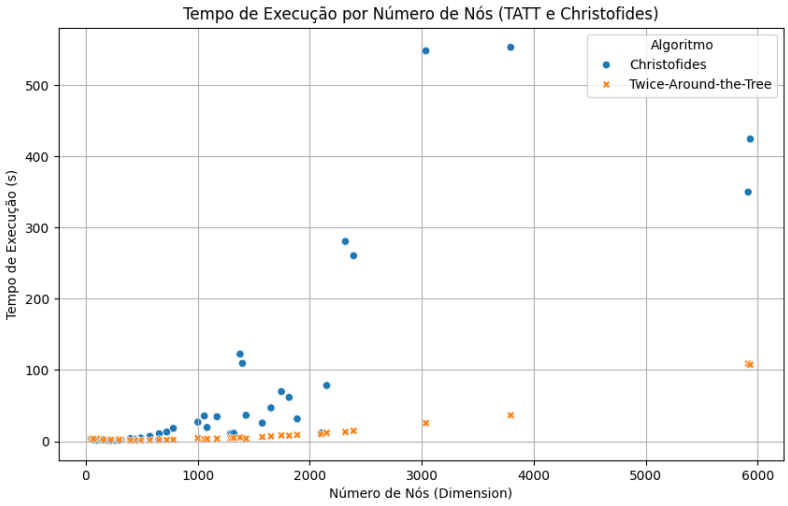

Figure 4. Comparativo - Tempo de Execução por Número de Nós (TATT e Christofides)

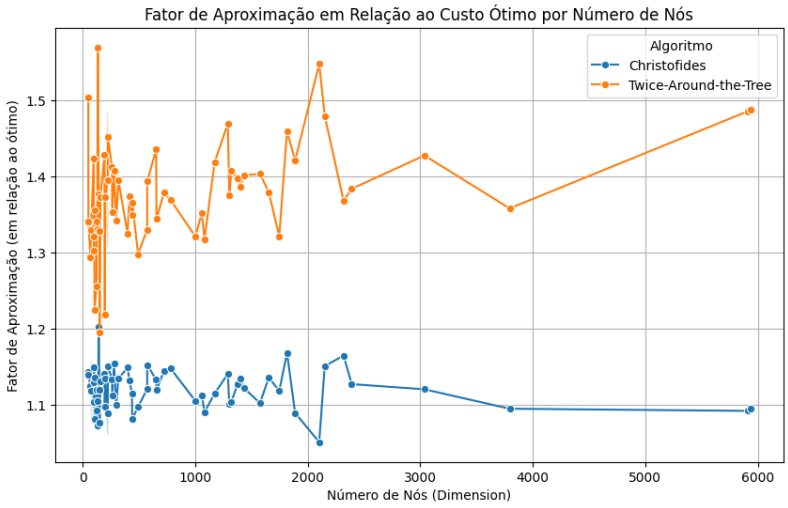

Figure 5. Comparativo - Fator de Aproximação (TATT e Christofides)

Portanto, é possível dizer que o algoritmo Twice-Around-the-Tree é mais eficiente em tempo, enquanto o Christofides se destaca em qualidade da solução. A escolha entre os dois depende da aplicação: o TATT é mais indicado para problemas onde o tempo de execução é crítico, enquanto o Christofides deve ser usado em problemas onde a qualidade da solução é o principal fator.

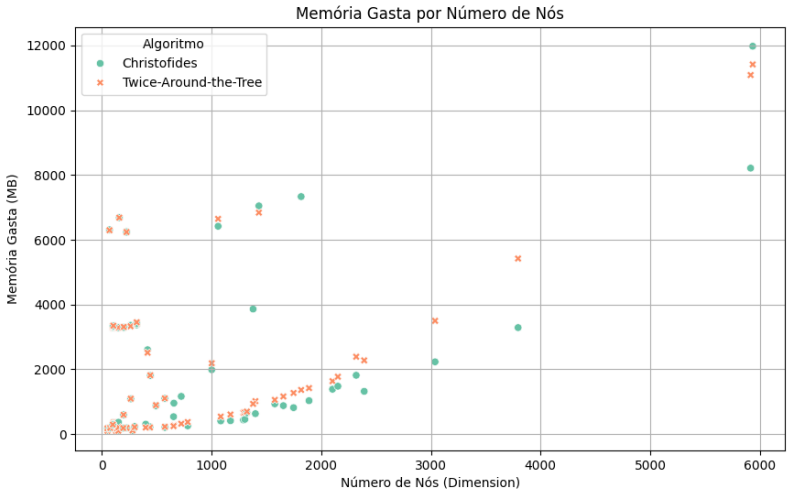

Figure 6. Comparativo - Fator de Aproximação (TATT e Christofides)

4. Conclusão

Neste trabalho, foram implementados e analisados três algoritmos para resolver o Problema do Caixeiro Viajante (TSP) em sua variação euclidiana: Branch-and-Bound, Twice-Around-the-Tree (TATT) e Christofides. Cada abordagem possui características específicas que as tornam adequadas para diferentes contextos e tamanhos de instâncias do problema.

O algoritmo Branch-and-Bound demonstrou ser uma solução exata eficiente para instâncias de menor porte, sendo capaz de garantir a solução ótima ao custo de uma complexidade exponencial. No entanto, sua aplicação prática é limitada devido ao rápido crescimento do espaço de busca, tornando-o impraticável para instâncias maiores, como evidenciado nos experimentos realizados.

Por outro lado, os algoritmos aproximados Twice-Around-the-Tree e Christofides apresentaram desempenhos significativamente superiores em termos de tempo de execução, mesmo para instâncias de maior porte. O TATT destacou-se pela simplicidade e baixa complexidade computacional, fornecendo soluções razoáveis de maneira eficiente. Já o algoritmo de Christofides, apesar de ser mais complexo, apresentou soluções de melhor qualidade com um fator de aproximação garantido de 1,5, tornando-se uma opção robusta para aplicações que requerem um equilíbrio entre qualidade e eficiência.

Os experimentos com as instâncias da TSPLIB confirmaram as propriedades teóricas dos algoritmos e permitiram comparar suas performances em um ambiente controlado. A escolha do algoritmo mais adequado depende, portanto, do contexto do problema, das restrições de tempo de execução e da qualidade esperada da solução. Para aplicações críticas, onde a solução ótima é essencial, o Branch-and-Bound pode ser viável em instâncias pequenas. Para problemas de maior escala ou quando uma solução aproximada de alta qualidade é aceitável, o algoritmo de Christofides é a alternativa mais apropriada.

References

[1] DICKSON, G. *Blossom Algorithm*. Disponível em: https://www.eecs.tufts.edu/~gdicks02/Blossom/Blossom_Algorithm.html. Acesso em: 07 jan. 2025.

[2] NETWORKX DEVELOPERS. *NetworkX Documentation*. Disponível em: https://networkx.github.io/documentation/latest/index.html. Acesso em: 07 jan. 2025.

[3] IGRAH DEVELOPERS. *iGraph*. Disponível em: https://igraph.org. Acesso em: 07 jan. 2025.

[4] ARBEX, M. *Escrita Científica*. Disponível em: https://homepages.dcc.ufmg.br/~mirella/doku.php?id=escrita. Acesso em: 13 jan. 2025.

[5] TSPLIB DEVELOPERS. *TSPLIB*. Disponível em: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Acesso em: 15 jan. 2025.

Table 1. Comparação de Qualidade e Tempo entre Christofides e TATT

	Qualidade (Christofides)	Qualidade (Twice-Around-the-Tree)	Tempo (Christofides)	Tempo (Twice-Around-the-Tree)	Memória (Christofides)	Memória (Twice-Around-the-Tree)
a280.tsp	1.1535923910750974	1.407413844555735	1.588	1.9467	103.515625	111.09375
berlin52.tsp	1.1394901808258793	1.3412907047879643	3.3783	3.3987	100.25	100.66015625
bier127.tsp	1.1198343449580517	1.341181296893765	2.0496	3.6006	103.91796875	96.05078125
ch130.tsp	1.1051187995438831	1.3304021736711864	1.9644	3.5259	99.9609375	99.13671875
ch150.tsp	1.0894664355500165	1.2765735554670907	2.0687	2.703	101.625	100.875
d1291.tsp	1.140339070294354	1.4696225922723294	10.5991	4.2394	435.85546875	659.25
d1655.tsp	1.135435499519306	1.3791056917466216	46.8145	6.8335	877.3984375	1161.70703125
d198.tsp	1.0973616248732347	1.217898165675931	1.2013	1.9193	595.2890625	594.82421875
d2103.tsp	1.0504638281314986	1.5479660858163535	11.4795	9.9816	1385.06640625	1633.10546875
d493.tsp	1.0968983584941412	1.297842661763193	4.5287	1.2338	871.9765625	893.16015625
d657.tsp	1.1191179319969295	1.343845962144337	10.5202	1.6775	955.02734375	267.98046875
eil101.tsp	1.1494394683252713	1.320416864157215	1.9545	3.4233	180.0390625	176.55859375
eil51.tsp	1.1427814645195398	1.5044629646129124	3.4322	3.4233	176.02734375	176.36718875
eil76.tsp	1.1581003677947284	1.3144113752066555	1.889	3.4261	176.84765625	176.2421875
fl1400.tsp	1.1345135104823936	1.3869689212101481	109.4367	4.9011	630.1640625	1016.95703125
fl1577.tsp	1.1018856680497202	1.403348315293885	25.5832	5.9257	933.95703125	1056.59765625
fl3795.tsp	1.09448815631476	1.3580158678055165	553.2724	36.449	3287.35546875	5420.08984375
fl417.tsp	1.131838319435361	1.3740135435791194	2.0717	1.1666	2604.421875	2512.54296875
gil262.tsp	1.1333701964393945	1.4121758522953278	1.6185	1.9575	3360.1640625	3328.97265625
kroA100.tsp	1.0944922778117776	1.2786241767322784	1.8789	3.433	3321.55078125	3281.7265625
kroA150.tsp	1.1193001184254252	1.3241806213640215	2.1317	2.617	3279.0078125	3282.1640625
kroA200.tsp	1.1441997589579367	1.3630777155813317	1.3093	1.8015	3284.8203125	3288.93359375
kroB100.tsp	1.084377148378703	1.1689264212856523	1.8121	3.4035	3288.1171875	3290.2578125
kroB150.tsp	1.1501870453110357	1.3836484477729756	1.1132	1.7821	3291.6328125	3293.07421875
kroB200.tsp	1.125050853017786	1.3829860338688	1.1924	1.8159	3296.04296875	3308.86328125
kroC100.tsp	1.1310969662215922	1.3478500825944204	1.8454	3.3994	3311.2421875	3316.58203125
kroD100.tsp	1.1077182004485164	1.2732833962819095	1.8722	3.3778	3320.3359375	3325.62890625
kroE100.tsp	1.0776965313207785	1.3824279067801104	1.8802	3.4035	3330.62109375	3335.95703125
lin105.tsp	1.135244161602442	1.3560337150576274	1.843	3.3692	3340.296875	3346.78515625
lin318.tsp	1.1244558884547449	1.3834600375230452	1.7785	1.9772	3373.48828125	3403.6484375
linhp318.tsp	1.1430585690135318	1.4063476095551113	1.8033	1.9731	3421.625	3452.1796875
nrw1379.tsp	1.1272479745790147	1.397582278070714	122.4098	5.1974	3860.9296875	933.171875
p654.tsp	1.1328322591037376	1.4355539480460753	2.5035	1.6548	536.9453125	245.65625
pcb1173.tsp	1.1142593117788397	1.418387345544943	34.4545	3.6415	412.17578125	605.2890625
pcb3038.tsp	1.120006578934273	1.4276141242317089	548.2876	25.4724	2230.7109375	3497.11328125
pcb442.tsp	1.0807194777048235	1.3660424299142642	3.0458	1.1804	1807.6015625	1815.359375
pr1002.tsp	1.1049551160041338	1.321177642478497	26.8532	4.3681	1986.703125	2186.48828125
pr107.tsp	1.0810596276686961	1.2242520057728703	2.7271	3.4644	360.1875	353.30859375
pr124.tsp	1.0916093281581203	1.2559877968538857	1.8971	3.517	355.2265625	356.890625
pr136.tsp	1.0723340240602681	1.5698109163978895	1.8637	1.8497	358.44921875	360.3203125
pr144.tsp	1.2027483659415852	1.3768440481741249	1.858	2.8541	361.203125	363.546875
pr152.tsp	1.0764413344809058	1.1943038188622066	2.0425	2.0711	365.01953125	210.19140625
pr226.tsp	1.1500174636964933	1.4519547800169565	1.1595	1.93	190.52734375	188.76171875
pr2392.tsp	1.126834740100962	1.3838323394299639	260.5204	14.5408	1319.76171875	2275.41796875
pr264.tsp	1.1125186208257827	1.3527392235541527	1.1045	2.8581	1087.10546875	1092.5
pr299.tsp	1.0991309537175626	1.341454518891923	1.6502	2.001	233.78515625	214.23046875
pr439.tsp	1.1148608513855556	1.34889204023657	2.1062	1.1917	224.3203125	204.62890625
pr76.tsp	1.0788148433877391	1.3437449907251222	3.4076	1.793	191.8828125	191.68359375
rat195.tsp	1.1399462858596485	1.428206307385696	1.092	1.8087	184.25390625	185.1015625
rat575.tsp	1.15203738299402	1.3936119051174758	7.5458	1.4013	204.64453125	226.4609375
rat783.tsp	1.1472716933589875	1.3688908876607937	18.0679	2.0218	255.32421875	372.421875
rat99.tsp	1.1291758701259091	1.4229769158966652	1.8341	3.3946	289.7265625	290.87890625
rd100.tsp	1.125852393317651	1.3643054128888372	1.9368	3.4335	290.375	292.359375
rd400.tsp	1.1485992817447195	1.3251997084151985	3.881	1.1254	305.78125	201.99609375
rl1304.tsp	1.100302011106245	1.3749172570706139	10.6782	4.431	454.99609375	672.328125
rl1323.tsp	1.1038594506134718	1.4079311455787364	11.1571	4.6341	655.98828125	694.1484375
rl1889.tsp	1.0889502393077495	1.4210436985784263	31.4098	8.8949	1030.6875	1419.43359375
rl5915.tsp	1.0916211668895648	1.485948324227153	349.9387	109.0256	8211.65625	11087.203125
rl5934.tsp	1.094523918296536	1.4881571384122843	424.42	107.3053	11976.33984375	11414.2265625
st70.tsp	1.1246846118784422	1.293854052847015	2.2944	3.3671	6308.1796875	6293.96875
ts225.tsp	1.0603205101173654	1.48456449728238	1.8887	1.8224	6253.296875	6234.43359375
tsp225.tsp	1.1174675407559624	1.3064179216763123	1.1843	1.8487	6234.34765625	6238.30078125
u1060.tsp	1.1116254098476481	1.3517312661960685	35.5212	2.4594	6417.8359375	6644.9140625
u1432.tsp	1.122056121468754	1.4017832290508727	36.4435	3.8062	7045.99609375	6837.93359375
u159.tsp	1.130744209273773	1.3732883652612995	1.9753	1.7944	6683.6171875	6682.30078125
u1817.tsp	1.1677463128197236	1.459791887806316	61.6377	7.8748	7332.421875	1362.80078125
u2152.tsp	1.1504183322525972	1.4797163686512498	78.299	11.7939	1477.46875	1770.41015625
u2319.tsp	1.1643949997214207	1.3683015206772815	280.6655	12.9812	1815.41015625	2388.58203125
u574.tsp	1.12010864585875	1.3299865042196892	6.7009	1.3087	1100.86328125	1102.90234375
u724.tsp	1.1442300739113045	1.3786606512269666	13.0225	1.7821	1164.1328125	320.46875
vm1084.tsp	1.089997206925756	1.3174772301748483	19.3688	3.2669	406.2578125	540.703125
vm1748.tsp	1.1179147006371812	1.3212025099607339	69.7244	8.2167	816.23046875	1270.62890625

Table 2. Análise Comparativa entre os Algoritmos TATT e Christofides

Métrica	Min	Max	Média	Desvio Padrão
Tempo de execução (s) - TATT	1.125400	109.025600	7.270417	17.986843
Tempo de execução (s) - Christofides	1.092000	553.272400	45.895688	116.047413
Qualidade da solução - TATT	1.168926	1.569811	1.369944	0.074386
Qualidade da solução - Christofides	1.050464	1.202748	1.118898	0.027803
Memória Gasta - TATT	96.050781	11414.226	2071.492	2463.333
Memória Gasta - Christofides	99.960938	11976.340	2078.278	2422.928