Relatório – Trabalho Prático 2
Soluções para problemas difícies

Gustavo Chaves Ferreira[1] **(2022043329), Júlio Guerra Domingues**[1] **(2022431280)**

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

Abstract. *This article presents the implementation and analysis of three algorithms applied to the Geometric Traveling Salesman Problem: an exact approach using the Branch-and-Bound method and two approximations based on the Twice-Around-the-Tree and Christofides algorithms. Experiments were conducted with instances from the TSPLIB library and reduced instances, evaluating performance in terms of execution time, memory usage, and solution quality. Results demonstrate that Branch-and-Bound is better suited for small instances, while Christofides stands out as the best balance between quality and efficiency in larger instances. The Twice-Around-the-Tree proved to be a viable alternative when speed is a priority, even with some precision loss. The analyses highlight that the choice of algorithm depends on the instance's characteristics and computational constraints.*

Resumo. *O presente artigo aborda a implementação e análise de três algoritmos aplicados ao problema do Caixeiro Viajante Geométrico: uma abordagem exata utilizando o método Branch-and-Bound e duas aproximações baseadas nos algoritmos Twice-Around-the-Tree e Christofides. Foram conduzidos experimentos com instâncias da biblioteca TSPLIB e instâncias reduzidas, avaliando desempenho em termos de tempo de execução, uso de memória e qualidade das soluções obtidas. Os resultados demonstram que o Branch-and-Bound é mais adequado para instâncias pequenas, enquanto o Christofides se destaca por oferecer o melhor equilíbrio entre qualidade e eficiência em instâncias maiores. O Twice-Around-the-Tree mostrou-se uma alternativa viável quando rapidez é prioritária, mesmo com perda de precisão. As análises reforçam que a escolha do algoritmo depende das características da instância e das restrições computacionais.*

1. Introdução

O *Traveling Salesman Problem* (TSP) é um problema bastante estudado dentro da área de complexidade computacional e que pode ser descrito da seguinte maneira: dado um conjunto de cidades, deseja-se determinar se existe uma rota para se visitar cada uma delas exatamente uma única vez e retornar ao ponto de partida. Ademais, o mesmo também pode ser abordado sob a ótica da otimização combinatória: busca-se determinar a menor rota que não viola as restrições descritas acima. Neste trabalho, serão apresentadas implementações de três algoritmos para se resolver o TSP com base em um problema de minimização. Vale ressaltar que as instâncias para teste estarão restritas àquelas classificadas como geométricas (onde considera-se que os pontos estão em um espaço euclidiano e nas quais a desigualdade triangular é sempre válida).

A primeira abordagem utiliza-se da estratégia conhecida como *Branch-and-Bound*. Através da exploração de árvores de decisão, realizando-se podas nos caminhos que garantidamente não levarão a uma solução melhor do que a encontrada até então, a mesma oferece uma forma de se determinar a solução ótima para um dado problema em que se busca maximizar ou minimizar uma função objetivo. Apesar de, no pior caso, o *Branch-and-Bound* ainda ter que visitar um número não polinomial de nós, em alguns casos o mesmo pode realizar podas que cortam de forma significativa o número de casos a serem explorados, gerando uma resposta em um tempo significativamente menor do que uma solução força bruta.

A segunda e terceira implementações descritas são consideradas algoritmos aproximativos. Ao se deparar com problemas *NP-difíceis*, como a versão de otimização do TSP, muitas vezes pode ser útil determinar soluções não necessariamente perfeitas, mas que possam ser encontradas em tempo polinomial determinístico e que estejam dentro de um fator de qualidade do ótimo real para cada instância.

Dado o valor da solução ótima $f(s*)$ de um problema de maximização e uma outra solução $f(s_a)$ que aproxima a primeira, definimos a razão de acurácia da última como:

$$r(s_a) = \frac{f(s^*)}{r(s_a)}$$

Por outro lado, tratando-se de um problema de minimização, dado o valor de sua solução ótima $f(s*)$ e uma outra solução $f(s_a)$ que aproxima a primeira, definimos a razão de acurácia da última como:

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

Um algoritmo *c*-aproximativo de tempo polinomial é definido como aquele onde a taxa de acurácia $r(s_a)$ da aproximação que ele produz não excede c para nenhuma instância do problema em questão:

$$r(s_a) \leq c$$

O melhor valor para o qual a desigualdade acima é válida é chamado de fator de aproximação.

O algoritmo conhecido como *Twice-Around-the-Tree* fundamenta-se no fato de que remover uma aresta de um ciclo Hamiltoniano origina uma árvore geradora. Ao percorrer especificamente a árvore geradora mínima com suas as arestas duplicadas (não considerando vértices que já foram visitados), ele é capaz de obter uma trajetória que custa, no máximo, duas vezes mais do que o ótimo. Logo, ele possui um fator de aproximação igual à 2.

Já aquele conhecido como algoritmo de *Christofides* inspira-se na mesma ideia. Porém, tendo em mãos a árvore geradora mínima (MST) de um grafo, ele a une ao emparelhamento perfeito de menor peso do subgrafo induzido pelos vértices de grau ímpar

da MST em questão. Feito isso, a estrutura resultante é percorrida em um circuito Euleriano, novamente ignorando-se vértices já visitados. O caminho obtido possui custo até 1.5 vezes maior do que o ótimo real. Esse é o seu fator aproximativo.

Informações detalhadas sobre o funcionamento de cada um deles podem ser encontradas em materiais como [Levitin 2007].

As características e o desempenho das algoritmos que usam as abordagens descritas foram avaliados em instâncias da biblioteca TSPLIB, disponbilizadas e descritas em [Reinelt], considerando-se as seguintes métricas: tempo de execução, uso de memória e qualidade das soluções obtidas (em comparação ao ótimo real).

O objetivo deste trabalho é compreender os desafios práticos de implementação, identificar cenários onde soluções exatas são aplicáveis e avaliar a eficiência de abordagens aproximativas em instâncias de maior escala. O relatório está estruturado para apresentar as escolhas de implementação, os experimentos realizados e os resultados obtidos.

2. Implementação

Optou-se pela utilização de representações otimizadas para grafos completamente conexos, típicos do TSP euclidiano. Para os algoritmos *Twice-Around-the-Tree* e de *Christofides*, utilizou-se a biblioteca Networkx [Developers], que simplifica a manipulação de grafos. No entanto, reconhece-se que sua otimização para grafos esparsos introduz sobrecarga desnecessária no contexto de grafos densos. Assim, empregou-se a matriz de adjacências no branch-and-bound, devido ao acesso rápido aos pesos das arestas e menor consumo de memória comparado a listas de adjacências com ponteiros.

Os métodos presentes no código fonte e que serão descritos a seguir podem ser encontradas no arquivo *Python* nomeado *main.py*, presente no mesmo repositório em que esse artigo se encontra.

Demais métodos ocasionalmente não discutidos aqui desempenham o papel de auxilares na leitura dos arquivos de entrada ou na execução dos testes propostos. O próprio código fonte contém explicações a respeito do funcionamento de cada um deles.

2.1. Algoritmo Twice-Around-the-Tree

A função *twice-around-the-tree* é a responsável pela implementação do mesmo. Um grafo é gerado a partir da matriz de adjacência de cada instância, aproveitando-se da bilbioteca Networkx. Utilizando-se também das facilidades ofertadas pela mesma, a MST de tal estrutura é encontrada. Por fim, o método *pre-order-dfs* implementa uma busca pré-ordem em profundidade na árvore geradora mínima, o que é equivalente a percorrer um circuito Euleriano na MST obtida não considerando vértices repetidos. O custo de tal percurso é retornado.

A complexidade do *Twice-Around-the-Tree* é dominada pela contrução da árvore citada, tendo custo total de $O(|E|lg(|V|))$, onde E é o conjunto de arestas do grafo de entrada e V o de vértices.

2.2. Algoritmo Christofides

A função *christofides* é a encarregada por implementar do algoritmo em questão. Novamente, um grafo é gerado com o auxílio da bilbioteca Networkx. A função *min-weight-*

matching encontra o emparelhamento mais barato entre os vértices de grau ímpar. Ademais, a MST da entrada é também computada. Então, após combinadas as duas estruturas, um caminhamento euleriano é realizado na união resultante a partir do método *compute-eulerian-circuit*, que usa uma lista para simular a estrutura de uma pilha e percorrer todas as arestas de acordo com a lista de vizinhos de cada um dos vértices. O custo do *tour* encontrado após repetições de vértices no caminhamento terem sido removidas é retornado.

A sua complexidade é dominada pelo cálculo do emparelhamento perfeito, sendo da ordem de $O(|V|^3)$, onde V é o conjunto de vértices (ou cidades) da instância em questão.

2.3. Algoritmo Branch-and-Bound

A função *branch-and-bound* contém toda a lógica responsável pela execução do mesmo (com chamadas à funções auxiliares).

Cada nó da árvore de espaços de busca gerada pelo algoritmo é implementado pela classe *Node*. Ela é a responsável por armazenar o caminho parcial representado por cada nodo, além de computar, no momento de instanciação de cada um dos objetos da classe, um limiar inferior para o custo do menor caminho que poderá ser obtido a partir dos vértices restantes com o auxílio do método *compute-bound*. Esse *bound* é calculado da seguinte maneira: para cada uma das cidades, soma-se em um acumulador $bound$ os custos das duas arestas de menor custo incidentes a ela. O resultado final é dividido por 2, já que nessa lógica arestas são contadas de forma duplicada, e a operação $\lceil bound \rceil$ é aplicada. Vale ressaltar que, nos casos em que duas arestas incidentes a uma cidade já tiverem sido definidas pelo caminho representado pelo nó atual, o valor dessas arestas substitui o valor das arestas de menor custo incidentes à cidade atual na soma do limiar. No caso em que apenas uma das arestas já tiver sido fixada, o custo da segunda aresta de menor custo é substituído no cálculo de $bound$.

A exploração da árvore que é iterativamente construída se dá com base na estratégia de *Best First*. Nela, a cada momento em que é preciso se tomar a decisão de qual novo nó expandir, aquele que possui o menor limiar é o selecionado. Tal lógica é implementada atráves do uso de uma fila de prioriedades. Ao longo da execução do TSP, nodos são adicionados e removidos da estrutura de dados para representar tal heurística.

Se em um dado momento, algum nó é gerado no nível $|V| - 1$ (considerando que o nível da árvore que contém a raiz é o 0), então o *tour* é finalizado adicionando-se o vértice de índice 0 no caminho parcial encontrado até então (note que pode-se iniciar o circuito Hamiltoniano em um vértice arbitrário, visto que o custo mínimo final será sempre o mesmo). Nesse instante, o custo do *loop* encontrado é calculado e, caso ele seja melhor do que o do *tour* mais barato encontrado até agora, ele passa a ser tratado com o ótimo parcial. A variável que armazena tal valor é inicializada para representar o maior valor positivo possível ('infinto positivo').

O diferencial do algoritmo de *Branch-and-Bound* encontra-se no fato de que nós cujos limiares sejam mais caros do que o valor de ótimo global parcial são podados (ou seja, retirados da fila de prioridades para não serem explorados). O uso da estratégia *Best First* almeja, dentre outras coisas, determinar o mais cedo possível uma solução ótima parcial para que as podas sejam feitas o quanto antes. Com isso, potencialmente, ramos

contendo uma quantidade exponencial de nós não precisarão ser explorados, e um atalho para a resposta final é tomado.

Para certos casos, espera-se que tal abordagem reduza exponencialmente o custo de execução do algoritmo final. Entretanto, tal afirmação não é válida para o caso geral que considera qualquer instância. Portanto, no pior caso, o custo de execução da função *branch-and-bound* é não polinomial, mais precisamente de $O(|V|!)$, sendo da mesma ordem de um algoritmo força bruta que exploraria uma a uma todas as possíveis combinações de caminhos que podem ser soluções do TSP (considerando um computador determinístico).

3. Experimentos

Os experimentos foram conduzidos com as instâncias geométricas disponíveis da TSPLIB. Cada algoritmo foi testado com tempo limite de 30 minutos por instância. Adicionalmente, foram criadas instâncias modificadas com 5 e 10 vértices a partir das instâncias `eil51` e `berlin52`, removendo vértices considerados de menor relevância. A relevância foi definida pelo grau de conectividade dos vértices na árvore geradora mínima, com vértices menos conectados sendo eliminados. Estas instâncias foram utilizadas para testar o funcionamento do *Branch-and-Bound* e compará-lo com os algoritmos aproximativos.

As métricas avaliadas incluem:

- **Custo**: Representa o valor total da solução encontrada para o problema do Caixeiro Viajante.
- **Razão com o Ótimo**: Calculada como $\frac{Custo_{encontrado}}{Custo_{ótimo}}$, é utilizada para medir a eficiência de algoritmos aproximativos em relação ao ótimo.
- **Tempo de Execução**: Monitorado em segundos, reflete o tempo total necessário para que o algoritmo encontre uma solução.
- **Uso de Memória**: Mede o pico de memória alocada pelo algoritmo durante sua execução, registrado em kilobytes (KB), utilizando a biblioteca `tracemalloc`.

4. Resultados e discussão

De forma geral, os resultados indicam que os algoritmos aproximativos oferecem um compromisso interessante entre tempo de execução e qualidade da solução, enquanto o *Branch-and-Bound* é limitado a instâncias menores devido ao seu crescimento exponencial de complexidade. Os resultados completos estão apresentados na Tabela **??**.

O Branch-and-Bound foi testado em instâncias reduzidas para avaliar sua viabilidade. Nas instâncias com 5 e 10 vértices, ele conseguiu encontrar a solução ótima dentro do limite de tempo. Contudo, em instâncias com 15 vértices, observou-se um aumento significativo no tempo de execução e no uso de memória, limitando sua escalabilidade. O algoritmo apresentou resultados consistentes com sua natureza exata.

O *Twice-Around-the-Tree* proporcionou soluções rápidas, respeitando o fator de aproximação de até 2 vezes. Sua eficiência em termos de tempo o torna ideal para casos onde a qualidade pode ser sacrificada em prol de maior velocidade.

O algoritmo de *Christofides* apresentou um equilíbrio entre tempo e qualidade, com fator de aproximação limitado a 1,5. Ele se mostrou eficiente em instâncias médias,

sendo uma boa alternativa ao *Branch-and-Bound* quando o tempo de execução é uma restrição.

Durante a execução de instâncias maiores (superiores a 10.000 vértices) houve encerramento do programa pelo sistema. Esse comportamento foi atribuído ao consumo excessivo de memória causado pelo tamanho das estruturas necessárias para armazenar e processar o grafo completo. Tal comportamento evidencia a limitação do sistema em alocar recursos suficientes para processar instâncias grandes, reforçando a necessidade de otimizações ou ajustes para cenários de alta escala.

4.1. Análise das Instâncias Modificadas

As instâncias reduzidas, criadas a partir de `eil51` e `berlin52`, foram úteis para validar o desempenho do *Branch-and-Bound*. Em comparação com aos algoritmos aproximativos, o Branch-and-Bound alcançou a solução ótima, enquanto o *Twice-Around-the-Tree* e o *Christofides* ofereceram soluções com razões de 1,8 e 1,4 em média, respectivamente. Tais achados reforçam a aplicabilidade do Branch-and-Bound em instâncias pequenas e dos algoritmos aproximativos em cenários maiores ou mais restritos em termos de recursos computacionais.

4.2. Resultados completos

Tabela 1. Resultados compilados dos experimentos.

Instância	Ótimo	TaT (Custo)	TaT (Tempo)	TaT (Memória)	TaT / Opt	Christ (Custo)	Christ (Tempo)	Christ (Memória)	Christ / Opt	BnB (Custo)	BnB (Tempo)	BnB (Memória)	BnB / Opt
eil51reduced5*	NA	50.55066424	0.001654148102	58.60253906	NA	50.55066424	0.002278089523	74.890625	NA	50.55066424	0.01001191139	16.54882813	NA
berlin52reduced5*	NA	2537.25798	0.0002069473267	9.59375	NA	2386.160048	0.0008659362793	24.50585938	NA	2361.063628	0.009743213654	16.73339844	NA
berlin52reduced10*	NA	3686.91391	0.0004870891571	25.046875	NA	3737.842905	0.001096725464	40.63085938	NA	3358.638414	203.2726009	27998.87793	NA
eil51reduced10*	NA	114.5220649	0.0004210472107	22.53125	NA	103.5043685	0.001681089401	48.39648438	NA	96.70421166	268.0258811	8682.241211	NA
eil51reduced15*	NA	175.4424733	0.0007708072662	49.4453125	NA	149.6758289	0.002133131027	75.36523438	NA	NA	NA	NA	NA
eil51reduced20*	NA	233.5119353	0.00146317482	74.9140625	NA	176.1999789	0.002990961075	101.7011719	NA	NA	NA	NA	NA
eil51	426	640.9012229	0.007119178772	537.3330078	1.5045	489.2899036	0.02821302414	680.3681641	1.1486	NA	NA	NA	NA
berlin52	7542	10116.0145	0.005463123322	450.359375	1.3413	8542.921553	0.02483510971	557.6855469	1.1327	NA	NA	NA	NA
st70	675	873.3514857	0.01139783859	822.3984375	1.2939	759.1699478	0.06095480919	997.8027344	1.1247	NA	NA	NA	NA
pr76	108159	145338.1145	0.01137804985	920.921875	1.3437	116683.5346	0.03684830666	1014.263672	1.0788	NA	NA	NA	NA
eil76	538	707.1533199	0.01171875	920.9921875	1.3144	620.6683014	0.07960772514	1147.638672	1.1537	NA	NA	NA	NA
rat99	1211	1723.225045	0.02140402794	1832.515625	1.423	1367.431979	0.08762383461	1879.169922	1.1292	NA	NA	NA	NA
rd100	7910	10791.65582	0.02144789696	1866.4375	1.3643	8809.720576	0.1835179329	2119.021484	1.1137	NA	NA	NA	NA
kroB100	22141	25881.19989	0.02037501335	1866.382813	1.1689	23994.53911	0.08157992363	1892.216797	1.0837	NA	NA	NA	NA
kroD100	21294	27113.29664	0.01996517181	1866.328125	1.2733	23536.36092	0.1113870144	1950.044922	1.1053	NA	NA	NA	NA
kroA100	21282	27211.67973	0.02197599411	1866.289063	1.2786	24213.26854	0.1175861359	2021.201172	1.1377	NA	NA	NA	NA
kroC100	20749	27966.54136	0.02064299583	1866.203125	1.3479	23483.88991	0.110517025	2023.716797	1.1318	NA	NA	NA	NA
kroE100	22068	30507.41905	0.02027273178	1866.195313	1.3824	23782.60705	0.149408102	2090.427734	1.0777	NA	NA	NA	NA
eil101	629	830.5422076	0.02228093147	1900.851563	1.3204	723.6014003	0.1596589088	2154.583984	1.1504	NA	NA	NA	NA
lin105	14379	19498.40879	0.02225494385	2046.484375	1.356	16432.03299	0.105768919	2068.021484	1.1428	NA	NA	NA	NA
pr107	44303	54238.03661	0.02285814285	2118.609375	1.2243	47908.94275	0.07548904419	2235.365234	1.0814	NA	NA	NA	NA
pr124	59030	74140.95965	0.03243613243	2799.585938	1.256	64667.63767	0.06787514687	2799.882813	1.0955	NA	NA	NA	NA
bier127	118282	158637.6062	0.03327584267	2924.46875	1.3412	132533.1804	0.2976391315	3258.349609	1.1205	NA	NA	NA	NA
ch130	6110	8128.757281	0.03510785103	3053.710938	1.3304	6777.779741	0.162569046	3054.007813	1.1093	NA	NA	NA	NA
pr136	96772	151913.742	0.03873515129	3326.421875	1.5698	103772.9123	0.06230664253	3326.71875	1.0723	NA	NA	NA	NA
pr144	58537	80596.32005	0.04129886627	3705.195313	1.3768	70611.60136	0.06640410423	3705.492188	1.2063	NA	NA	NA	NA
kroA150	26524	35122.5668	0.04650878906	4005.4375	1.3242	29939.70284	0.5760228634	4147.023438	1.1288	NA	NA	NA	NA
ch150	6528	8333.47217	0.06368589401	4147.765625	1.2766	7186.518258	0.1837399006	4005.921875	1.1009	NA	NA	NA	NA
kroB150	26130	36154.73394	0.0452606678	4005.414063	1.3836	29946.87288	0.4434890747	4019.076172	1.1461	NA	NA	NA	NA
pr152	73682	87998.69398	0.04572606087	4103.96875	1.1943	79369.84655	0.06727194786	4104.265625	1.0772	NA	NA	NA	NA
u159	42080	57787.97441	0.05071377754	4471.226563	1.3733	47766.16199	0.2446401119	4471.523438	1.1351	NA	NA	NA	NA
rat195	2323	3317.723252	0.08047890663	7471.039063	1.4282	2664.564807	0.4612710476	7471.335938	1.147	NA	NA	NA	NA
d198	15780	19218.43305	0.08068084717	7692.429688	1.2179	17357.35977	0.5256800652	7692.726563	1.1	NA	NA	NA	NA
kroB200	29437	40710.95988	0.08167004585	7828.765625	1.383	33140.89354	0.6434969902	7829.0625	1.1258	NA	NA	NA	NA
kroA200	29368	40030.86635	0.08266806602	7828.75	1.3631	33348.59614	1.196289063	7829.210938	1.1355	NA	NA	NA	NA
tsp225	3916	5115.932581	0.1322879791	9679.234375	1.3064	4431.44741	0.6077868938	9679.546875	1.1316	NA	NA	NA	NA
ts225	126643	188009.7016	0.1000938416	9678.164063	1.4846	136160.85	0.1448969841	9678.054688	1.0752	NA	NA	NA	NA
pr226	80369	116692.1537	0.1028840542	9754.75	1.452	93175.38542	0.4637179375	9755.046875	1.1593	NA	NA	NA	NA
gil262	2378	3358.154177	0.1418380737	12788.87109	1.4122	2708.215572	1.504615068	12789.16797	1.1389	NA	NA	NA	NA
pr264	49135	66466.84175	0.1399981976	13012.90234	1.3527	54663.60243	0.6444511414	13155.51953	1.1125	NA	NA	NA	NA
a280	2579	3629.720305	0.2030739784	14768.26953	1.4074	2969.889947	1.061098099	14626.24609	1.1516	NA	NA	NA	NA
pr299	48191	64646.03472	0.1821599007	16647.22266	1.3415	53310.8896	1.516469955	16647.51953	1.1062	NA	NA	NA	NA
linhp318	41345	58145.44192	0.2089390755	18792.57422	1.4063	47363.14497	2.585279226	18935.19141	1.1456	NA	NA	NA	NA
lin318	42029	58145.44192	0.2568199635	18934.89453	1.3835	47363.14497	1.880457878	18792.87109	1.1269	NA	NA	NA	NA
rd400	15281	20250.37674	0.3424730301	33095.69922	1.3252	17640.93502	7.911396027	33238.31641	1.1544	NA	NA	NA	NA
fl417	11861	16297.17464	0.4345431328	35787.48828	1.374	13436.60216	3.158316851	35645.62891	1.1328	NA	NA	NA	NA
pr439	107217	144624.1579	0.4947140217	39258.09766	1.3489	120685.5993	2.38212204	39258.41016	1.1256	NA	NA	NA	NA
pcb442	50778	69364.90251	0.4098482132	39746.40234	1.366	55509.77131	6.002887964	39888.61328	1.0932	NA	NA	NA	NA
d493	35002	45427.08885	0.6653842926	48844.09766	1.2978	38719.17753	10.20934391	48844.39453	1.1062	NA	NA	NA	NA
u574	36905	49083.15194	0.9258520603	64911.28516	1.33	41357.74223	16.20611906	64912.28516	1.1207	NA	NA	NA	NA
rat575	6773	9438.933433	0.8429470062	64976.47266	1.3936	7841.003495	18.64202118	65117.25391	1.1577	NA	NA	NA	NA
p654	34643	49731.89542	1.049026966	82852.45703	1.4356	39246.56028	2.797849894	82852.05078	1.1329	NA	NA	NA	NA
d657	48912	65730.1937	1.166237354	83553.89453	1.3438	55143.86863	24.17999601	83412.73828	1.1274	NA	NA	NA	NA
u724	41910	57779.66789	1.535465956	113532.4414	1.3787	47809.28446	30.97141314	113532.7383	1.1408	NA	NA	NA	NA
rat783	8806	12054.45316	1.672040939	130534.418	1.3689	10150.34868	46.68613219	130533.8789	1.1527	NA	NA	NA	NA
pr1002	259045	342244.4624	2.706872225	204112.3789	1.3212	287967.6865	69.12945199	203971.2852	1.1117	NA	NA	NA	NA
u1060	224094	302914.8664	3.225511074	226338.1289	1.3517	250977.0696	94.12412906	226337.5195	1.12	NA	NA	NA	NA
vm1084	239297	315268.3487	3.24745512	236078.832	1.3175	263730.7859	48.0208478	236079.1289	1.1021	NA	NA	NA	NA
pcb1173	56892	80694.89286	3.925287008	273036.9023	1.4184	63903.09362	86.80434895	273036.293	1.1232	NA	NA	NA	NA
d1291	50801	74658.29731	4.581877947	325584.9805	1.4696	57997.40356	22.65194821	325585.2773	1.1417	NA	NA	NA	NA
rl1304	252948	347782.5703	4.962929964	332546.6445	1.3749	280824.8427	22.77898812	332546.0352	1.1102	NA	NA	NA	NA
rl1323	270199	380421.5876	5.007185221	341280.7305	1.4079	301003.4589	23.26288986	341281.0273	1.114	NA	NA	NA	NA
nrw1379	56638	79156.26507	5.49861908	418823.2227	1.3976	64641.87972	310.4406788	418823.5195	1.1413	NA	NA	NA	NA
fl1400	20127	27915.52348	5.574913979	429912.5664	1.387	22828.8296	288.2477729	429912.8633	1.1342	NA	NA	NA	NA
u1432	152970	214430.7805	5.765496016	447212.2852	1.4018	172355.9945	91.31684589	447071.1914	1.1267	NA	NA	NA	NA
fl1577	22249	31223.09667	6.815891027	531124.9727	1.4033	24573.52852	58.06174111	531265.7539	1.1045	NA	NA	NA	NA
d1655	62128	85681.07842	7.83594799	579611.832	1.3791	70835.64841	115.562202	579612.1289	1.1402	NA	NA	NA	NA
vm1748	336556	444658.6319	9.107143879	639710.9258	1.3212	379674.5729	172.4559751	639711.2227	1.1281	NA	NA	NA	NA
u1817	57201	83501.55577	9.769303083	685044.207	1.4598	66832.96594	132.1699638	685044.5039	1.1684	NA	NA	NA	NA
rl1889	316536	449811.4882	11.61416698	735668.3477	1.421	346928.444	70.74978709	735668.6445	1.096	NA	NA	NA	NA
d2103	80450	124533.8716	13.17249513	895210.707	1.548	84509.32914	16.63635898	895211.0039	1.0505	NA	NA	NA	NA
u2152	64253	95076.21583	14.53773594	932949.957	1.4797	74400.08365	191.5840771	932950.2539	1.1579	NA	NA	NA	NA
u2319	234256	320532.841	17.61318803	1069827.238	1.3683	272024.8262	738.7754338	1069827.535	1.1612	NA	NA	NA	NA
pr2392	378032	523132.9069	18.42902422	1134181.91	1.3838	428445.1292	679.2560239	1134182.207	1.1334	NA	NA	NA	NA
pcb3038	137694	196573.8992	30.14813495	1989655.488	1.4276	155294.0591	1460.5423	1989655.785	1.1278	NA	NA	NA	NA
fl3795	28772	NA	NA	NA	NA	31563.15541	1394.288095	2970355.23	1.097	NA	NA	NA	NA
fnl4461	182566	254671.6275	182.4892581	3994348.949	1.395	NA	NA	NA	NA	NA	NA	NA	NA
rl5915	565530	840348.3558	734.4058356	7583729.973	1.4859	620359.8462	1315.266221	7583730.27	1.097	NA	NA	NA	NA
rl5934	556045	827482.336	691.7196009	7626153.223	1.4882	611131.1902	1509.895182	7626153.52	1.0991	NA	NA	NA	NA

Nota: Instâncias marcadas com '*' foram criadas a partir da redução das menores instâncias originais, removendo vértices de menor relevância. **Abreviações:** **TaT** = *Twice-Around-the-Tree*, **Christ** = *Christofides*, **BnB** = *Branch-and-Bound*, **Custo** = Custo da solução encontrada, **Tempo** = Tempo de execução (segundos), **Memória** = Memória máxima utilizada (KB), **/ Opt** = Razão entre o custo encontrado e o ótimo.

Os experimentos mostraram que não existe uma abordagem única que resolva de forma ideal todos os cenários possíveis. A seleção do método mais adequado está intrinsicamente ligada às restrições específicas de cada problema e aos requisitos de precisão, tempo de execução e consumo de recursos computacionais. Para instâncias de menor escala, em que a obtenção de uma solução exata é imprescindível, o *Branch-and-Bound* se mostrou como a escolha mais apropriada. Já para problemas de maior escala, o *Christofides* apresentou-se como a opção mais robusta, equilibrando qualidade e eficiência

de maneira consistente. Por outro lado, o *Twice-Around-the-Tree* é particularmente vantajoso em cenários onde a rapidez na obtenção da solução é a principal prioridade, ainda que com um comprometimento maior na qualidade da resposta encontrada.

5. Conclusão

Este trabalho explorou três abordagens distintas para o problema do Caixeiro Viajante Geométrico, destacando os desafios e as limitações de cada uma. O *Branch-and-Bound* se provou ser eficiente para instâncias menores, oferecendo soluções ótimas em tempo prático, mas sua escalabilidade é limitada pelo crescimento não polinomal de sua complexidade. Já os algoritmos aproximativos *Twice-Around-the-Tree* e *Christofides* demonstraram um equilíbrio entre tempo de execução e qualidade da solução, sendo mais adequados para instâncias maiores, em especial nos casos em que uma resposta exata é computacionalmente inexequível de ser obtida, mas onde ainda assim uma resposta (ainda que com uma margem de erro moderada) precisa ser obtida.

A realização de todas as etapas do trabalho prático, incluindo o desenvolvimento de código e do relatório final, favoreceram uma maior compreensão do conteúdo trabalhado em sala de aula ao longo da disciplina, em especial no último terço do curso, e permitiram o contato dos autores com instâncias reais, as quais exigem um maior grau de elaboração sob o ponto de vista algorítmico para serem solucionadas.

Referências

Developers, N. Software for complex networks.

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

Reinelt, G. Tsplib: A library of sample instances for the tsp and related problems - universität heidelberg.