Trabalho Prático 2
Problema do Caixeiro Viajante

Iago Gabino Gonçalves[1]

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

iagogabino@ufmg.br

Abstract. *This paper explores the resolution of the Traveling Salesman Problem using three methods: Branch and Bound, Twice Around the Tree, and the Christofides algorithm. We describe each technique, focusing on aspects such as complexity and data structures. Experiments are conducted to compare the performance of these methods, as well as memory usage and the quality of approximation (when applicable).*

Resumo. *Este artigo explora a resolução do Problema do Caixeiro Viajante usando três métodos: Branch and Bound, Twice Around the Tree e o algoritmo de Christofides. Descrevemos cada técnica, focando em aspectos como complexidade e estruturas de dados. Foram realizados experimentos para comparar o desempenho dos métodos, além da utilização de memória e qualidade da aproximação (quando aplicável).*

1. Introdução

O Problema do Caixeiro Viajante (TSP) é um dos desafios mais emblemáticos em otimização combinatória e teoria dos grafos. Ele busca determinar a rota mais curta que permite a um caixeiro visitar uma série de cidades e retornar ao ponto de origem, passando apenas uma vez por cada cidade. Com a complexidade do problema crescendo exponencialmente com o aumento do número de cidades, encontrar a solução ótima se torna um desafio computacional significativo.

Para abordar este problema, este estudo explora três métodos algorítmicos: Branch and Bound, que é uma técnica exaustiva conhecida por sua precisão, mas de alta complexidade computacional; e os métodos de aproximação Twice Around the Tree e o algoritmo de Christofides, ambos oferecendo soluções mais eficientes em tempo de execução, ainda que com uma precisão potencialmente menor.

Esta análise compreende a descrição detalhada de cada algoritmo na seção de Implementação, seguida pela apresentação dos experimentos e resultados, onde comparamos o desempenho, uso de memória e qualidade da aproximação dos métodos. A conclusão do trabalho sintetiza os insights obtidos, destacando as situações em que cada técnica se mostra mais vantajosa para resolver o TSP.

2. Implementação

Nesta seção, detalhamos a implementação dos três métodos escolhidos para resolver o Problema do Caixeiro Viajante. Discutiremos as escolhas feitas em termos de estruturas de dados, a lógica por trás das estimativas de custo, e as particularidades de cada algoritmo.

2.1. Branch and Bound

O Branch and Bound (BnB) é uma técnica de otimização exaustiva que explora sistematicamente todas as soluções possíveis para encontrar a melhor. A escolha deste algoritmo deve-se à sua capacidade de fornecer soluções precisas, embora possa ser computacionalmente intensiva.

Para representar o grafo do problema, utilizamos uma matriz de adjacência, convertida a partir de uma estrutura de dados graph fornecida pela biblioteca *tsplib95*. A matriz de adjacência permite um acesso rápido e direto ao custo de cada aresta entre dois nós. Cada elemento adj_matrix[i][j] representa o custo da aresta entre os nós i e j, facilitando a computação dos custos dos caminhos durante a execução do algoritmo.

A função *calculate_bound* no algoritmo Branch and Bound é vital para estimar o custo inferior (bound) de um caminho parcial no Problema do Caixeiro Viajante. Ela inicia calculando o custo acumulado das arestas já percorridas no caminho; caso encontre uma aresta inexistente (indicada por zero na matriz de adjacência), retorna infinito, sinalizando um caminho inviável. Posteriormente, a função adiciona ao custo a menor aresta possível para cada nó ainda não visitado. Este passo envolve selecionar a segunda aresta mais barata que sai de cada nó não visitado, ignorando a aresta de custo zero para si mesmo, para estimar o custo mínimo adicional para incluir esses nós no caminho. Caso um nó não tenha conexões viáveis, a função retorna infinito. O valor resultante, que é a soma dos custos do caminho atual e dos custos mínimos estimados para os nós não visitados, serve como um limite inferior para avaliar a viabilidade de continuar explorando esse caminho, ajudando a focar a busca nos caminhos mais promissores e descartando aqueles que certamente não levarão à solução ótima.

Foi implementado o método Best-First Search para expandir os nós na árvore de busca. Ao contrário do Depth-First Search, o Best-First Search prioriza a expansão dos nós com o menor limite inferior (bound), o que é coerente com a busca pela solução ótima no menor tempo possível. Isso é implementado usando uma fila de prioridades (heapq em Python), onde os nós são automaticamente ordenados pelo seu valor de bound, com complexidade de inserção/remoção $O(\log n)$, onde n é o número de elementos no heap.

A complexidade do Branch and Bound é difícil de determinar de forma exata, pois depende fortemente da estrutura do problema específico e da eficácia dos limites inferiores utilizados. Teoricamente, em pior caso, pode aproximar-se da complexidade de uma busca exaustiva, que é fatorial ($O(n!)$) em relação ao número de cidades. No entanto, na prática, o uso eficiente de limites inferiores pode reduzir significativamente o espaço de busca, melhorando o desempenho do algoritmo.

2.2. Twice Around The Tree

O Twice Around the Tree (TATT) é um método de aproximação para o Problema do Caixeiro Viajante, caracterizado por sua eficiência em instâncias grandes.

O algoritmo inicia com a construção de uma Árvore Geradora Mínima (MST), essencial para conectar todos os vértices do grafo com o menor peso total possível sem formar ciclos. Em seguida, é realizada uma busca em profundidade (DFS) em pré-ordem a partir do primeiro nó, resultando em um caminho que visita cada nó da MST uma vez.

Este caminho é então transformado em um ciclo hamiltoniano, adicionando cada nó não visitado ao ciclo e ignorando os já visitados. O peso total do ciclo é calculado somando os pesos das arestas entre cada par de nós consecutivos no ciclo, utilizando os pesos do grafo original.

A construção da Árvore Geradora Mínima (MST) é realizada utilizando o algoritmo de Kruskal, com a biblioteca *NetworkX*, com uma complexidade de tempo de $O(|E| \log |V|)$, onde $|E|$ é o número de arestas e $|V|$ é o número de vértices no grafo. A DFS sobre a MST tem complexidade de tempo de $O(|V|_{MST} + |E|_{MST})$, considerando $|V|_{MST}$ e $|E|_{MST}$ como o número de vértices e arestas na MST, respectivamente.

A complexidade de espaço do algoritmo TATT é influenciada pela armazenagem da MST e pelo armazenamento do caminho e ciclo hamiltoniano. A MST requer espaço para $|V|_{MST}$ vértices e até $|V|_{MST} - 1$ arestas, enquanto o caminho e ciclo hamiltoniano necessitam de espaço para armazenar até V vértices cada.

Quanto ao fator de aproximação, o TATT é um algoritmo 2-aproximado que não garante a obtenção da solução ótima, mas oferece uma solução aproximada. A qualidade da solução aproximada depende da estrutura específica do grafo. Em alguns casos, o TATT pode se aproximar bastante da solução ótima, enquanto em outros, a discrepância pode ser maior. Este fator é uma consideração importante ao escolher o TATT para instâncias específicas do Problema do Caixeiro Viajante, especialmente quando a eficiência de tempo é prioritária em relação à precisão absoluta da solução.

2.3. Algoritmo de Christofides

O algoritmo de Christofides é uma abordagem de aproximação reconhecida por sua eficiência na resolução do Problema do Caixeiro Viajante. Este método começa com a construção de uma Árvore Geradora Mínima (MST) usando a biblioteca NetworkX, com mesma complexidade analisada anteriormente.

Após a construção da MST, o algoritmo identifica os vértices com grau ímpar na MST e forma um subgrafo contendo esses vértices. Em seguida, realiza um emparelhamento de peso mínimo neste subgrafo, garantindo a maximização do número de emparelhamentos. O emparelhamento resultante é então combinado com a MST para formar um multigrafo, que inclui arestas duplicadas se necessário.

A próxima etapa é encontrar um circuito euleriano neste multigrafo, utilizando novamente as capacidades do NetworkX. Este circuito é transformado em um ciclo hamiltoniano, percorrendo o circuito e visitando cada vértice apenas uma vez. O peso total do ciclo hamiltoniano é calculado somando os pesos das arestas conforme percorremos este ciclo.

De forma diferente do TATT, o algoritmo de Christofides tem complexidade dominada não pela construção da MST, mas sim pelo emparelhamento mínimo no subgrafo induzido pelos vértices de grau ímpar. Este passo possui complexidade $O(|V|^3)$.

Quanto à complexidade de espaço do algoritmo de Christofides, ela é determinada principalmente pela armazenagem de várias estruturas de dados durante a execução do algoritmo. Inicialmente, a MST é construída e armazenada, exigindo espaço para até $|V|$ vértices e $|V| - 1$ arestas. Em seguida, o subgrafo contendo os vértices de grau ímpar é criado e um emparelhamento de peso mínimo é encontrado, o que pode adicio-

nar até $|V|/2$ arestas adicionais ao espaço requerido. Finalmente, o multigrafo resultante da combinação da MST com o emparelhamento é utilizado para encontrar um circuito euleriano, o que pode potencialmente duplicar o número de arestas armazenadas. Portanto, a complexidade de espaço do algoritmo de Christofides é dominada pelo número de vértices e arestas que precisam ser armazenados, o que, no pior caso, é proporcional a $O(|V| + |E|)$, considerando $|E|$ como o número total de arestas no grafo original.

O algoritmo de Christofides é particularmente notável pelo seu fator de aproximação, oferecendo uma solução que é, no pior caso, 1.5 vezes o comprimento do ciclo ótimo para grafos métricos. Esta eficiência torna o algoritmo de Christofides uma escolha valiosa para instâncias do Problema do Caixeiro Viajante onde uma solução aproximada é suficiente e o tempo de execução é uma consideração crítica.

3. Experimentos e Resultados

3.1. Branch and Bound

A solução baseada em BnB foi testada separadamente, uma vez que durante os testes com a *TSPLIB* o algoritmo não encontrou uma solução em menos de 30 minutos para nenhuma instância.

Dessa forma, foram criadas 4 instâncias de teste, com dimensões 10, 15, 20 e 30. Abaixo está a tabela para a execução do algoritmo nessas instâncias:

Instância	Custo	Tempo de Execução (s)	Memória (MB)
test10.tsp	2826	0.203	58.64
test15.tsp	2705	23.04	154.45
test20.tsp	N/A	N/A	N/A
test30.tsp	N/A	N/A	N/A

Tabela 1. Resultados da execução do algoritmo

Este aumento no tempo de execução e no uso de memória pode ser atribuída à natureza exaustiva do algoritmo Branch and Bound. A complexidade de tempo do algoritmo, em pior caso, é exponencial, e o espaço de soluções cresce rapidamente à medida que o número de cidades (vértices no problema do TSP) aumenta. O Branch and Bound explora sistematicamente todas as combinações possíveis para encontrar a rota mais curta, o que pode se tornar inviável computacionalmente para instâncias maiores. Além disso, o uso de memória é impactado pela necessidade de armazenar e gerenciar um número crescente de subproblemas na árvore de busca.

Portanto, enquanto o Branch and Bound é eficaz para instâncias menores do TSP, oferecendo soluções precisas, sua aplicabilidade a instâncias maiores é limitada pela escalada exponencial no tempo de execução e no uso de memória.

Por outro lado, algumas melhorias poderiam ser implementadas, como aquelas focadas em poda eficiente da árvore de busca e gerenciamento de memória. A implementação de técnicas de poda avançadas, como heurísticas refinadas para estimativas de limites inferiores, pode ajudar a eliminar rapidamente caminhos subótimos, reduzindo o espaço de busca. Além disso, a adoção de estruturas de dados mais eficientes pode diminuir o uso de memória significativamente. A paralelização do processo de

busca, distribuindo tarefas entre múltiplos processadores, também pode acelerar a busca pela solução ótima, tornando o Branch and Bound mais viável para instâncias maiores do problema.

3.2. TATT e Christofides

De maneira diferente de BnB, os algoritmos aproximativos analisados conseguem resultados razoáveis na *TSPLIB* em tempos viáveis. A tabela com os resultados detalhados se encontra na Seção 5 (Tabela 2).

A análise dos resultados mostra que, embora o algoritmo de Christofides geralmente tenha um desempenho superior em termos de qualidade da solução, o TATT se destaca por sua rapidez (Figura 1). O TATT, sendo mais rápido, oferece soluções em tempo significativamente menor, o que pode ser vantajoso em situações onde o tempo de resposta é crítico. Por outro lado, o algoritmo de Christofides tende a encontrar soluções mais próximas do ótimo, justificando seu tempo de execução mais longo.

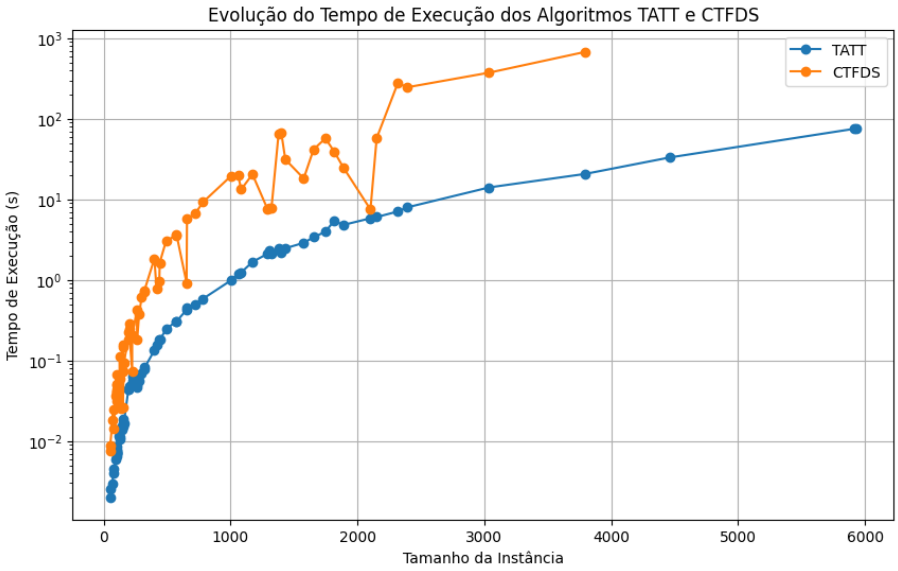

Figura 1. Tempo de execução em escala logarítmica dos algoritmos

Uma análise da diferença entre TATT e CTFDS pode ser visualizada na Figura 2. O gráfico indica a diferença percentual entre o custo obtido em cada instância pelos dois algoritmos. Ou seja, uma diferença de 30% indica que o algoritmo de Christofides retornou uma solução 30% melhor que TATT.

Particularmente, para instâncias maiores, observamos que o TATT pode não manter a mesma qualidade de solução que o Christofides, mas ainda assim se mostra como uma opção viável quando há restrições de tempo. Em contrapartida, o algoritmo de Christofides, mesmo sendo mais lento, mantém uma consistência na qualidade das soluções, aproximando-se do ótimo com uma margem de erro menor, como pode ser observado na Figura 3.

Esta diferença nos faz considerar o contexto de aplicação ao escolher entre os dois algoritmos. Para aplicações que exigem soluções de alta qualidade e onde o tempo de execução é menos crítico, o algoritmo de Christofides se mostra mais apropriado. No

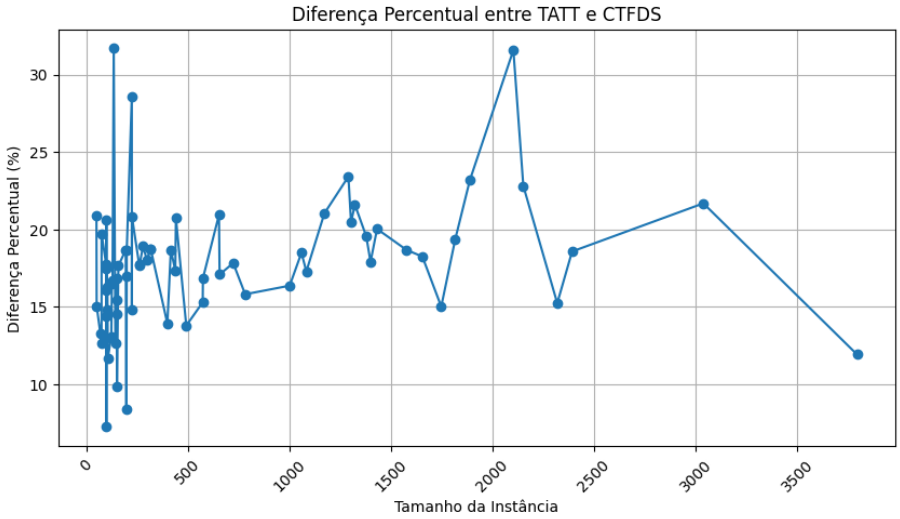

Figura 2. Diferença percentual entre os resultados dos algoritmos

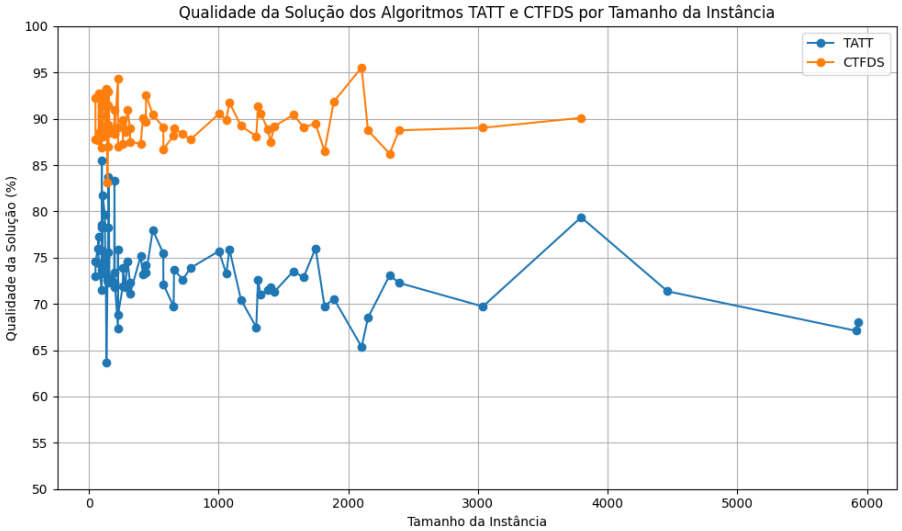

Figura 3. Qualidade da solução (percentual em relação ao ótimo)

entanto, para cenários onde o tempo é um fator limitante, o TATT pode ser a escolha mais adequada, oferecendo uma solução aceitável em um curto período de tempo.

Em relação ao uso de memória, a Figura 4 demonstra um aumento progressivo conforme o tamanho das instâncias se expande, para ambos os algoritmos. Isso se deve pela característica dos dois algoritmos, que gastam um espaço linear em relação ao número de vértices e arestas das Árvores Geradoras Mínimas e, no caso de Christofides, em relação ao subgrafo induzido e ao emparelhamento de peso mínimo, como discutido anteriormente. Ainda que o segundo algoritmo tenha que reservar um espaço extra para o subgrafo e o emparelhamento, os dois algoritmos parecem gastar uma quantidade de memória muito similar, indicando que um gerenciamento de memória eficiente está sendo feito.

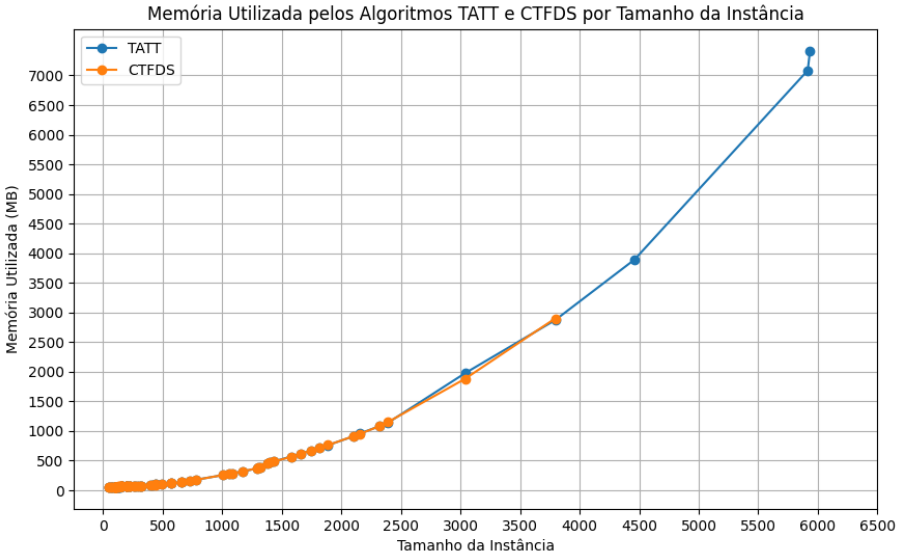

Figura 4. Utilização de Memória (MB) dos algoritmos

4. Conclusão

Após uma análise meticulosa dos experimentos realizados com os algoritmos Twice Around The Tree, Christofides e Branch and Bound, pode-se dizer que cada um oferece benefícios distintos que são melhor aplicados em diferentes contextos do Problema do Caixeiro Viajante. O Branch and Bound, apesar de garantir a solução ótima, mostra-se impraticável para instâncias de grande escala devido à sua complexidade exponencial tanto em tempo quanto em uso de memória. Em contrapartida, os algoritmos aproximativos TATT e Christofides oferecem uma compensação eficaz entre o tempo de execução e a qualidade das soluções, com Christofides frequentemente alcançando resultados mais próximos do ótimo. Notavelmente, o algoritmo TATT se destaca em cenários onde o tempo é um recurso limitado, fornecendo soluções rápidas e razoavelmente eficazes.

5. Tabela de Comparação TATT e Christofides (CTFDS)

Instância	Algoritmo	Resultado	Tempo (s)	Memoria (MB)	Qualidade
eil51.tsp	tatt	584	0.0020079612731933594	56.71484375	72.95%
eil51.tsp	ctfds	462	0.0075130462646484375	57.33984375	92.21%
berlin52.tsp	tatt	10114	0.002507448196411133	56.984375	74.57%
berlin52.tsp	ctfds	8591	0.008812427520751953	57.4375	87.79%
st70.tsp	tatt	888	0.0029942989349365234	57.1640625	76.01%
st70.tsp	ctfds	770	0.01800060272216797	58.35546875	87.66%
eil76.tsp	tatt	696	0.004510641098022461	57.04296875	77.30%
eil76.tsp	ctfds	608	0.024331331253051758	58.5234375	88.49%
pr76.tsp	tatt	145336	0.003998756408691406	57.77734375	74.42%
pr76.tsp	ctfds	116684	0.014076709747314453	58.16015625	92.69%
rat99.tsp	tatt	1693	0.005999088287353516	58.140625	71.53%
rat99.tsp	ctfds	1393	0.03629803657531738	58.8984375	86.93%
kroA100.tsp	tatt	27210	0.0065152645111083984	58.46875	78.21%
kroA100.tsp	ctfds	23293	0.04358482360839844	59.33984375	91.37%
kroB100.tsp	tatt	25885	0.007706642150878906	58.65234375	85.54%
kroB100.tsp	ctfds	24012	0.03156113624572754	59.01171875	92.21%
kroC100.tsp	tatt	27968	0.0075168609619140625	58.73046875	74.19%
kroC100.tsp	ctfds	23470	0.040023088455200195	58.83984375	88.41%
kroD100.tsp	tatt	27112	0.00702667236328125	58.59375	78.54%
kroD100.tsp	ctfds	23583	0.040873050689697266	59.140625	90.29%
kroE100.tsp	tatt	29965	0.006854057312011719	58.46484375	73.65%
kroE100.tsp	ctfds	23783	0.04877519607543945	59.0	92.79%
rd100.tsp	tatt	10790	0.00700831413269043	58.54296875	73.31%
rd100.tsp	ctfds	8906	0.06662797927856445	59.3125	88.82%
eil101.tsp	tatt	830	0.0072062015533447266	58.51171875	75.78%
eil101.tsp	ctfds	707	0.05163073539733887	59.05078125	88.97%
lin105.tsp	tatt	19495	0.008513212203979492	58.73828125	73.76%
lin105.tsp	ctfds	16320	0.04161500930786133	59.125	88.11%
pr107.tsp	tatt	54237	0.007272243499755859	58.62890625	81.68%
pr107.tsp	ctfds	47891	0.034822702407836914	59.5703125	92.51%
pr124.tsp	tatt	74139	0.01139688491821289	59.4765625	79.62%
pr124.tsp	ctfds	64438	0.027036666870117188	59.984375	91.61%
bier127.tsp	tatt	158626	0.010601997375488281	59.83203125	74.57%
bier127.tsp	ctfds	132448	0.1120920181274414	61.21484375	89.30%
ch130.tsp	tatt	8129	0.011290788650512695	59.7890625	75.16%
ch130.tsp	ctfds	6773	0.05926370620727539	60.10546875	90.21%
pr136.tsp	tatt	151904	0.014548063278198242	60.53515625	63.71%
pr136.tsp	ctfds	103771	0.02502274513244629	61.1484375	93.26%
pr144.tsp	tatt	80599	0.013682365417480469	60.56640625	72.63%
pr144.tsp	ctfds	70404	0.026152849197387695	61.05859375	83.14%
ch150.tsp	tatt	8347	0.016304492950439453	60.62890625	78.21%
ch150.tsp	ctfds	7135	0.07322025299072266	61.27734375	91.49%
kroA150.tsp	tatt	35119	0.015024662017822266	60.71484375	75.53%
kroA150.tsp	ctfds	29688	0.14765095710754395	62.0546875	89.34%

Instância	Algoritmo	Resultado	Tempo (s)	Memoria (MB)	Qualidade
kroB150.tsp	tatt	36150	0.016619443893432617	60.71484375	72.28%
kroB150.tsp	ctfds	30053	0.1561570167541504	62.04296875	86.95%
pr152.tsp	tatt	87995	0.01902174949645996	60.953125	83.73%
pr152.tsp	ctfds	79311	0.026068687438964844	61.36328125	92.90%
u159.tsp	tatt	57791	0.01654791831970215	61.44140625	72.81%
u159.tsp	ctfds	47586	0.0943303108215332	62.67578125	88.43%
rat195.tsp	tatt	3234	0.04345202445983887	64.0078125	71.83%
rat195.tsp	ctfds	2630	0.18305325508117676	65.0078125	88.33%
d198.tsp	tatt	18936	0.04352998733520508	64.9765625	83.33%
d198.tsp	ctfds	17347	0.22710943222045898	66.05859375	90.97%
kroA200.tsp	tatt	40028	0.046103715896606445	65.4765625	73.37%
kroA200.tsp	ctfds	33232	0.2888987064361572	66.09375	88.37%
kroB200.tsp	tatt	40703	0.0477139949798584	64.91015625	72.32%
kroB200.tsp	ctfds	33119	0.24585652351379395	66.203125	88.88%
ts225.tsp	tatt	188008	0.06235647201538086	67.21484375	67.36%
ts225.tsp	ctfds	134282	0.07414579391479492	67.6484375	94.31%
tsp225.tsp	tatt	5161	0.0530092716217041	66.671875	75.88%
tsp225.tsp	ctfds	4397	0.20845985412597656	67.640625	89.06%
pr226.tsp	tatt	116694	0.0560755729675293	67.56640625	68.87%
pr226.tsp	ctfds	92415	0.19899225234985352	68.296875	86.97%
gil262.tsp	tatt	3308	0.049599409103393555	69.55859375	71.89%
gil262.tsp	ctfds	2724	0.42756152153015137	70.42578125	87.30%
pr264.tsp	tatt	66470	0.046376705169677734	70.73828125	73.92%
pr264.tsp	ctfds	54662	0.17998456954956055	72.03515625	89.89%
a280.tsp	tatt	3592	0.05641937255859375	71.1640625	71.80%
a280.tsp	ctfds	2913	0.3735079765319824	70.6171875	88.53%
pr299.tsp	tatt	64645	0.06813430786132812	73.50390625	74.55%
pr299.tsp	ctfds	52969	0.60317063331604	73.640625	90.98%
lin318.tsp	tatt	58138	0.08179545402526855	76.296875	72.29%
lin318.tsp	ctfds	47246	0.7256131172180176	76.71484375	88.96%
linhp318.tsp	tatt	58138	0.07898473739624023	75.87890625	71.12%
linhp318.tsp	ctfds	47246	0.7093472480773926	76.76953125	87.51%
rd400.tsp	tatt	20331	0.13332915306091309	89.31640625	75.16%
rd400.tsp	ctfds	17501	1.831794261932373	91.2890625	87.32%
fl417.tsp	tatt	16200	0.15719866752624512	90.33984375	73.22%
fl417.tsp	ctfds	13175	0.7810888290405273	90.55859375	90.03%
pr439.tsp	tatt	144623	0.1802980899810791	95.99609375	74.14%
pr439.tsp	ctfds	119525	0.968463659286499	94.25390625	89.70%
pcb442.tsp	tatt	69223	0.17926502227783203	96.18359375	73.35%
pcb442.tsp	ctfds	54868	1.6330785751342773	96.89453125	92.55%
d493.tsp	tatt	44884	0.24514222145080566	105.19140625	77.98%
d493.tsp	ctfds	38697	3.0764973163604736	104.76171875	90.45%
u574.tsp	tatt	48902	0.30798935890197754	117.1015625	75.47%
u574.tsp	ctfds	41424	3.5676157474517822	117.84375	89.09%
rat575.tsp	tatt	9393	0.3061513900756836	114.65234375	72.11%
rat575.tsp	ctfds	7811	3.6246349811553955	114.6953125	86.71%

Instância	Algoritmo	Resultado	Tempo (s)	Memoria (MB)	Qualidade
p654.tsp	tatt	49699	0.42573118209838867	132.60546875	69.71%
p654.tsp	ctfds	39292	0.9165177345275879	133.078125	88.17%
d657.tsp	tatt	66342	0.44495511054992676	134.0625	73.73%
d657.tsp	ctfds	55007	5.740518569946289	135.5703125	88.92%
u724.tsp	tatt	57721	0.49324607849121094	163.3046875	72.61%
u724.tsp	ctfds	47431	6.732421159744263	163.13671875	88.36%
rat783.tsp	tatt	11921	0.5832662582397461	177.71484375	73.87%
rat783.tsp	ctfds	10036	9.229254245758057	177.12890625	87.74%
pr1002.tsp	tatt	342216	0.9827172756195068	254.14453125	75.70%
pr1002.tsp	ctfds	286203	19.338133573532104	254.64453125	90.51%
u1060.tsp	tatt	305849	1.1822197437286377	273.15234375	73.27%
u1060.tsp	ctfds	249289	19.887624979019165	276.3515625	89.89%
vm1084.tsp	tatt	315277	1.2378864288330078	282.9140625	75.90%
vm1084.tsp	ctfds	260770	13.284540891647339	282.87109375	91.77%
pcb1173.tsp	tatt	80761	1.6718637943267822	316.953125	70.44%
pcb1173.tsp	ctfds	63767	20.44710946083069	319.7890625	89.22%
d1291.tsp	tatt	75282	2.1077749729156494	367.9765625	67.48%
d1291.tsp	ctfds	57655	7.612111330032349	369.12109375	88.11%
rl1304.tsp	tatt	348309	2.294562578201294	376.1875	72.62%
rl1304.tsp	ctfds	277037	7.770598411560059	376.19140625	91.30%
rl1323.tsp	tatt	380427	2.1455650329589844	385.49609375	71.03%
rl1323.tsp	ctfds	298272	7.850752353668213	385.91015625	90.59%
nrw1379.tsp	tatt	79188	2.4907078742980957	458.1484375	71.52%
nrw1379.tsp	ctfds	63703	65.38710761070251	456.8359375	88.91%
fl1400.tsp	tatt	28022	2.214573383331299	466.33984375	71.83%
fl1400.tsp	ctfds	23005	67.52929735183716	465.48828125	87.49%
u1432.tsp	tatt	214417	2.468414545059204	485.09375	71.34%
u1432.tsp	ctfds	171469	31.267587900161743	485.15625	89.21%
fl1577.tsp	tatt	30257	2.8853845596313477	568.26953125	73.53%
fl1577.tsp	ctfds	24606	18.149416208267212	561.078125	90.42%
d1655.tsp	tatt	85285	3.3903191089630127	612.0546875	72.85%
d1655.tsp	ctfds	69721	41.80337190628052	611.45703125	89.11%
vm1748.tsp	tatt	442756	3.965996026992798	670.640625	76.01%
vm1748.tsp	ctfds	376275	57.04537320137024	671.203125	89.44%
u1817.tsp	tatt	82013	5.417550086975098	711.8203125	69.75%
u1817.tsp	ctfds	66125	39.219348669052124	712.6171875	86.50%
rl1889.tsp	tatt	448611	4.8109235763549805	761.3671875	70.56%
rl1889.tsp	ctfds	344580	24.889561891555786	762.09375	91.86%
d2103.tsp	tatt	123086	5.818941831588745	911.0390625	65.36%
d2103.tsp	ctfds	84209	7.491679668426514	912.21875	95.54%
u2152.tsp	tatt	93801	6.031611919403076	955.828125	68.50%
u2152.tsp	ctfds	72407	57.54789710044861	947.23046875	88.74%
u2319.tsp	tatt	320506	7.067569732666016	1079.35546875	73.09%
u2319.tsp	ctfds	271669	278.2410237789154	1080.95703125	86.23%
pr2392.tsp	tatt	523107	7.946911573410034	1139.8125	72.27%
pr2392.tsp	ctfds	425940	245.75655579566956	1151.19921875	88.75%

Instância	Algoritmo	Resultado	Tempo (s)	Memoria (MB)	Qualidade
pcb3038.tsp	tatt	197498	14.058192729949951	1974.4296875	69.72%
pcb3038.tsp	ctfds	154670	373.28386330604553	1880.15625	89.02%
fl3795.tsp	tatt	36256	20.673224687576294	2871.0234375	79.36%
fl3795.tsp	ctfds	31935	675.753710269928	2898.76953125	90.10%
fnl4461.tsp	tatt	255829	33.085	3892.44	71.36%
fnl4461.tsp	ctfds	N/A	N/A	N/A	N/A
rl5915.tsp	tatt	842903	75.016	7084.31	67.09%
rl5915.tsp	ctfds	N/A	N/A	N/A	N/A
rl5934.tsp	tatt	817449	74.568	7415.55	68.02%
rl5934.tsp	ctfds	N/A	N/A	N/A	N/A
rl11849.tsp	tatt	N/A	N/A	N/A	N/A
rl11849.tsp	ctfds	N/A	N/A	N/A	N/A
usa13509.tsp	tatt	N/A	N/A	N/A	N/A
usa13509.tsp	ctfds	N/A	N/A	N/A	N/A
brd14051.tsp	tatt	N/A	N/A	N/A	N/A
brd14051.tsp	ctfds	N/A	N/A	N/A	N/A
d15112.tsp	tatt	N/A	N/A	N/A	N/A
d15112.tsp	ctfds	N/A	N/A	N/A	N/A
d18512.tsp	tatt	N/A	N/A	N/A	N/A
d18512.tsp	ctfds	N/A	N/A	N/A	N/A

Tabela 2. Comparação TATT e CTFDS

Referências

BrainKart contributors (2023). Approximation algorithms for the traveling salesman problem. [Online; acessado em 06/12/2023].

JavaTpoint contributors (2023). Traveling salesperson problem using branch and bound. [Online; acessado em 04/12/2023].

Levitin, A. (2007). *Introduction to the Design Analysis of Algorithms*. Pearson, 3rd edition.

Wikipedia contributors (2023). Christofides algorithm — Wikipedia, the free encyclopedia. [Online; acessado em 06/12/2023].