Análise comparativa de algoritmos para o TSP Euclidiano

Jean Lucas Almeida Mota[1,*]

[a] *Universidade Federal de Minas Gerais, Belo Horizonte, Brazil*

Abstract

Este trabalho tem como objetivo explorar e analisar os aspectos práticos de algoritmos para solucionar o problema do caixeiro viajante euclidiano (TSP). Os algoritmos usados foram: *Branch-and-Bround*, *Twice-around-the-tree* e *Christofides*. Estes foram comparados segundo o tempo, o espaço e qualidade da solução gerada.

1. Introdução

Na computação, atualmente, não há solução polinomial para uma série de problemas. Alguns desses, classificados como problemas NP-completo, só possuem solução polinomial em uma maquina de Turing não determinística [1]. No entanto, isso não inviabiliza e também não diminui o empenho dos acadêmicos em buscar soluções suficientemente rápidas para os problemas NP-completo. Dessa forma, neste trabalho são comparados três tipos de algoritmos para problemas difíceis. São eles: *Branch-and-Bround*, *Twice-around-the-tree* e *Christofides*.

O intuito deste artigo é fazer a analise comparativa entre os algoritmos. Dessa forma, a fim de facilitar o estudo, as analises foram feitas com relação ao problema do Caixeiro Viajante na sua versão euclidiana usando as métricas Euclidiana e Manhatan. As características analisadas foram o tempo de execução, o espaço usado e a qualidade da solução gerada.

Com base nas analises comparativas, foi observado que o algoritmo *Branch-and-Bround* possui a melhor solução, mas o pior desempenho em tempo, sendo viável apenas para instâncias pequenas. Já os algoritmos *Twice-around-the-tree* e *Christofides* não possuem uma solução tão ótima, mas possuem o tempo de execução extremamente superior, fazendo deles viáveis para instâncias grandes. Entre os dois últimos, temos que a solução do *Christofides* possui uma qualidade maior, visto o seu desempenho, este pode ser considerado o melhor dos 3 algoritmos analisados.

Os algoritmos e as técnicas usadas neste trabalho foram vistos em [4] e [3].

O restante do artigo está organizado da seguinte maneira. A seção 2 apresenta alguns conceitos básicos e estruturas utilizadas. A seção 3 detalha a implementação. A seção 4 apresenta uma análise comparativa e discute os resultados, enquanto a seção 5 conclui o trabalho.

*Corresponding author

Email address: jeanlk@ufmg.br (Jean Lucas Almeida Mota)

2. Conceitos Básicos e Estruturas Usadas

Todo o trabalho foi realizado sob um caso particular do problema do *Caixeiro Viajante*. Este problema consiste em, dado um grafo ponderado qualquer, encontrar o caminho de menor custo partindo de um vértice qualquer, visitando todos os outros vértices uma única vez e retornando ao inicial. Atualmente, não existe um algoritmo geral e viável para este problema. No entanto, para o caso particular onde os pesos das arestas são dados através de uma métrica, existem algoritmos. Dessa forma, para este caso particular, temos o *Caixeiro Viajante Euclidiano*

O algoritmo de *Branch-and-Bround* possui o paradigma de explorar o espaço de busca de forma eficiente. Para cada nó da árvore de busca, temos um limiar(bound), uma estimativa para o custo máximo ou mínimo que aquele ramo da árvore pode tomar. A partir desse limiar o algoritmo decide explorar ou abandonar aquele nó. Já os algoritmos *Twice-around-the-tree* e *Christofides* são algoritmos aproximativos, ou seja, trocam a otimalidade da solução pela velocidade de execução. São algoritmos que estimam a qualidade da sua solução com base na ótima e admitem uma 'margem de erro' na sua solução.

A estimativa de custo de um caminho foi definida como: Para cada um dos vértices, pegue a soma do peso das duas menores arestas incidente a ele e adicione ao custo total. Depois, tome a função teto deste total dividido por dois.

$$EstimativaCusto = \left\lceil \frac{(V_1+V_1')+(V_2+V_2')+\cdots+(V_n+V_n')}{2} \right\rceil$$

Para a modelagem e implementação do *Caixeiro Viajante Euclidiano* foi escolhida estrutura *Grafo*, por causa da definição do próprio problema. E para a busca no espaço de soluções a estrutura usada foi a árvore de busca, pela facilidade de exploração armazenamento de informações nos vértices. Essas estruturas foram extraídas da biblioteca NetworkX. Já as estruturas simples, como listas, filas de prioridade e tuplas foram implementadas.

3. Implementação

O programa o qual foi implementado os três algoritmos foi desenvolvido na linguagem python versão 3.11.1.

A fim de modularizar a implementação, foram implementadas quatro classes, uma de mais alto nível e três de baixo nível. São elas *PathFinder*, *Bnb*, *Tat*, *Chr*.

PathFinder é a classe de mais alto nível, onde foram feitas as operações de leitura da entrada, criação da instância, escolha do algoritmo a ser utilizado e output da solução.

Bnb é a classe de baixo nível onde foi feita a implementação do algoritmo *Branch-and-Bround*. Possui as funções Bound, que recebe uma solução parcial e calcula a estimativa de custo do caminho a partir dessa solução parcial. E Bnb que implementa o algoritmo de busca.

Tat é a classe de baixo nível onde foi feita a implementação do algoritmo *Twice-around-the-tree*. Possui as funções Tat, que implementa o algoritmo de busca. E BuscaPreOrder, que recebe uma arvore e um nó inicial e realiza a busca PreOrder na árvore.

Chr é a classe de baixo nível onde foi feita a implementação do algoritmo *Christofides*, bem semelhante a classe *Tat*. Possui a função Chr, que implementa o algoritmo de busca.

Para execução no Windows foi executado o seguinte comando: *python tsp.py [algoritmo] [instancia] [metrica]*. As siglas usadas estão na tabela 1

	Sigla
Branch-and-Bround	bnb
Twice-around-the-tree	tat
Christofides	chr
Euclidiana	euc
Manhattan	man

Table 1: Tabela de siglas para execução do programa.

Exemplo de execução do algoritmo *Branch-and-Bround* para instância 2^4 com a métrica euclidiana: *python tsp.py bnb 4 euc*

4. Análise e Resultados

Neste artigo os algoritmos *Branch-and-Bround*, *Twice-around-the-tree* e *Christofides* com a métrica Euclidiana e Manhattan serão comparados segundo o tempo, espaço e qualidade da solução. Para isso foram feitos 5 experimentos para cada uma das instâncias de tamanho 2^4, 2^5, 2^6, 2^7, 2^8, 2^9 e 2^{10}, e retirada a média. Há de ser observar que, durante os experimentos, foi estipulado um teto máximo de tempo de execução de 30 minutos para cada experimento.

Para o algoritmo *Branch-and-Bround* não foi possivel executar para instâncias maiores que 4, uma vez que o mesmo excedeu o tempo limite. A Figura 1 mostra o gráfico do algoritmo em relação ao tempo e a Figura 2 mostra a memória usada.

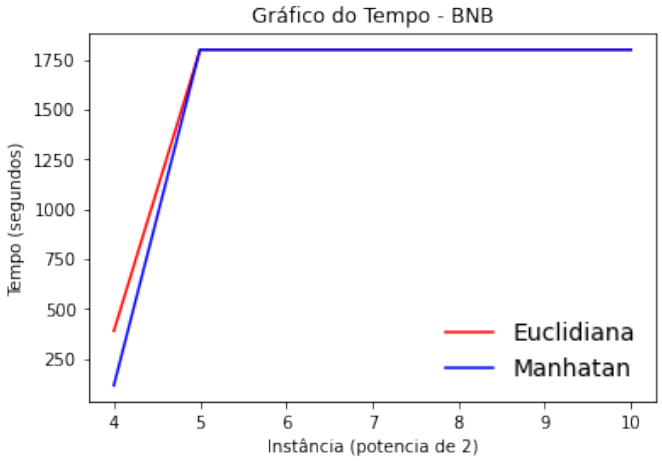

Figure 1: Tempo de execução do algoritmo *Branch-and-Bround* para 10 instâncias diferentes.

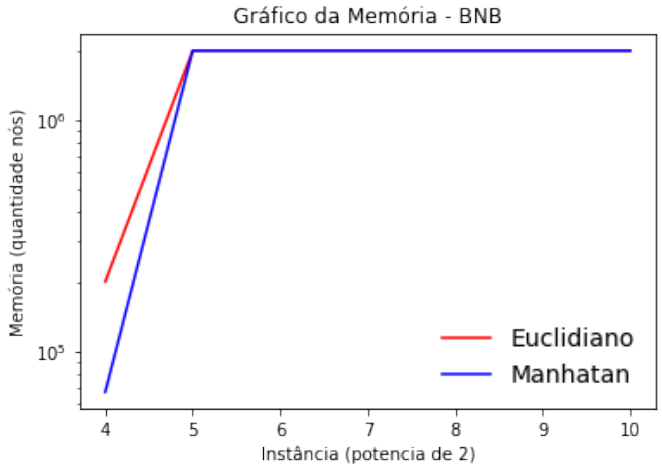

Figure 2: Memória usada pelo algoritmo *Branch-and-Bround* para 10 instâncias diferentes.

Com esses dados podemos observar que o *Branch-and-Bround* não é eficiente para grandes instâncias do problema do *Caixeiro Viajante*. Uma vez que já na instância 2^5 ele ultrapassa o limite estipulado. No entanto, a qualidade de sua solução é ótima e será usada para verificar a qualidade dos outros algoritmos.

Para o algoritmo *Twice-around-the-tree*, a figura 3 mostra o crescimento do tempo de execução a medida que aumentamos o tamanho da entrada. Enquanto a figura 4 mostra um crescimento linear do uso da memória pelo algoritmo a medida que a entrada cresce.

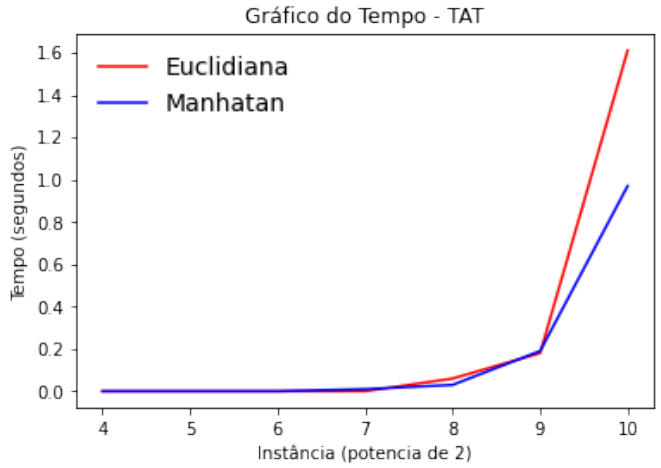

Figure 3: Tempo de execução do algoritmo *Twice-around-the-tree* para 10 instâncias diferentes.

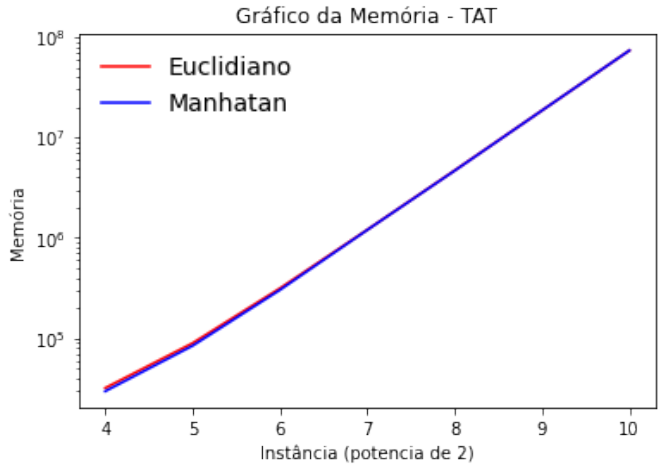

Figure 4: Memória usada pelo algoritmo *Twice-around-the-tree* para 10 instâncias diferentes.

Quanto a otimalidade do algoritmo, temos que comparar sua solução com a solução do *Branch-and-Bround*. No entanto, como temos apenas a solução para instancia 2^4, foi feito mais alguns experimentos com instâncias menores. Como mostra a tabela 2, com instâncias muito pequenas o algoritmo possui uma qualidade excelente. No entanto, a medida que as instâncias aumentam, a qualidade começa a cair. Segundo [1] e [2], o *Twice-around-the-tree* para entradas maiores mantém uma qualidade fixa de 2.0 da ótima.

	BNB	TAT	Qualidade
2^2	2219.3	2219.3	1.0
2^3	2313.9	2558.3	1.1
2^4	2898.2	3486.3	1.2

Table 2: Tabela comparativa sobre a qualidade da solução do *Twice-around-the-tree*

Para o algoritmo *Christofides*, a figura 5 mostra o crescimento do tempo de execução. Enquanto a figura 6 mostra um crescimento linear do uso da memória pelo algoritmo.

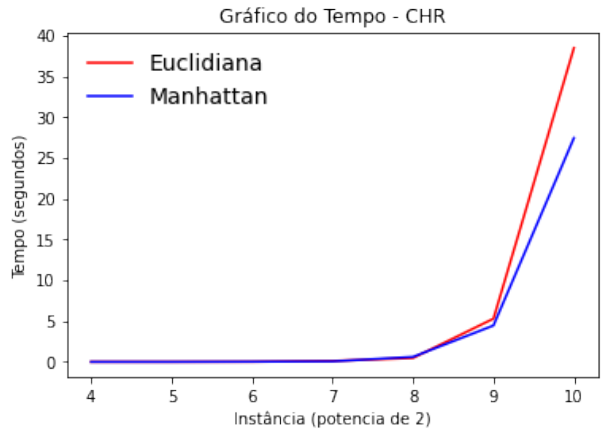

Figure 5: Tempo de execução do algoritmo *Christofides* para 10 instâncias diferentes.

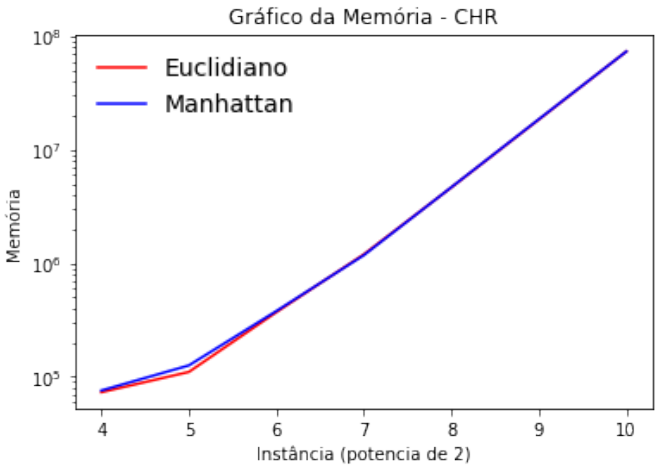

Figure 6: Memória usada pelo algoritmo *Christofides* para 10 instâncias diferentes.

Quanto a otimalidade do algoritmo, temos a mesma restrição de analise que o *Twice-around-the-tree*, podendo analisar apenas para a isntância 2^4. Dessa forma, a tabela 3 mostra uma analise restrita do decaimento da qualidade a medida que a entrada cresce. Segundo [1] e [2], o algoritmo *Christofides* para instâncias maiores manterá uma qualidade de 1.5 da ótima.

	BNB	CHR	Qualidade CHR
2^2	2245.0	2245.0	1.0
2^3	2893.2	3097.0	1.07
2^4	4021.9	4263.7	1.06

Table 3: Tabela comparativa sobre a qualidade da solução do *Christofides*

4.1. Análise Comparativa

Ao comparar o tempo de execução de todos os algoritmos, podemos perceber uma discrepância entre o *Branch-and-Bround* e os outros. Veja na figura 7 e 8.

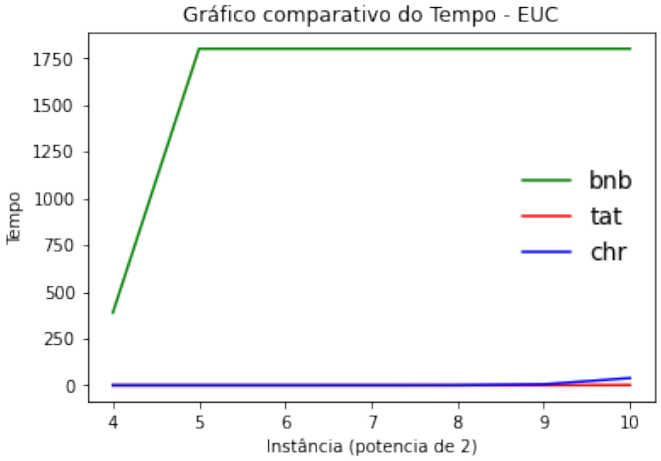

Figure 7: Tempo de execução de todos os algoritmos usando a métrica euclidiana para 10 instâncias diferentes.

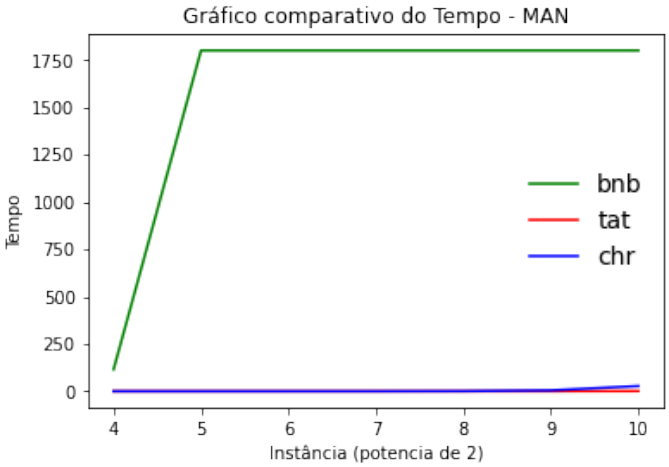

Figure 8: Tempo de execução de todos os algoritmos usando a métrica de Manhattan para 10 instâncias diferentes.

Com relação a memória, não foi possível gerar dados para o *Branch-and-Bround*. No entanto, para os outros algoritmos temos uma linearidade dos dados. Como pode ser visto nas figuras 9 e 10.

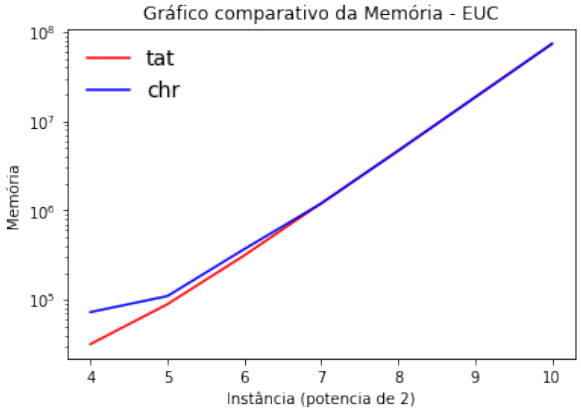

Figure 9: Memória usada pelo algoritmos com a métrica Euclidiana para 10 instâncias diferentes.

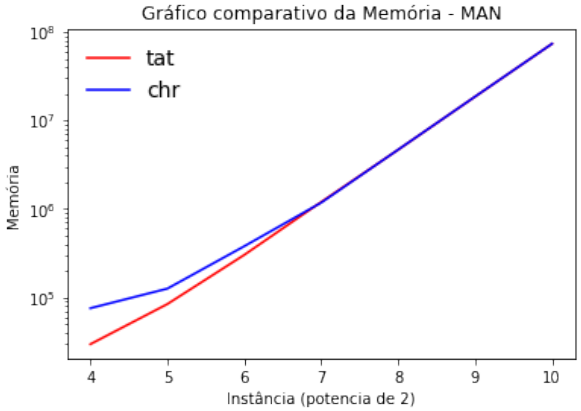

Figure 10: Memória usada pelo algoritmos com a métrica de Manhattan para 10 instâncias diferentes.

Com relação a qualidade das soluções, foi criada duas tabelas. A tabela 4 mostra a qualidade dos algoritmos *Twice-around-the-tree* e *Christofides* perante a solução ótima do *Branch-and-Bround*. Enquanto a tabela 5 mostra a comparação entre o custo das soluções dos dois algoritmos aproximativos para todas as instâncias.

	BNB	TAT	CHR	Qualid. TAT	Qualid. CHR
2^2	2234.4	2234.4	2234.4	1.0	1.0
2^3	1844.5	2156.8	2075.1	1.16	1.12
2^4	3828.8	4728.1	4247.8	1.23	1.10

Table 4: Tabela comparativa sobre a qualidade da solução.

	TAT	CHR	TAT / CHR
2^4	4334.62	3730.36	1.16
2^5	6030.00	5147.94	1.17
2^6	8117.52	7254.45	1.18
2^7	11572.68	10335.56	1.19
2^8	16464.22	14577.12	1.12
2^9	22935.40	20263.58	1.13
2^10	31949.20	28248.89	1.13

Table 5: Tabela comparativa sobre o custo da solução dos algoritmos *Twice-around-the-tree* e *Christofides*.

5. Conclusão

Dessa forma, podemos perceber que, por mais que o *Branch-and-Bround* possua uma solução ótima, a vantagem é claramente dos algoritmos aproximativos. Uma vez que possuem uma boa solução e uma performance muito superior, tanto em tempo de execução quanto na quantidade de memória usada, como é mostrado nas figuras 9 e 10. Enquanto o *Branch-and-Bround* ultrapassa o limite de 30 minutos, os algoritmos *Twice-around-the-tree* e *Christofides* demoram menos de 1 segundo.

Percebe-se também que as métricas influenciam principalmente no tempo de execução para instâncias grandes. Como fica evidente nas figuras 3 e 5, a partir da instância 2^9 a métrica euclidiana começa a crescer mais rapidamente que a Manhattan. Isso se deve principalmente as operações de divisão e raiz exigidas por essa métrica. No entanto, quanto a memória, não houve diferenças entre as métricas.

Na tabela 4 podemos vislumbrar uma qualidade inferior do *Twice-around-the-tree* em relação ao *Christofides* a medida que a instância aumenta. No entanto, na tabela 5 observamos as soluções de ambos para grandes entradas e nota-se que a otimalidade do *Christofides* em relação ao *Twice-around-the-tree* se mantêm maior, mas essa diferença não cresce.

Comparando os dois algoritmos aproximativos, temos uma vantagem maior na utilização do *Christofides*. Uma vez que sua solução se aproxima mais da ótima enquanto gasta o mesmo tempo e a mesma memória que o *Twice-around-the-tree*.

Assim, concluo que o algoritmo *Branch-and-Bround*, mesmo que tenha a melhor solução, na prática, não é a melhor escolha para a solução do TSP Euclidiano. Salvo caso seja extremamente necessário uma solução ótima. Na grande maioria dos casos, concluo que o melhor algoritmo para o TSP Euclidiano é o *Christofides*, usando a métrica Manhattan quando for necessário uma solução mais rápida e a métrica Euclidiana quando for necessário representar distâncias do mundo real de maneira mais verossímil. Este algoritmo obteve resultados muito satisfatórios tanto na execução quanto na implementação. Mesmo que o *Christofides* seja mais complexo de implementar que o *Twice-around-the-tree*, através da biblioteca NetworkX essa complexidade pode ser remediada.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2022.

[2] Levitin, A. *Introduction to Design and Analysis of Algorithms, 2/E*. Pearson Education India, 2008.

[3] Professor Luiz Chaimowicz. Slide sobre busca em espaço de estados (2022). Disciplina Introdução a Inteligência Artificial DCC642 UFMG, 2022.

[4] Professor Renato Vimieiro. Slide sobre soluções aproximadas para problemas difíceis (2022). Disciplina Algoritmos 2 DCC207 UFMG, 2022.