UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Graduação em Sistemas de Informação

Pedro de Oliveira Guedes

Análise comparativa de algoritmos para solução do problema do “Caixeiro Viajante”

Belo Horizonte

2023

Resumo

O presente artigo busca realizar a avaliação de desempenho de diferentes algoritmos utilizados para a solução do problema do caixeiro viajante, abordando uma solução exata e duas aproximativas. Os algoritmos serão avaliados quanto à proximidade da solução encontrada em relação à ótima, o tempo necessário para chegar àquela solução e a memória utilizada por eles.

Essa análise se faz necessária, principalmente, para mostrar as diferenças de execução real dos algoritmos em relação às suas respectivas complexidades assintóticas, que, apesar de importantes, são muito abstratas e dificilmente refletem cenários do mundo real. Ao fim, os três algoritmos implementados terão os resultados comparados entre si, apresentando reflexões sobre a utilização dos mesmos.

Palavras-chave: Caixeiro Viajante. Algoritmos aproximativos. Análise comparativa.

SUMÁRIO

Introdução

O problema do Caixeiro Viajante é um dos mais famosos da computação, consistindo em otimizar a rota de um vendedor entre pontos de interesse, partindo de um ponto inicial e retornando a ele ao fim, para que o custo seja mínimo ao final da viagem. Esse algoritmo é resolvido por Máquinas de Turing Determinísticas com complexidade de tempo $O(2^n)$, mas poderia ser resolvido não-deterministicamente em tempo $O(n)$, fazendo com que ele pertença à classe NP. Além disso, já foi provado que ele também é NP-Difícil, o que faz com que ele seja um problema NP-Completo.

Por ser NP-Completo, a solução desse problema em tempo polinomial para máquinas determinísticas pode levar a grandes descobertas no campo da computação, já que todos os outros problemas NP-Completos, que são redutíveis a ele, poderiam também ser reduzidos. No campo prático, ainda não foi possível desenvolver tal algoritmo, mas já existem alternativas que fazem com que o tempo de computação seja consideravelmente reduzido para instâncias grandes.

O propósito deste artigo é implementar três desses algoritmos, sendo um deles exato e dois aproximativos, para realizar comparações quanto à proximidade da solução fornecida e a ótima, além do tempo e memória consumidos para a computação das instâncias.

Para atingir esse objetivo, o restante do artigo foi organizado em seções, sendo esta a seção 1. A seção 2 trata dos conceitos básicos para o entendimento do artigo, explicando-os e relacionando com os objetivos. A seção 3 trata dos algoritmos implementados em mais detalhes, abordando escolhas de bibliotecas e decisões de projeto relacionadas. A seção 4 é relativa à modelagem dos experimentos a serem feitos, que serão utilizados para realizar a análise comparativa. A seção 5 aborda a análise comparativa de fato, abrigando a grande maioria dos gráficos e mostrando as variações entre algoritmos. A seção 6 conclui o artigo, mostrando os resultados obtidos de forma resumida.

Conceitos básicos

Para compreender este artigo por completo, é necessário antes entender o que é o problema do caixeiro viajante e porquê ele é tão importante para a computação. Assim como mencionado anteriormente, esse é um problema de otimização, onde se busca a melhor forma, ou sequência de passos, para realizar uma ação.

Para defini-lo de forma mais completa, primeiro assuma a existência de diversos pontos de interesse no espaço, que podem ser cidades, lojas, pessoas, entre outros. Assuma então a existência de um vendedor, ou caixeiro, que parte de um desses pontos, devendo visitar todos os outros e retornar ao ponto inicial, sem que nenhum seja repetido. Assuma também que para sair de um ponto e chegar até outro, existe um custo, que pode ser representado como a distância entre eles, o tempo de deslocamento, dificuldade, entre outras grandezas. O problema do caixeiro viajante consiste em encontrar a rota que minimize o custo total da viagem, respeitando as restrições impostas.

Quando a teoria dos problemas NP-Completos surgiu, o caixeiro viajante foi um dos primeiros a ser provado como NP-Difícil, o que significa que ele é redutível a qualquer outro problema NP-Completo, assim como todo outro problema NP-Completo também é redutível a ele. Essa última afirmação é o que faz com que esse problema seja tão estudado na computação, pois ela implica que, se um algoritmo polinomialmente eficiente for desenvolvido para o caixeiro viajante, todos os outros problemas NP-Completos também poderão ser resolvidos em tempo polinomial.

Além da importância teórica do problema já apresentada, existe também a importância prática, já que ele é um problema de logística do mundo real, sendo aplicado para otimizar as rotas de entrega de distribuidoras. Apesar de ainda não ter sido desenvolvido um algoritmo polinomial, já existem diversos algoritmos que solucionam o problema em tempo relativamente eficiente, dadas algumas restrições.

Por fim, os algoritmos existentes podem ser divididos em dois grandes grupos, que são os exatos e aproximativos. Os algoritmos exatos são aqueles que encontram a solução exata para o problema, verificando todas as possibilidades inerentes à instância e excluindo as menos eficientes, até que essa solução seja alcançada. Já os aproximativos não verificam exaustivamente as possibilidades, ao invés disso eles utilizam heurísticas para encontrar um valor que seja próximo o suficiente do ótimo.

Algoritmos e técnicas implementadas

A análise comparativa se dará entre um algoritmo exato, que foi construído utilizando a técnica *Branch and Bound*, e dois algoritmos aproximativos, que utilizam a árvore geradora mínima dos grafos para estimar o peso mínimo que leva o caixeiro a visitar todos os pontos de

interesse. Ao longo desta seção, cada um dos algoritmos será discutido, em subtópicos próprios, quanto ao funcionamento e decisões de estruturas de dados empregadas.

Todos os algoritmos foram desenvolvidos na linguagem Python, versão 3.12, usando os pacotes nativos da linguagem e os pacotes que vêm em adição à ferramenta Jupyter Notebook, com exceção da biblioteca NetworkX, que é externa foi utilizada para a manipulação dos grafos.

Algoritmo *Branch and Bound*

A técnica "*Branch and Bound*" de desenho de algoritmos é utilizada, principalmente, para problemas de otimização, buscando evitar a exploração completa e exaustiva de todas as possibilidades disponíveis. Essa técnica consiste na definição de estimativas de melhor caso a cada nova possibilidade, privilegiando a exploração de ramos da árvore de possibilidades que sejam válidos tenham melhor estimativa, até que uma solução completa e válida seja atingida. A partir desse momento, toda possibilidade cuja estimativa de melhor caso foi computada é comparada com a solução, caso a estimativa não apresente um valor melhor do que a solução atual, aquela possibilidade deixa de ser explorada, reduzindo a quantidade de passos total do algoritmo.

Utilizando esta técnica, uma das partes mais importantes do desenvolvimento é a criação da função de cálculo da estimativa de melhor caso. Para realizar esse cálculo, são somados para cada ponto de interesse, doravante denominados vértices, os dois menores pesos de conexão, doravante denominados arestas, disponíveis. Essa soma ocorre porque cada vértice, para ser completamente acessado, precisa ter uma aresta incidente sobre ele, representando a chegada ao vértice, e outra que saia dele e leve o caixeiro para o próximo. Porém, como essas arestas são potencialmente computadas duas vezes, já que uma aresta que liga os vértices "u" e "v" pode ser contabilizada como de saída do vértice "u" e entrada de "v", a soma total é dividida por 2 ao fim.

Como essa estimativa é calculada a cada nova possibilidade, ao passar por um dos vértices do grafo, é possível que já tenham sido incluídas arestas conectadas a ele na solução parcial das possibilidades exploradas. Quando isso ocorre, o algoritmo de estimativa soma as arestas daquele vértice já incluídas na solução parcial, utilizando a heurística das menores arestas remanescentes apenas no caso de aquele vértice não ter duas arestas incluídas na possibilidade.

Para gerenciar as possibilidades sendo exploradas, foi criada uma classe customizada, indicando o nível da árvore em que aquela possibilidade se encontra, as arestas escolhidas e o custo acumulado com elas, além da estimativa de custo para o restante dos vértices do grafo. Essa classe é inserida em um *heap*, que é implementado e controlado pela biblioteca *NetworkX* anteriormente mencionada, utilizando a estimativa como chave de comparação para encontrar a possibilidade mais promissora, garantindo o funcionamento da técnica *best-first search*.

Apesar das heurísticas apresentadas acima, este algoritmo não deixa de ter a complexidade assintótica marcada por $O(2^n)$, já que pode ser necessário visitar todas as possibilidades de caminho para que o ótimo seja encontrado.

Algoritmo *Twice Around The Tree*

Esse foi o primeiro algoritmo aproximativo implementado, para o funcionamento dele devem ser seguidas algumas restrições leves nas instâncias, que frequentemente refletem cenários do mundo real. Entre elas, está o fato de que somente são aceitas instâncias que configuram grafos completos, não havendo melhor caso garantido em caso contrário. Para garantir essa restrição, os grafos são construídos em uma função específica, que garante a criação de um grafo completo, sem arestas auto-incidentes, ou seja, não são feitas arestas que saem de um vértice "v" e retornem a esse mesmo vértice.

Além disso, os vértices e arestas do grafo devem seguir princípios euclidianos, onde o custo de viagem é sempre igual ou maior a 0, a viagem entre dois vértices tem custo independente do ponto de partida e os custos seguem o princípio de desigualdade triangular. Este último significa que não pode existir um caminho "$u \rightarrow v \rightarrow w$", que seja menor do que um caminho "$u \rightarrow w$". Essas restrições são garantidas pelos provedores das instâncias, que serão melhor discutidos na seção de "Experimentos".

Esse algoritmo consiste em construir uma árvore geradora mínima, que é o menor conjunto de arestas, com o menor peso possível, que conecta todos os vértices do grafo. Obtidas essas arestas, é feito um caminhamento pré-ordem na árvore e construído um ciclo hamiltoniano com os vértices. O peso desses vértices é então somado, levando à aproximação final para aquela instância. A aproximação fornecida por esse algoritmo é garantida a ser, no máximo, o dobro da melhor, ou seja, se o custo ótimo para realizar a rota do caixeiro viajante for igual a

"*X*", o fornecido por esse algoritmo será no máximo "2*X*". Tanto a manipulação de grafos, quanto os algoritmos auxiliares necessários, foram utilizados da biblioteca *NetworkX*.

Algoritmo de *Christofides*

Esse algoritmo é o segundo aproximativo implementado e segue as mesmas restrições impostas ao "*Twice Around The Tree*", tendo as mesmas garantias concedidas para funcionamento. Também de forma similar ao anterior, ele utiliza a árvore geradora mínima e constrói um ciclo hamiltoniano para realizar a aproximação. Porém, antes da construção do ciclo, existem alguns passos intermediários. O primeiro deles é selecionar na árvore geradora mínima, os vértices que possuem grau ímpar, ou seja, somente vértices que possuem uma quantidade ímpar de arestas conectando eles aos outros da árvore, obtendo um subgrafo do original com esses vértices.

Com o subgrafo obtido, é construído um *matching* perfeito mínimo, que consiste em selecionar o conjunto de arestas daquele subgrafo que contemplem todos os vértices presentes, enquanto o custo total é minimizado. Com esse *matching* obtido, as arestas selecionadas dele que não estiverem presentes na árvore geradora mínima são adicionadas a ela. Por fim, é realizado um caminhamento pré-ordem na árvore, adicionando os pesos das arestas à solução final a ser retornada. Assim como no anterior, a biblioteca *NetworkX* foi utilizada para a manipulação dos grafos e aplicação dos algoritmos auxiliares.

Experimentos

Todos os experimentos foram executados em ambiente *Windows*, com um processador de 8 núcleos e 16 threads, com velocidade base de 1,80 GHz. O ambiente dispunha de 20 GB de memória RAM, sendo 19,4 GB disponíveis para execução dos processos. Para garantir a integridade posterior do sistema de teste, caso os algoritmos utilizassem mais de 95% da memória RAM disponível, eles seriam interrompidos e a instância avaliada teria dados indisponíveis de resultado. Além disso, como a execução dos algoritmos pode demorar muitas horas para finalizar, foi definido um limite máximo de 30 minutos, no qual, caso alguma instância precisasse de mais tempo do que isso, os resultados para ela também seriam indisponíveis.

Buscando otimizar o processo de teste, foi definida uma última restrição além das apresentadas acima. Nela, era assumido que, se uma instância do problema do caixeiro

viajante possui "N" vértices e levou um tempo "T_N" para executar, como todas as instâncias são construídas da mesma maneira, se uma delas possuir "$N + 1$" vértices, salvo comportamentos inesperados do hardware, ela provavelmente levará um tempo $T_{N+1} \geq T_N$. Dessa forma, caso o tempo limite seja excedido para a computação do resultado de alguma instância, todas as instâncias maiores ou iguais a ela não serão executadas por serem potencialmente mais lentas.

As instâncias do problema do caixeiro viajante utilizadas para os testes foram obtidas no site da Universidade Pública Alemã de *Heidelberg*, devidamente referenciada ao fim deste artigo. Elas são garantidas pela universidade como instâncias euclidianas, que satisfazem as restrições impostas pelos algoritmos aproximativos. Essas instâncias foram fornecidas aos algoritmos em ordem crescente da quantidade de vértices, tendo a ordenação feita em código.

Análise comparativa

A menor instância do problema do caixeiro viajante disponível para testes possuía 51 vértices, levando a um máximo de 2^{51} (mais de 2,25 quatrilhões) possibilidades. Ao fornecer essa instância para o algoritmo exato, implementado com a técnica *Branch and Bound*, foram excedidos os 30 minutos de limite e nenhum resultado foi retornado, ou seja, esse algoritmo deixa de ser eficiente ainda para instâncias ainda pequenas do problema.

Ao realizar a bateria de testes para o algoritmo aproximativo *Twice Around The Tree*, foi possível executar em tempo hábil 73 das 78 instâncias disponíveis, compreendendo problemas com até 5.934 vértices. O crescimento do tempo apresentou comportamento exponencial, com baixa variação para instâncias com até 1.000 vértices. O tempo total para executar todos esses testes foi de aproximadamente 5 minutos (294,5 segundos), com o tempo individual médio de 4 segundos, o que é relativamente baixo.

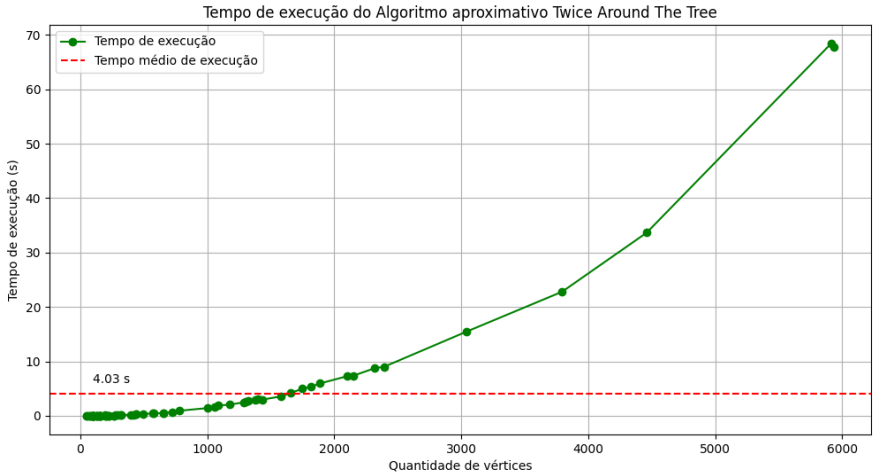

Figura 1: Gráfico de linha mostrando a relação tempo v.s. tamanho para o algoritmo *Twice Around The Tree*.

Apesar do tempo de execução deste algoritmo ser relativamente baixo, o consumo de memória realizou saltos muito grandes de uma instância a outra para tamanhos superiores a 3.000 vértices, necessitando de mais de 16 GB ao todo (16.805,926 MB). As instâncias necessitam individualmente de, em média, 230 MB para a execução completa, um valor que é muito possivelmente gerado pelos outliers que ocorrem acima do limite de 3.000 vértices anteriormente mencionado.

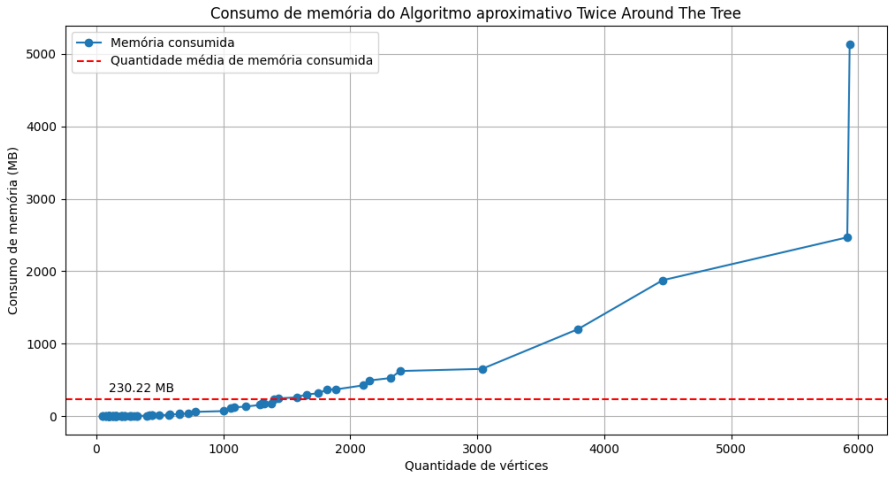

Figura 2: Gráfico de linha mostrando a relação memória v.s. tamanho para o algoritmo *Twice Around The Tree*.

A instância denominada "*rl11849*", que possui 11.849 vértices passou a consumir mais de 16 GB de memória sozinha, levando o consumo de memória acima dos 95% previamente

definidos para os experimentos, o que fez com que a execução fosse interrompida manualmente.

Já no algoritmo aproximativo de *Christofides*, os resultados obtidos da bateria de testes para tempo foram muito maiores em relação aos anteriores, levando mais de uma hora (3.992 segundos) para finalizar a execução dos testes. A média de tempo para cada uma das instâncias foi de aproximadamente um minuto (57 segundos), apresentando oscilações muito fortes no tempo a partir de 1.000 vértices.

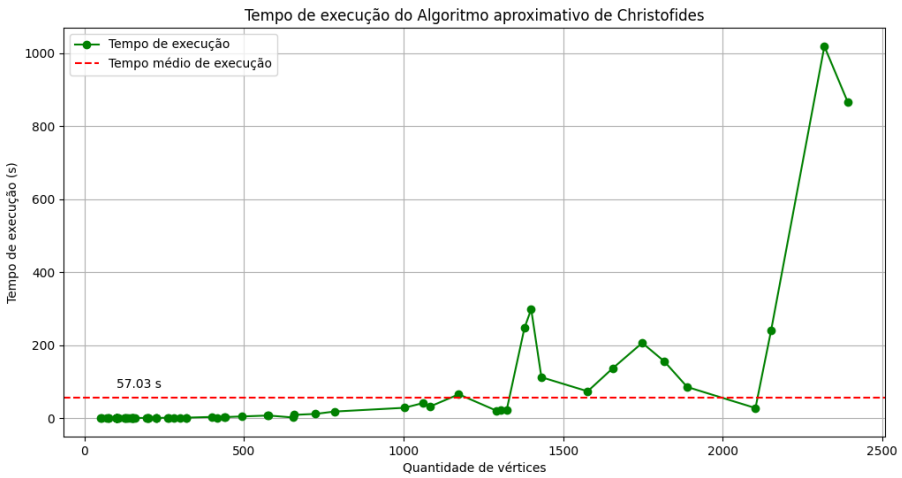

Figura 3: Gráfico de linha mostrando a relação tempo v.s. tamanho para o algoritmo de *Christofides*.

A instância denominada "*pcb3038*", que possui 3.038 vértices, levou mais de 30 minutos para executar, ocasionando a interrupção automática da bateria de testes. A memória consumida total foi mais de 8 GB (8.373,151 MB), com memória média por instância de aproximadamente 120 MB (119,62 MB), apresentando fortes oscilações na memória consumida a partir de 1.500 vértices.

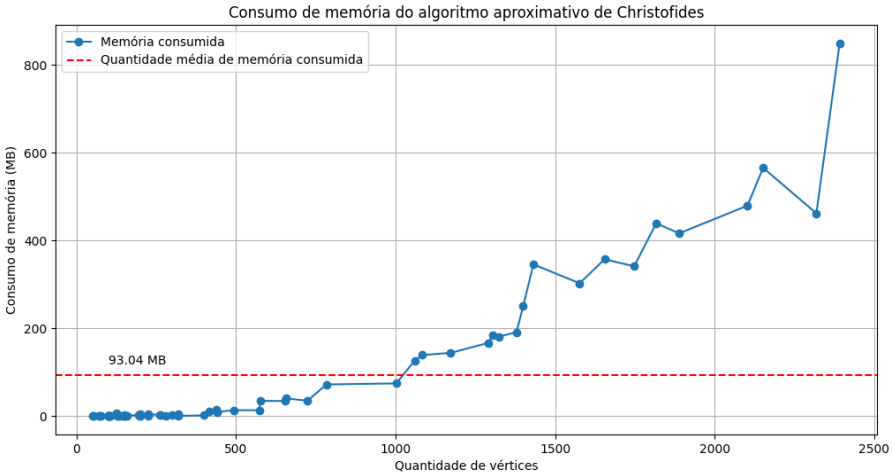

Figura 4: Gráfico de linha mostrando a relação memória v.s. tamanho para o algoritmo de *Christofides*.

Apesar dos valores apresentados serem próximos da metade dos valores obtidos para o algoritmo *Twice Around The Tree*, essa diferença é muito influenciada pelo fato de não terem sido registrados os resultados para instâncias acima de 3.000 vértices. O gráfico abaixo mostra a proximidade entre os valores para instâncias de tamanho igual.

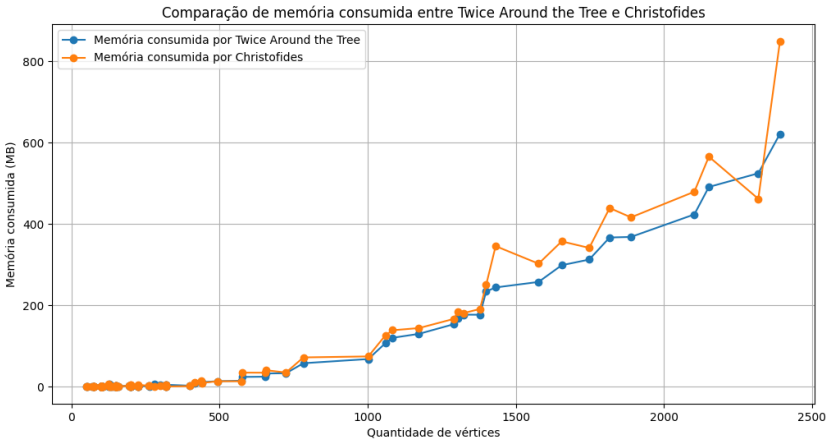

Figura 5: Gráfico de linha comparando a memória consumida pelo algoritmo *Twice Around The Tree* e *Christofides*.

Findas as comparações quanto a tempo e espaço, deve-se também comparar a proximidade dos resultados obtidos pelos algoritmos em relação ao resultado ótimo conhecido. O algoritmo Twice Around The Tree teve fator de aproximação máximo igual a 1,57, o que significa que as soluções apresentadas por ele foram, no máximo, 1,57 vezes pior do que a solução ótima conhecida. Já para o algoritmo de Christofides, o fator máximo foi de 1,6, contrariando a definição inicial do fator de aproximação máximo ser de 1,5. Apesar disso,

tanto o fator mínimo de aproximação do Christofides quanto o médio obtiveram magnitude menor do que o Twice Around The Tree.

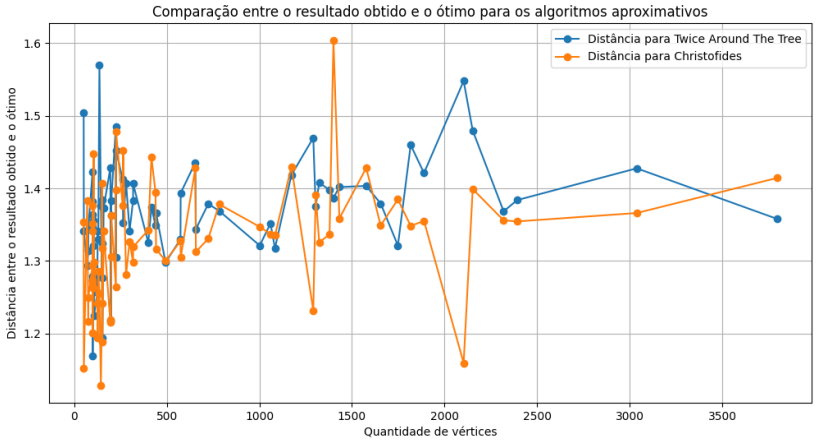

Figura 6: Gráfico de linha comparando a distância entre a solução dos algoritmos *Twice Around The Tree* e *Christofides* em relação a ótima conhecida.

Conclusão

Ao longo do presente artigo, foram feitas diversas definições, tanto sobre o problema do Caixeiro Viajante, quanto às técnicas de solução relativas a ele. Com a execução dos experimentos sobre os algoritmos implementados, foi possível obter informações que podem guiar a utilização de algum deles em problemas do mundo real.

De forma geral, o algoritmo de *Christofides* apresentou melhores resultados, com um fator de aproximação médio em relação à solução ótima de 1,33 contra 1,36 do algoritmo *Twice Around The Tree*. Em relação ao quesito de memória, ambos possuíram consumos bastante parecidos para instâncias de mesmo tamanho, sendo relativamente equivalentes. A diferença maior está no quesito de tempo, já que o algoritmo *Twice Around The Tree* apresentou tempo médio de execução de 4,03 segundos, o que é mais de 14 vezes mais rápido do que o de *Christofides*, com média igual a 57,03.

Levando em consideração esses fatores e o fato de que o algoritmo exato implementado com a técnica *Branch and Bound* não foi capaz de executar para nenhuma instância, é possível afirmar que o algoritmo Twice Around The Tree é mais indicado para situações em que a precisão não é tão importante. Já que ele é capaz de entregar uma solução relativamente próxima da ótima em tempos muito menores do que o de *Christofides*, que é preferível em situações com necessidade de precisão um pouco mais altas.

Referências

HAGBERG, A. et al. **Exploring Network Structure, Dynamics, and Function using NetworkX**. [s.l: s.n.]. Disponível em: <https://conference.scipy.org/proceedings/SciPy2008/paper_2/full_text.pdf>.

JÜNGER, M.; REINELT, G.; RINALDI, G. Chapter 4 The traveling salesman problem. **Handbooks in Operations Research and Management Science**, v. 7, p. 225–330, 1 jan. 1995.

REDDY, P.; DERDAR, A. **Minimal Spanning Tree**. [s.l: s.n.]. Disponível em: <https://www.ijert.org/research/minimal-spanning-tree-IJERTV6IS030189.pdf>. Acesso em: 10 dez. 2023.

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG. **TSPLIB**. Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html>. Acesso em 5 dez. 2023.

WEISSTEIN, E. W. **Perfect Matching**. Disponível em: <https://mathworld.wolfram.com/PerfectMatching.html>. Acesso em: 10 dez. 2023.