Implementação e Análise Comparativa de Algoritmos para o Problema do Caixeiro Viajante Métrico

André Luiz Moreira Dutra

27 de agosto de 2021

Resumo

O Problema do Caixeiro Viajante é um dos mais conhecidos e consolidados problemas em Ciência da Computação, cuja solução tem diversas aplicações práticas. O problema consiste em, dado um conjunto de cidades e as distâncias entre cada par das mesmas, definir o menor trajeto que passe por cada cidade uma única vez e volte ao início. Abordamos neste artigo o problema por meio de três algoritmos diferentes: Branch-and-Bound, Twice-Around-the-Tree e o Algoritmo de Christofides, trazendo uma análise comparativa dos resultados e recursos utilizados por cada algoritmo utilizando diferentes métricas de distância. Pelos resultados da análise foi possível observar que, de maneira geral, o Branch-and-Bound, embora trouxesse os melhores resultados, tornou-se inviável para a maioria das instâncias, enquanto, entre os demais, o Twice-Around-the-Tree é executado com o menor tempo e o Algoritmo de Christofides traz a melhor aproximação.

1 Introdução

Formulado pela primeira vez no século XIX pelo matemático W. R. Hamilton, o Problema do Caixeiro Viajante (ou TSP, abreviação para Travelling Salesman Problem, conforme comumente citado na literatura) se resume a, dadas as distâncias entre cada par de um conjunto de pontos, definir o menor caminho que passe por todos os pontos e volte ao início. Uma das principais motivações para a solução do problema se encontra em seu próprio nome, no caso de um entregador que precisa sair de um armazém, passar uma única vez por um determinado conjunto definido de locais, entregando as mercadorias, e voltar ao armazém, passando pelo menor caminho possível. Nesse contexto, o TSP é essencial em modelar problemas de logística espaciais. No entanto, o TSP também aparece em situações menos triviais, como na representação de figuras como Séries de Fourier contínuas no plano complexo [Pol18]. Tendo em mente as diversas aplicações do problema, este trabalho objetiva resolvê-lo em tempo viável com o máximo de precisão possível.

Formalmente, o Problema do Caixeiro Viajante se define por, dado um grafo completo de arestas ponderadas, encontrar seu ciclo Hamiltoniano de peso total mínimo. O TSP é um problema de otimização da classe NP-difícil, o que significa que, a menos que P = NP, é impossível encontrar uma solução exata para

uma instância do problema em tempo viável polinomial. No entanto, podemos abordar o problema de outras formas, seja minimizando o tempo de execução de algoritmos exatos não polinomiais, seja implementando algoritmos polinomiais que encontrem resultados aproximados. Desse modo, apresentamos a implementação de três algoritmos: o Branch-and-Bound, exato e não polinomial no pior caso, e os algoritmos Twice-Around-The-Tree e Algoritmo de Christofides, polinomiais com resultado aproximado.

A definição original do TSP não traz quaisquer limitações físicas ou geométricas aos atributos das arestas, podendo estas assumir pesos de quaisquer valores reais possíveis, e o grafo pode ou não ser direcionado. Dessa forma, para atender aos pré-requisitos dos algoritmos implementados, tratamos neste artigo de um caso específico do TSP, o Problema do Caixeiro Viajante Métrico, no qual, além da definição padrão do problema, há a restrição de que as arestas devem ser positivas e formar uma métrica, ou seja, a distância entre dois vértices é a mesma em ambas as direções, e dados três vértices quaisquer no grafo, as arestas que os conectam obedecem à desigualdade triangular. Essa definição implica a representação do problema por um grafo não-direcionado. Embora a restrição torne o problema menos geral, a complexidade do problema não é restringida, de modo que o mesmo continue sendo NP-difícil. Todas as referências feitas ao TSP a partir deste ponto estarão se referindo ao caso métrico específico tratado.

Além da implementação, os algoritmos foram testados em diferentes instâncias, definidas selecionando um número de pontos aleatórios no plano e atribuindo ao grafo correspondente diferentes métricas de distância entre cada par de pontos, ainda obedecendo às retrições do TSP Métrico. Para cada tamanho, métrica e algoritmo, os resultados e recursos de tempo e espaço gastos foram coletados, analisados e comparados. A implementação e análise dos resultados de cada algoritmo são o foco principal deste artigo.

O texto deste trabalho está organizado da seguinte maneira: na seção 2 são apresentadas as definições e detalhes de implementação de cada um dos algoritmos abordados. Na seção 3, são apresentados os métodos e métricas utilizados nos experimentos realizados com cada algoritmo, bem como uma análise comparativa dos algoritmos em relação aos resultados obtidos. Por fim, a seção 4 apresenta as principais conclusões tiradas sobre cada implementação com relação aos experimentos realizados.

2 Algoritmos

Conforme introduzido, foram selecionados três algoritmos de tratamento do TSP para implementação: Branch-and Bound, Twice-Around-the-Tree e o Algoritmo de Christofides. Esta seção se concentra em descrever os detalhes teóricos e práticos da implementação de cada um desses algoritmos. Em todos os algoritmos, a entrada considerada foi uma matriz de adjacência do grafo representativo do problema, na forma de um numpy array, e a saída uma dupla contendo a solução do problema, representada por uma lista de vértices em ordem de caminhamento (incluindo a repetição do vértice inicial no fim) e o custo da solução.

Todos os algoritmos abordados foram implementados na linguagem Python, considerando a versão 3.7.0. Para a manipulação de matrizes, foi usada a biblioteca Numpy, e para a manipulação de grafos

foi usada a bliblioteca NetworkX. A escolha dessas bibliotecas foi feita levando em consideração a ampla gama de funcionalidades trazidas por elas no âmbito da manipulação dessas estruturas, bem como o fato de serem bibliotecas padrão em grande parte das distribuições da linguagem. Além disso, todos os algoritmos polinomiais mencionados ao longo da definição dos algoritmos correspondem às implementações usadas em suas funções por suas respectivas bibliotecas, garantindo uma correspondência entre os resultados teóricos e os práticos.

2.1 Branch and Bound

Após a apresentação do problema, o algoritmo mais ingênuo para encontrar uma solução exata seria testar todos os ciclos possíveis e selecionar o menor entre eles utilizando uma árvore de tentativas. Nesse caso, dado um ponto de início, definimos ele como raiz da árvore e a dividimos em galhos terminados em nós representando todos os possíveis pontos aos quais o caixeiro pode seguir. A partir deles adicionamos mais galhos para os próximos pontos, até que um ciclo seja formado. Então, dentre todos os ciclos, nós folhas da árvore, selecionamos aquele cujo custo total é menor.

Apesar de retornar uma solução exata, esse algoritmo faria iterações de ordem fatorial, o que só seria viável para instâncias muito pequenas. Embora a complexidade do problema não nos permita fugir do paradigma de Backtracking sem provar que P = NP, podemos melhorar o algoritmo estimando, a cada solução parcial gerada na árvore de tentativas, o custo mínimo de todos os ciclos que a contêm. Desse modo, após encontrar um ciclo completo, podemos ignorar todos os nós na árvore de tentativas cujo mínimo é maior que o custo do ciclo que já temos, bem como todas as suas subárvores filhas. Repetimos o processo encontrando ciclos de custo cada vez menor e eliminando as subárvores piores que seus custos, até que não sobrem mais nós para serem percorridos e a solução mínima exata seja encontrada. Essa é a essência do algoritmo Branch-and-Bound.

Para definir uma implementação, é necessário, primeiro, definir a função de custo mínimo. É notável que, em um ciclo completo, cada vértice tem exatamente duas arestas adjacentes, a que liga ao vértice de entrada e a que liga ao vértice de saída, que, por definição, são únicos. O custo de um ciclo completo pode ser calculado, então, somando os pesos das duas arestas do ciclo adjacentes a cada vértice e dividindo por dois para eliminar repetições (a aresta ab seria contada duplamente como ba, por exemplo). Se não sabemos quais arestas adjacentes a determinado vértice pertencem à solução, sabemos que a soma de ambas é no mínimo a soma das duas arestas de menor peso adjacentes ao vértice. Igualmente, se conhecemos apenas uma aresta adjacente ao vértice, a soma é no mínimo a soma dela à aresta de menor peso restante. Assim, podemos definir a nossa função de custo mínimo para uma solução parcial como a soma dos pesos das duas menores arestas adjacentes a cada vértice fora da solução, que são desconhecidos, somados aos pesos duas arestas adjacentes a cada vértice interior, ambas conhecidas, somadas à aresta conhecida mais a menor aresta restante dos dois vértices das extremidades, que possuem apenas uma aresta conhecida, dividido por dois. Assim, qualquer ciclo que contenha a solução parcial terá a soma de arestas de cada vértice no mínimo igual

à soma mínima definida pela função, tornando a soma total uma função de custo mínimo para a solução parcial.

É também necessário definir a forma de caminhar pela árvore de tentativas, bem como a respectiva estrutura de representação da árvore que favoreça o caminhamento. É notável que os cortes na árvore só podem ser feitos após um ciclo completo ser definido. Assim, é mais vantajoso para o algoritmo fazer um caminhamento por profundidade na árvore, pois assim mais soluções são acessadas e mais subárvores são cortadas. Dessa forma, a estrutura ideal para representar a árvore é uma pilha, na qual os nós filhos de cada nó são empilhados logo após o nó ser acessado, garantindo o caminhamento por profundidade.

Embora Thomas H. Cormen *et al*, em seu livro *Introduction to Algorithms*, recomendem acessar os nós da árvore em ordem de custo mínimo, utilizando um heap binário para a representação da árvore [CLRS01], foi observado nos testes que, na prática, uma solução parcial menor tem grandes chances de ter um custo mínimo menor que uma solução parcial com mais vértices, então o algoritmo gastava muito tempo incluindo soluções parciais no heap antes de alcançar um ciclo completo e iniciar os cortes, funcionando quase como uma busca por largura na árvore e neutralizando a eficiência sugerida pelo método. Além disso, o custo de inserção e remoção no heap não são constantes, e o alto número de soluções parciais adicionadas simultaneamente ao heap sobrecarrega a memória em instâncias maiores. Assim, a pilha foi escolhida como estrutura de representação da árvore, pois, além de favorecer o algoritmo, ela tem custo de inserção e remoção constantes e atinge tamanho máximo quadrático no número de vértices (n filhos para cada um dos n níveis da árvore), não sobrecarregando a memória tanto quanto o custo espacial potencialmente exponencial do heap.

Como alguns detalhes adicionais, o peso da aresta que liga um vértice a ele próprio foi definido como infinito, para evitar que a distância nula do vértice a ele próprio fosse favorecida em relação às distâncias do vértice a outros vértices. Além disso, como o algoritmo encontra ciclos, a sequência correspondente ao caminhamento pelo ciclo na outra direção pode ser ignorada (o ciclo abcda e o ciclo adcba são essencialmente os mesmos, por exemplo). Assim, para evitar o caminhamento por ciclos repetidos, foi definido que o vértice de índice 2 só deve ser adicionado à solução parcial após o vértice de índice 1 ter sido adicionado, pois cada ciclo onde o 1 vem antes do 2 define uma sequência pelo seu caminho inverso onde o 2 vem antes do 1, que são justamente as que queremos ignorar.

Tendo todos esses detalhes em mente, o algoritmo foi implementado no módulo `Branch_n_bound.py` como a função `TSP_Branch_and_Bound`, recebendo como parâmetros a matriz de adjacência A do grafo em forma de um numpy array. Para o cálculo da função de custo mínimo, é feita uma lista com os índices dos vértices finais das duas arestas de menor peso adjacentes a cada vértice. A função bound, que retorna o custo mínimo, recebe essa lista, a matriz de adjacência e a solução parcial e calcula o custo mínimo da solução parcial conforme especificado anteriormente. Então usando uma classe Node, criada contendo o custo total mínimo, o nível, o custo parcial e a solução parcial de um nó, o nó raiz é definido contendo o vértice inicial 0 na solução parcial, custo nulo, nível 1 e custo mínimo definido conforme a função bound. Em seguida, o nó raiz é inserido na pilha, que foi implementada usando a própria lista padrão da linguagem, que possui os

métodos `append` e `pop` de inserção e remoção em seu fim constantes, e o melhor custo e melhor solução são definidos como infinito e uma lista vazia, respectivamente. Então, enquanto a pilha não estiver vazia, o nó do topo é desempilhado e, caso seu nível seja maior que o número de vértices, indicando que ele é um ciclo, a melhor solução é atualizada caso seu custo seja menor. Se o nível for menor que o número de vértices, indicando que ela é uma solução parcial, e seu custo mínimo for menor que o custo da melhor solução, são inseridos na pilha todos os seus filhos possíveis que tenham custo mínimo menor que a solução. Se o nível for igual ao número de vértices, indicando que todos os vértices foram caminhados, o vértice inicial é adicionado ao final da solução parcial para fechar o ciclo e inserido na pilha. Após todas as iterações, a função retorna a lista contendo a solução encontrada e seu respectivo custo.

2.2 Twice-Around-the-Tree

Embora o algoritmo anterior encontre uma solução exata, ele ainda é potencialmente exponencial, visto que caminha pela árvore exponencial de tentativas por um número indeterminado de iterações. Assim, uma outra abordagem para a solução do problema se dá no uso de algoritmos de complexidade de tempo polinomial, mas que encontram soluções aproximadas para o problema. Nesse contexto, trazemos como primeiro algoritmo aproximativo para o TSP o Twice-Around-the-Tree.

A árvore geradora de um grafo é um de seus subgrafos conexos que forma uma árvore. A árvore geradora mínima, ou AGM, do grafo é a árvore geradora de menor custo total do grafo, e pode ser encontrada em tempo polinomial $\mathcal{O}(V^2 \log V)$ em um grafo completo pelo algoritmo de Kruskal [Kru56]. A vantagem da AGM é que, como todo subgrafo conexo contém uma árvore geradora, o custo de um subgrafo conexo é no mínimo igual ao custo da AGM, inclusive o subgrafo contendo o ciclo solução do TSP. Assim, podemos usá-la para encontrar uma solução aproximada para o problema.

Suponha que temos a árvore geradora mínima do grafo em questão. Se criarmos um multigrafo duplicando as arestas da AGM, um circuito euleriano passando por todas as arestas do multigrafo custaria o dobro do custo da AGM, que é menor que o dobro do custo da solução do TSP. Considerando que os pesos das arestas formam uma métrica, ou seja, a desigualdade triangular se aplica entre as arestas do grafo, podemos cortar caminho ignorando os vértices repetidos do circuito euleriano, o que geraria um caminho sem repetições de custo total no máximo igual ao dobro do custo da AGM, que é menor que o dobro da solução ótima do TSP, encontrando uma solução aproximada de custo no máximo igual ao dobro da melhor possível. Considerando que o circuito euleriano ignorando vértices repetidos é equivalente a um caminhamento pré-ordem na AGM, o Twice-Around-the-Tree consiste portanto, em encontrar a AGM do grafo, realizar um caminhamento pré-ordem na AGM e retornar os vértices na ordem do caminhamento, mais o vértice inicial no fim para fechar o ciclo. De todos os passos, o que domina o custo de tempo do algoritmo é a árvore geradora mínima, levando a uma complexidade final $\mathcal{O}(V^2 \log V)$.

A biblioteca de manipulação de grafos Networkx, mencionada anteriormente, traz ferramentas nativas para realizar todas essas operações. Assim, o algoritmo foi implementado no módulo `Twice_around_the_tree.py`

como a função `TSP_Twice_Around_the_Tree`, que recebe a matriz de adjacência do grafo na forma de um numpy array e cria um objeto graph da biblioteca pela função `from_numpy_array`. Em seguida, ele encontra a AGM do grafo usando a função `minimum_spanning_tree` e lista seu caminhamento pré-ordem usando a função `dfs_preorder_nodes`. Por fim, ele adiciona o vértice inicial ao fim da lista, soma o custo total do ciclo gerado e retorna a solução e seu custo.

2.3 Algoritmo de Christofides

A aproximação feita no algoritmo anterior se dá cortando caminho pelo circuito euleriano formado dobrando a árvore geradora mínima (vide o nome, indicando o circuito dobrado em torno da árvore). O matemático Nicos Christofides observou, entretanto, que como os únicos vértices da AGM impedindo-a de conter um circuito euleriano são os vértices de grau ímpar, basta adicionar ao grafo arestas apenas entre esses vértices, e o grafo resultante conterá um circuito euleriano potencialmente menor e, consequentemente, uma aproximação melhor [Chr76].

Um matching perfeito em um grafo é um subconjunto das arestas do grafo que conecta todos os vértices de modo que cada vértice seja adjacente a exatamente uma aresta. O matching perfeito de custo mínimo pode ser encontrado em tempo polinomial cúbico no número de vértices, usando o algoritmo de Blossom [Edm65]. Christofides propõe, então, uma melhoria do Twice-Around-the-Tree criando um multigrafo união da AGM com o matching perfeito de custo mínimo no subgrafo contendo apenas os vértices de grau ímpar na AGM. Assim, a adição de uma aresta adjacente a cada vértice de grau ímpar torna todos os graus pares e permite a realização do circuito euleriano.

Note que a única configuração que permite um matching perfeito é a ligação de vértices em pares por cada aresta, logo, o número de vértices do subgrafo deve ser par. No entanto, nesse caso o matching perfeito sempre irá existir, pois o número de vértices de grau ímpar em um grafo sempre é par. Tomando o subgrafo de vértices ímpares, todo circuito hamiltoniano nele define um matching perfeito alternando suas arestas. Assim, o matching perfeito de peso mínimo tem peso no máximo igual à metade do circuito hamiltoniano de peso mínimo do subgrafo de vértices ímpares, que é no máximo igual ao circuito hamiltoniano de peso mínimo no grafo original, que é a solução ótima do TSP. Assim, o algoritmo de Christofides encontra uma solução de custo no máximo igual ao custo do circuito euleriano formado, que é igual à soma dos custos da AGM e do matching perfeito, que são no máximo iguais à solução do TSP e à sua metade, respectivamente, encontrando uma solução no máximo 1.5 vezes maior que a ótima, reduzindo pela metade a faixa de erro do Twice-Around-the-Tree.

Desse modo, o algoritmo de Christofides consiste em encontrar a árvore geradora mínima do grafo, eliminar do grafo todos os vértices de grau par na AGM, encontrar o matching perfeito de peso mínimo no subgrafo restante, adicionar as arestas do matching perfeito na AGM, com os pesos originais, fazer um circuito euleriano na AGM com as arestas adicionais e retornar o circuito eliminando os vértices repetidos (assim como no algoritmo anterior, esse passo exige que os pesos formem uma métrica). Como a única

alteração feita no algoritmo Twice-Around-the-Tree foi a adição do algoritmo de matching perfeito de custo mínimo, de complexidade cúbica maior que a do algoritmo anterior, a complexidade de tempo é dominada pelo custo do matching, $\mathcal{O}(V^3)$.

Assim como no Twice-Around-the-Tree, a biblioteca NetworkX traz funções nativas que permitem realizar todos os passos do Algoritmo de Christofides. Portanto, o algoritmo foi implementado no módulo como a função `TSP_Christofides`, que recebe a matriz de adjacência do grafo na forma de um numpy array e cria um objeto G da classe graph da biblioteca pela função `from_numpy_array`. Em seguida, ela encontra a AGM de G usando a função `minimum_spanning_tree`, representa ela como um objeto da classe MultiGraph e todo nó de grau par da árvore é eliminado do grafo original. Embora a biblioteca não tenha uma função para realizar o matching perfeito de peso mínimo, ela tem a função `max_weight_matching`, que, se chamada com `max_cardinality` = True, encontra o matching de cardinalidade máxima (nesse caso, o matching perfeito), de peso máximo. Assim, a chamada da função para o grafo com seus pesos negativos encontra o matching perfeito de peso máximo negativo, que define o matching perfeito de peso mínimo positivo no grafo (o que só pode ser feito se todos os pesos das arestas do grafo forem positivos). Com isso, os pesos de todas as arestas do grafo restante são multiplicados por -1 e a ele é aplicada a função `max_weight_matching`, encontrando as arestas do matching perfeito de peso mínimo. Essas arestas são adicionadas no multigrafo da AGM, com os pesos originais, e nele é feito um circuito euleriano usando a função `eulerian_circuit` começando do vértice 0. Por fim, a função retorna como solução os vértices do circuito euleriano sem as repetições mais o vértice inicial no fim fechando o ciclo, e o custo do ciclo encontrado.

3 Experimentos

3.1 Métricas de Análise

Além da implementação dos algoritmos, esse trabalho objetiva analisar a performance de cada um em relação a vários aspectos. Para tal, foram criadas 7 instâncias aleatórias do problema, de números de vértices 2^i para $i = 4, 5, \ldots, 10$. Para garantir a restrição métrica do problema, esses vértices foram selecionados como pontos aleatórios no plano, de dimensões entre 0 e 100. Em seguida, os pesos das arestas entre os vértices foram selecionados como duas métricas diferentes: a distância euclidiana entre os pontos e a distância Manhattan, ou distância do táxi, definida como a soma dos módulos das diferenças das coordenadas (interpretada como a distância mínima entre dois pontos sendo permitida apenas a movimentação paralela aos eixos do plano). Essa escolha se deu como forma de observar se, apesar dos limites dos algoritmos, diferentes algoritmos se beneficiam de diferentes métricas. Além disso, a distância Manhattan é uma poderosa ferramenta na modelagem de pontos organizados em grades, como as ruas de uma cidade, por exemplo, então observar o funcionamento dos algoritmos usando essa métrica pode levantar aspectos da aplicação prática dos mesmos a problemas de logística reais. As instâncias foram criadas usando o módulo `gerador_instancias.py`, e salvas como arquivos de texto no diretório Instancias.

Cada algoritmo foi chamado para os 14 grafos gerados, e para cada chamada foram coletados em um csv o custo total da solução, o tempo total de execução e a memória máxima alocada. O tempo em segundos foi coletado usando a função `default_timer` da biblioteca `timeit`, e a memória máxima em kb foi coletada usando o parâmetro `ru_maxrss` do método `getrusage` da biblioteca `resource`. Algoritmos cujo tempo de execução excedeu 10 minutos foram considerados sem solução, e o timeout das funções foi feito usando a biblioteca `signal`.

Como a alocação da memória em Python é organizada pela própria linguagem, cada chamada de algoritmo foi feita em um módulo separado, para evitar que a memória alocada para uma chamada fosse usada para outra e não contabilizada. Dessa forma, o módulo `run_alg.py` recebe como parâmetros do terminal o algoritmo a ser usado, o tamanho logarítmico da instância e a métrica usada e chama o respectivo algoritmo para a respectiva instância, conforme especificado, e guarda os resultados em um csv. As soluções também são guardadas em um diretório Solucao. Para amarrar as chamadas, um script bash `main.sh` chama o gerador de instâncias, cria o csv e chama `run_alg.py` para cada instância e algoritmo possíveis. As subseções seguintes tratam dos resultados obtidos nos experimentos para cada parâmetro analisado.

3.2 Qualidade da Solução

Com relação à qualidade da solução retornada por cada algoritmo, evidentemente o algoritmo Branch-and-Bound teve a melhor performance, como esperado, já que sua solução é exata. No entanto, ele só conseguiu alcançar a solução ótima nas intâncias de tamanho 2^4, ultrapassando o tempo máximo estipulado em todos os outros casos. A Figura 1 apresenta os resultados obtidos para cada métrica com relação à qualidade da solução:

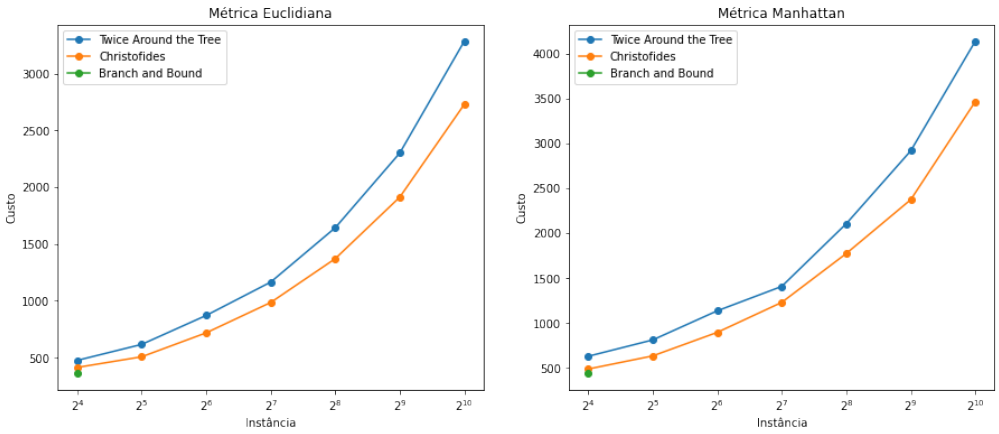

Figura 1: Custo da solução de cada instância encontrado por cada algoritmo

Embora o Branch-and-Bound tenha encontrado a melhor solução em ambos os casos da primeira instância, é evidente que a qualidade dos demais algoritmos não distoa muito da solução ótima. Na instância de tamanho 2^4, para a qual o ótimo foi encontrado, a solução do algoritmo Twice-Around-the-Tree foi 31%

pior que a ótima, para a métrica euclidiana, e 43% pior que a ótima para a métrica Manhattan, enquanto o algoritmo de Christofides excedeu a ótima em 14% e 10%, para as respectivas métricas. Em todos os casos as margens de erro de cada algoritmo foram menos que a metade da demonstrada para cada algoritmo. No caso específico do algoritmo de Christofides as soluções tiveram faixa de erro de 4 a 5 vezes menores que o esperado, um resultado notável.

Com relação às métricas, o algoritmo Twice-Around-the-Tree teve um desempenho pior em relação à ótima para a métrica Manhattan do que para a métrica euclidiana. Isso possivelmente se dá pelo fato de que, na métrica Manhattam, muitas vezes a distância de um caminho entre três vértices uvw não é encurtada fazendo o caminho direto uw (como no caso de três vértices de um triângulo retângulo, por exemplo). Assim, o passo do caminhamento pré-ordem tende a diminuir menos o custo do caminho euleriano para a métrica Manhattan que para a euclidiana. Pelo mesmo motivo, é conjecturado que o algoritmo de Christofides teve um melhor desempenho em relação à ótima para a métrica Manhattam porque as distâncias eliminadas do caminho euleriano com o matching perfeito em relação às demais arestas do grafo são maiores na métrica Manhattan do que na métrica euclidiana. É importante ressaltar, no entanto, que a escassez de resultados exatos pelo Branch-and-Bound não nos permite tirar conclusões concretas sobre a relação dos demais algoritmos com as respectivas soluções ótimas.

Comparando os algoritmos aproximativos entre si, tanto para a métrica euclidiana quanto para a métrica Manhattam a solução dada pelo Twice-Around-the-Tree em cada instância foi cerca de 20% a 25% pior que a solução retornada pelo algoritmo de Christofides. Esse resultado, embora ligeiramente melhor, está em consonância com o valor esperado considerando seus coeficientes de aproximação. Como o primeiro retorna uma solução no máximo duas vezes maior que a ótima, e o segundo no máximo 1.5 vezes, o custo do primeiro é 0.5 vezes a ótima a mais que o segundo, ou seja, $\frac{0.5}{1.5} = 33.3\%$ pior. A diferença entre as qualidades dos dois algoritmos foi ligeiramente maior para a métrica Manhattam (cerca de 24 a 26%) do que para a métrica euclidiana (cerca de 19 a 22%), trazendo mais evidências para a hipótese de que o Twice-Around-the-Tree e o Algoritmo de Christofides têm desempenho, respectivamente, pior e melhor em relação à ótima para métrica Manhattan que para a métrica euclidiana.

3.3 Tempo Gasto

Com relação ao tempo de execução, conforme observado no item anterior, o algoritmo Branch-and-Bound teve a pior performance entre todos os algoritmos, alcançando a solução ótima apenas nas intâncias de tamanho 2^4, e ultrapassando o tempo máximo estipulado em todos os outros casos. Esse resultado também era esperado considerando a complexidade de tempo do algoritmo. A Figura 2 apresenta os resultados obtidos para cada métrica com relação ao tempo de execução.

Primeiramente, é evidente que na instância de tamanho 2^4, na qual o Branch and Bound terminou, o algoritmo foi muito mais rápido para a métrica Manhattan que para a métrica euclidiana, quase 50% mais rápido, em valores relativos, e cerca de três minutos e meio, em valores absolutos, o que em tempo

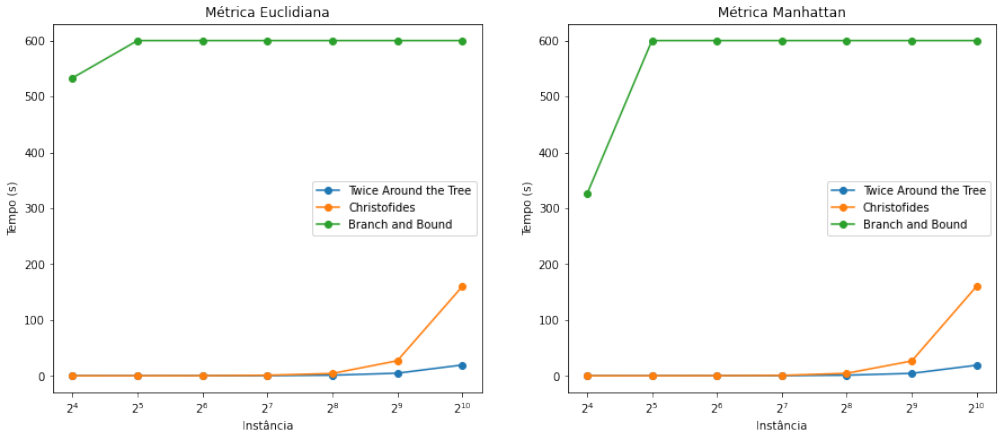

Figura 2: Tempo gasto por cada algoritmo em cada instância

computacional é extremamente alto. A principal hipótese para a causa dessa diferença é justamente o comportamento da distância Manhattan com relação à distância de caminhos intermediários entre vértices. Conforme explicado no item anterior, muitas vezes a distância Manhattan do caminho feito pelos vértices uvw é igual à distância direta uw. Assim, após encontrar um ciclo completo, as chances de as soluções parciais empilhadas contendo os vértices mais próximos do fim do ciclo terem custo próximo ou maior que o do ciclo são altas, permitindo que soluções parciais sejam cortadas mais próximas da raiz e o algoritmo execute em tempo menor. No entanto, como no caso anterior, a falta de mais casos para sustentar a hipótese não nos permite tirar conclusões concretas.

Normalizando o eixo x e ignorando os altos tempos de execução do Branch and Bound, na figura 3, podemos ver que as complexidades de cada algoritmo polinomial se refletem nos gráficos de seus tempos de execução. O algoritmo de Christofides, de complexidade $\mathcal{O}(V^3)$, apresenta uma curva muito mais acentuada que o Twice-Around-the-Tree, de complexidade $\mathcal{O}(V^2 \log V)$, que na Figura 2 aparenta formar uma reta, mas cuja curva sutil pode ser observada na figura 3. Apesar de a proporção das qualidades das soluções retornadas pelos algoritmos aproximativos se manter quase constante ao longo do tempo, conforme observado no item anterior, a proporção do tempo gasto por cada algoritmo cresce, evidentemente, em um fator quase linear. Assim, para instâncias suficientemente grandes, o alto tempo de execução do Algoritmo de Christofides deixa de ser viável tendo em vista sua melhoria constante na qualidade, de modo que o Twice-Around-the-Tree seja mais vantajoso nesse aspecto.

3.4 Memória máxima alocada

Com relação à memória máxima alocada, o algoritmo Branch-and-Bound novamente teve o pior desempenho, mas a discrepância não foi tão grande quanto na complexidade de tempo, devido à adaptação feita com uso da pilha. A Figura 4 apresenta os resultados obtidos para cada métrica com relação à memória máxima alocada, em kB:

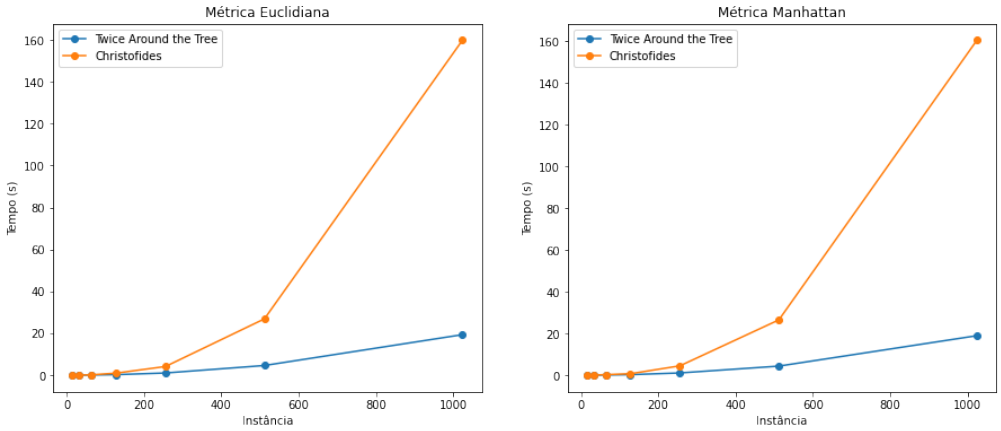

Figura 3: Tempo gasto por cada algoritmo polinomial em cada instância

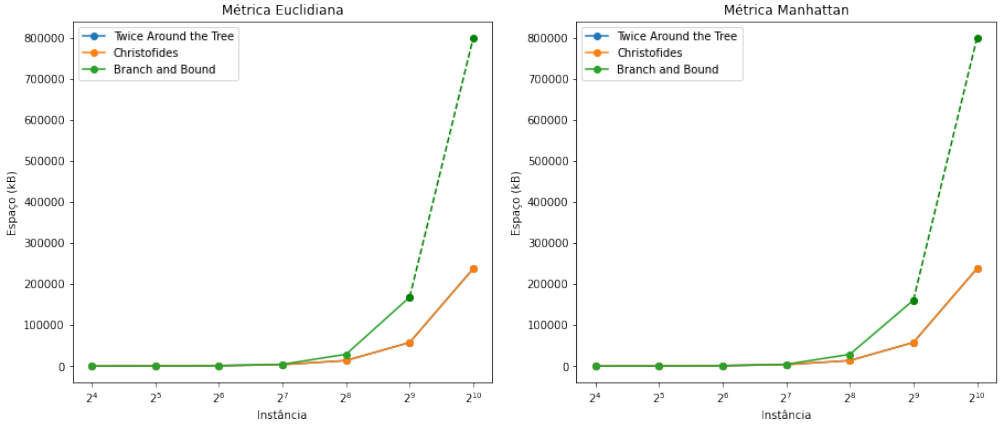

Figura 4: Memória máxima gasta por cada algoritmo em cada instância (Escala Logarítmica)

O pontilhado no gráfico do algoritmo Branch-and-Bound indica que o último valor não foi de fato coletado, e sim previsto a partir dos demais valores e das circunstâncias de execução do programa. Isso se dá pelo fato de que a execução dos casos de tamanho 1024 sobrecarregou a memória do computador antes que os dados pudessem ser coletados, possivelmente devido a outros processos do sistema operacional sendo executados simultaneamente, então os valores só puderam ser previstos. É notável que o Branch-and-Bound também excedeu os algoritmos aproximativos em relação à alocação de memória, mas ele alocou quantidades muito próximas aos algoritmos aproximativos nas instâncias menores. Embora as curvas nos gráficos apontem para um custo espacial exponencial, isso é apenas uma impressão causada pelos valores do eixo x organizados em proporção logarítmica, para equalizar as distâncias entre cada instância. O custo espacial do Branch-and-Bound ainda é quadrático, conforme pode ser visto na Figura 5, cujo eixo foi normalizado.

É interessante observar, também, que as curvas de espaço dos dois algoritmos aproximativos se coincidem quase que completamente nos gráficos. Isso se dá pelo fato de que, em ambos, a complexidade de espaço é dominada pelo espaço ocupado pelo grafo e o espaço ocupado pela AGM dele, ambos objetos

da classe NetworkX. Nos dois algoritmos, o espaço total alocado não supera, a nível de kilobytes, o espaço alocado para as duas estruturas, gerando duas curvas coincidentes. Como eles têm complexidade de espaço linear no tamanho do grafo e o grafo é completo, a complexidade dos algoritmos também é quadrática, mas gera uma parábola muito mais suave que a do Branch and Bound, como também pode ser observado na Figura 5.

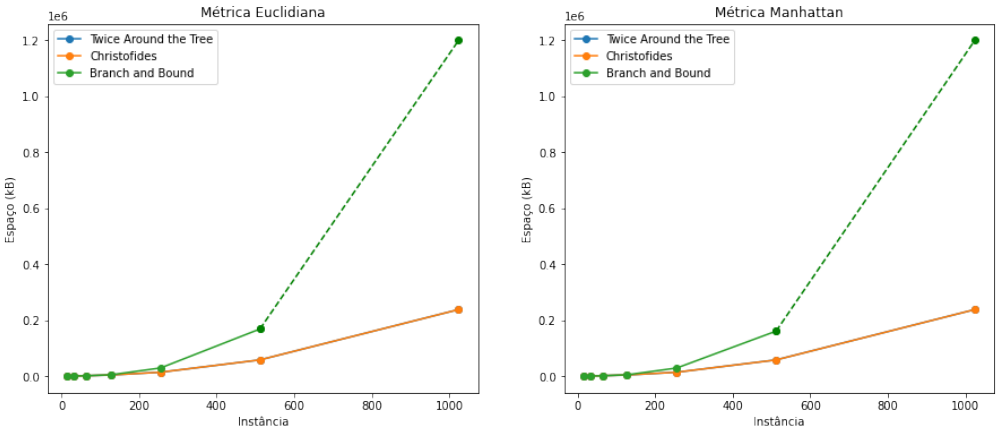

Figura 5: Memória máxima gasta por cada algoritmo em cada instância (Escala Padrão)

4 Conclusão

Embora o Problema do Caixeiro Viajante pertença à classe NP-difícil, é possível abordá-lo de modo a obter soluções satisfatórias para aplicações práticas. Os algoritmos apresentados e implementados neste artigo trazem soluções que favorecem diferentes aspectos, que puderam ser vistos em mais detalhe nos testes. Os resultados dos experimentos nos permitiram identificar que, dentre todos os algoritmos analisados, o Branch-and-Bound, embora se destaque pela qualidade ótima de sua solução, apresentou o maior gasto de tempo e espaço, e só foi capaz de alcançar a solução da menor instância dos testes no tempo estipulado. Mesmo estendendo o tempo limite de execução do algoritmo, ele se torna inviável para instâncias ainda na casa das dezenas, devendo ser utilizado apenas em intâncias pequenas nas quais a solução ótima é essencial. A implementação do algoritmo também nos permitiu refletir sobre como a escolha da estrutura de representação do algoritmo pode influenciar a sua complexidade de espaço. Foi observado a partir dos resultados que entre os algoritmos polinomiais aproximativos, o Algoritmo de Christofides apresenta uma aproximação melhor que o Twice-Around-the-Tree em todas as instâncias, mas sua vantagem em relação à qualidade do Twice-Around-the-Tree é constante, enquanto sua desvantagem em relação ao tempo cresce com o tamanho da instância. Assim, o Algoritmo de Christofides apresenta o melhor desempenho no caso geral, enquanto o Twice-Around-the-Tree é o ideal para instâncias grandes demais para a complexidade do Algoritmo de Christofides. Em ambos os casos, ficou evidente que a complexidade de espaço de ambos os algoritmos é quase idêntica.

Com relação às métricas utilizadas, os resultados sugerem um melhor desempenho do Algoritmo de Christofides na qualidade da solução em relação à ótima utilizando a métrica Manhattan, enquanto o Twice-Around-the-Tree se saiu melhor em relação à ótima utilizando a métrica Euclidiana. A diferença na qualidade dos algoritmos aproximativos foi ligeiramente maior para a métrica de Manhattan, o que sustenta essa hipótese. O Branch-and-Bound demonstrou um tempo de execução consideravelmente melhor utilizando a métrica Manhattan, mas não é possível concluir se essa melhora é causada pela métrica ou se é específica da instância observada, devido à carência de soluções exatas para as demais instâncias. Por esse mesmo motivo, as comparações dos algoritmos aproximativos em relação à ótima também requerem mais dados para sustentar as hipóteses levantadas. Fora esses pontos, as métricas não apresentaram mudanças significativas no tempo de execução ou memória alocada pelos algoritmos.

Referências

[Chr76] Nico Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *National Technical Information Service*, Feb 1976.

[CLRS01] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1), 1956.

[Pol18] Burkard Polster. Epicycles, complex fourier series and homer simpson's orbit. https://www.youtube.com/watch?v=qS4H6PEcCCA, Jul 2018. Acesso em: 27 de Agosto de 2021.