Algoritmos para o problema do Caixeiro Viajante - TSP
Soluções para problemas difíceis

Iasmin Correa Araújo

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

iasminaraujo@dcc.ufmg.br

Abstract. *The purpose of this article is to discuss and compare different solutions to complex problems, with a particular focus on the traveling salesman problem. Two approximate solutions are presented - Twice Around the Tree and the Christofides' Algorithm - and an exact solution, with Branch and Bound. In order to compare, the TSPLIB library's same instances are used.*

Resumo. *Este artigo tem como objetivo discutir e comparar as diferentes soluções para problemas difíceis, especificamente o problema do caixeiro viajante. São apresentadas duas soluções aproximativas - Twice Around the Tree e o Algoritmo de Christofides - e uma solução exata, com o Branch and Bound. Para fins de comparação, são utilizadas as mesmas instâncias, disponibilizadas na biblioteca TSPLIB.*

1. Introdução

Na área da computação, dentre os problemas que podem ser computados, existem aqueles que podem ser executados em tempo polinomial em uma máquina determinística (problemas da classe P), e aqueles que são computados em tempo polinomial em uma máquina não determinística. Esses últimos são definidos pertencentes à classe NP. Como não existe fisicamente uma máquina não determinística, os problemas NP são problemas difíceis de serem resolvidos pelos computadores existentes até a atualidade.

Um exemplo de problema NP é o problema do caixeiro viajante (Travelling Salesman Problem - TSP), que consiste em, dado um grafo completo com vértices representando as várias cidades e cada uma das arestas com o custo de ir de uma cidade a outra, queremos encontrar o caminho que passe por todas as cidades uma única vez e retorne à cidade de partida, usando o menor custo possível. Isso corresponde a encontrar o menor circuito hamiltoniano no grafo completo. A demonstração de porque esse problema é NP fica a cargo do leitor.

Por se tratar de um problema difícil, poderíamos tentar resolver este problema utilizando a força bruta. Porém, isso nos daria sempre um custo exponencial de tempo, mais especificamente um custo fatorial (temos n formas de escolher a primeira aresta, n-1 para a segunda, e assim por diante). Por tal motivo, existem algoritmos que nos auxiliam no tratamento desses problemas, como o backtracking e o branch and bound. Neste trabalho, será implementado e analisado o algoritmo branch and bound. Em termos gerais, esse algoritmo consiste em trabalhar com estimativas de soluções, sempre expandindo soluções que tenham a melhor estimativa, e "podar" soluções que sejam piores do que as já obtidas.

Ainda que essa estratégia seja interessante e possa nos fazer chegar a soluções mais rapidamente, no pior caso é possível que continuemos gastando um tempo exponencial para chegar à solução. Por esse motivo, se não precisamos exclusivamente de soluções ótimas, fornecidas por algoritmos exatos, podemos recorrer a algoritmos aproximados. Em particular, o problema do caixeiro viajante não pode ser aproximado por um fator constante, exceto se P=NP. No entanto, para instâncias em que a função de custo (distância entre duas cidades) é dada por uma mética, existem algoritmos aproximativos conhecidos, que podem não fornecer a solução ótima do problema, mas exploram tal caracerística para encontrar soluções relativamente próximas da ótima, informando um fator de aproximação. Neste trabalho, serão explorados os algoritmos Twice Around the Tree e Christofides.

2. Implementações

Os três algoritmos para solucionar o problema do caixeiro viajante foram implementados na linguagem Python, considerando a distribuição padrão Anaconda 22.9. Além disso, para lidar com manipulação de grafos nos algoritmos, foi utilizada a biblioteca Networkx. Os detalhes de implementação de cada algoritmo serão descritos separadamente abaixo.

No código anexo ao GitHub, segue o arquivo bnb.py, além de uma tabela com os resultados. No arquivo, há uma função para criação de grafos e as funções relacionadas a cada um dos algoritmos implementados. Para executar o algoritmo, basta executar ”python bnb.py ¡nome-arquivo.tsp¿”. Por padrão, está chamada a função Twice Around The Tree, mas caso seja necessário modificar, basta trocar a chamada de função no código.

2.1. Branch and Bound

Neste algoritmo, foi utilizado o branch and bound com estimativa igual a soma das duas arestas de menor peso incidentes em cada vértice. Como cada aresta é contada duas vezes, dividimos esse valor por dois. Como estratégia para um corte mais rápido, é proposta uma solução trivial com circuito hamiltoniano 1 - 2 - 3 - ... - n - 1, sendo n o tamanho da instância. Para o branch das soluções, são geradas as permutações de cada vértice, ou seja, estando em um vértice, para onde podemos ir e com qual custo isso é possível.

A fim de equilibrar tempo e memória, foi utilizada a estratégia best-first, uma vez que a prioridade era o tempo. Se fosse utilizado o depth-first, o algoritmo poderia demorar ainda mais. Caso a prioriade fosse a memória, seria melhor utilizar depth first.

2.2. Algoritmo Twice Around The Tree

Para este algoritmo, foi utilizada a estratégia vista em sala de aula. Dado um grafo G, é computada uma árvore geradora mínima. Nessa aresta, cria-se um hipergrafo com arestas de ida e volta de um vértice para outro e encontra-se o menor circuito euleriano. Porém, esse circuito têm vértices repetidos. Como estamos lidando com métricas, podemos apenas remover essas repetições, sem prejuízos.

O Twice Around The Tree tem fator de aproximação 2, ou seja, os resultados que ele apresenta podem ser no máximo 2 vezes o valor ótimo do problema. Isso será melhor observado na experimentação.

2.3. Algoritmo de Christofides

Neste algoritmo, também foi utilizada a estratégia vista em aula. No algoritmo de Christofides, ao invés de adicionarmos todos os vértices de volta à árvore geradora mínima, devemos olhar anteriormente para o subgrafo induzido pelos vértices de grau ímpar e encontrar o emparelhamento (matching) mínimo perfeito a partir dele. Depois, adicionamos somente as arestas do matching à AGM. Por fim, procedemos de forma idêntica ao Twice Around the Tree: computamos o circuito euleriano com o hipergrafo e removemos o que é repetido, obtendo um circuito hamiltoniano.

Dessa forma, o fator de aproximação do algoritmo se torna 1,5. Ou seja, ele pode resultar em uma solução até 1,5 vezes o valor ótimo. Isso também será visto melhor na próxima seção.

3. Experimentos e discussão

Para a experimentação, os algoritmos foram executados com as instâncias disponibilizadas na biblioteca TSPLIB, especificamente instâncias cuja função de custo era a distância euclidiana 2D, para que fosse possível executar todos os algoritmos implementados.

O tempo de execução de cada instância era limitado a 30 minutos. Caso contrário, os dados deveriam ser colocados como NA. Por esse motivo, os dados de todas as instâncias do algoritmo Branch and Bound não estão disponibilizados.

Com relação aos algoritmos aproximativos, havia um total de 78 instâncias disponibilizadas. No entanto, dessas instâncias, foi possível coletar os dados de 70 delas: as 8 maiores instâncias disponibilizadas não puderam ser executadas. Devido ao tamanho das instâncias, a memória não conseguiu armazenar todos os dados para computar o programa. O tamanho máximo computado pelos algoritmos aproximativos foi 3795 cidades. Ao trabalhar com um número a partir de 4461 cidades, a memória foi excedida. é importante ressaltar que, à primeira vista, parece uma quantidade pequena de cidades, porém, deve-se observar que passa a ser preciso criar arestas de cada um dos vétices já existentes para cada um dos novos, além de também conectar cada vértice novo ao recém-criado. Isso gera um aumento significativo no espaço utilizado, e deve ser levado em consideração.

Dessa forma, analisaremos instâncias até um limite de 4000 cidades, com relação apenas aos algoritmos aproximativos. Com relação às metricas computadas, foi observado o tempo de execução em segundos, a memória em MB utilizada durante a execução do algoritmo e a qualidade das soluções. Cada um desses aspectos será discutido em uma subseção abaixo.

3.1. Ambiente de execução

O programa foi testado em ambiente Linux, utilizando o WSL com Ubuntu 20.04. Dados do Processador: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz 3.11 GHz Memória RAM: 16GB.

3.2. Análise do tempo de execução

Sobre o tempo de execução, é importante comentar inicialmente sobre o Branch and Bound. Como havia limite de tempo de 30 minutos de execução, nenhuma instância

foi executada no tempo determinado. Isso reforça, na prática, o que é dito em teoria: o problema do caixeiro viajante é um problema com custo fatorial, que em notação Big-O é $O(2^n)$. Um problema de custo exponencial com relativamente poucas instâncias, como o primeiro da lista, com 51 cidades, tem um custo de tempo elevadíssimo, e por isso não é possível computar rapidamente sua solução.

Dito isso, podemos analisar o tempo gasto pelos algoritmos Twice Around the Tree(TAT) e Christofides (CHR). Abaixo está o gráfico do tamanho das instâncias pelo tempo de execução de cada uma delas.

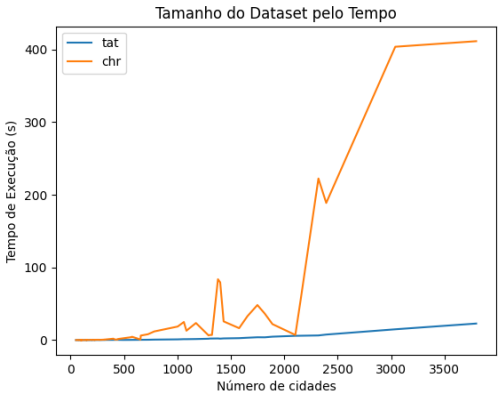

Figure 1. Gráfico do tamanho de todas as instâncias relacionado ao tempo de execução

Podemos analisar a diferença entre os tempos de execução de um algoritmo quadrático - o TAT e um cúbico - o CHR. Quanto maiores as instâncias, mais tempo é gasto no algoritmo de Christofides, enquanto no TAT a curva de crescimento cresce bem mais lentamente. Aqui, é válido observar que, pelo tempo, é bem viável lidar com instâncias de tamanhos grandes. Possivelmente, no TAT, que é $O(n^2)$, as instâncias de mais de 4000 cidades rodariam relativamente rápido, caso houvesse mais memória disponível. Esse não é o caso do algoritmo de Christofides.

3.3. Análise da memória utilizada

A memória utilizada nos dois algoritmos irá diferir devido ao número de arestas adicionado à árvore geradora mínima: enquanto no TAT adiciona-se os n vétices ”de volta” da AGM, no CHR adiciona-se apenas arestas do matching do subgrafo induzido pelas arestas de grau ímpar. Na prática, foram obtidos os resultados a seguir:

É possível ver que, na prática, a memória utilizada foi praticamente a mesma. Para fins de melhor visualização, foram coletados os dados de instâncias até 300 cidades, obtendo outro gráfico:

Aqui, percebe-se que, para estas instâncias, o TAT se comporta ligeiramente melhor que o CHR, gastando menos memória. Porém, os gastos não diferem tanto, mas é um resultado justificado: o CHR fornece um resultado mais aproximado. Para isso, precisa gastar mais tempo e memória para obter bons resultados.

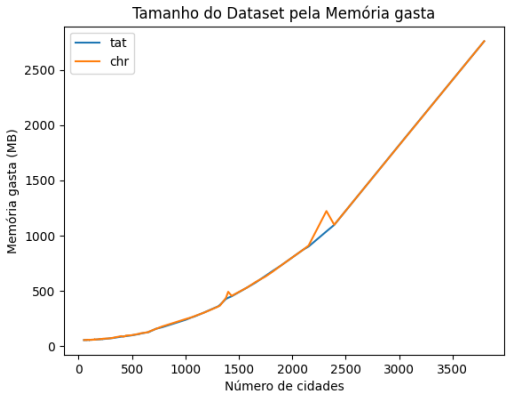

Figure 2. Gráfico do tamanho de todas as instâncias relacionado à memória gasta

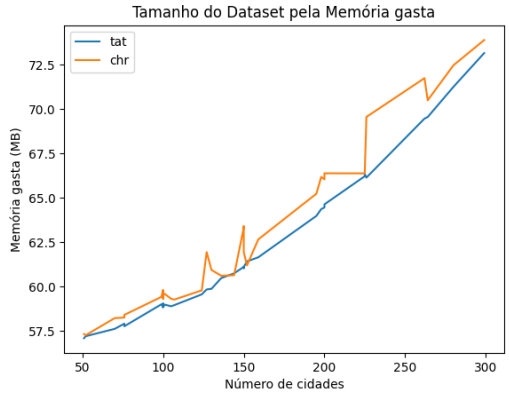

Figure 3. Gráfico do tamanho de todas as instâncias relacionado à memória gasta

3.4. Análise da qualidade das soluções obtidas

Por fim, podemos analisar a qualidade das soluções obtidas pelos algoritmos. Também foi disponibilizado o valor das soluções ótimas pela biblioteca TSPLIB. Em alguns casos, esse ótimo era um intervalo, e foi pegado o valor médio desse intervalo. Feito isso, foi calculada a qualidade das soluções, fazendo a divisão do custo obtido pelo valor ótimo, vindo da biblioteca.

Com tudo isso computado, foram obtidos os resultados:

É possível observar que, para estas instâncias, as qualidades das soluções foram muito boas: o TAT teve os valores mais frequentes oscilando entre 1.3 e 1.5, e o CHR no valor 1.1. Isso reforça a teoria de que o algoritmo aproximativo pode, **no pior caso**, ter um limite para uma solução, mas esse limite pode ou não aparecer nos cálculos. Por exemplo, o TAT, para essas instâncias, se comportou muito bem, tendo um valor máximo de aproximação de 1,6, que pode ser bastante distante de 2. O mesmo ocorre para o CHR.

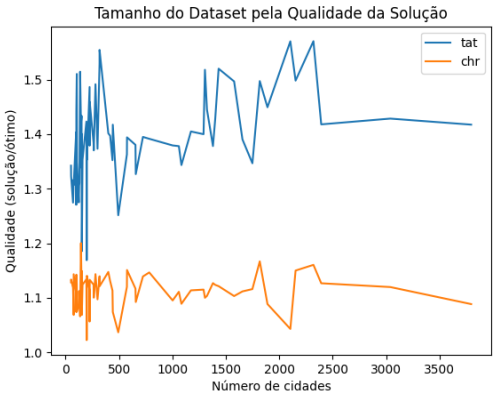

Figure 4. Gráfico do tamanho de todas as instâncias relacionado à qualidade obtida

4. Conclusão

Tendo em vista que nenhuma das instâncias executou em menos de 30 minutos no algoritmo Branch and Bound, é necessário discutir qual a prioridade desejada ao tentar encontrar a solução para um problema: tempo, memória, ou precisão - introduzida com os algoritmos aproximativos.

Com a execução dos experimentos, é possível verificar que as soluções obtidas com algoritmos aproximativos, a depender das instâncias utilizadas, podem ser muito próximas das soluções exatas. Quanto mais precisos tentarmos ser, mais recursos serão gastos. Por isso o algoritmo de Christofides gasta mais memória e tempo, mas consegue ser mais preciso que o Twice Around the Tree.

Dessa forma, se a prioridade é o algoritmo com a solução ótima, não existe outra forma de obtê-la, senão dispondo de muito tempo e muita memória, e sabendo que para determinadas instâncias será impossível que a mesma pessoa inicie a execução de um programa e veja seu resultado. Porém, se a exatidão ainda é uma prioridade mas tempo e memória são mais escassos, é melhor que se escolha utilizar algum algoritmo aproximativo. Se temos um pouco mais de recursos, teremos mais qualidade com o Algoritmo de Christofides. Caso contrário, ainda temos o TAT para fornecer uma boa solução aproximada, que é **no máximo** duas vezes o ótimo. Como vimos, para várias instâncias ela consegue uma resposta bem próxima ao ótimo, como 1,5.

References

Vimieiro, R.(2023). Slides virtuais e códigos da disciplina de Algoritmos II. Disponibilizados via Teams. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Levitin, A. Introduction To Design And Analysis Of Algorithms, 2/E. [s.l.] Pearson Education India, 2008.

Reference — NetworkX 3.2.1 documentation. Disponível em: https://networkx.org/documentation/stable/reference/. Acesso em: 11 dez. 2023.