Trabalho Prático 2 da Disciplina Algoritmos II
Soluções para problemas difíceis - Caixeiro Viajante

Ana Luísa Araújo Bastos[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

ana.araujo@dcc.ufmg.br

Abstract. *This report describes the implementation and evaluation of algorithms for the Traveling Salesman Problem (TSP), in the Euclidean variation - where the vertices are points in the 2D Cartesian plane and the edge weights are the Euclidean distances between the points - including the exact algorithm branch-and-bound, and the approximations twice-around-the-tree and Christofides. The codes were implemented in Python3 and evaluated in terms of execution time, space usage and solution quality. Furthermore, an analysis of the experiments will be presented, discussing the results and conclusions obtained.*

Resumo. *Este relatório descreve a implementação e avaliação de algoritmos para o Problema do Caixeiro Viajante (TSP), na variação euclidiana - onde os vértices são pontos no plano cartesiano 2D e os pesos das arestas são as distâncias euclidianas entre os pontos- incluindo o algoritmo exato branch-and-bound,e os aproximativos twice-around-the-tree e Christofides. As implementações foram realizadas em Python3 e avaliadas em termos de tempo de execução, uso de espaço e qualidade da solução. Além disso, será apresentada uma análise dos experimentos, discutindo os resultados e conclusões obtidas.*

1. Introdução

O Problema do Caixeiro Viajante (TSP) é um desafio clássico na área de otimização combinatória, com ampla aplicação em diversas áreas, incluindo logística, transporte e design de circuitos. Consiste em encontrar o caminho mais curto que visita cada cidade exatamente uma vez e retorna à cidade de origem. O TSP faz parte da classe de problemas NP-Difícil, o que significa que não existe um algoritmo capaz de resolver o problema em tempo polinomial em uma Máquina de Turing Determinística.

Neste contexto, o presente trabalho aborda as implementações e análises de desempenho de três algoritmos distintos para solucionar o TSP euclidiano. Nessa variação, vértices são pontos no plano e o custo de uma aresta é a distância euclidiana entre os vértices em que ela incide.

Os algoritmos escolhidos são: branch-and-bound, uma abordagem exata que explora de maneira inteligente o espaço de soluções para tentar evitar o custo exponencial, o twice-around-the-tree e o algoritmo de Christofides, sendo os últimos heurísticas aproximativas que obtém soluções próximas à ótima com um custo muito menor.

Este relatório descreve detalhes da implementação de cada algoritmo, estruturas de dados escolhidas e outras considerações importantes. Além disso, apresenta os resultados obtidos em experimentos conduzidos com conjuntos de teste da TSPLIB, analisando o desempenho em termos de tempo, espaço e qualidade da solução.

2. Implementações

Em todas as implementações, a biblioteca Networkx é utilizada para a construção do grafo não direcionado com os dados das instâncias de teste, esta é a principal estrutura de dados, e o custo das arestas é calculado pela distância euclidiana entre os vértices. Outros detalhes específicos de cada algoritmo serão apresentados nas subseções a seguir.

2.1. Branch-and-Bound

A implementação do Branch-and-Bound utiliza a estratégia best-first-search com um heap de máximo para explorar os nós mais promissores primeiro e tentar podar mais ramos. O grafo completo é representado por uma matriz de adjacência. A classe Node é empregada para representar os nós na árvore de busca, contendo informações como limiar (bound), custo acumulado, nível na árvore e o caminho percorrido até o nó.

O algoritmo inicia criando o nó raiz e adicionando-o à fila de prioridade. Uma solução inicial é gerada, considerando um caminho trivial, como uma tentativa de podar mais ramos previamente e diminuir o custo da execução. A função bound é responsável por calcular o limiar de cada nó, estimando o custo total com os vértices ainda não incluídos no caminho como sendo o teto da soma das 2 menores arestas de cada vértice dividida por 2, somado ao custo do caminho atual.

Durante a execução, o algoritmo remove o nó com o menor limiar. Se o nó for uma solução completa, a melhor solução é atualizada, caso necessário. Se não for uma folha, mas o limiar for menor que a melhor solução atual, gera e adiciona os filhos na fila de prioridade. Se for uma folha, gera e adiciona o filho que fecha o ciclo na fila de prioridade.

Embora o algoritmo retorne a solução ótima, sua complexidade total é elevada (O($n!$)), indicando que pode ser extremamente demorado, especialmente para instâncias grandes do problema do caixeiro viajante.

2.2. Twice-Around-the-Tree

A idéia por trás do Twice-Around-the-Tree é que, para o problema do TSP euclidiano, a árvore geradora mínima do grafo é uma boa aproximação do custo mínimo. A implementação apoia-se nas funções da biblioteca Networkx, que é bastante útil para manipulação de grafos.

A função recebe o grafo com os vértices de entrada e uma raiz e calcula a árvore geradora mínima (AGM) usando a função *minimum_spanning_tree* da Networkx. Depois, grava em uma lista o caminhamento em pré-ordem, começando pela raiz, na AGM e adiciona o vértice inicial no fim da lista para fechar o ciclo. Esse será o ciclo que o caixeiro viajante deverá percorrer para obter o menor custo estimado. O custo total é calculado pela função *path_cost()*, que, dado o grafo original e o caminho a ser percorrido, calcula a soma dos pesos das arestas utilizadas.

Esse é um algoritmo dito 2-aproximado, ou seja, o caminho encontrado por ele pode ter custo até 2 vezes o custo ótimo. Já em relação ao custo assintótico, o algoritmo tem um ganho muito grande em comparação à solução exata, tendo o último custo O(2^n) e o primeiro custo O($n^2 logn$), uma vez que gerar a AGM é a operação mais custosa do algoritmo.

2.3. Algoritmo de Christofides

A terceira implementação é baseada no algoritmo de Christofides, outra heurística aproximada, que parte da mesma ideia do Twice-Around-the-Tree de gerar a AGM, mas melhora o fator de aproximação ao encontrar um matching perfeito de peso mínimo entre os vértices de grau ímpar na AGM e construir o multigrafo com as arestas da AGM e do matching antes de construir o circuito hamiltoniano. Essa abordagem faz com que o fator de aproximação diminua para 1,5.

Novamente a Networkx é amplamente utilizada, para garantir a eficiência e legibilidade do algoritmo. Primeiro, é calculada a AGM do grafo, usando a mesma função do algoritmo anterior, em seguida, encontra-se os vértices com grau ímpar, que farão parte do subgrafo induzido (a função *subgraph()* é a responsável por gerar o subgrafo e os vértices de grau ímpar são encontrados passando por todos os vértices eguardando aqueles cujo grau -atributo do grafo- não seja divisível por 2). Depois é usada a função *min_weight_matching()* da biblioteca para computar o matching de peso mínimo no subgrafo e criado o mulltigrafo (também uma função da biblioteca) que recebe tanto as arestas da AGM quanto as do subgrafo induzido. Após isso, *eulerian_circuit()* retorna o circuito euleriano do mutligrafo. Por fim, resta eliminar os vértices duplicados e substituir subcaminhos u-w-v por arestas u-v, para isso, cada aresta é percorrida, identificando se o vértice de destino w de uma aresta uw já foi visitado e é o mesmo da aresta vx seguinte, nesse caso, insere-se a aresta ux, ao invés de inserir uw e vx (w=v) e os vértices são marcados como visitados. Ao final, teremos o circuito com menor custo encontrado pelo algoritmo.

As principais estruturas de dados incluem: grafo completo não direcionado ,árvore Geradora Mínima, subgrafo induzido e multigrafo que são representados por estruturas de dados da NetworkX; matching perfeito, representado por um conjunto de arestas (lista de tuplas (vértice de entrada, vértice de saída)); e a lista final_path, usada para representar o caminho final.

O algoritmo de Christofides, além de ganhar em aproximação, possui o mesmo custo do Twice-Around-the-Tree, uma vez que o custo ainda é dominado pela criação da AGM.

3. Experimentos e Resultados

Foram realizados experimentos utilizando um conjunto de 78 testes da TSPLIB, considerando instâncias com a função de custo como a distância euclidiana. Foram avaliados o desempenho dos três algoritmos em termos de tempo, espaço e qualidade da solução. Todos os testes foram feitos em uma máquina com Windows 11, processador Intel(R) Core(TM) i5 de 10ª geração e memória RAM de 8gb após uma reinicialização do sistema. A versão do Python é a 3.12 e todas as bibliotecas estão nas versões mais recentes disponíveis. Os testes possuem limite de tempo de 30 minutos, sendo encerrados após esse período.

3.1. Algoritmo Branch-and-Bound

Para o Branch-and-Bound, mesmo a menor instância, com 51 vértices, teve tempo superior a 30 minutos, por esse motivo, não foi possível coletar informações.

3.2. Twice-Around-the-Tree (TATT)

Para o Twice-Around-the-Tree, foram coletados os dados dos testes feitos com as 73 menores instâncias, após isso o programa foi encerrado com o erro ”Memory error”. As informações foram adicionadas a uma planilha e utilizadas para gerar gráficos relacionando o tempo, o fator de aproximação e a memória utilizada com o número de nós do grafo. Os resultados são apresentados a seguir.

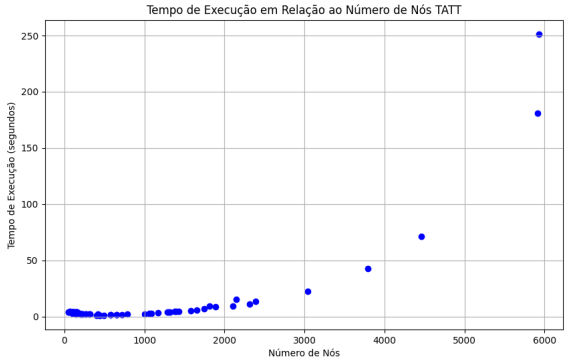

Figura 1. Gráfico do Tempo de Execução em Relação ao Número de Vértices/Nós para o TATT

Como é possível perceber, o tempo cresce de acordo com o número de nós (vértices) e não possui muitos pontos fora da curva. A média de tempo entre todos os datasets que demoraram menos que 30 minutos foi 11.45 segundos, que se distancia muito dos 251,29 segundos necessários para encontrar a solução para o dataset rl5934.tsp, que possui 5934 vértices.

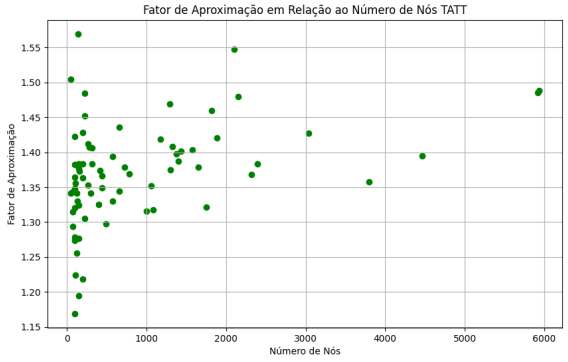

Figura 2. Gráfico do Fator de Aproximção em Relação ao Número de Vértices/Nós para o TATT

Já em relação ao fator de aproximação, que indica a qualidade da solução (quanto menor, melhor), pode-se perceber que ele está pouco relacionado ao número de vértices

que o grafo possui, e apesar do fator de aproximação provado para o Twice-Around-the-Tree ser 2, nenhum dataset teve uma solução com custo maior que 1.55 vezes o valor da ótima, o que indica uma boa qualidade do algortimo em relação à solução.

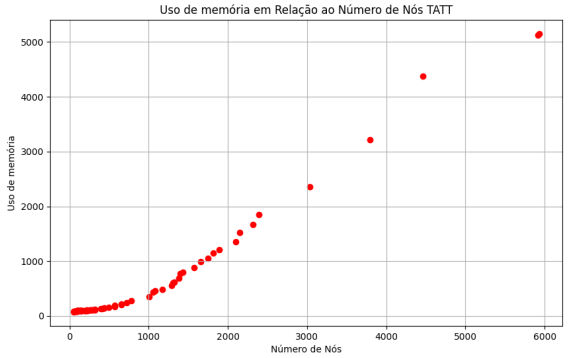

Figura 3. Gráfico do Uso de Memória em Relação ao Número de Vértices/Nós para o TATT

No que se refere ao uso de memória, no entanto, novamente é possível encontrar forte relação com a quantidade de nós. O último dataset que foi possível analisar, ocupou cerca de 5GB de memória, e o seguinte a ele, que possui 11849 nós, já ultrapassou a quantidade de memória disponível no computador, sendo assim, em um computador com as mesmas configurações do utilizado, pode-se dizer que o algoritmo está limitado, por um fator de memória, a datasets com cerca de 6000 vértices.

3.3. Algoritmo de Christofides

Para o Christofides, os primeiros 70 testes terminaram a execução, mas o 71º foi abortado por exceder o limite de tempo. Assim como para o algoritmo anterior, todos os dados foram coletados em uma planilha e traduzidos em gráficos. Os gráficos estão dispostos a seguir.

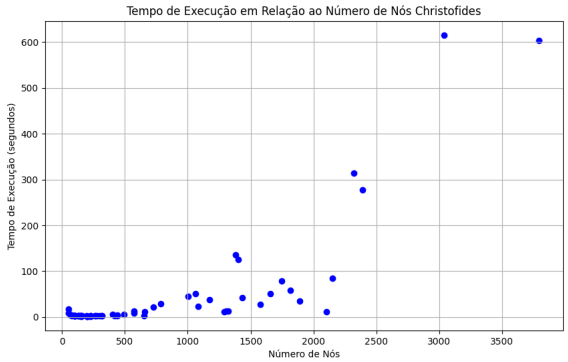

Figura 4. Gráfico do Tempo de Execução em Relação ao Número de Vértices/Nós para o Christofides

Como pode-se ver, para o Christofides, não é possível encaixar uma função bem definida na relação entre tempo e nós, mas em média, o tempo tende a aumentar com o número de vértices. Também foi calculada a sua média, obtendo 40.74 segundos, ou

seja, maior que a do algoritmo anterior, provavelmente porque, embora os dois tenham a mesma complexidade assintótica, o Christofides realiza um número maior de operações.

Para esse algoritmo, o limite se contra no tempo gasto. Considerando que o último teste cujo tempo gasto estava dentro do limite foi aquele para o dataset fl3795.tsp com 3795 vértices e o seguinte, com o 4461 nós já ultrapassou os 30 minutos, pode-se dizer que esse algoritmo termina em pouco tempo para grafos de até 4000 vértices.

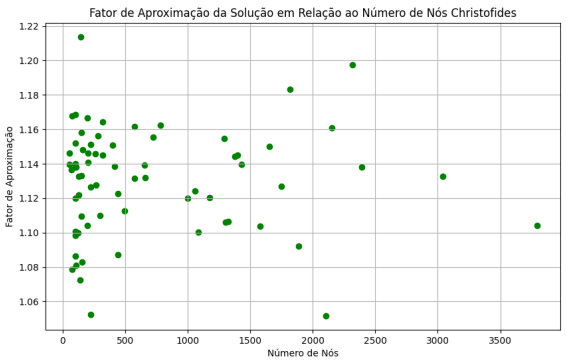

Figura 5. Gráfico do Fator de Aproximção em Relação ao Número de Vértices/Nós para o Christofides

Acerca do fator de aproximação, como esperado, o algoritmo possui resultados melhores que os do anterior, sendo, no teste com pior resultado, 1.22 vezes pior que a solução ótima.

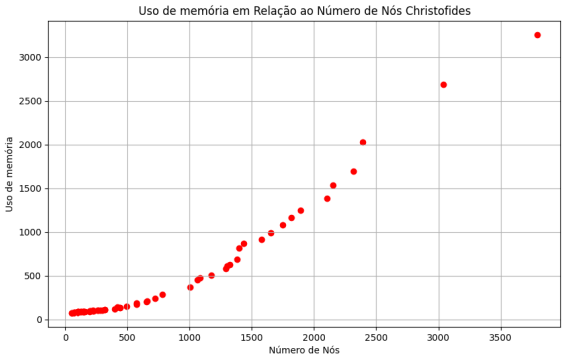

Figura 6. Gráfico do Uso de Memória em Relação ao Número de Vértices/Nós para o Christofides

Com respeito à memória, é possível perceber que quantidade de vértices e memória estão proporcionalmente relacionados e podem escalar rapidamente.

3.4. Comparação

Para uma comparação mais clara entre os algoritmos, foram plotados três gráficos que unem os dois algoritmos para as métricas avaliadas e uma tabela com estatíticas dos testes.

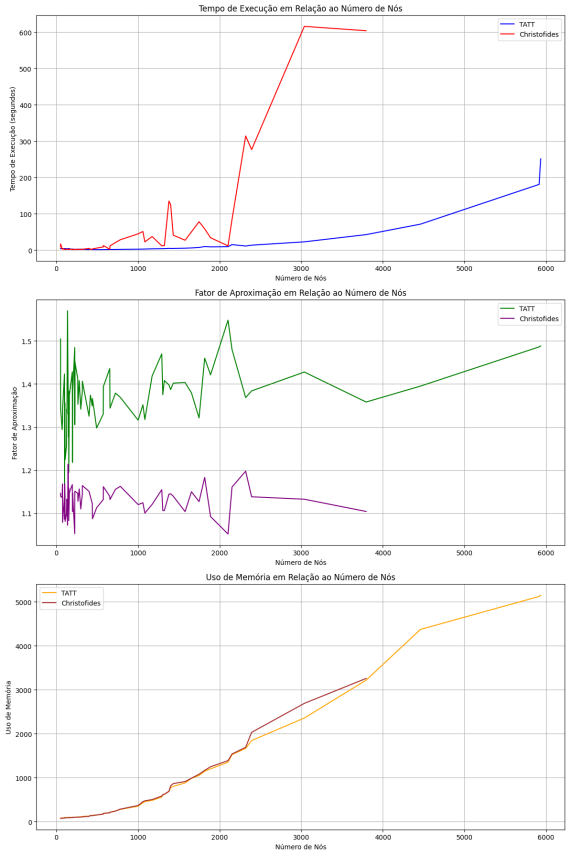

Figura 7. Gráfico da Avaliação das Três Métricas para os Dois Algorirmos

Tabela 1. Análise comparativa entre TATT e Christofides

Métrica	Min	Max	Média	Desvio Padrão
Tempo de execução (s) - TATT	1,2518	251,298	11,4536	36,3742
Uso de memória (KB) - TATT	74,2734	5154,5352	592,5579	1070,1306
Qualidade da solução - TATT	1,1689	1,5698	1,3702	0,0740
Tempo de execução (s) - Christofides	1,2620	615,767	40,7406	112,2087
Uso de memória (KB) - Christofides	75,0273	3258,4570	419,6848	628,0683
Qualidade da solução - Christofides	1,0518	1,2138	1,1299	0,0310

A partir desses gráficos e da tabela, pode-se concluir que o Twice-Around-the-Tree é bem mais eficiente e estável em questão de tempo, mas a qualidade da solução dada por ele é inferior à do Christofides, já em relação à memória, os dois parecem ter um comportamento parecido, para o dataset fl3795.tsp de 3795 nós, o último gastou aproximadamente 3.3GB e no primeiro foram gastos aproximadamente 3,2GB.

4. Conclusão

O trabalho abordou a resolução do Problema do Caixeiro Viajante por meio de três algoritmos: Twice-Around-the-Tree, Christofides e Branch-and-Bound. A escolha entre os algoritmos deve considerar os trade-offs entre tempo de execução, recursos disponíveis

e qualidade da solução. O Twice-Around-the-Tree é uma opção viável para instâncias de tamanho moderado, oferecendo soluções de qualidade em tempos razoáveis. Já o Christofides, apesar de demandar mais tempo, proporciona soluções de melhor qualidade, sendo adequado para casos em que a precisão é mais importante e dispõe-se de um tempo maior. Por fim, o Branch-and-Bound, embora não tenha resultados para os testes, permanece como uma opção para a obtenção de soluções exatas, mas deve-se estar ciente que a quantidade de recursos demandada é consideravelmente maior, uma vez que, para 51 nós já não foi possível computar a solução em 30 minutos, enquanto que os outros dois chegaram a 5934 vértices para o primeiro e 3795 para o segundo.

Referências

Levitin, Anahy. (2011) "Introduction to the Design and Analysis of Algorithms 3th (third) edition", In: Pearson Education, United States of America.

Goodrich, Michael T. and Tamassia, Roberto (2014) "Algorithm Design and Applications 1st Edition", In: John Wiley & Sons ltd., United States of America.