Problema do Caixeiro Viajante
TP2 - Algoritmos II

Daniel Andrade Carmo[1]

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

Abstract. *This article discuss the NP-Hard Travelling Salesman problem and possible alternative solutions to solve the euclidean variant of it. The main objective is to compare exact and approximate approaches in terms of time, space and quality of three different algorithms implemented.*

Resumo. *Este artigo discute o problema NP-Difícil do Caixeiro Viajante e possíveis soluções alternativas para resolver a variante euclidiana do mesmo. O principal objetivo é comparar abordagens exatas e aproximativas em termos de tempo, espaço e qualidade das soluções apresentadas dos três algoritmos implementados.*

1. Introdução

Este projeto aborda o Problema do Caixeiro Viajante (TSP) que consiste em determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem ao final do trajeto. Esta definição pode ser também descrita como o problema de se encontrar o circuito Hamiltoniano de peso mínimo em um grafo.

O TSP possui inúmeras aplicações práticas, como em logística, por exemplo, e, por ser um problema NP-Completo, requer abordagens aproximativas para ser resolvido em tempo hábil. Portanto, discutiremos uma abordagem exata, usando o algoritmo baseado na técnica de Branch-and-Bound, e a compararemos com duas abordagens aproximativas para resolver a variante euclidiana do TSP, utilizando os algoritmos Twice-Around-The-Tree e Christofides.

2. Problema do Caixeiro Viajante Euclidiano

O Problema do Caixeiro viajante consiste em determinar o circuito Hamiltoniano de peso mínimo em um grafo em que as arestas têm pesos determinados. Isto é, encontrar o menor caminho que a partir de um vértice percorre todos os outros apenas uma única vez e retorna para o nó inicial, completando o circuito.

Para que possamos utilizar os algoritmos de Christofides e Twice-Around-The-Tree, devemos considerar um caso particular do TSP, em que a função de custo do grafo seja uma métrica. Na prática, muitas instâncias do problema no mundo real satisfazem esta restrição. O problema moldado desta forma é chamado de Problema do Caixeiro Viajante Euclidiano, que definiremos formalmente a seguir.

Seja $G(V, E)$ um grafo com o conjunto V de vértices e o conjunto E de arestas. Seja c a função de uscto do grafo.

1. $\forall i, j \in V : \mathrm{c}(i, j) \geq 0$

2. $\forall i, j \in V : \mathrm{c}(i, j) = 0 \Leftrightarrow i = j$
3. $\forall i, j \in V : \mathrm{c}(i, j) = \mathrm{c}(j, i)$
4. $\forall i, j, k \in V : \mathrm{c}(i, j) \leq \mathrm{c}(i, k) + \mathrm{c}(k, j)$

Como lidamos com o problema euclidiano, o custo de cada aresta é a distância euclidiana entre dois vértices i e j do grafo no espaço, que pode ser formalmente calculada por:

Distância Euclidiana: $\mathrm{c}(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

3. Algoritmos

Como dito anteriormente, para este trabalho foram implementados três algoritmos para que fossem comparadas as soluções dos mesmos nos quesitos tempo, memória gasta e qualidade da solução. Como o Problema do Caixeiro Viajante é um problema NP-Difícil, não podemos resolvê-lo de forma determinística rapidamente, o que exige que utilizemos abordagens aproximativas como as que serão descritas a seguir. Como já demonstrado por seus criadores, os algoritmos aqui apresentados retornam, no máximo, soluções duas vezes piores que o ótimo garantido.

3.1. Branch and Bound: o algoritmo exato

A ideia para o algoritmo de Branch and Bound (BNB) surgiu de uma adaptação do algoritmo de Backtracking tradicional. No BNB é adicionada ao algoritmo a capacidade de calcular, para cada nó da árvore de busca, um limiar (bound) para o valor da função objetivo de todas as soluções que puderem ser obtidas a partir deste nó. Além disso, é armazenado o valor da melhor solução já explorada até o momento. Dessa forma, podemos realizar podas na árvore de busca se o limiar de um nó for pior que uma solução melhor já encontrada, economizando em espaço e tempo no cálculo da solução exata do problema.

Para o Problema do Caixeiro Viajante, calculamos o limiar para a função objetivo da forma que descreveremos a seguir. O algoritmo é uma adaptação direta do algoritmo de backtracking para a computação de um Circuito Hamiltoniano. Precisamos incluir uma forma de podar ramos que levem a caminhos mais longos/menos custosos que os já vistos. Como devemos sempre chegar e sair de um vértice no grafo, podemos usar como estimativa as somas das duas arestas de menor peso incidentes em cada vértice e, como cada aresta seria contada duas vezes, dividimos esse valor por 2.

Para calcular de maneira eficiente o limiar da função objetivo de cada nó, guardamos, a cada novo nó criado, qual a solução de caminho foi proposta para que o bound fosse calculado. Dessa forma, foi necessário apenas atualizar as arestas correspondentes ao dois últimos nós do conjunto solução parcial. Assim, o custo de atualização da solução do limiar se tornou constante e a complexidade de espaço $O(|V|)$ para cada nó da árvore de busca.

Como estrutura de dados para armazenar os nós e definir qual a ordem de exploração das soluções, optei por utilizar uma Priority Queue. Esta estrutura de dados, da forma como é implementada pela biblioteca padrão de Python, gasta tempo $O(\log n)$ para inserir e remover nós da fila. A prioridade dos nós é dada pelo tamanho da solução, priorizando os melhores limites inferiores calculados. Isto é o que caracteriza uma busca best-first. Assim, não são feitas muitas computações desnecessárias, convergindo mais rapidamente para a solução ótima exata e economizando tempo e recursos computacionais.

Também foram ignorados caminhos que resultassem na mesma solução mas com a ordem reversa.

O algoritmo de Branch and Bound implementado, por fim, retorna a solução ótima, isto é, o ciclo hamiltoniano de menor custo no grafo de entrada. No entanto, pela natureza combinatória do problema, percebemos, durante os testes, que não é muito interessante utilizar essa abordagem para problemas com grafos grandes, já que a complexidade do algoritmo se explode em ordem de $O(n!)$, como em uma abordagem por força bruta.

3.2. Twice-Around-The-Tree

O algoritmo Twice-Around-The-Tree é uma solução proposta para resolvermos o Problema do Caixeiro Viajante Euclidiano de uma forma extremamente mais rápida, abrindo mão da exatidão da solução final proposta. A ideia é encontrar a Árvore Geradora Mínima (MST) do grafo em questão e percorrê-la em pré-ordem, encontrando uma solução aproximada para o circuito Hamiltoniano de peso mínimo.

O principal custo envolvido na execução do Twice-Around-The-Tree é o cálculo da MST, que pode ser eficientemente feito se utilizarmos os algoritmos $O(|E|\cdot \log|V|)$, de Prim ou Kruskal. Para percorrermos os nós da MST em pré-ordem, utilizamos de uma busca em profundidade, que tem custo $O(|E|+|V|)$. Logo, o custo total do algoritmo Twice-Around-The-Tree é dominado por $O(|E|\cdot \log|V|)$.

Este algoritmo, apesar de não retornar a melhor solução exata, possui uma complexidade ordens de grandeza menor que o algoritmo de Branch and Bound, que é $O(n!)$, mantendo uma garantia de ser, no máximo, duas vezes pior que o algoritmo exato, como provado por seus criadores para o Problema do Caixeiro Viajante Euclidiano. Apesar de não ser uma solução ideal, pode ser útil em inúmeras situações da vida real, em que precisamos calcular um circuito Hamiltoniano de peso mínimo.

3.3. Algoritmo de Christofides

O algoritmo de Christofides é uma modificação do algoritmo Twice-Around-The-Tree que reduz ainda mais a diferença entre a solução do algoritmo aproximativo e a solução ótima exata.

O algoritmo Twice-Around-The-Tree constrói um circuito Hamiltoniano a partir do circuito Euleriano no multigrafo com as arestas da MST duplicadas. Partindo desta ideia, Christofides percebeu que poderíamos transformar um circuito Euleriano em um Hamiltoniano ao encontrar o matching perfeito de peso mínimo entre os vértices de grau ímpar na MST, e construir um multigrafo com as arestas da MST e do matching. Depois, podemos transformar o circuito Euleriano no Hamiltoniano da mesma forma que fizemos no algoritmo de Twice-Around-The-Tree.

O principal custo envolvido no cálculo feito pelo algoritmo de Christofides é a computação do matching de peso mínimo, que é feito pelo algoritmo de Edmonds em $O(|V|^2 \cdot |E|)$ com complexidade linear no tamanho da entrada. Isto torna o algoritmo mais custoso que o Twice-Around-The-Tree.

No entanto, apesar de uma maior complexidade, o algoritmo de Christofides é sabidamente mais preciso que o Twice-Around-The-Tree, garantindo, como provado por

Christofides, soluções, no máximo, 1.5 vezes piores que as soluções ótimas exatas. Isto pode ser vantajoso em problemas da vida real em que uma maior precisão é mais importante que o tempo de execução do algoritmo, o que é frequentemente visto na prática.

4. Experimentos

Para demonstrar na prática os aspectos de tempo, espaço e precisão dos algoritmos previamente discutidos, foram realizados testes com instâncias do Problema do Caixeiro Viajante Euclidiano da biblioteca TSPLIB95, da Universidade de Heidelberg, na Alemanha. Os algoritmos foram executados em 78 instâncias do problema utilizando as bibliotecas Time e Tracemalloc para avaliar o uso de tempo e memória na prática. Além disso, foi utilizado o limiar ótimo para cada instância, fornecido pela TSPLIB95, para metrificar o desempenho dos algoritmos aproximativos.

Inicialmente, os algoritmos foram executados, um por um, para cada instância do TSP, respeitando um limite máximo de 30 minutos de execução. Caso o algoritmo executasse por mais de 30 minutos sem retornar uma resposta, a instância do problema era deixada de lado e outra instância era testada. Ao iniciar os testes, se tornou evidente que o algoritmo de Branch and Bound não executaria para nenhuma das instâncias do conjunto de testes propostos, uma vez que excedeu os 30 minutos nas duas menores instâncias, a eil51 e a berlin52. No entanto, os algoritmos Twice-Around-The-Tree e o Christofides executaram em menos de 30 minutos na maioria das instâncias, tendo problemas em poucos casos.

4.1. Tempo de execução

Cada algoritmo foi executado para cada uma das 78 instâncias do dataset da TSPLIB95. No entanto, o algoritmo de Branch and Bound não executou para nenhuma das instâncias do conjunto de testes em menos de 30 minutos, o que, de forma clara, demonstra na prática a complexidade de $O(n!)$ citada anteriormente. A menor instância possuía um grafo de 51 vértices, então, na tentativa de validar a corretude do algoritmo, busquei instâncias de tamanhos menores, que funcionaram corretamente. Apenas instâncias de menos de 20 vértices foram capazes de serem avaliadas em menos de 30 minutos, sendo todos os outputs para o Branch and Bound iguais a NA, o que demonstrou a inviabilidade de utilizarmos este algoritmo em problemas reais de tamanho considerável.

Observando o gráfico na Figura 1, podemos notar que para instâncias pequenas, os algoritmos Twice-Around-The-Tree e Christofides gastaram quase a mesma quantidade de tempo para nos retornar uma solução. No entanto, para instâncias de tamanho maior que 200 nós, o tempo de execução explodiu para o algoritmo de Christofides e a curva se distanciou abruptamente do algoritmo Twice-Around-The-Tree. Isto pode ser explicado pela diferença de complexidade assintótica entre os dois algoritmos. Apesar de serem baseados na mesma ideia, o cálculo do matching de peso mínimo no algoritmo de Christofides introduz o fator quadrático $O(|V|^2 \cdot |E|)$, que, na prática, foi bem mais lento que o Twice-Around-The-Tree, que tem complexidade $O(|E| \cdot \log|V|)$. Dada esta diferença de complexidade de tempo, o algoritmo de Christofides não conseguiu executar instâncias de mais de 4000 nós em menos de 30 minutos, sendo o gráfico truncado na última instância completamente resolvida por este algoritmo.

A partir dos experimentos, podemos observar que se a instância for muito grande e o tempo de processamento fizer diferença em um problema prático, talvez seja mais

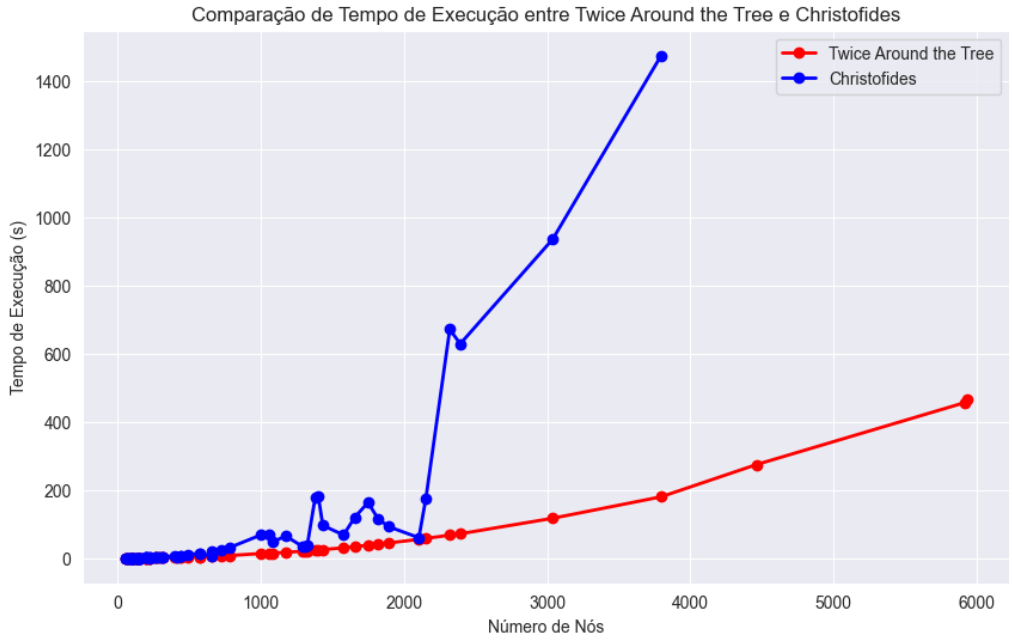

Figure 1. Tempo gasto para execução dos algoritmos de acordo com o número de nós no grafo

interessante utilizar a abordagem aproximativa do Twice-Around-The-Tree, tendo um resultado, no máximo, duas vezes pior que o ótimo. O uso do algoritmo de Branch and Bound é inviável se a instância for grande, já que, apesar de computar a solução ótima exata, a execução do algoritmo toma consideravelmente mais tempo.

4.2. Consumo de memória

Para o consumo de memória, os resultados não foram muito diferentes do esperado. O algoritmo de Branch and Bound, apesar de não poder ser executado em menos de 30 minutos, demonstrou um aumento significativo no consumo de memória para instâncias maiores. Para a instância fl3795, por exemplo, que conta com 3795 nós, o kernel do Python foi morto pelo sistema operacional, ao tentar alocar mais de 50gb de memória para execução do algoritmo. Isto pode ser explicado pelo custo $O(|V|)$ para cada nó da árvore de busca, como explicado anteriormente.

Podemos observar, a partir dos experimentos da TSPLIB95, seus resultados na Figura 2. Não há uma diferença significativa no consumo de memória entre os dois algoritmos aproximativos utilizados, uma vez que os algoritmos têm a mesma ideia base, diferenciando-se apenas no cálculo do matching de peso mínimo feito pelo algoritmo de Christofides. Os dois algoritmos podem ser usados na prática para instâncias consideravelmente grandes, uma vez que consumiram, na maior instância com quase 6000 nós, apenas 6gb de memória total.

4.3. Proximidade do limiar ótimo

Como previamente ressaltado, os algoritmos de CHRItofides e Twice-Around-The-Tree são ambos aproximativos para solucionar o Problema do Caixeiro Viajante. O Twice-Around-The-Tree garante sempre uma solução no máximo 2 vezes pior que o limiar ótimo

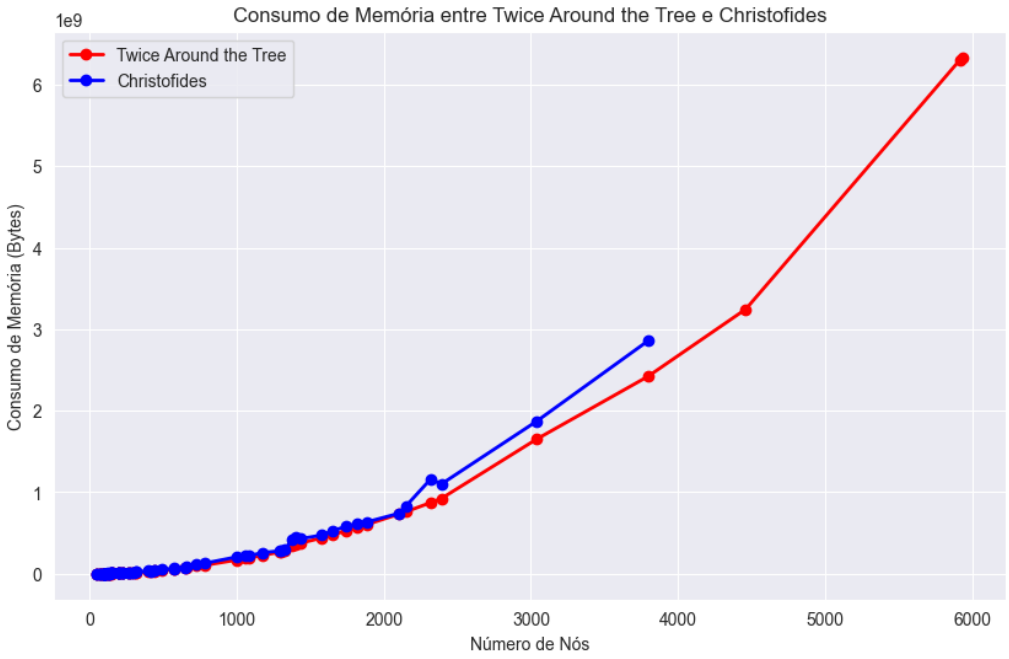

Figure 2. Consumo de memória de acordo com o número de nós do grafo

e o algoritmo de Christofides sempre garante uma solução 1.5-aproximada para a versão euclidiana do TSP.

Na Figura 3 podemos observar um gráfico plotado a partir dos resultados obtidos pelas instâncias do dataset da TSPLIB95. Como o TSP consiste em encontrar o circuito Hamiltoniano de peso mínimo no grafo, quanto menor a solução, melhor.

Na prática, pudemos observar que os algoritmos aproximativos implementados têm, na maioria dos casos, soluções muito próximas do limiar ideal para os problemas, não tendo chegado nenhuma vez ao caso em que a solução do Twice-Around-The-Tree é duas vezes pior que o limiar estipulado. Isto demonstra que para problemas do mundo real, em que a precisão não é extremamente necessária, os algoritmos aproximativos provavelmente têm melhor custo-benefício se comparados a algoritmos que calculam respostas exatas para o problema.

5. Conclusão

Neste estudo, exploramos a implementação de algoritmos para resolver o Problema do Caixeiro Viajante euclidiano. Implementamos três abordagens distintas: Branch and Bound, Christofides e Twice Around the Tree, cada uma apresentando diferentes características em termos de precisão e eficiência computacional.

O algoritmo Branch and Bound, apesar de sempre retornar a solução exata, devido a sua complexidade assintótica elevada não pôde executar nas instâncias do dataset de testes. O uso excessivo de tempo e memória demonstrou que, na prática, se tivermos instâncias médias/grandes, não vale a pena utilizar este algoritmo, sendo mais rápido e menos custoso o uso de abordagens aproximativas. Como em muitas situações do mundo real a precisão extrema não é necessária, o uso de algoritmos como o Twice-Around-The-Tree ou CHRItofides pode ser benéfico.

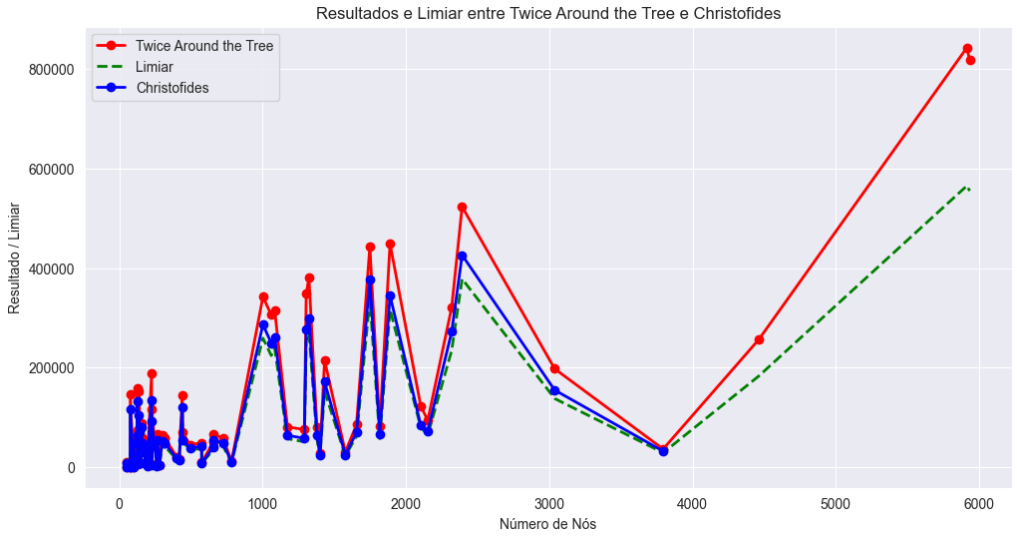

Figure 3. Proximidade entre o resultado aproximado e o limiar ótimo

Tanto o Twice Around the Tree quanto o algoritmo de Christofides mostraram eficácia na obtenção de soluções aproximadas. O primeiro, com sua abordagem de construção de uma árvore geradora mínima e busca em profundidade, revelou-se especialmente rápido, embora menos preciso. Enquanto isso, o algoritmo de Christofides, com um fator aproximativo de 1.5 em relação ao ótimo, obteve soluções mais precisas, sacrificando um pouco o tempo de execução e apresentando um uso ligeiramente maior de memória.

Ao analisar o desempenho, observamos que o Twice Around The Tree apresentou tempos de execução significativamente menores em comparação ao Christofides para instâncias maiores, além de uma ligeira vantagem em relação ao uso de memória. No entanto, o Christofides destacou-se ao produzir soluções mais próximas do ótimo, sendo uma melhor opção para situações reais em que uma maior precisão é mais importante que o tempo em que obtemos a solução para o Problema do Caixeiro Viajante.

Portanto, cada algoritmo apresenta benefícios e malefícios entre precisão, eficiência e uso de memória, sendo adequado para diferentes tamanhos de instâncias do Problema do Caixeiro Viajante. A escolha do algoritmo ideal dependerá dos requisitos específicos de cada aplicação, considerando a relação entre a qualidade da solução desejada, o tempo disponível para sua obtenção e as restrição de memória disponível no momento de execução.

Instância	Algoritmo	Nós	Limiar	Resultado	Qualidade	Tempo(s)	Bytes
eil51	TATT	51	426	584	1.37x	0.04	617165
eil51	CHRI	51	426	462	1.08x	0.06	747445
berlin52	TATT	52	7542	10114	1.34x	0.04	518952
berlin52	CHRI	52	7542	8591	1.14x	0.06	721336
st70	TATT	70	675	888	1.32x	0.07	850420
st70	CHRI	70	675	770	1.14x	0.09	1145938
eil76	TATT	76	538	696	1.29x	0.07	941108
eil76	CHRI	76	538	608	1.13x	0.10	1358677
pr76	TATT	76	108159	145336	1.34x	0.08	1142972
pr76	CHRI	76	108159	116684	1.08x	0.10	1338118
rat99	TATT	99	1211	1693	1.40x	0.12	1749547
rat99	CHRI	99	1211	1393	1.15x	0.19	2330178
kroA100	TATT	100	21282	27210	1.28x	0.13	1949024
kroA100	CHRI	100	21282	23293	1.09x	0.22	2529012
kroB100	TATT	100	22141	25885	1.17x	0.12	1947908
kroB100	CHRI	100	22141	24012	1.08x	0.19	2383052
kroC100	TATT	100	20749	27968	1.35x	0.13	1939164
kroC100	CHRI	100	20749	23470	1.13x	0.21	2150836
kroD100	TATT	100	21294	27112	1.27x	0.12	1949436
kroD100	CHRI	100	21294	23583	1.11x	0.25	2548963
kroE100	TATT	100	22068	29965	1.36x	0.13	1951916
kroE100	CHRI	100	22068	23783	1.08x	0.25	2600277
rd100	TATT	100	7910	10790	1.36x	0.12	1935496
rd100	CHRI	100	7910	8906	1.13x	0.28	2628230
eil101	TATT	101	629	830	1.32x	0.12	1850660
eil101	CHRI	101	629	707	1.12x	0.20	2600604
lin105	TATT	105	14379	19495	1.36x	0.14	2089876
lin105	CHRI	105	14379	16320	1.13x	0.23	2593032
pr107	TATT	107	44303	54237	1.22x	0.17	2236720
pr107	CHRI	107	44303	47891	1.08x	0.21	2836566
pr124	TATT	124	59030	74139	1.26x	0.20	2795240
pr124	CHRI	124	59030	64438	1.09x	0.23	3024388
bier127	TATT	127	118282	158626	1.34x	0.20	2946484
bier127	CHRI	127	118282	132448	1.12x	0.49	4107058
ch130	TATT	130	6110	8129	1.33x	0.22	3006828
ch130	CHRI	130	6110	6773	1.11x	0.33	3682354
pr136	TATT	136	96772	151904	1.57x	0.23	3279836
pr136	CHRI	136	96772	103771	1.07x	0.26	3756040
pr144	TATT	144	58537	80599	1.38x	0.26	3590236
pr144	CHRI	144	58537	70404	1.20x	0.31	4083350
ch150	TATT	150	6528	8347	1.28x	0.26	3785668
ch150	CHRI	150	6528	7135	1.09x	0.40	4067700
kroA150	TATT	150	26524	35119	1.32x	0.28	3893368
kroA150	CHRI	150	26524	29688	1.12x	0.70	5098663
kroB150	TATT	150	26130	36150	1.38x	0.28	3898100

Table 1 – Continuação

Instância	Algoritmo	Nós	Limiar	Resultado	Qualidade	Tempo(s)	Bytes
kroB150	CHRI	150	26130	30053	1.15x	0.67	5049867
pr152	TATT	152	73682	87995	1.19x	0.28	3969988
pr152	CHRI	152	73682	79311	1.08x	0.34	4471372
u159	TATT	159	42080	57791	1.37x	0.30	4295776
u159	CHRI	159	42080	47586	1.13x	0.51	4614644
rat195	TATT	195	2323	3234	1.39x	0.48	6578716
rat195	CHRI	195	2323	2630	1.13x	0.86	7990990
d198	TATT	198	15780	18936	1.20x	0.47	7180184
d198	CHRI	198	15780	17347	1.10x	0.99	8808770
kroA200	TATT	200	29368	40028	1.36x	0.50	7399572
kroA200	CHRI	200	29368	33232	1.13x	1.35	9396710
kroB200	TATT	200	29437	40703	1.38x	0.51	7399496
kroB200	CHRI	200	29437	33119	1.13x	1.20	7843160
ts225	TATT	225	126643	188008	1.48x	0.62	9057020
ts225	CHRI	225	126643	134282	1.06x	0.73	9885438
tsp225	TATT	225	3919	5161	1.32x	0.59	8484452
tsp225	CHRI	225	3919	4397	1.12x	1.03	10114186
pr226	TATT	226	80369	116694	1.45x	0.65	9101784
pr226	CHRI	226	80369	92415	1.15x	1.08	9627104
gil262	TATT	262	2378	3308	1.39x	0.88	10848596
gil262	CHRI	262	2378	2724	1.15x	1.78	13709552
pr264	TATT	264	49135	66470	1.35x	0.90	11935404
pr264	CHRI	264	49135	54662	1.11x	1.30	12283156
a280	TATT	280	2579	3592	1.39x	0.93	12175188
a280	CHRI	280	2579	2913	1.13x	1.84	14905954
pr299	TATT	299	48191	64645	1.34x	1.21	14871796
pr299	CHRI	299	48191	52969	1.10x	2.74	18131050
lin318	TATT	318	42029	58138	1.38x	1.38	16567760
lin318	CHRI	318	42029	47246	1.12x	3.20	20453434
linhp318	TATT	318	41345	58138	1.41x	1.08	16550704
linhp318	CHRI	318	41345	47246	1.14x	2.98	20422272
rd400	TATT	400	15281	20331	1.33x	2.08	28473912
rd400	CHRI	400	15281	17501	1.15x	6.90	35654694
fl417	TATT	417	11861	16200	1.37x	2.10	30643152
fl417	CHRI	417	11861	13175	1.11x	4.11	31259100
pr439	TATT	439	107217	144623	1.35x	2.43	33838848
pr439	CHRI	439	107217	119525	1.11x	4.90	37783742
pcb442	TATT	442	50778	69223	1.36x	2.30	34188688
pcb442	CHRI	442	50778	54868	1.08x	7.08	41603900
d493	TATT	493	35002	44884	1.28x	2.89	41044508
d493	CHRI	493	35002	38697	1.11x	11.11	51200310
u574	TATT	574	36905	48902	1.33x	3.92	54054520
u574	CHRI	574	36905	41424	1.12x	13.96	64604786
rat575	TATT	575	6773	9393	1.39x	4.04	51110664
rat575	CHRI	575	6773	7811	1.15x	13.87	63513994

Table 1 – Continuação

Instância	Algoritmo	Nós	Limiar	Resultado	Qualidade	Tempo(s)	Bytes
p654	TATT	654	34643	49699	1.43x	5.09	68022616
p654	CHRI	654	34643	39292	1.13x	6.72	68999884
d657	TATT	657	48912	66342	1.36x	5.25	68966520
d657	CHRI	657	48912	55007	1.12x	21.19	82252238
u724	TATT	724	41910	57721	1.38x	6.56	95627628
u724	CHRI	724	41910	47431	1.13x	24.88	112493754
rat783	TATT	783	8806	11921	1.35x	7.73	104355408
rat783	CHRI	783	8806	10036	1.14x	31.33	128455762
pr1002	TATT	1002	259045	342216	1.32x	13.67	168833316
pr1002	CHRI	1002	259045	286203	1.10x	68.62	206528394
u1060	TATT	1060	224094	305849	1.36x	13.83	186575728
u1060	CHRI	1060	224094	249289	1.11x	68.07	223805514
vm1084	TATT	1084	239297	315277	1.32x	14.48	194176884
vm1084	CHRI	1084	239297	260770	1.09x	48.16	219217142
pcb1173	TATT	1173	56892	80761	1.42x	17.23	223023176
pcb1173	CHRI	1173	56892	63767	1.12x	64.88	257384350
d1291	TATT	1291	50801	75282	1.48x	20.33	265221020
d1291	CHRI	1291	50801	57655	1.13x	34.15	280090834
rl1304	TATT	1304	252948	348309	1.38x	20.86	270835144
rl1304	CHRI	1304	252948	277037	1.10x	35.94	284268050
rl1323	TATT	1323	270199	380427	1.41x	21.61	278052956
rl1323	CHRI	1323	270199	298272	1.10x	36.45	291966958
nrw1379	TATT	1379	56638	79188	1.40x	25.00	349409232
nrw1379	CHRI	1379	56638	63703	1.12x	178.52	416443182
fl1400	TATT	1400	20127	28022	1.39x	23.73	355898716
fl1400	CHRI	1400	20127	23005	1.14x	183.40	454997830
u1432	TATT	1432	152970	214417	1.40x	25.16	373977788
u1432	CHRI	1432	152970	171469	1.12x	96.10	428211878
fl1577	TATT	1577	22226.5	30257	1.36x	30.48	438433560
fl1577	CHRI	1577	22226.5	24606	1.11x	68.02	472997430
d1655	TATT	1655	62128	85285	1.37x	33.83	478864540
d1655	CHRI	1655	62128	69721	1.12x	119.95	528852458
vm1748	TATT	1748	336556	442756	1.32x	38.07	528398736
vm1748	CHRI	1748	336556	376275	1.12x	163.38	581978370
u1817	TATT	1817	57201	82013	1.43x	40.78	563662052
u1817	CHRI	1817	57201	66125	1.16x	116.30	614157070
rl1889	TATT	1889	316536	448611	1.42x	44.39	605627788
rl1889	CHRI	1889	316536	344580	1.09x	92.91	636841314
d2103	TATT	2103	80201.0	123086	1.53x	55.15	731153916
d2103	CHRI	2103	80201.0	84209	1.05x	60.09	740049788
u2152	TATT	2152	64253	93801	1.46x	57.92	760831036
u2152	CHRI	2152	64253	72407	1.13x	175.06	832213618
u2319	TATT	2319	234256	320506	1.37x	67.68	872551188
u2319	CHRI	2319	234256	271669	1.16x	671.94	1161535406
pr2392	TATT	2392	378032	523107	1.38x	71.58	923159276

Table 1 – Continuação

Instância	Algoritmo	Nós	Limiar	Resultado	Qualidade	Tempo(s)	Bytes
pr2392	CHRI	2392	378032	425940	1.13x	629.44	1100320426
pcb3038	TATT	3038	137694	197498	1.43x	117.13	1649676484
pcb3038	CHRI	3038	137694	154670	1.12x	936.78	1868855962
fl3795	TATT	3795	28747.5	36256	1.26x	180.11	2419313888
fl3795	CHRI	3795	28747.5	31935	1.11x	1475.78	2854912218
fnl4461	TATT	4461	182566	255829	1.40x	274.77	3245936868
rl5915	TATT	5915	565285.0	842903	1.49x	456.72	6300101520
rl5934	TATT	5934	555057.5	817449	1.47x	465.30	6334939440

Table 1: Resultados experimentais: Twice-Around-The-Tree (TATT) e Christofides (CHRI)

References

[1] GeeksforGeeks. (2023, December 10). Kruskal's Minimum Spanning Tree (MST) Algorithm. Retrieved from https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

[2] GeeksforGeeks. (2023, December 10). Prim's Algorithm for Minimum Spanning Tree (MST). Retrieved from https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

[3] Universität Heidelberg. (2023, August 17). TSPLIB 95. [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[4] Wikipedia. (2023, December 10). Christofides algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Christofidesalgorithm

[5] Renato Vimieiro. Slides vistos em sala de aula na disciplina de Algoritmos II em 2023/2.