Análise de algoritmos para a computação de rotas para o Problema do Caixeiro Viajante métrico

Vinicius Silva Gomes[1]

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

{vinicius.gomes}@dcc.ufmg.br

Abstract. *This article describes and analyzes the results of an experimentation process based on three different implementations of algorithms that compute routes for the metric Travelling Salesman Problem. The first algorithm is based on the Branch-and-Bound strategy for exact solutions and the other two are approximate algorithms: the Twice Around the Tree algorithm and the Christofides algorithm.*

Resumo. *Este artigo descreve e analisa os resultados de um processo de experimentação baseado em três implementações diferentes de algoritmos que computam rotas para o Problema do Caixeiro Viajante métrico. O primeiro algoritmo é baseado na estratégia de Branch-and-Bound para soluções exatas e os outros dois são algoritmos aproximados: o algoritmo Twice Around the Tree e o algoritmo de Christofides.*

1. Introdução

Problemas relacionados a geração de rotas que satisfaçam algumas condições são relativamente comuns no dia-a-dia. Distribuidoras, por exemplo, precisam estabelecer uma rota que conecte todos os pontos desejados e, ao final, retorne para o ponto de início. Nesse meio, é desejável que o caminho percorrido seja o menor possível, uma vez que isso indica que a distribuição será mais rápida e o gasto com combustível será menor.

Esse problema em específico, conhecido na área da computação como o Problema do Caixeiro Viajante (ou *Travelling Salesman Problem/TSP*, em inglês), é bastante famoso e estudado, visto que se trata de um problema com as mais diversas aplicações. No entanto, a computação dessas rotas ainda é um problema computacionalmente "difícil".

Apesar de poder ser enunciado de diversas formas, de modo geral, mas sem perder a devida especificidade, o Problema do Caixeiro Viajante pode ser descrito como problema de um vendedor que precisa sair de uma cidade e passar por todas as outras na área de interesse [Wikipedia 2022b]. Ele pode passar por cada cidade apenas uma vez e, ao final, deve voltar para a cidade da qual ele começou. Queremos que essas restrições sejam cumpridas e que, além disso, o caminho escolhido seja "satisfatoriamente pequeno".

Por se tratar de um problema NP-Difícil, i.e., problema com solução polinomial numa máquina de Turing não-determinística, o problema do Caixeiro Viajante é bastante complicado de ser computado em uma máquina determinística, uma vez que não conhecemos maneiras de replicar escolhas não-determinísticas e, por isso, somos obrigados a

fazer uma enumeração completa das escolhas do problema para que ele possa ser computado [Levitin 2012]. Isso faz com que soluções ingênuas (e até mesmo as mais elaboradas) possuam complexidade de tempo exponencial e, portanto, para algumas instâncias, demorariam mais tempo do que é desejado/disponível.

Para tanto, estratégias foram desenvolvidas de modo que o problema possa ser computado sem que todas as escolhas sejam analisadas. Em outras palavras, algumas abordagens podam a árvore de escolhas e fazem com que o problema, na prática, seja computado mais rapidamente. Além disso, existem casos onde uma solução exata não é necessária, apenas uma aproximação "boa o suficiente" basta. Nesses casos, soluções aproximadas podem ser utilizadas para resolver o problema e, ainda assim, apresentarão um resultado razoável.

Dessa forma, este artigo visa apresentar três algoritmos diferentes para computar rotas para o Problema do Caixeiro Viajante. A primeira delas é baseada na estratégia de *Branch-and-Bound* e retorna uma solução exata para a instância enquanto as outras duas são algoritmos aproximativos, sendo o primeiro deles o algoritmo *Twice Around the Tree* e o outro o algoritmo de *Christofides*. Além disso, todos esses algoritmos serão submetidos a experimentos para que possam ser comparados, especialmente quanto ao uso de tempo e espaço em relação a qualidade da solução.

2. Modelagem Computacional

Como sugerido pelo própria definição, o Problema do Caixeiro Viajante trata de um problema que pode ser muito bem modelado por um grafo. De forma simplista, um grafo $G(V, E)$ pode ser descrito como uma estrutura matemática composta por um conjunto não vazio de vértices V e arestas E, onde as arestas nada mais são que duplas (u, v) onde $u, v \in V$. Essas arestas podem possuir pesos e podem ser usadas para estabelecer relações arbitrárias entre os nós que as delimitam [Wikipedia 2022a].

Nesse sentido, o *TSP* pode ser mapeado para um grafo, de modo que as cidades que devem ser percorridas serão os nós no grafo que representa a instância e as arestas serão os caminhos que ligam cada cidade, sendo os pesos de cada aresta a distância entre as cidades que delimitam esse caminho.

Além disso, por definição do problema, temos que existem caminhos que liguem todas as cidades entre si. Assim, o grafo que representa uma instância é um grafo completo, unidirecional e ponderado. Dessa forma, é possível representar, de maneira semântica e formal, cada configuração do problema.

3. Restrições do Problema

Para restringir o escopo do *TSP* e, acima de tudo, favorecer a computação dos algoritmos aproximativos, estaremos considerando instâncias do *TSP* que satisfazem alguns requisitos geométricos com relação as distâncias entre as cidades. O *TSP* sob esse conjunto de restrições é conhecido como Problema do Caixeiro Viajante Métrico.

Essas restrições se aplicam ao peso das arestas do grafo que representa a instância. Estaremos considerando que esses pesos, ou seja, que a função de custo do grafo seja uma métrica. Por ser uma métrica, ela deverá satisfazer as seguintes condições (onde u, v e w são vértices do grafo):

- $c(u, v) \geq 0$ (todas as distâncias são positivas);
- $c(u, v) = 0$ se, e somente se, $u = v$ (distância só é 0 se é de um nó para ele mesmo);
- $c(u, v) = c(v, u)$ (propriedade de simetria);
- $c(u, v) \leq c(u, w) + c(w, v)$ (também conhecida como Desigualdade Triangular).

Com a instância do problema satisfazendo essas condições, teremos a garantia de que os algoritmos aproximados funcionarão de acordo com o seu fator de aproximação [Vimieiro 2022], uma vez que o *TSP*, no caso geral, não possui fator de aproximação constante a menor que $P = NP$. Além disso, como muitas instâncias, na prática, satisfazem essas condições (aplicações em cenários reais normalmente envolvem métricas como funções de custo), não há prejuízo ao considerar apenas instâncias com essas características.

Para enriquecer o processo de experimentação e testar diferentes situações, as cidades serão considerados como pontos no plano e duas métricas diferentes de distância entre elas serão consideradas: a distância euclidiana (em linha reta) e a distância Manhattan (em forma de *grid*). Na seção 5 (experimentação) esses pontos serão melhor detalhados.

4. Implementações

Três implementações diferentes de algoritmos que computam possíveis rotas para o problema serão analisadas. Como dito anteriormente, a primeira é baseada na estratégia de *Branch-and-Bound* e retorna uma solução exata para a instância, ou seja, retorna a menor rota que cumpre as especificações que foram apresentadas.

As outras implementações, por sua vez, são algoritmos aproximados bastante conhecidos: o algoritmo *Twice Around the Tree* e o algoritmo de *Christofides*. Essas duas implementações possuem abordagens diferentes (apesar de semelhantes em certos pontos) e fatores aproximativos diferentes. Eventualmente, todos desses algoritmos podem gerar a solução ótima para a instância, no entanto, não há garantia que isso sempre irá acontecer.

Quanto ao programa, especificamente, todos os algoritmos foram implementados utilizando a linguagem de programação **Python**, se apoiando, ainda, nas bibliotecas *NetworkX* e *NumPy*, que são bastante conhecidas na linguagem, para a construção e manipulação dos grafos, vetores e matrizes ao longo do código. Esses pacotes garantem que a implementação das funções intermediárias utilizadas pelos algoritmos seja a mais eficiente possível.

A seguir, cada um dos algoritmos será melhor destrinchado e o pseudocódigo para cada um deles também será apresentado.

4.1. Solução baseada em *Branch-and-Bound*

Se inspirando bastante em outro paradigma de design de algoritmos, a estratégia de *branch-and-bound* atua explorando os nós na árvore de soluções candidatas.

A grande diferença em relação ao paradigma de *backtracking* é que, além de fazer as podas baseadas no escopo do problema (soluções parciais que ferem alguma condição do problema), soluções baseadas em *branch-and-bound*, que normalmente são aplicadas

em problemas de otimização, possuem podas baseadas em uma estimativa, o que permite que a árvore de soluções candidatas seja explorada de maneira bem mais rápida.

Nesse sentido, uma solução parcial para esse problema é um conjunto de nós que representam as cidades percorridas na solução, além da ordem que eles aparecem na solução representar a ordem que eles devem ser percorridos (a distância total é calculada considerando o peso da aresta que liga os vértices i e $i + 1$, $i + 1$ e $i + 2$ e assim por diante). Como existem arestas entre todos os vértices, essa condição sempre é satisfeita.

São consideradas soluções parciais inválidas aquelas soluções que não contemplem todos os vértices presentes no grafo, que não terminem no mesmo vértice que iniciou o trajeto e soluções que contemplem algum vértice mais de uma vez no trajeto. Muitas dessas soluções já serão desconsideradas (ou não aparecerão na árvore de soluções) pois o algoritmo já será implementado de modo a evitar esses nós infrutíferos.

O algoritmo, então, irá explorar a árvore de soluções candidatas a partir de uma estratégia conhecida por*best-first-search*. A ideia é aplicar uma busca em largura (*breadth-first-search* ou BFS) considerando uma fila de prioridades. Assim, soluções parciais com estimativas melhores são priorizadas na hora da exploração. O objetivo é fazer com que o espaço de busca seja percorrido de maneira mais ágil, com podas mais otimizadas e eficientes [Levitin 2012].

Para tanto, é necessário uma função de estimativa que seja realista de acordo com a instância e ainda possa ser computada de maneira relativamente eficiente para cada nó da árvore de soluções.

A função utilizada se baseia na ideia de que o caixeiro vai entrar numa cidade e depois irá sair dela, por isso consideramos, para cada vértice, as duas menores arestas incidentes a ele. Ao final, o peso dessas arestas escolhidas é somado e dividido por 2 (cada aresta seria contada, potencialmente, duas vezes).

A medida que as soluções parciais crescem, as arestas existentes entre os nós da solução vão sendo fixadas na solução, para assegurar que a métrica continue realista a medida que mais cidades sejam visitadas.

O algoritmo que implementa essa função se baseia na matriz de adjacência com as distâncias entre os vértices e na solução parcial. Inicialmente, as arestas fixadas pela solução parcial são inseridas em uma variável auxiliar que armazena todas as arestas escolhidas. Em seguida, para aqueles nós que ainda não tenham 2 arestas escolhidas, a matriz de adjacência é percorrida na linha referente a esse vértice e as menores arestas disponíveis são escolhidas até que ele possua 2 arestas designadas. O cálculo permanece o mesmo: todas as arestas escolhidas são somadas e o resultado é dividido por 2.

O algoritmo 1 e a figura 1 mostram o pseudocódigo da função que calcula a estimativa utilizada e o pseudocódigo do algoritmo desenvolvido para resolver o *TSP* a partir de uma solução baseada em *branch-and-bound*.

O algoritmo para a função de *bound* insere infinito nas células da matriz de adjacências que forem escolhidas para a solução para impedir que elas sejam escolhidas novamente em outra iteração.

Algorithm 1 Função de *bound* do algoritmo

Input: A matriz de adjacências W, a solução parcial S e o número de vértices N
Output: Estimativa para o tamanho do caminho, considerando a solução parcial

```
total = 0
selected ← ∅
for i = 0 to |S| − 1 do
    start = S[i]
    end = S[i + 1]
    selected[start] ← W[start][end]
    selected[end] ← W[end][start]
    W[start][end] = ∞
end for
for each key ∈ selected do
    if |key| ≠ 2 then
        while |key| ≤ 2 do
            idx ← MINIMAL-INDEX(W[key])
            selected[key] ← W[key][idx]
            W[key][idx] = ∞
        end while
    end if
end for
for each (u, v) ∈ E in selected do
    total += WEIGHT(u, v)
end for
return ceil(total/2)
```

4.2. Algoritmo *Twice Around the Tree*

O algoritmo *Twice Around the Tree* é um algoritmo 2-aproximado, ou seja, a solução retornada por ele pode ser até 100% pior que a solução ótima. Esse algoritmo é inspirado na ideia de que a árvore geradora mínima do grafo é uma aproximação boa de um caminho mais curto no grafo.

Sendo assim, a ideia principal do algoritmo é computar a árvore geradora mínima da instância e aproveitar as propriedades do fato da função de custo do grafo ser uma métrica para reduzir as distâncias percorridas e, assim, obter uma boa aproximação para o circuito a ser percorrido.

Dessa forma, o primeiro passo do algoritmo é computar a árvore geradora mínima. Em seguida, essa AGM será percorrida por um caminhamento pré-ordem e os nós visitados e a ordem em que eles forem visitados será armazenados. Ao final, teremos um caminho com alguns nós repetidos que saem do vértice fonte e retornam a ele no final.

No entanto, nesse caminho, alguns nós são repetidos e, pela Desigualdade Triangular, sabemos que que $c(u, v) \leq c(u, w) + c(w, v)$. Logo, esses vértices repetidos podem ser removidos do caminho em prol de manter a propriedade que cada nó só é visitado uma única vez e, ainda assim, reduzir a distância percorrida graças a desigualdade triangular [Goodrich and Tamassia 2006].

```
Procedure bnb-tsp(A,n)
  root = (bound([0]), 0, 0, [0]);
  queue = heap([root]);
  best = ∞;
  sol = ∅;
  while queue ≠ ∅ do
    node = queue.pop();
    if node.level > n then
      if best > node.cost then best = node.cost; sol = node.s;
    end
    else if node.bound < best then
      if node.level < n then
        for k = 1 to n − 1 do
          if k ∉ node.s and A[node.s[−1]][k] ≠ ∞ and
           bound(node.s ∪ {k}) < best then
            queue.push((bound(node.s ∪ {k}, node.level +
              1, node.cost + A[node.s[−1]][k], node.s ∪ {k}))
          end
        end
      end
      else if A[node.s[−1]][0] ≠ ∞ and bound(node.s ∪ {0}) < best
       and ∀i ∈ node.s then
        queue.push((bound(node.s ∪ {0}, node.level + 1, node.cost +
          A[node.s[−1]][0], node.s ∪ {0}))
      end
    end
  end
```

Figura 1. Pseudocódigo da solução baseada em branch-and-bound para o *TSP* métrico. Imagem tirada de [Vimieiro 2022].

Em essência, a ideia é que, ao realizar o caminhamento pré-ordem, estamos na verdade percorrendo um circuito euleriano (circuito que passa por todas as arestas do grafo e começa e termina no mesmo vértice) no multígrafo com as arestas da AGM duplicadas. O que, em efeito, acaba sendo o mesmo que realizar uma busca em profundidade (*depth-first-search, ou DFS*) na AGM e, ao final, acrescentar o nó fonte ao caminho.

Por fim, será obtido um circuito hamiltoniano (circuito que passa por todos os vértices do grafo e começa e termina no mesmo vértice) com a ordem em que os nós devem ser visitados para que as restrições do *TSP* sejam satisfeitas e a distância total percorrida seja otimizada. Como último passa, basta apenas consultar quais arestas estão presentes no caminho e somar o peso delas, para que seja possível obter a distância total percorrida.

O pseudocódigo 2 evidencia o processo de obtenção do caminho aproximado e da distância total percorrida nesse caminho.

4.3. Algoritmo de *Christofides*

Semelhante ao algoritmo *Twice Around the Tree*, o algoritmo de *Christofides* é, na verdade, um aperfeiçoamento da ideia executada pelo *Twice Around the Tree*.

O algoritmo de *Christofides* inicia a computação calculando qual seria a árvore geradora mínima para o grafo da instância, assim como o *Twice Around the Tree*, mas, ao

Algorithm 2 Algoritmo *Twice Around the Tree*

Input: O grafo G que define a instância do problema
Output: Os nós que o caminho irá percorrer e a distância desse caminho

```
best = 0
T ← MINIMUM-SPANNING-TREE(G)
path ← PREORDER-DFS(T)
for each (u, v) ∈ E in path do          ▷ Percorre as arestas do caminho
    best += WEIGHT(u, v)                ▷ Acumula os pesos das arestas
end for
return path, best
```

invés de executar um caminho euleriano num multígrafo resultante das arestas da AGM duplicadas, o algoritmo de *Christofides* torna essa "duplicação"de arestas mais eficiente, o que resulta no fator de aproximação de 1.5, 50% melhor que o do *Twice Around the Tree*.

A grande diferença, portanto, está na política de escolha das arestas duplicadas. O que o algoritmo de *Christofides* faz é encontrar um *matching* perfeito de peso mínimo no grafo induzido pelos nós de grau ímpar na AGM (conjunto de arestas disjuntas no grafo composto pelos nós com grau ímpar, de modo que a escolha das arestas minimize a soma dos pesos).

Na prática, essa escolha faz com que as arestas resultantes do *matching* sejam uma escolha mais inteligente e menos custosa do que apenas duplicar todas as arestas da AGM. Por fim, essas arestas obtidas serão inseridas na AGM, formando um multígrafo a partir da AGM e das arestas do *matching*.

O final do algoritmo permanece igual ao *Twice Around the Tree*, no qual um circuito euleriano é computado sobre esse multígrafo (DFS executada sobre ele), os nós repetidos são removidos graças a desigualdade triangular e o nó fonte é inserido ao final do caminho, tornando ele um circuito hamiltoniano.

O pseudocódigo 3 evidencia o processo de obtenção do caminho aproximado a partir do algoritmo de *Christofides* e da distância total percorrida nesse caminho.

Algorithm 3 Algoritmo de *Christofides*

Input: O grafo G que define a instância do problema
Output: Os nós que o caminho irá percorrer e a distância desse caminho

```
best = 0
T ← MINIMUM-SPANNING-TREE(G)
O ← ODD-DEGREE-SUBGRAPH(T)
N ← MINIMUM-WEIGHT-MATCHING(O)
M ← MULTIGRAPH(T, N)
path ← PREORDER-DFS(M)
for each (u, v) ∈ E in path do          ▷ Percorre as arestas do caminho
    best += WEIGHT(u, v)                ▷ Acumula os pesos das arestas
end for
return path, best
```

5. Processo de Experimentação

O processo de experimentação será orientado por um gerador de instâncias para o problema do *TSP* métrico. Esse gerador irá criar instâncias com 2^k nós $\forall k \in \mathbb{N}$ onde $4 \leq k \leq 10$.

Os nós do grafo foram gerados como pontos de coordenadas (x, y) num plano 2D e, a medida que um novo ponto era gerado, a distância euclidiana e a distância Manhattan entre ele e os pontos previamente computados era calculada e inserida nas respectivas matrizes.

Ao final, duas matrizes simétricas diferentes serão computadas, sendo cada elemento $d_{i,j}$ o peso da aresta (i, j) (que conecta os vértices i e j). Uma das matrizes é composta pela distância euclidiana entre as coordenadas dos vértices e a outra pela distância Manhattan entre as coordenadas dos vértices. Estas duas matrizes foram geradas para cada k diferente e armazenadas em disco. Como pode ser notado, cada matriz é, na verdade, uma matriz de adjacência que representam a instância criada pelo gerador.

Após isso, cada um dos três algoritmos será executado sobre as entradas produzidas, considerando os seguintes parâmetros: tamanho da entrada e tipo de métrica de distância. Dessa forma, ao final, serão obtidas 42 amostras diferentes com os resultados para cada execução. Será interessante armazenar, para cada algoritmo, o tamanho da instância, o tipo de métrica de distância considerada, o caminho retornado, a distância total desse caminho, o tempo de execução do algoritmo (em milissegundos) e o espaço utilizado pelo algoritmo (pico de espaço utilizado em bytes).

Para coletar as informações de tempo e espaço, foram utilizados os pacotes *timeit* e *tracemalloc*. O tempo de execução de cada algoritmo foi medido individualmente pra cada entrada e salvo em disco.

Essas informações serão coletadas pelo programa e armazenadas, para que depois possam ser importadas e interpretadas. O tempo de execução máximo para cada abordagem é de 30 minutos, após isso o programa será interrompido, os resultados serão preenchidos como NA (*not available*, ou indisponível) e a próxima instância será processada.

Para interpretação dos resultados, os dados coletados passarão por um processo de análise exploratória, utilizando como base a biblioteca *Pandas*, para que eles possam ser interpretados e mais informações sobre eles possam ser abstraídas. Estamos interessados em tentar determinar, de maneira geral, uma métrica de qualidade de cada algoritmo; relações entre o tipo de métrica/tamanho da entrada e o desempenho do algoritmo; etc.

6. Resultados Obtidos

Após o processo de experimentação, os dados coletados foram analisados e algumas informações puderam ser abstraídas. Primeiramente, a solução baseada em *branch-and-bound* só concluiu sua computação em menos de 30 minutos para a instância com 2^4 vértices. Para as outras instâncias, o algoritmo não foi capaz de terminar sua execução em tempo hábil.

Além disso, para essa instância onde o algoritmo de *branch-and-bound* terminou sua computação, a diferença entre as distâncias calculadas pelos algoritmos aproximados em relação a solução exata foi relativamente pequena (no máximo 392 unidades de

distância para o *Twice Around the Tree* em relação a solução exata, considerando a métrica de distância Manhattan).

Por outro lado, a diferença de tempo de computação é bastante pertinente. Enquanto o *Twice Around the Tree* e o algoritmo de *Christofides* demoraram entre 3 e 11 milissegundos para computar o caminho e a distância, a solução exata demorou entre 3 e 5 minutos. Além do tempo, o espaço gasto também é importante ser levado em conta, uma vez que para uma instância com 2^4 vértices, a solução exata gastou aproximadamente 130 *megabytes* de memória. Essa quantidade de memória só chegou perto de ser gasta nas últimas instâncias presentes na experimentação, com 2^9 e 2^{10} vértices.

Para o restante das entradas, a análise segue apenas para os dois algoritmos aproximativos. Para facilitar a comparação entre eles, as figuras 2 e 3 mostram a comparação do uso de tempo e espaço, respectivamente, em relação a cada tamanho de entrada. Os gráficos foram produzidos considerando os resultados para a distância euclidiana, mas as curvas se mantiveram bastante semelhantes para a distância Manhattan também.

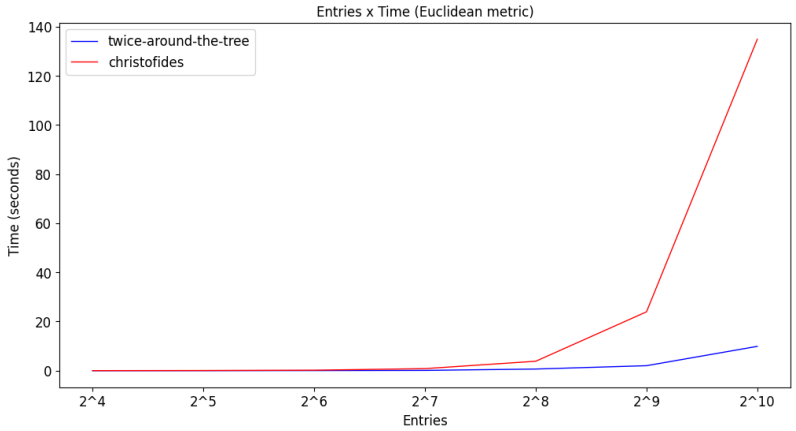

Figura 2. Gráfico de linha com a comparação do uso de tempo de cada algoritmo aproximado em relação ao tamanho da instância.

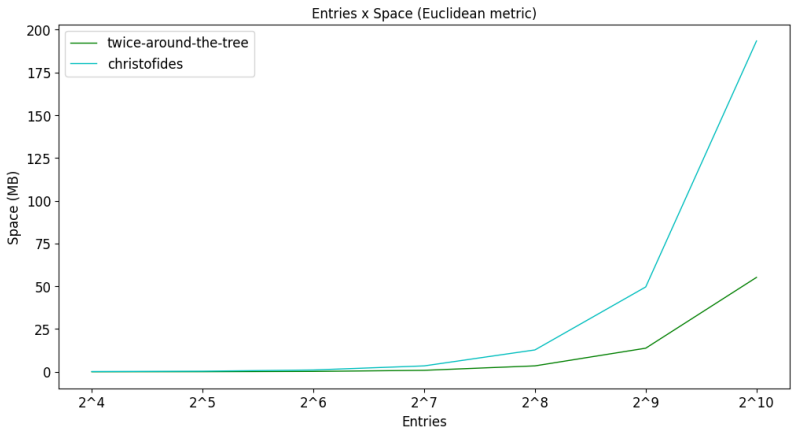

Figura 3. Gráfico de linha com a comparação do uso de espaço de cada algoritmo aproximado em relação ao tamanho da instância.

Como pode ser observado, a medida que o tamanho da instância aumenta, o gasto de tempo e de memória para computar o caminho e o custo dele aumenta com uma curva

bastante íngreme. Isso indica que, para instâncias maiores ainda, o comportamento deve se manter e a computação ficaria cada vez mais cara.

No entanto, apesar da diferença parecer bastante acentuada, o tempo máximo gasto para computar o caminho e a distância total desse caminho para o *TSP* foi de pouco mais de 2 minutos, o que é bem pouco quando se compara que a solução exata sequer pôde ser computada para instâncias com mais de 2^4 nós.

Por fim, foi feita uma análise em relação ao tamanho do caminho retornado para cada algoritmo em relação ao tamanho da instância. A figura 4 mostra que a diferença é, na verdade, bastante sutil e dificilmente percebida, dado a magnitude dos tamanhos retornados. Para facilitar a visualização, a figura 5 mostra a diferença de distância retornada entre o algoritmo *Twice Around the Tree* e o algoritmo de *Christofides*.

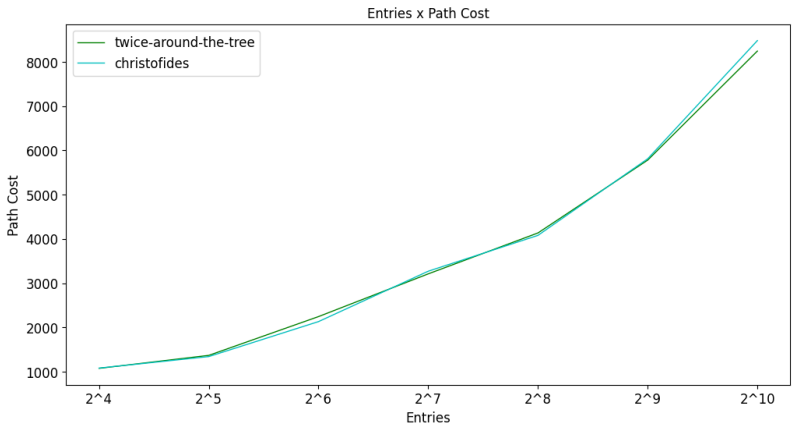

Figura 4. Gráfico de linha com a comparação do tamanho, em unidades de distância, do caminho retornado por cada algoritmo aproximativo.

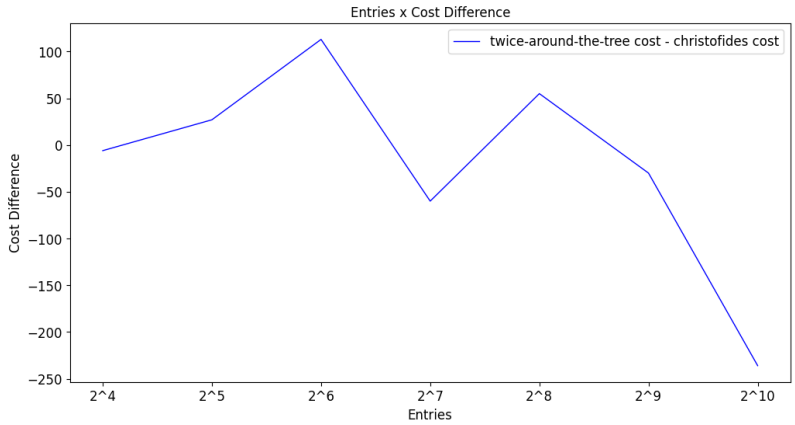

Figura 5. Gráfico de linha com a diferença entre o tamanho dos caminhos retornados pelos algoritmos aproximados.

Como pode ser observado, a distância, em termos de magnitude, é bem pequena e. Curiosamente, para a maior instância presente no experimento, o caminho retornado pelo *Twice Around the Tree*, que possui um fator de aproximação maior e, portanto, espera-se que possua um desempenho ligeiramente pior que o algoritmo de *Christofides*, foi menor, sendo a maior diferença de distância observada.

7. Conclusão

Após o processo de experimentação e a análise dos resultados obtidos, algumas conclusões puderam ser tiradas. Por se tratar de um problema difícil computacionalmente, foi notável a demora para computar as respostas para ele, mesmo que aproximadas. Isso mostra que, em geral, computar a solução exata quase nunca vai ser algo viável.

Com exceção de situações onde a solução exata é estritamente necessária, o número de vértices no grafo da instância é pequeno ou a solução pode ser previamente computada e aproveitada; a solução baseada em *branch-and-bound* dificilmente poderá ser aplicada.

Além disso, a diferença entre as soluções aproximadas só passou a ser mais perceptível com instâncias maiores, onde a diferença no tamanho do caminho, o tempo gasto na computação e a memória utilizada começaram a ser mais pertinentes.

Assim, temos que para instâncias pequenas, o desempenho dos algoritmos aproximados é bastante semelhante, sendo cada um deles uma boa escolha para computar o caminho e a melhor distância possível, de acordo com o fator de aproximação.

No entanto, para grafos maiores, a relação entre tempo/memória e qualidade da solução deve ser melhor observada. O algoritmo *Twice Around the Tree*, de modo geral, gasta bem menos recursos que o algoritmos de *Christofides*, entregando bons resultados em um tempo bem menor e utilizando bem menos espaço, como pode ser observado nas figuras da seção anterior. Dessa forma, em situações onde a velocidade com que esse caminho é computado é o mais importante, o *Twice Around the Tree* pode ser a melhor escolha. Em contrapartida, se a garantia de que a solução é a melhor possível é o mais importante, o algoritmo de *Christofides* pode ser a escolha mais segura.

Por fim, foi possível concluir que a escolha do algoritmo, seja aquele que retorna a solução exata ou àqueles que são aproximados, depende bastante do tamanho da instância e das necessidades da aplicação. Cada um deles possui suas vantagens e desvantagens e a percepção delas em relação ao cenário em que o algoritmo vai ser aplicado é fundamental para que o melhor resultado seja obtido.

Referências

Goodrich, M. T. and Tamassia, R. (2006). *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 6th edition.

Levitin, A. (2012). *Introduction to The Design & Analysis of Algorithms*. Pearson Education, 3th edition.

Vimieiro, R. (2022). *Conjunto de slides da disciplina Algoritmos II*. Departamento de Ciência da Computação – Universidade Federal de Minas Gerais.

Wikipedia (2022a). Teoria dos grafos. `https://pt.wikipedia.org/wiki/Teoria_dos_grafos`. Acesso em: 08 de dez. de 2022.

Wikipedia (2022b). Travelling salesman problem. `https://en.wikipedia.org/wiki/Travelling_salesman_problem`. Acesso em: 08 de dez. de 2022.