Trabalho Prático 2 - soluções para problemas difíceis

Izadora Monken Ganem, Mateus Cursino Gomes Costa

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
DCC207 – Algoritmos 2 – Prof. Renato Vimieiro

izaganem@ufmg.br, mateuscgcosta@ufmg.br

1. Introdução

O problema do Caixeiro Viajante consiste em encontrar uma rota para o caixeiro partir de uma cidade e retornar a ela passando por todas as cidades existentes exatamente uma vez percorrendo a menor distância possível. Esse problema pertence à classe NP-difícil e pode ser modelado por um grafo completo, no qual os vértices representam as cidades e as arestas representam o caminho entre elas com seu peso representando a distância entre as cidades; a solução consiste em encontrar o circuito Hamiltoniano – inclui todos os vértices uma única vez, exceto o inicial = final – de menor peso no grafo gerado. Neste trabalho prático, analisamos soluções para o problema do caixeiro viajante geométrico baseadas em 3 algoritmos estudados em sala de aula:

- Branch-and-bound: solução exata com estratégia de redução da complexidade.
- Twice around the tree: solução aproximada com fator de aproximação igual a 2.
- Christofides: solução aproximada com fator de aproximação igual a 1,5.

avaliando o desempenho de cada algoritmo com base no custo de tempo, no custo de espaço e na qualidade da solução retornada.

2. Implementação

A implementação deste trabalho prático foi feita em Python 3.12 e conta com quatro arquivos .py, um para cada algoritmo, um para interpretação da entrada e um .ipynb para testes, coleta de dados e análise de resultados. As decisões tomadas durante o desenvolvimento serão avaliadas com mais profundidade a seguir.

2.1. Branch-and-bound

Existem duas funções, Bound(G, caminho) e BnB(G, n, r), com G sendo um grafo, n o número de vértices e r o vértice inicial(root). Bound faz uma varredura linear do caminho fornecido e inclui o peso de todas as arestas presentes nele. Em seguida, para todo vértice não presente no caminho, a aresta de menor custo saindo do vértice é somada ao valor da saída. Caso o caminho fornecido tenha alcançado o tamanho máximo, o peso da aresta entre o vértice de início e o último selecionado é somado e o valor é retornado. Essa implementação foi feita baseada no cálculo de estimativa de custo visto em sala, porém foi decidido essa maneira de escolher uma aresta por vértice para que fosse evitado comparar se uma aresta mínima já havia sido somada no caso de escolher duas arestas por vértice em um caminho incompleto.

A função BnB é a implementação do branch and bound. Cada vértice do grafo é representado por uma 4-tupla (nível, bound, custo, caminho). Foi tomada a decisão de uso

de uma fila com min heap da biblioteca heapq para manter a ordenação da fila e avançar a busca. Como a 4-tupla segue uma ordem lexicográfica de ordenação, ou seja, numa comparação de duas tuplas, primeiro é comparado o primeiro termo, depois o segundo e assim por diante, o branch and bound foi implementado com uma busca em profundidade, pois nível é o primeiro valor de um vértice e o desempate é feito pelo menor bound. A busca em profundidade foi escolhida para que o melhor bound pudesse ser atualizado quanto antes, evitando que buscas desnecessárias fossem feitas. Para permitir o uso de um min heap, nível cresce no sentido negativo, logo o primeiro nível da árvore de busca do algoritmo é representado pelo valor -1, por exemplo. A variável bound recebe o valor retornado da função Bound para aquele nó da exploração, custo é a soma dos pesos das arestas do caminho correspondente e caminho é uma lista de inteiros que representam os vértices em ordem de inserção.

O nó raiz da árvore de busca é iniciado com (0,bound([1]),0,[1]) e o melhor bound iniciado com o custo do caminho [1,2,3,...,n,1], assim o espaço de busca já é possivelmente reduzido. Além disso, uma condição de que o vértice 4 nunca pode ser incluso antes do vértice 3 foi adicionada, assim casos como [1,2,3,4,1] e [1,4,3,2,1] não são ambos explorados, dividindo o espaço de busca por 2. A implementação do branch and bound foi feita de forma similar ao visto em sala, porém ao invés de adicionar nós à árvore para avaliar se eles têm bound menor que o melhor quando eles forem consultados, o bound deles é avaliado antes de adicioná-los à árvore, evitando assim inserções e heapefys desnecessários.

2.2. Twice-around-the-tree

Inicialmente, a biblioteca NetworkX foi utilizada para computar a árvore geradora mínima T do grafo G utilizando o algoritmo de Prim $((O(M + NlogN))$ e para computar uma lista com os vértices ordenados pela pré-ordem de visitação da dfs $(O(N))$, observa-se que essa função retorna a lista sem repetição dos vértices. Em seguida, o vértice inicial foi adicionado ao final dessa lista e uma função que calcula o peso do caminho foi chamada $(O(N))$ para retornar a solução.

Para o grafo original e para a árvore geradora mínima foram utilizadas as estruturas fornecidas pela biblioteca NetworkX devido à sua facilidade de manipulação e suporte a operações específicas. Já para o circuito hamiltoniano (caminho final) e para a lista com os vértices em pré-ordem foram utilizadas as listas nativas do python, também escolhidas por facilidade de manipulação.

Assim, o custo total do algoritmo é $((O(M + NlogN))$, pois a etapa dominante da complexidade é a construção da árvore geradora mínima.

2.3. Christofides

Inicialmente, utilizou-se a biblioteca NetworkX para computar a árvore geradora mínima T do grafo G por meio do algoritmo de Prim e para gerar o multigrafo g da MST. Em seguida, foram computados os vértices que possuem grau ímpar de T e o matching perfeito de peso mínimo no subgrafo induzido pelos vértices de grau ímpar foi computado utilizando a biblioteca NetworkX. Seguidamente, as arestas do matching perfeito de peso mínimo foram adicionadas ao multigrafo g e o circuito euleriano foi computado rodando uma DFS recursiva em g, pegando as arestas em ordem de visitação. Posteriormente,

cada aresta do circuito euleriano foi visitada, removendo vértices repetidos do caminho, criando atalhos da forma $u - w - v - > u - v < custo >$. Por fim, o vértice inicial foi adicionado ao final do caminho, retornando assim o circuito hamiltoniano de menor peso e uma função que calcula o peso do caminho foi chamada, retornando assim a solução final.

Para o grafo original, o multigrafo e a árvore geradora mínima, foram utilizadas as estruturas fornecidas pela biblioteca NetworkX devido à sua facilidade de manipulação e suporte a operações específicas. Já para o circuito hamiltoniano (caminho final) e o circuito euleriano, optou-se pelo uso de listas nativas do Python, também escolhidas pela simplicidade e eficiência na manipulação direta dos dados.

Logo, o custo total do algoritmo é $O(k^3)$, sendo k o número de vértices de grau ímpar da árvore geradora mínima, pois a etapa dominante da complexidade nesse algoritmo é a computação do matching perfeito de peso mínimo, onde foi utilizada a biblioteca NetworkX que segue a implementação padrão do algoritmo de Edmunds.

3. Experimentos e Resultados

3.1. Branch-and-Bound

Branch and bound é um algoritmo com crescimento exponencial em tempo em relação ao tamanho da entrada, como a menor entrada presente no TSPLIB é de 51 vértices, já era esperado que nenhum dos casos de teste executasse em menos de 30 minutos. Ao invés disso, 8 testes foram feitos com entradas de tamanho [5,10,11,12,13,14,15,16] para que o comportamento do algorítimo pudesse ser estudado.

Como é possível observar na Figura 1, o comportamento do tempo de execução é exponencial, como esperado. Durante o desenvolvimento do trabalho, tivemos mais de uma versão de BnB e um fator determinante para que as decisões de implementações finais fossem feitas foi o tempo de execução do algoritmo. Nas versões anteriores o tempo gasto por execução era quase duas vezes maior do que o alcançado na versão final. Essa melhora foi alcançada mudando a busca para busca em profundidade com desempate baseado no menor bound, na refatoração da função bound para evitar construções de listas desnecessárias e para diminuir o número de comparações dentro de laços e na refatoração do BnB, fazendo a avaliação do bound de um nó antes de qualquer outra coisa e apenas adicionando nós com bound menor que o ótimo na fila.

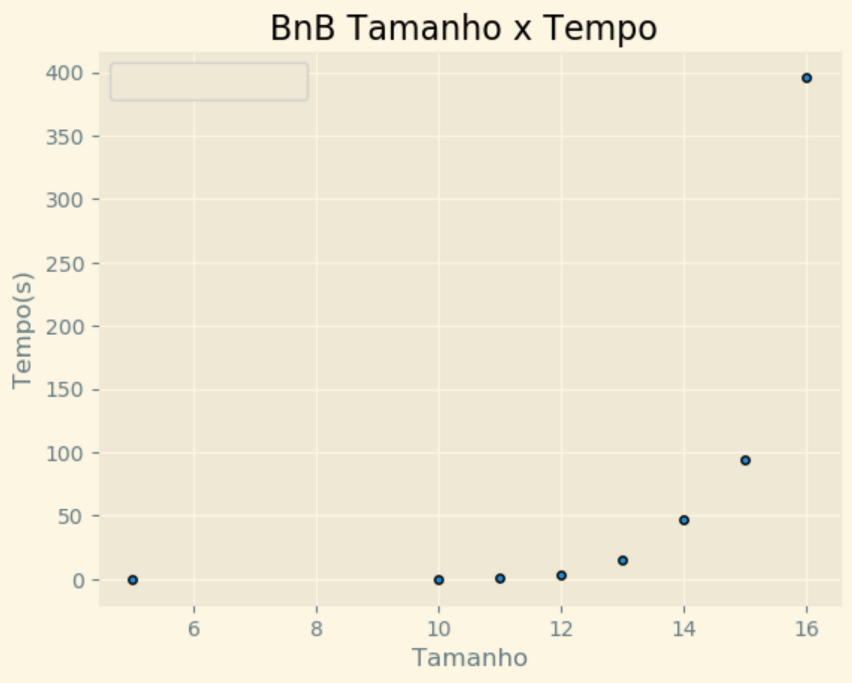

Figure 1.

Já na Figura 2, podemos ver um comportamento que aparenta linear, porém isso se deve ao tamanho reduzido das instâncias. Caso fossem feitos mais testes é esperado que esse comportamento se aproxime de uma curva exponencial imperfeita, visto que a árvore de nós do espaço de busca não é completa.

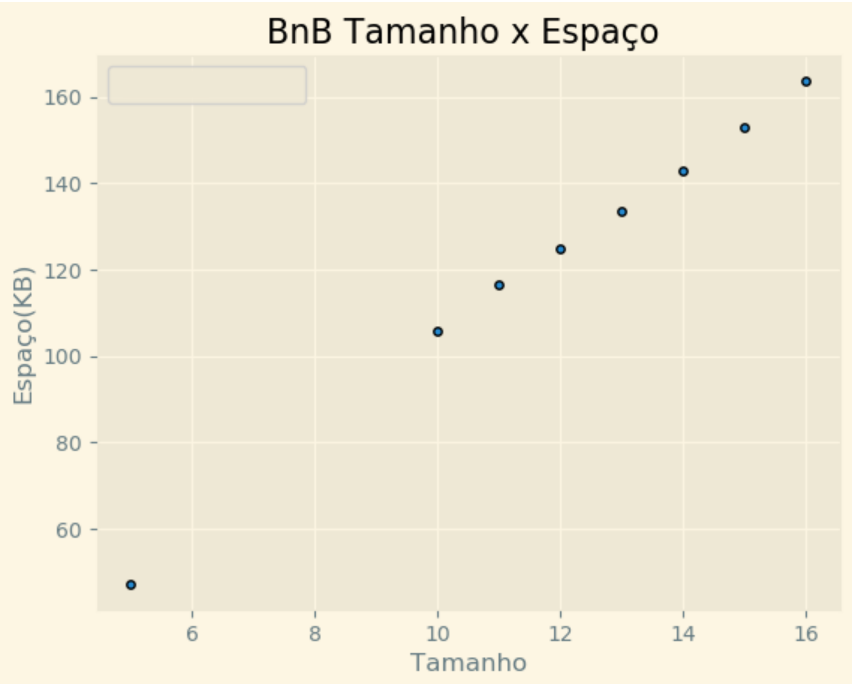

Figure 2.

3.2. Christofides e Twice-around-the-tree

As análises de desempenho, tempo e espaço do Christofides e do Twice-Around-the-Tree serão feitas de forma comparativa entre eles, visto que ambos são algoritmos aproximativos para o problema do TSP. Dado o limite de meia hora de execução, Os 70° a 74° maiores casos não executaram Christofides dentro do limite de tempo e os 75° a 78° não executaram nem Christofides nem Twice-around-the-tree, totalizando 8 falhas no Christofides e 4 no TAtT.

A complexidade assintótica do Christofides é maior do que do TAtT e isso é confirmado pela Figura 3, na qual os pontos se aproximam de uma curva $f(x) - x^3$ e $g(x) = nlog(n)$, respectivamente.

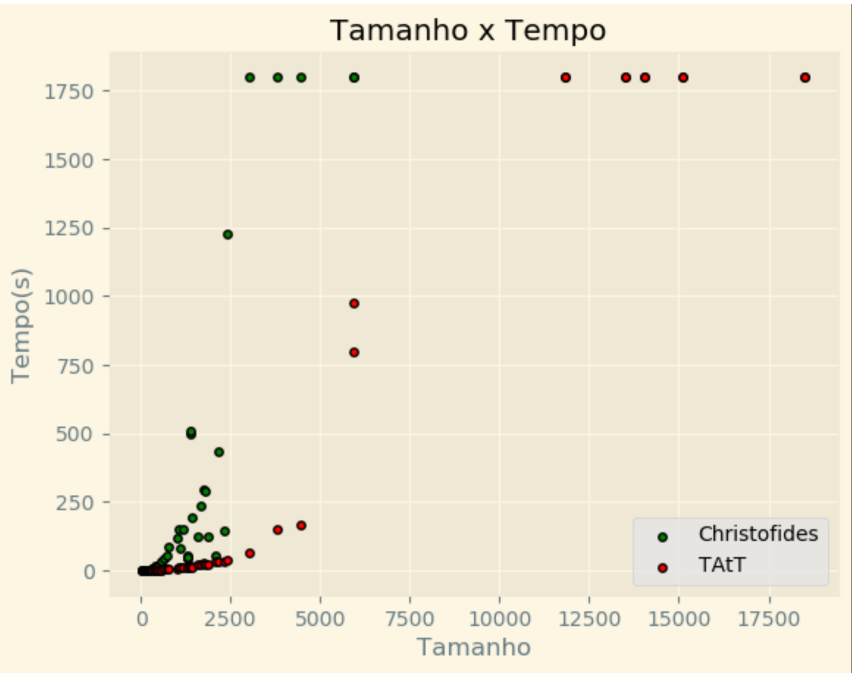

Figure 3.

Como os casos de mais vértices tomam muito mais tempo que os menores, um segundo gráfico (Figura 4) foi plotado para que uma melhor visualização seja possível.

Quanto ao consumo de espaço em memória, ambos algoritmos usaram praticamente a mesma quantidade de espaço. Isso se deve pois ambos usam de um mesmo grafo completo e mais um segundo grafo auxiliar com a mesma quantidade de arestas e vértices, seja usando o matching mínimo para adicionar arestas ou duplicando as arestas da MST. No gráfico (Figura 5), os pontos verdes foram feitos levemente maiores que os vermelho para que fosse possível ver que eles se sobrepõem em todos os pontos. O consumo de espaço também não alcançou valores muito altos, pois os maiores casos não foram executados em tempo hábil.

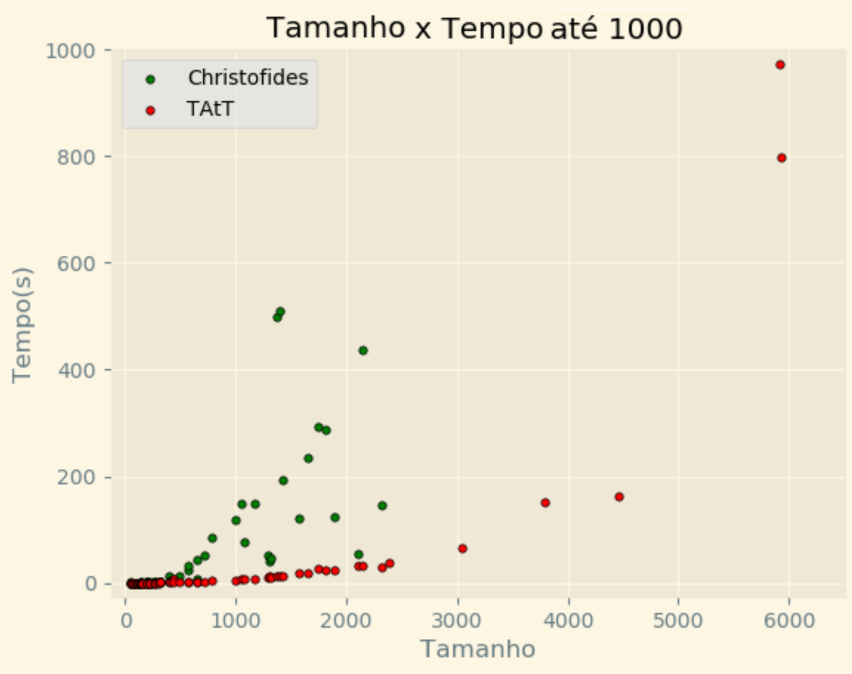

Figure 4.

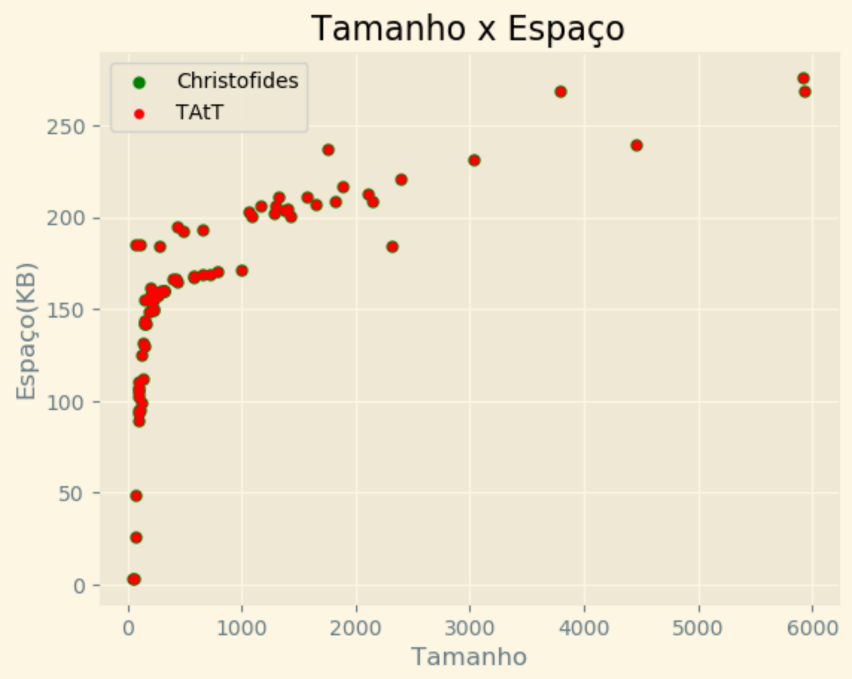

Figure 5.

A taxa de aproximação ao ótimo atingida em ambos algoritmos respeita os limites provados em sala, como esperado. Christofides nem sempre tem uma taxa de aproximação melhor do que TAtT, porém nos casos em que TAtT ultrapassa a linha dos 1.5x Christofides o supera por muito como visto na Figura 6.

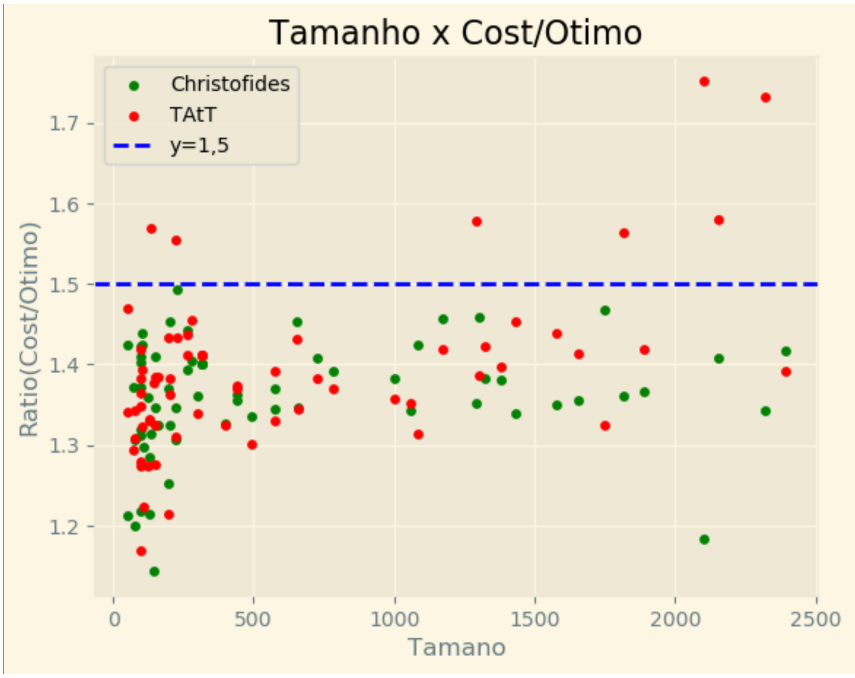

Figure 6.

4. Conclusões

Em conclusão, foi possível observar que os algoritmos foram bem implementados. O algoritmo Branch-and-Bound apresentou o comportamento esperado, sendo capaz de executar em menos de 30 minutos apenas para instâncias pequenas, enquanto os algoritmos aproximativos foram capazes de processar instâncias significativamente maiores nesse mesmo intervalo de tempo com resultados alinhados com os fatores de aproximação das soluções: 1,5 para o algoritmo de Christofides e 2 para o Twice-Around-the-Tree. No entanto, o desempenho do Twice-Around-the-Tree para instâncias maiores foi melhor, devido a sua menor complexidade de tempo.

Também, foi possível perceber a complexidade do problema do caixeiro viajante e as dificuldades na implementação dos algoritmos para resolvê-lo, principalmente no branch-and-bound, onde tivemos que fazer diversas alterações e pensar em mudanças com o objetivo de reduzir a complexidade computacional ao máximo.

Por fim, o trabalho foi fundamental para a consolidação prática dos algoritmos aprendidos em sala de aula.

5. Referências

References

Levitin, A. V. (2002). Introduction to the design and analysis of algorithms. In *https://dl.acm.org/doi/10.5555/579301*. Addison-Wesley Longman Publishing Co., Inc.

Vimieiro, R. (2024). Slides de aula 2024/2. In *Moodle*. UFMG.