Análise Metodológica das Resoluções do Problema do Caixeiro Viajante

Thaís F. da Silva[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

thaisfdasilva159@gmail.com

Abstract. *This paper explores the Traveling Salesman Problem (TSP), a classic combinatorial optimization challenge, in relation to three main algorithms: Branch and Bound, Twice Around The Tree, and the Christofides algorithm. The experiments were focused on the execution time, memory usage, and accuracy of each of the mentioned methods above.*

Resumo. *Este trabalho explora o Problema do Caixeiro Viajante (PCV), um clássico desafio de otimização combinatória, em relação a 3 principais algoritmos Branch and Bound, Twice Around The Tree e o algoritmo de Christofides. Os experimentos foram focados no tempo de execução, utilização de memória e precisão de cada um dos métodos citados acima.*

1. Introdução

O Problema do Caixeiro Viajante (PCV) é um clássico desafio de otimização combinatória, onde o objetivo é encontrar o caminho mais curto que passe por todas as cidades de um conjunto uma única vez e retorne para a cidade de origem. Formalmente falando, dado um conjunto de cidades e as distâncias entre elas, o objetivo é encontrar o caminho de peso mínimo que passe uma única vez por cada cidade e volte para a origem.

Devido ao fato de não ser um problema de fácil resolução, diversos métodos foram implementados para se alcançar um resultado aproximado do ótimo, sendo os mais famosos: *Branch and Bound*, *Twice Around The Tree* e o algoritmo de *Christofides*. Neste artigo serão apresentados essas três implementações específicas juntamente com as suas vantagens e desvantagens para cada uso.

2. Implementação

2.1. Bibliotecas Utilizadas

NumPy: é uma biblioteca fundamental em Python para computação numérica. Ela fornece suporte para arrays multidimensionais, operações matemáticas de alto desempenho e funcionalidades para trabalhar com álgebra linear, transformadas de Fourier e números aleatórios. A principal estrutura de dados em NumPy é o array, que permite a realização eficiente de operações vetorizadas.

tsplib95: a biblioteca é específica para lidar com instâncias do Problema do Caixeiro Viajante (PCV). Ela oferece funcionalidades para carregar, manipular e analisar conjuntos de dados no formato TSPLIB, um formato padrão para representar instâncias

do PCV. Com o tsplib95, é possível acessar informações sobre as cidades, distâncias entre elas e outras propriedades relevantes para o PCV.

NetworkX: é uma biblioteca utilizada para a criação, análise e visualização de estruturas de redes (grafos) em Python. O NetworkX fui principalmente utilizado na execução dos algoritmos de *Christofides* e *Twice Around The Tree* para manipulação de grafos (gerar subgrafos, obter o grau de um vértice, etc...) e execução de algoritmos auxiliares (DFS, Matching, MST, etc...) Outras (sys, time, csv, os): As demais bibliotecas foram utilizadas para ler os dados de entrada, marcar o tempo de execução, gerar o csv com os resultados, executar os testes para todos os datasets escolhidos.

2.2. Algoritmos

Todos os algoritmos a seguir foram executados levando em consideração os pseudo códigos apresentados em sala de aula, e que foram anexados abaixo após a explicação resumida de cada método.

Branch and Bound TSP: O *Branch and Bound* é um método de busca exaustiva que divide o problema em subproblemas menores, utilizando uma estratégia de árvore. Ele explora sistematicamente as soluções, realizando podas para descartar ramos que levam a soluções sub-ótimas. A complexidade é exponencial no pior caso, mas as podas ajudam a reduzir o espaço de busca. O resultado pode ser uma solução ótima para problemas de otimização combinatória, dependendo da formulação do problema.

Christofides: O algoritmo de *Christofides* é uma heurística específica para o PCV. Ele combina uma solução aproximada inicial com um refinamento baseado em emparelhamentos perfeitos e árvores mínimas. Apresenta complexidade polinomial, sendo mais eficiente que métodos exatos para instâncias grandes do PCV. Ele garante um fator de aproximação de 1.5-aproximado para o TSP euclidiano, tornando-se uma heurística eficiente e amplamente utilizada para o PCV.

Twice Around The Tree: O *Twice Around The Tree* é uma heurística para o Problema do Caixeiro Viajante (PCV). Ele cria um circuito hamiltoniano inicial na árvore de expansão mínima, duplica as arestas e aplica um processo de refinamento para melhorar a solução. Sua complexidade é polinomial, tornando-se mais eficiente que métodos exatos como o *Branch and Bound* garantindo um fator de aproximação de 2-aproximado, sendo eficiente para instâncias práticas do PCV.

3. Experimentos e Resultados

Foram realizados dois experimentos com o objetivo de analisar o tempo de execução, uso de memória e precisão em relação ao valor ótimo para os três algoritmos. O primeiro teste foi realizado utilizando 10 datasets escolhidos aleatoriamente dentre aqueles com o número de cidades inferior a 500, já o segundo foi realizado utilizando todos os datasets disponíveis.

3.1. Ambiente de Teste:

O programa foi desenvolvido na linguagem python e testado em um computador com as seguintes especificações:

- Sistema Operacional: Linux Ubuntu 22.04
- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
- RAM: 8,00 GB (utilizável: 7,80 GB)

3.2. *Branch and Bound* TSP:

Nenhuma das instâncias de teste finalizaram a execução em 30 minutos, nem mesmo a menor de todas utilizadas com 52 cidades a serem exploradas. Por esse motivo não foi possível obter dados para as próximas analises.

3.3. Tempo de Execução:

Ao compararmos o tempo de execução para os dois algoritmos podemos perceber que para um numero pequeno de cidades ambos gastam aproximadamente a mesma quantidade de tempo, já para instâncias maiores o algoritmo *Twice Around The Tree* (tatt) é mais rápido e constante do que o *Christofides* (chr).

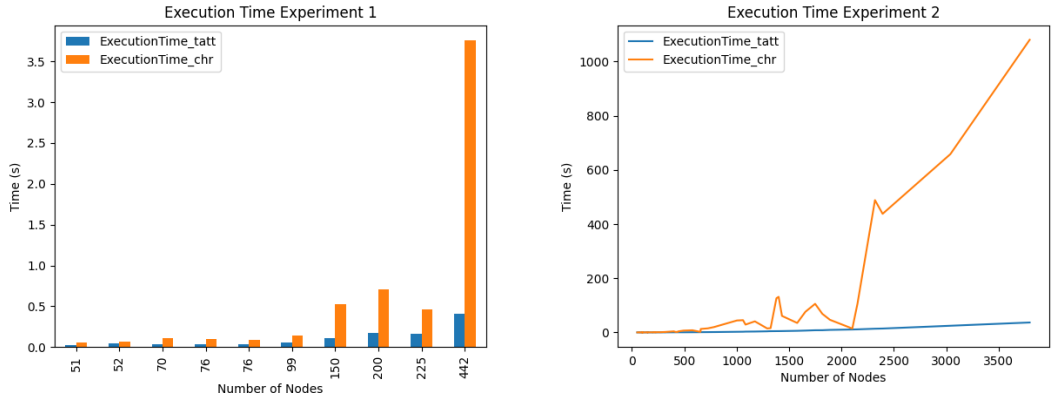

Figura 1. Tempo de Execução para os dois experimentos

3.4. Uso de memória:

Em relação ao consumo de memoria ao longo da execução dos algoritmos ambos os algoritmos tiveram gastos similares, mas como podemos observar nas figuras abaixo o algoritmo *Twice Around The Tree* (tatt) possui um gasto um pouco superior ao *Christofides* (chr). Por esse motivo o *Christofides* pode ser melhor em casos de instâncias grandes, poupando um pouco de memória ao aumentar o tempo de execução.

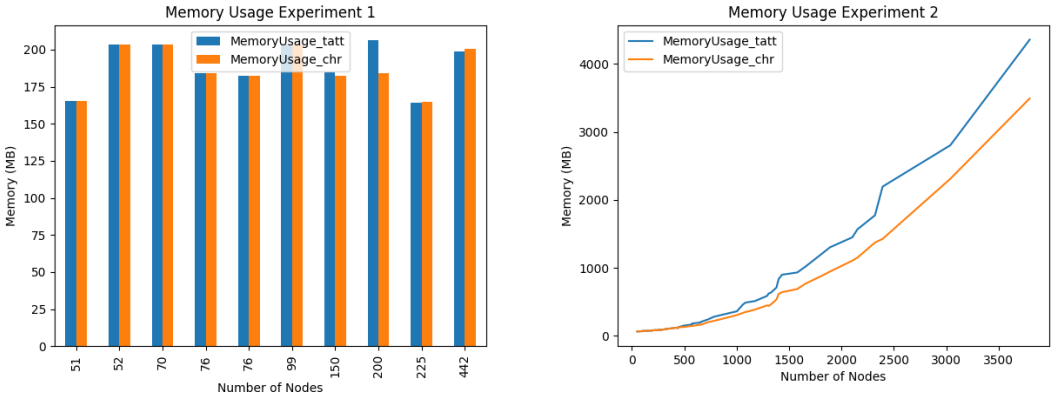

Figura 2. Tempo de Execução para os dois experimentos

3.5. Precisão:

Analisando agora apenas a precisão dos algoritmos em relação ao valor ótimo, podemos concluir que o algoritmo *Christofides* (chr) produz resultados mas semelhantes aos valores ótimos do que o *Twice Around The Tree* (tatt). Na figura abaixo podemos perceber que o desvio médio do algoritmo *Christofides* é menor, além de comprovar que ele é 1.5 aproximado enquanto o *Twice Around The Tree* é 2 aproximado.

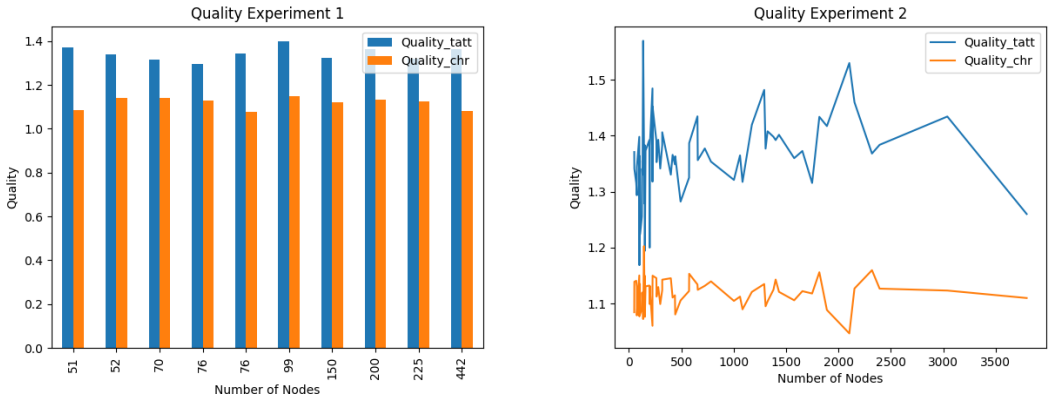

Figura 3. Tempo de Execução para os dois experimentos

Por fim, abaixo pode-se observar a tabela do experimento 1 com os dados responsáveis por gerar os gráficos de barra. Nela os dados foram separados por dataset, indo da menor para a maior instancia, e é possível observar o algoritmo utilizado *Twice Around The Tree* (tatt) ou *Christofides* (chr), o tempo de execução, uso de memória e a qualidade do resultado produzido em relação ao ótimo.

Dataset	Algorithm	ExecutionTime (s)	MemoryUsage (Mb)	Quality Result / Optimum
eil51	tatt	0,02831935883	165,203125	1,370892019
	chr	0,0552277565	165,203125	1,084507042
berlin52	tatt	0,04145407677	203,5585938	1,341023601
	chr	0,06768369675	203,5585938	1,139087775
st70	chr	0,1041760445	203,5585938	1,140740741
	tatt	0,03902983665	203,5585938	1,315555556
eil76	chr	0,09986233711	184,3476563	1,130111524
	tatt	0,03850388527	184,3476563	1,293680297
pr76	tatt	0,03810954094	182,4726563	1,343725441
	chr	0,08533406258	182,4726563	1,078819146
rat99	tatt	0,05256199837	203,5585938	1,398018167
	chr	0,1405088902	203,5585938	1,150289017
kroA150	tatt	0,1129887104	184,4726563	1,324046147
	chr	0,5243279934	182,4726563	1,119288192
kroA200	tatt	0,1739151478	206,3085938	1,362980114
	chr	0,7085852623	184,3476563	1,131571779
tsp225	chr	0,4627695084	164,953125	1,122829418
	tatt	0,1628885269	163,953125	1,317926456
pcb442	tatt	0,4100399017	198,8085938	1,363247863
	chr	3,75713253	200,5585938	1,080546693

Figura 4. Tabelas do experimento 1

4. Conclusão

Essa breve investigação em relação a algoritmos aproximados e exatos para o Problema do Caixeiro Viajante possibilitou uma percepção da aplicação de diversas estratégias aproximativas vistas ao longo do período.

Com os resultados obtidos com foco em tempo de execução, uso de memória e precisão é possível concluir que o algoritmo *Twice Around The Tree* é melhor para instancias pequenas onde a perda de precisão não é um problema relevante. Já o algoritmo *Christofides* possibilitou um ganho de memória considerável, e por mais que gaste mais tempo do que o *Twice Around The Tree*, ainda é uma opção mas rápida do que *Branch and Bound* sem uma perda de precisão considerável. Por fim, o algoritmo *Branch and Bound* proporciona um resultado exato mas é necessário uma quantidade de memória e tempo consideravelmente maior do que os algoritmos aproximativos, sendo assim útil para casos de instâncias pequenas e uma extrema necessidade de uma precisão exata.

References

Software for Complex Networks — NetworkX 3.2.1 documentation. Disponível em: ¡https://networkx.org/documentation/stable/index.html¿. Acesso em: 11 dez. 2023.

tsplib95. Disponível em: ¡https://pypi.org/project/tsplib95/¿. Acesso em: 11 dez. 2023.