Trabalho Prático 2 – Soluções para problemas difíceis

Thiago de Assis Lima[1]

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

thiagoassis99@ufmg.br

Abstract. *This article analyzes different solutions to the traveling salesman problem, evaluating exact solutions, with the branch-and-bound algorithm, and approximate ones, with the twice-around-the-tree and christofides algorithms. In the case of approximations, the Christofides algorithm presents itself as a superior choice when results closer to the optimum are expected, but as a result it takes much more processing time. Therefore, the Twice around the tree algorithm can be a suitable solution when time is an important variable.*

Resumo. *Este artigo analisa diferentes soluções para o problema do caxeiro viajante, avaliando soluções exatas, com o algoritmo de branch-and-bound, e aproximativas, com os algoritmos twice-around-the-tree e christofides. No caso dos aproximativos, o algoritmo de Christofides se apresenta como uma escolha superior quando espera-se resultados mais próximos do ótimo, porém com isso ele gasta bem mais tempo de processamento. Portanto, o algoritmo de Twice around the tree pode ser uma solução adequada quando o tempo é uma variável importante.*

1. Introdução

Este artigo tem como objetivo analisar diferentes formas de se resolver o problema do caixeiro viajante com distâncias euclidianas. Para isso, foi usado algoritmos conhecidos na literatura, como o Branch and Bound para soluções exatas e, para os aproximativos, os algoritmos de Christofides e Twice around the tree.

Será feita uma comparação para avaliar qual a melhor estratégia para rodar o Branch and Bound em dois aspectos: tempo e espaço, seja com *best-first* ou *depth-first*.

Ainda, será avaliado como os algoritmos de Christofides e Twice around the tree se comportam ao comparar seus resultados obtidos, tempo e espaço gastos.

2. Arquitetura do projeto

O projeto possui a seguinte estrutura de pastas:

- `out_{algorithm}`: Pasta onde são armazenados os resultados gerados pelos algoritmos executados. Os arquivos dentro dessa pasta incluem dados sobre o melhor caminho encontrado, o custo mínimo, o tempo de execução e o uso de memória.
- `src`: Pasta principal que contém os códigos-fonte do projeto. Dentro de `src`, tem os seguintes arquivos:

  - `branch_and_bound.py`: Implementação do algoritmo Branch and Bound para resolver o TSP.
  - `twice_around_the_tree.py`: Implementação do algoritmo Twice Around the Tree para resolver o TSP de forma aproximada.
  - `christofides.py`: Implementação do algoritmo de Christofides, que também oferece solução aproximada.
  - `utils.py`: Contém funções auxiliares para gerar instâncias aleatórias do problema e para ler instâncias de entrada do TSP.
  - `main.py`: Arquivo principal que executa os algoritmos, realizando a chamada aos métodos implementados nos outros arquivos.
  - `comparative.py`: Código auxiliar e independente, utilizado para gerar resultados comparativos entre os algoritmos e para criar os gráficos de desempenho presentes nesse artigo.
- `tsp_examples`: Pasta que contém os arquivos de entrada utilizados pelos algoritmos para testar diferentes instâncias do problema do TSP.

2.1. Execução do projeto

A execução do projeto deve ser feita a partir de `main.py`, ele possui os seguintes argumentos esperados no terminal:

- `action`: Define qual algoritmo será utilizado para resolver o TSP. As opções disponíveis são:
  - `bnb`: Executa o algoritmo Branch and Bound.
  - `christofides`: Executa o algoritmo de Christofides.
  - `bfs`: Executa o algoritmo de busca em largura (BFS).
  - `twice`: Executa o algoritmo Twice Around the Tree.
  - `generate_random_examples`: Gera exemplos aleatórios de instâncias do TSP.
- `--tsp_folder`: Define a pasta que contém os arquivos de entrada do TSP. O valor padrão é `tsp_examples`.
- `--out_folder`: Especifica a pasta onde os resultados dos algoritmos serão salvos. Caso não seja fornecido, o script criará uma pasta com o nome `out_{algorithm}`.

O script processa todos os arquivos `.tsp` presentes na pasta definida por `--tsp_folder`. Para cada arquivo, o algoritmo selecionado será executado, e o tempo de execução, o uso de memória, o custo e caminho encontrados serão registrados. Além disso, o script possui um limite de tempo de 30 minutos para cada execução. Caso o tempo de execução ultrapasse esse limite, um arquivo de resultado com ”NA” será gerado. Os resultados de cada execução são armazenados na pasta de saída (`out_{algorithm}`).

3. Detalhes do funcionamento e implementação do Branch and Bound

O algoritmo Branch and Bound para resolver o problema do Caixero Viajante é uma técnica funciona por meio da exploração do espaço de soluções. Ele divide o problema em subproblemas menores, estimando uma solução ótima parcial para cada subproblema usando um limite inferior (*bound*). A partir disso, ele realiza a poda de subproblemas cujos limites indicam que não podem gerar uma solução melhor que a melhor já encontrada. Esse processo continua até que todos os subproblemas viáveis sejam explorados,

fazendo com que a solução ótima seja provavelmente encontrada de forma mais rápida do que uma simples solução ingênua, evitando a exploração desnecessária de caminhos que não levariam a uma solução melhor.

Neste trabalho foi explorado duas abordagens diferentes: *best-first* e *depth-first*.

3.1. Abordagem Best-First

A abordagem *best-first* utiliza uma fila de prioridade para explorar os nós com o menor limite inferior primeiro, garantindo que o algoritmo explore as soluções mais promissoras antes.

A função `bnb_tsp` começa com um nó raiz contendo o caminho inicial $[0]$, o custo inicial (zero) e o limite inferior calculado usando a função `bound`. A cada iteração, o nó com o menor limite inferior é retirado da fila de prioridade e expandido. Ela pode ser descrita pelos seguintes passos:

- Inicializa a fila de prioridade com o nó raiz.
- Expande os nós retirados da fila, atualizando o custo e o limite inferior.
- Se um caminho completo for encontrado, verifica se ele é melhor que a melhor solução encontrada até o momento.
- Realiza poda dos nós cujos limites inferiores são maiores que o custo da melhor solução encontrada.

O grafo é representado por uma matriz de adjacência, onde cada elemento $[i][j]$ contém o custo da aresta entre as cidades i e j. Para a fila de prioridade, foi usada o `heapq` disponível em Python.

3.2. Abordagem Depth-First

A função `bfs_tsp` é responsável por resolver o TSP usando a abordagem *depth-first*. Ela utiliza a função `dfs_with_bound` para explorar todos os caminhos possíveis de maneira recursiva.

A função `dfs_with_bound` explora todos os caminhos possíveis a partir de um caminho inicial. Para cada caminho, calcula o custo e verifica se ele é um caminho completo. Caso o custo do caminho seja melhor que a melhor solução encontrada, o caminho é armazenado como a melhor solução. A função também calcula o limite inferior e realiza poda quando o limite inferior de um caminho é maior ou igual ao melhor custo encontrado até o momento.

A função `bfs_tsp` pode ser descrita pelos seguintes passos:

- Inicializa a busca com o nó raiz contendo o caminho inicial $[0]$ e o custo zero.
- Chama a função `dfs_with_bound` para explorar todos os caminhos possíveis.
- Atualiza a melhor solução se um caminho melhor for encontrado.
- Realiza poda quando o limite inferior de um caminho for maior ou igual ao melhor custo encontrado.

3.3. Função Bound

A função `bound` calcula o limite inferior de um caminho parcialmente construído. O limite inferior é baseado na soma dos custos das arestas já visitadas e nas menores distâncias não visitadas para as cidades restantes. Isso ajuda a estimar o custo mínimo necessário para completar o caminho. Essa função de estimativa é usada por ambas as abordagens.

4. Detalhes do funcionamento e implementação do Twice around the tree

O algoritmo *Twice Around the Tree* é uma heurística para resolver o problema do Caixeiro Viajante (TSP). A ideia central do algoritmo é criar uma solução aproximada usando uma árvore geradora mínima (MST) do grafo, gerando um caminho de pré-ordem. Mas como é preciso voltar ao ponto inicial e o problema é Euclidiano, então o último vértice é ligado ao primeiro, fechando um ciclo Hamiltoniano a partir dessa árvore. O fator de aproximação para esse algoritmo é 2, ou seja, os resultados obtidos serão no máximo 2 vezes pior que o ótimo.

O algoritmo começa carregando um grafo G a partir de um arquivo de entrada, utilizando a função `read_tsp_file(file)`. O grafo G contém nós representando as cidades e arestas com pesos correspondentes às distâncias euclidianas entre elas.

- **Grafo:** O grafo G é representado por um objeto do tipo `Graph` da biblioteca `networkx`. Cada nó representa uma cidade, e cada aresta tem um peso associado representando a distância entre as cidades.
- **Construção da Árvore Geradora Mínima (MST):** A árvore geradora mínima é gerada utilizando o algoritmo de Prim (`nx.minimum_spanning_tree`). O MST é uma subárvore do grafo que conecta todos os nós com o menor custo total possível, sem formar ciclos. A função `minimum_spanning_tree` do pacote `networkx` é usada para calcular o MST com base nos pesos das arestas.
- **Percorrendo o MST em Pré-Ordem:** A partir da raiz da árvore (um nó arbitrário do grafo), a função `nx.dfs_preorder_nodes` é usada para gerar uma sequência de nós visitados em um percurso de busca em profundidade. Esse percurso é armazenado na lista `preorder_nodes`.
- **Construção do Ciclo Hamiltoniano:** O ciclo Hamiltoniano é gerado pela união da lista de nós em pré-ordem com o primeiro nó da lista, garantindo que o ciclo se feche, retornando à cidade de origem.
- **Cálculo do Custo Total:** O custo total do ciclo Hamiltoniano é calculado somando os pesos das arestas entre os nós consecutivos do ciclo. A função de custo é somada dentro de um loop que percorre as arestas do ciclo.

5. Detalhes do funcionamento e implementação de Christofides

O algoritmo de Christofides é semelhante ao do Twice around the tree, mas possui algumas modificações que garantem um fator de aproximação 1.5, ou seja, o valor obtido será no máximo 1.5 vezes pior que o ótimo. Em resumo, o algoritmo tem os seguintes elementos e suas funcionalidades:

- **Grafo:** Representado por um objeto `Graph` da biblioteca `networkx`, da mesma forma que o algoritmo anterior.
- **Árvore Geradora Mínima (MST):** Uma árvore geradora mínima é construída a partir do grafo original utilizando a função `nx.minimum_spanning_tree`. Esta árvore conecta todos os vértices com o menor custo possível.
- **Pareamento Perfeito Mínimo de Vértices de Grau Ímpar:** O algoritmo identifica todos os vértices de grau ímpar no MST e cria um subgrafo induzido por esses vértices (outra instânica de `Graph` da lib `networkx`). Em seguida, encontra o pareamento perfeito de menor peso nesse subgrafo usando `nx.min_weight_matching`.

- **Construção do Circuito Euleriano e do Caminho Hamiltoniano:** As arestas do MST e do pareamento são combinadas para formar um multigrafo. Um circuito euleriano é extraído deste multigrafo, e os vértices duplicados são removidos para construir um ciclo Hamiltoniano aproximado. Para computar o multigrafo e o circuito euleriano, foram usadas funcionalidades da lib `networkx`.
- **Cálculo do Custo Total**: O custo é calculado da mesma forma descrita pelo algoritmo anterior, somando os pesos das arestas do caminho Hamiltoniano.

6. Discussão e Resultados para Branch and Bound

O algoritmo de Branch and bound funciona para encontrar exatamente a solução ótima do problema, por isso ele gasta muito mais tempo. Não foi possível dentro desse trabalho analisar qualquer instância da biblioteca TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/).

Primeiramente, foi tentado usar as menores instâncias disponíveis, como: `st70`, `pr76`, `eil51` e `berlin52`. Mas todas elas gastaram mais de 30 minutos e, no fim, foi retornado "NA". Foi usada as duas abordagens implementadas, *best-first* e *depth-first*, as duas sem sucesso. O máximo conseguido foi ao pegar a instância `eil51` e remover seus pontos até ter apenas 12, obtendo um tempo de processamento próximo de 32 minutos.

Por isso, para conseguir ainda fazer uma análise, foi criada instâncias aleatórias de diversos tamanhos, aumentando gradualmente até não ser possível mais obter resultados dentro do tempo limite. Ao chegar em 12 pontos, algumas instâncias conseguiram ser obtidas, mas depois de mais de 10 tentativas sequenciais, todas elas estavam gastando um pouco mais de 30 minutos. Portanto, para fazer a análise, foram geradas 10 instâncias de cada uma das seguintes configurações: 5, 7, 9 e 11 pontos; resultando em 40 no total.

Portanto as análises sobre o Branch and Bound serão feitas para avaliar qual o melhor de se usar: *best-first* ou *depth-first*. Explorando os arquivos disponíveis no github desse projeto, é possível verificar que todas as instâncias geraram o mesmo valor ótimo, mas não obrigatoriamente eles retornaram o mesmo caminho, pois podemos ter mais de uma forma de chegar no ótimo e isso vai depender das podas feitas por cada estratégia.

6.1. Análise de tempo

O algoritmo de Branch and Bound deve gastar mais tempo conforme temos mais pontos para se explorar. Na figura 1, podemos ver que o tempo aumenta em muito ao mudar de 9 para 11 pontos. Isso acontece independentemente se estamos olhando para estratégias de *best-first* ou *depth-first*. Essa tendência ainda continua de forma exponencial, pois se com 11 pontos foi gasto, em média, cerca de 10 minutos, para 12 foi além de 30 minutos, já que o algoritmo se interrompeu com o limite de tempo, também em ambos os casos.

Um segundo resultado a se destacar é que, a partir de 11 pontos, o *best-first* (representado pelas bolas azuis da figura 1), no geral, todas gastaram mais tempo que o *depth-first*. Isso sugere que, para abordagens aleatórias do TSP Euclidiano, o *depth-first* se apresenta como uma alternativa melhor.

6.2. Análise de espaço

Com a figura 2, não é possível ver que o aumento de pontos vai também impactar no aumento de espaço necessário.

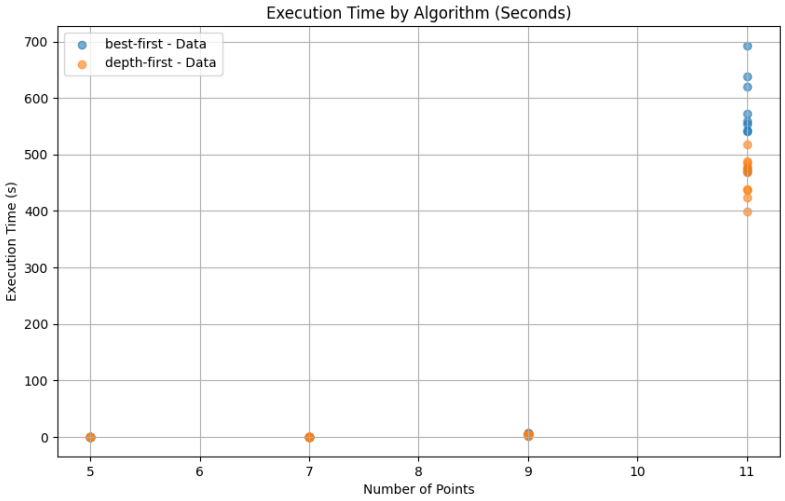

Figure 1. Tendência do tempo gasto pelo número de pontos

Contudo, isso não é verdade, porque o gasto de espaço é dominado pelo custo de se criar o grafo incial com todas as arestas entre todos os vértices. Dessa forma, o gasto de espaço obtido para o algoritmo de Branch and Bound deverá aumentar da mesma forma que aconteceu com os algoritmos aproximativos, vide figura 4. A curva fica ainda mais inclinada com mais pontos, próximo de 1000. Dessa forma, mesmo sendo impossível chegar nesse valor com o Branch and Bound na máquina usada para os testes, podemos concluir que instâncias maiores gastariam mais espaço.

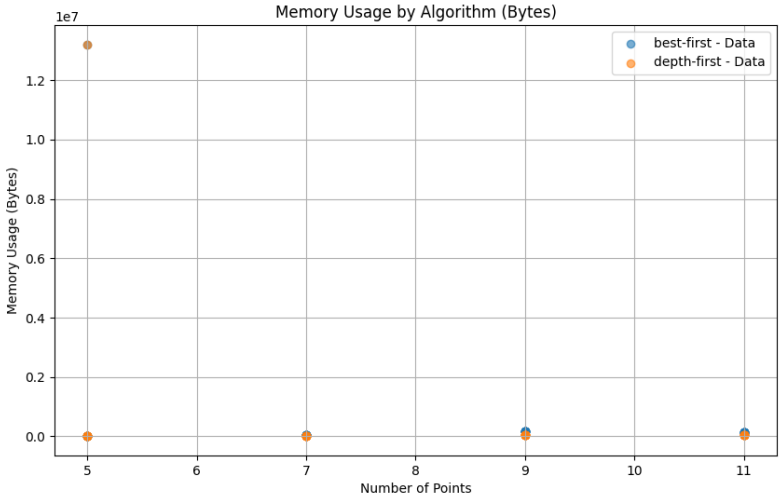

Figure 2. Tendência do espaço gasto pelo número de pontos

7. Discussão e Resultados para os Aproximativos

Com o objetivo de comparar os dois algoritmos, foi usada instâncias disponíveis na biblioteca TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/). Porém, a

máquina utilizada para fazer os testes tinha sua própria limitação e, por isso, não foi possível testar todos os arquivos de exemplo. Foi feito um corte para instâncias que tinham no máximo 6000 pontos, já que a partir disso a memória RAM do computador estava se esgotando antes mesmo do tempo limite de 30 minutos. Nas seções abaixo serão discutidas como os algoritmos se comportam em diferentes aspectos: tempo, espaço e fator de aproximação.

7.1. Análise de tempo

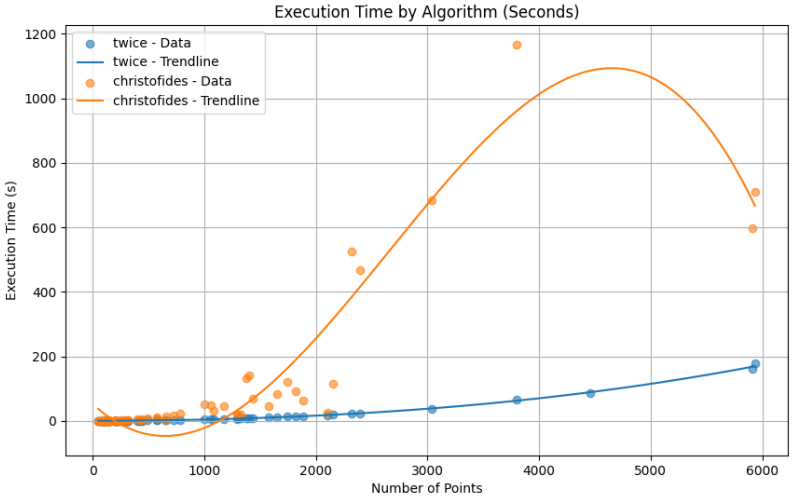

Figure 3. Tendência do tempo gasto pelo número de pontos

Uma primeira análise a se fazer é verificar se existe uma relação entre o tempo gasto e o número de pontos de uma instância. Como é possível ver na figura 3, tanto o algoritmo de Christofides quanto ao Twice around the tree possuem uma tendência a gastar mais tempo na medida que temos mais pontos para se verificar. Isso era um resultado esperado, já que ao aumentar um único ponto cria-se diversas arestas para o grafo.

Contudo, uma observação importante, é que isso é apenas uma tendência e não uma regra geral, pois instâncias maiores podem, sim, gastar menos tempo. Com os resultados do algoritmo de Christofides, isso ficou bem evidenciado, havia uma tendência de crescimento exponencial ao considerar instâncias entre 3000 e 4000 pontos, mas próximo de 6000 pontos é possível notar que o tempo gasto foi consideravelmente menor do que com instâncias próximas a 4000 pontos.

Com o algoritmo Twice around the tree não foi possível evidenciar isso com o conjunto de testes, ele obedeceu bem a tendência de aumentar o tempo de acordo com o tamanho da instância. Uma possível explicação para isso se dá na construção do subgrafo induzido de grau ímpar usada no Christofides, pois se nas instâncias maiores ele encontrar menos vértices de grau ímpar, então o tempo para processamento será realmente menor.

Ainda de acordo com a figura 3, é claramente perceptível que o algoritmo Twice around the tree é melhor quando avaliamos apenas o tempo gasto para processamento. Eles se difereciam ainda mais quando se avalia instâncias maiores, um caso específico que

isso ocorreu foi para instâncias próximas a 4000 pontos, onde o algoritmo de Christofides gastou muito mais tempo. Inclusive, ele não conseguiu completar o processamento para a instância `fnl4461` dentro dos 30 minutos, enquanto o Twice around the tree o fez em cerca de 87 segundos.

7.2. Análise de espaço

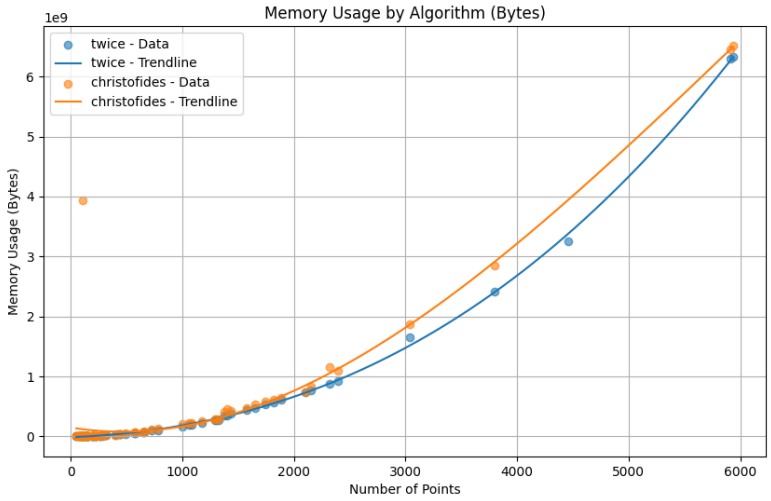

Figure 4. Tendência do espaço gasto pelo número de pontos

O espaço gasto para processar as instâncias do TSP foram dominadas pelo tamanho necessário de armazenamento do grafo. Quanto mais pontos há na instância, mais arestas foram criadas para ligá-los, já que cada novo ponto se liga a todas os outros vértices. Por isso, o esperado é que ao aumentar uma instância é aumentada também o espaço gasto de armazenamento de forma exponencial.

Conforme pode-se ver na figura 4, o comportamento esperado é de fato o obtido. Há um erro na medição de uma instância pequena no algoritmo de Christofides, onde uma bolinha laranja ficou deslocada de sua curva, mas isso pode ter sido um problema da própria biblioteca `tracemalloc`, porque as instâncias foram executadas uma após a outra sem interrupção, então talvez ela acabou acumulando resultados anteriores.

Outro ponto importante dessa análise é que os dois algoritmos aproximativos gastaram um espaço muito semelhante ao comparar duas instâncias iguais quaisquer. Não é exatamente igual, porque no Christofides, por exemplo, há uma criação do grafo induzido adicional, mas como dito antes, o custo de espaço é dominado pelo grafo inicial do problema. Portanto, não é possível dizer que o algoritmo de Christofides é muito pior que o outro quando se compara apenas o custo de espaço.

7.3. Análise do fator de aproximação

O Twice around the tree possui um fator de aproximação 2, já o de Christofides é 1.5. Com isso, o esperado é que o segundo obtenha melhores resultados do que o primeiro. Novamente, como se verifica na figura 5, ao analisar as instâncias de diferentes tamanhos, o algoritmo de Christofides é melhor se analisarmos apenas o fator de aproximação obtido.

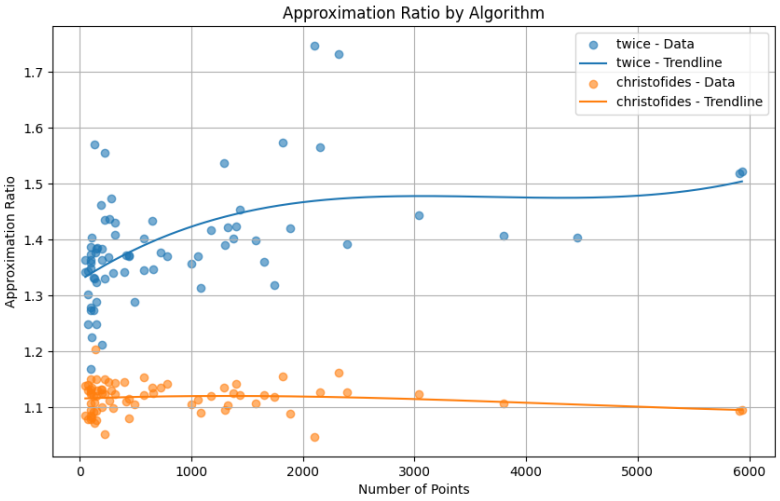

Figure 5. Tendência do fator de aproximação pelo número de pontos

A qualidade obtida por Christofides é muito maior do que no Twice around the tree, existe uma tendência quase de uma reta horizontal se aproximando de 1.1, ou seja, muito próximo do valor ótimo esperado. Não é possível concluir uma relação direta entre o tamanho da instância e a qualidade obtida, inclusive o pior caso aconteceu com uma instância pequena, onde é se vê um ponto isolado próximo do fator 1.2, enquanto para instâncias muito grande (próximo de 6000), os fatores foram praticamente 1.1.

Os resultados para o Twice around the tree foram um pouco mais caóticos. Existe uma leve tendência para se obter resultados piores ao aumentar o tamanho da instância, mas isso não é uma regra e os pontos estão muito distantes da curva encontrada. Especialmente para instâncias menores, onde foram mais testadas, tem-se uma maior discrepância de valores.

8. Conclusão

O algoritmo de Branch and Bound possui diferentes abordangens que podem ser mais vantajosas do que outras, no geral, se considerar casos aleatórios, a escolha de *depth-first* é mais rápida do que *best-first*. De qualquer forma, esse algoritmo produz uma solução viável, mas ainda muito difícil de rodar para instâncias maiores do que 12 pontos em computadores pessoais. Portanto, os algoritmos aproximativos podem ser úteis.

Para os algoritmos aproximativos, conclui-se que o algoritmo de Christofides possui uma qualidade muito maior que o Twice around the tree, mas com isso ele gasta também mais tempo. Portanto, a escolha de um ou outro vai depender do tamanho do problema e a qualidade esperada. Em resumo, se o tempo não for muito problemático, então é preferível escolher Christofides, mas se for uma instância muito grande e que esteja gastando muito tempo, então o Twice around the tree pode ser uma alternativa melhor. Sobre o custo de espaço, a diferença entre eles não é muito relevante, então pode-se considerar apenas o *tradeoff* entre tempo e qualidade.

[Levitin 1999]

References

Levitin, A. (1999). Branch-and-bound. In Goldstein, M., editor, *Introduction to The Design and Analysis of Algorithms*, pages 432–440. Addison-Wesley.