Análise de Estratégias para solução do Problema do Caixeiro Viajante Geométrico: Algoritmos Aproximativos e Backtracking

Cecília Junqueira V. M. Pereira, Felipe L. Gomide

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

{felipe.gomide, junqueira.cecilia}@dcc.ufmg.br

Abstract. *This article analyzes Branch and Bound approaches and approximate algorithms for solving the Traveling Salesman problem. The algorithms Twice-around-the-tree, Christofides as well as Branch and Bound in the variants Best-First and Depth-First were discussed and implemented. The TSPLIB library was used as input to execute the code produced, and information on execution time, memory usage and quality of output was captured. The results were then evaluated comparatively, assessing the positive and negative points of each approach.*

Resumo. *Esse artigo analisa abordagens de Branch and Bound e algoritmos aproximativos para solucionar o problema do Caixeiro Viajante. Os algoritmos Twice-around-the-tree, Christofides bem como o Branch and Bound nas variantes Best-First e Depth-First foram discutidos e implementados. A biblioteca TSPLIB foi utilizada como entrada para execução do código produzido, e foram capturadas informações de tempo de execução, uso de memória e qualidade das respostas. Os resultados foram então avaliados comparativamente, avaliando pontos positivos e negativos de cada abordagem.*

1. Introdução

Algoritmos aproximativos são um paradigma para lidar com problemas difíceis, que consiste em desenvolver um algoritmo que soluciona o problema de forma rápida (polinomial), porém não de forma exata. O grau de aproximação de um algoritmo aproximativo indica a relação entre a pior solução possível do algoritmo em relação ao ótimo. Um algoritmo 2-aproximativo, ou seja, de grau 2, trará uma resposta no máximo duas vezes pior que a original.

Branch-and-Bound é um paradigma para solucionar problemas difíceis. Ele se baseia em explorar as diferentes soluções combinatórias possíveis em um problema. Porém, esses algoritmos tentam otimizar o tempo da busca ao definir estimativas para soluções parciais do problema, de forma que, se já existe uma solução válida melhor que a estimativa de uma combinação da solução, essa deixa de ser explorada.

Esse trabalho busca então estudar essas diferentes abordagens para solucionar o problema do Caixeiro Viajante Euclidiano. Esse problema NP-Difícil consiste em encontrar um circuito hamiltoniano mínimo em um grafo completo, ou seja, buscar o menor caminho (em relação ao peso das arestas) que passe por todos os vértices do grafo e retorne ao ponto inicial.

A condição de Euclidiano adicionada ao problema consiste em afirmar que todos os pontos do grafo estão presentes no mesmo plano bidimensional, logo, os pesos das arestas correspondem à distância euclidiana das coordenadas dos vértices adjacentes. Essa característica da função de custo do grafo indica que essa é uma métrica, ou seja, segue as seguintes condições:

$$c(u, v) \geq 0$$
$$c(u, v) = 0 \iff u = v$$
$$c(u, v) = c(v, u)$$
$$c(u, v) \leq c(u, w) + c(w, v)$$

Tais condições são importantes para as demonstrações do grau de aproximação dos algoritmos. Sem elas, a qualidade da solução aproximativa dependeria da instância do problema. Suponha, por exemplo, um algoritmo que toma uma solução S, tendo e como a última aresta escolhida. O grau de aproximação do algoritmo dependeria então do valor arbitrário atribuído a e. Porém, tratando-se de uma métrica, a última condição descrita, chamada de desigualdade triangular, impõe que esse valor seja limitado superiormente pelo valor das outras arestas adjacentes às extremidades de e.

2. Algoritmos de Estudo

No desenvolvimento do trabalho foram implementados os algoritmos aproximativos *Twice-around-the-tree* (TaT) e Christofides e também o algoritmo de *Branch-and-Bound* específico para o Caixeiro Viajante, em duas implementações distintas. Mais informações quanto ao funcionamento, decisões de implementação e grau de aproximação destes serão dispostas nas subseções seguintes (2.1 a 2.3).

2.1. Twice-around-the-tree

O algoritmo *Twice-around-the-tree* consiste em gerar uma árvore geradora mínima do grafo e realizar um passeio nessa árvore. Uma árvore geradora mínima é um subgrafo acíclico de custo mínimo, que pode ser obtido a custo $O(n^2)$ através do algoritmo de *Primm* [Cormen 2009].

Algorithm 1 Twice-around-the-tree(G)

1- Selecione um vértice $r \in G.V$ como raiz
2- Calcule a árvore geradora mínima T de G partindo da raiz, utilizando Primm
3- Realize um passeio H na árvore geradora T
4- Retorne um conjunto de vértices, que correspondem à ordem de visita dos vértices pelo passeio H.

2.2. Algoritmo de Christofides

O algoritmo de *Christofides* possui um grau de aproximação de $3/2$, ou seja, melhor que o *Twice-around-the-tree*. Seu funcionamento também se baseia na travessia da árvore geradora mínima, porém, o faz de forma mais sofisticada. O algoritmo se baseia em gerar um circuito euleriano na árvore e remover as repetições de vértices, assim tornando-o

um circuito hamiltoniano melhor ou igual ao euleriano, pela condição da desigualdade triangular. Porém, uma condição do circuito euleriano é que todos os seus vértices devem possuir grau par, logo é necessário adicionar novas arestas à árvore para satisfazer essa condição.

O custo assintótico do algoritmo é dominado pelo custo de realizar o emparelhamento mínimo dos vértices ímpares. Pode ser resolvido através do algoritmo *Edmonds Blossom*, que possui custo $O(n^3)$.

Algorithm 2 Christofides(G)

1- Selecione um vértice $r \in G.V$ como raiz
2- Calcule a árvore geradora mínima T de G partindo da raiz, utilizando Primm
3- Gere o subgrafo induzido I de G contendo apenas os vértices de grau ímpar em T 4- Calcule o emparelhamento mínimo M de I
5- Gere o multigrafo G' com a união das arestas de T e M
6- Compute o circuito euleriano de G'
7- Realize o relaxamento do circuito, removendo repetições de vértices

2.3. Branch-and-Bound

Como mencionado na introdução, os algoritmos Branch-and-Bound buscam explorar todo o espaço de busca da solução, a partir da expansão de soluções parciais e de estimativas ou limiares do resultado das soluções parciais para avaliar se seus ramos de soluções são promissores ou não.

No caso do problema do caixeiro viajante, em que se deseja minimizar o custo do circuito, é necessária uma estimativa que seja um limiar inferior para a solução propriamente dita. Assim, uma solução já conhecida que é melhor que a estimativa de uma solução parcial será sempre melhor que qualquer solução nova encontrada a partir da solução parcial. Essa estimativa então é dada pela soma das duas menores arestas de cada vértice (entrada e saída), dividida por 2, já que arestas de saída e entrada seriam duplicadas. Como a solução é inteira, é utilizado o teto (arredondamento para cima) dessa somatória.

Além disso, existem diferentes heurísticas de exploração da árvore de soluções candidatas. Pode-se explorar a árvore por profundidade ou por qualidade, acessando os nós de melhor estimativa primeiro. Nesse trabalho, foram estudadas essas duas possibilidades,

3. Design de Estudo

Os algoritmos foram implementados na linguagem de programação Python, utilizando a biblioteca *Networkx* para modelagem dos grafos e utilizamos sua implementação dos algoritmos auxiliares das soluções aproximativas (Primm e Edmonds-Blossom). Essa decisão foi tomada pois o segundo possui uma complexidade elevada na implementação, o que não condiz dos objetivos buscados no estudo.

Quanto ao Branch-and-Bound, esse foi implementado de duas formas. A primeira utiliza uma fila de prioridade, ou *heap binário*, para explorar primeiro as soluções parciais de melhor estimativa. A segunda funciona de forma recursiva, realizando a exploração

pós-ordem da árvore. As duas soluções se baseiam em heurísticas, logo não existe uma solução evidentemente melhor. Apesar disso, existe uma vantagem da solução recursiva em memória, já que ambas as abordagens buscam explorar as $O(n!)$ possíveis soluções distintas. A fila de prioridade então pode possuir $O(n!)$ nós, com custo de inserção $O(\log n!)$, já a solução recursiva tem como limite de uso de memória a profundidade máxima da pilha de execução, já que só aloca memória no registro de ativação das chamadas recursivas, e, como o tamanho dos circuitos do grafo é n, a profundidade máxima da pilha é n e o custo em memória é $O(n)$.

A partir das implementações feitas, foram realizados testes a partir das instâncias do problema fornecidas pela biblioteca TSPLIB [Reinelt 1991], e foram realizadas medições quanto a tempo de execução, uso de memória e qualidade das soluções nos algoritmos aproximativos. As implementações e tabela com os resultados dos testes realizados podem ser encontrados no link: https://github.com/FelipeGomide/ALG2-TSP.

4. Resultados

Os dados extraídos após a execução das instâncias nos três algoritmos distintos estão presentes no GitHub, bem como uma planilha com todos esses dados. A partir dos resultados foi possível gerar gráficos e tabelas que elucidam o comportamento dos algoritmos aproximativos, sendo possível assim realizar comparações em questão de memória, tempo de execução e qualidade das soluções.

As implementações do Branch and Bound se mostraram muito ruins nas instâncias fornecidas pela biblioteca [Reinelt 1991]. Nenhuma das duas estratégias foi capaz de terminar a execução da menor instância disponível (51 vértices), em menos de 90 minutos de tempo de execução. Por isso, foram feitos estudos com instâncias ainda menores do problema, utilizando os n primeiros vértices da instância "berlin52".

A tabela 1 mostra os resultados dos testes com pequenas instâncias do Caixeiro Viajante para as duas implementações do Branch and Bound. Nesse caso, é perceptível como a navegação na árvore buscando priorizar a qualidade é pior em tempo de execução e memória, se comparado com a recursiva. Os resultados confirmam a tese inicial do custo fatorial da fila de prioridade, que seria bastante pior que o custo linear dos registros de ativação.

	Recursivo		Fila de Prioridade	
Tamanho	Tempo (s)	Memória (KB)	Tempo (s)	Memória (KB)
5	0.165	0.92	0.158	0.96
6	0.160	1.10	0.161	4.16
7	0.170	1.38	0.208	36.45
8	0.250	1.66	0.599	325.0
9	0.878	1.85	4.286	2838.5
10	6.285	2.06	39.499	25252.70

Table 1. Performance das implementações do Branch and Bound.

As figuras 1 e 2 mostram o comportamento em tempo dos algoritmos, variando a quantidade de vértices da instância, e a segunda possui um recorte nos eixos X e Y, para

que fosse possível visualizar o desempenho das instâncias menores. A figura 1 expõe uma curva de crescimento exponencial dos tempos de execução, o que coincide com o crescimento no número de arestas do grafo, ainda, que o crescimento com Christofides é bem mais acelerado. É importante ressaltar que o tempo de execução foi limitado a 30 minutos, ou 1800 segundos, o que explica o patamar alcançado após as instâncias de 11 mil vértices com Christofides.

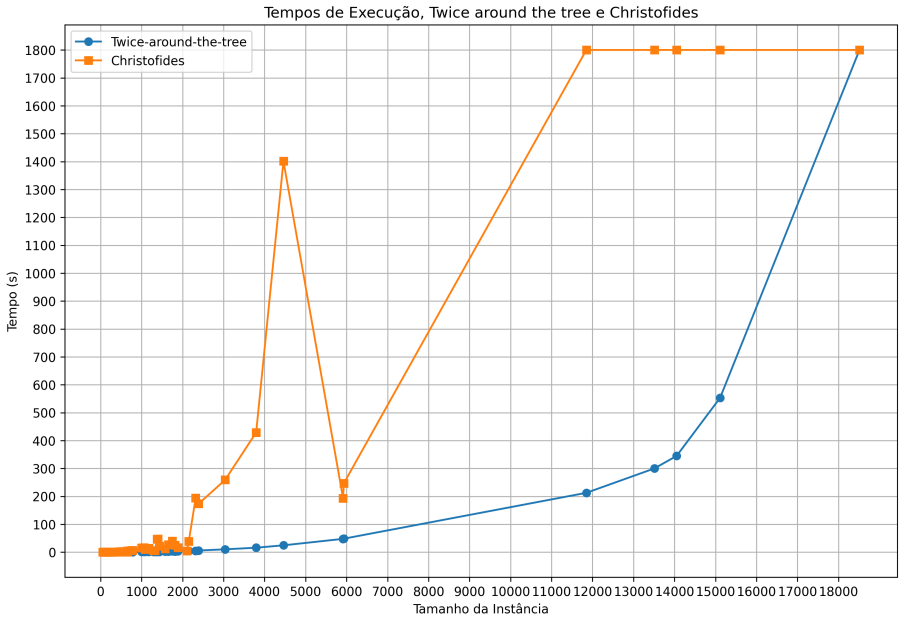

Figure 1. Tempos de Execução

Além disso, a figura 2 mostra que o tempo de execução do Twice around the tree segue uma curva muito mais comportada que o Christofides, do qual o tempo pode variar bastante de acordo com a natureza da instância. Isso pode ser resultado do seu custo assintótico ser dominado pelo custo assintótico cúbico do emparelhamento dos vértices de grau ímpar da árvore, o que indica que instâncias de muitos vértices ímpares serão mais custosas.

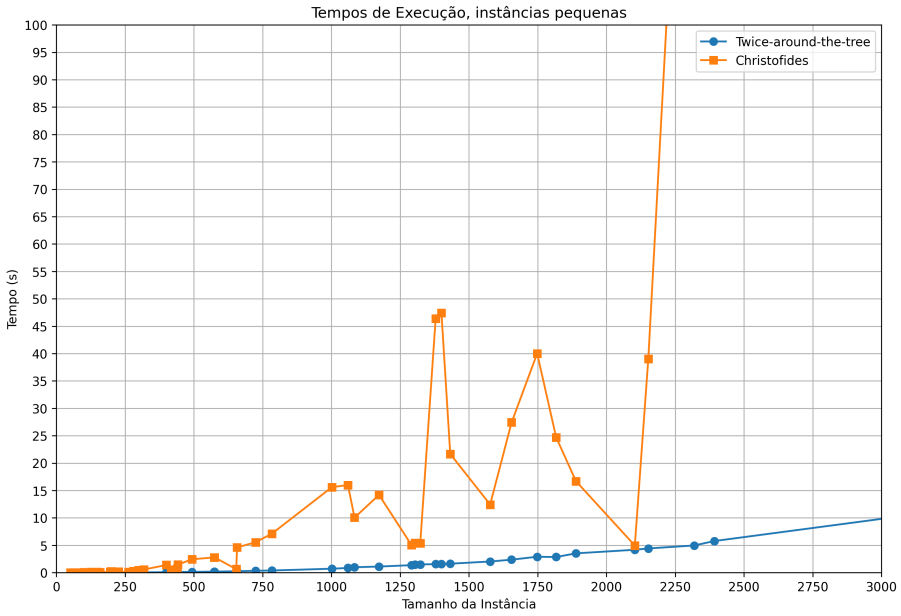

Figure 2. Tempos de Execução, instâncias pequenas

A figura 3 apresenta as curvas de uso de memória dos algoritmos. O fato curioso é que as curvas estão sobrepostas, logo a utilização de memória pelos algoritmos é basicamente idêntica, a curva tem um crescimento exponencial, acompanhando o número de arestas do grafo completo. Aqui, não existe informação quanto a instâncias de valor maior que 6.000 pois o *profiler* de memória afetou a velocidade de execução, o que levou menos instâncias a serem executadas no limite de 30 minutos.

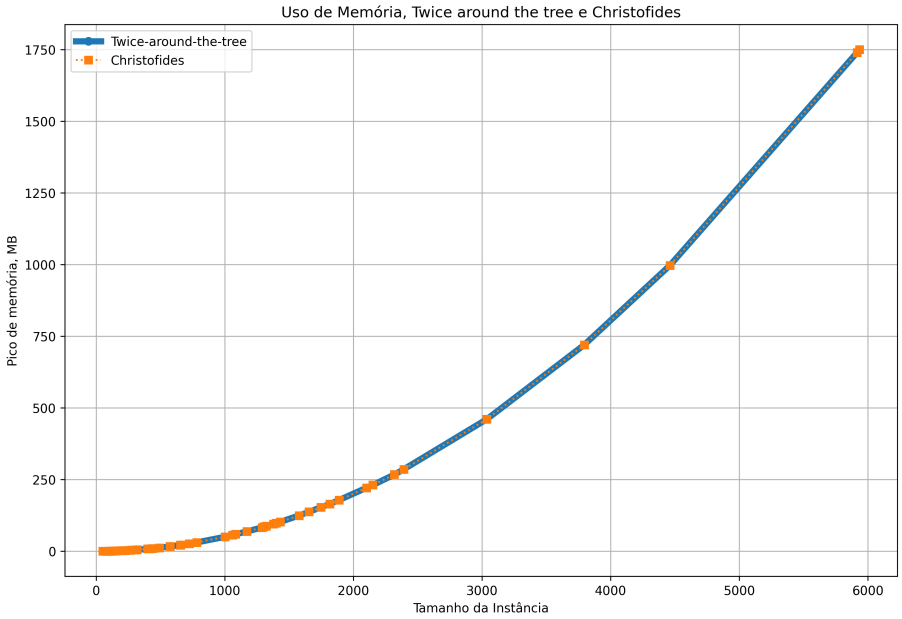

Figure 3. Picos de uso de memória

É possível verificar na figura 4 o grau de qualidade dos algoritmos aproximativos,

obtida dividindo a solução encontrada pelo valor ótimo fornecido pela biblioteca. Em ambos os casos, a qualidade média é bem inferior ao grau de aproximação do algoritmo, o que indica que os algoritmos são muito bons em grafos bem comportados. Além disso, o Christofides mostrou resultados bem melhores que o Twice-around-the-tree, porém, por ser muito mais custoso em tempo, é preciso analisar quando optar por uma das soluções. Em instâncias pequenas, de até 2 mil vértices, a diferença em tempo não é tão substancial, logo o ganho em qualidade é vantajoso, e, a partir desse momento, o Twice-around-the-tree passa a ser uma escolha melhor por ser mais rápido e melhor comportado em tempo.

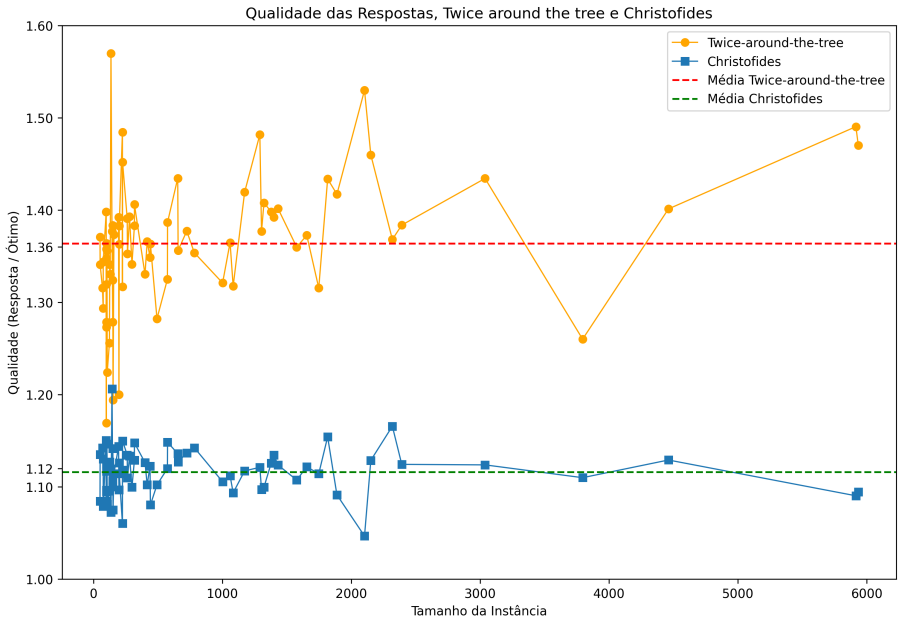

Figure 4. Qualidade das soluções aproximativas

References

Cormen, Thomas H; Leiserson, C. E. R. R. L. S. C. (2009). *Introduction to Algorithms*. PHI Learning, 3rd edition.

Reinelt, G. (1991). Tsplib—a traveling salesman problem library. *ORSA journal on computing*, pages 376–384.