Trabalho Prático 2

Gabriela Tavares Barreto

Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil

gbarreto@ufmg.br

1 Introdução

Este trabalho teve como objetivo implementar diferentes soluções para o problema do caixeiro viajante. O problema da caixeiro viajante consiste em tentar determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem. Ele é um problema de otimização NP-difícil, inspirado na necessidade dos vendedores em realizar entregas em diversos locais (as cidades) percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível. Ou seja, tem aplicações muito práticas no cotidiano.

A proposta do trabalho consistia em implementar três algoritmos diferentes para o problema, sendo dois deles aproximativos: o algoritmo de Christofides e Twice-around-the-three; um exato, utilizando a técnica de branch and bound. No decorrer deste relatório, serão apresentados detalhes sobre a abordagem utilizada para resolver o problema, o algoritmo implementado, sua complexidade computacional e uma análise dos resultados obtidos.

2 Método

O programa foi desenvolvido na linguagem Python, seguindo a distribuição determinada para o trabalho. O computador utilizado tem como sistema operacional o Windows 11, com 12 GB de RAM.

3 Implementação

3.1 Funções auxiliares

As primeiras funções desenvolvidas foram funções para abrir e ler os arquivos com as instâncias do problema. As instâncias eram provenientes da seguinte fonte: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`; e elas possuem um formato padrão. Isso permitiu criar uma função única de leitura dos arquivos, para automatizar a realização dos experimentos nos grafos posteriormente. Essa função chama-se `ler_arquivo`, e ela recebe o caminho de um arquivo com uma instância do problema, o lê e retorna o número de vértices da instância junto a uma lista com as coordenadas de cada vértice. A seguir, a descrição de outras funções criadas para auxiliar no tratamento dos dados de cada instância ou na execução dos algoritmos.

- `distancia_euclidiana(a, b)`: calcular a distância euclidiana entre os vértices `a` e `b`.

- `calcula_arestas(d, dados)`: calcula as distâncias euclidianas entre todos vértices da lista `dados`, e os retorna em uma lista.
- `custo_circuito(c, G, n)`: dado os vértices de um circuito na lista c, calcula o custo desse caminho de acordo com os pesos de cada aresta entre os vértices no grafo `G`.

3.2 Estruturas dados e biblioteca Networkx

Como a natureza do problema é de grafos, foi escolhida a biblioteca Networkx para criar objetos do tipo Graph que representassem as instâncias do problema. Essa estrutura foi escolhida porque a biblioteca é eficiente na representação de grafos, visto que por baixo dos panos o grafo é representado como uma tabela Hash e tem tempo de acesso constante $O(1)$. Assim, é uma estrutura com tempo de resposta rápido, além de já estar pronta, o que facilitou desenvolver as soluções. Além disso a biblioteca Networkx disponibiliza uma série de funções que podem ser aplicadas a grafos, como Prim e Kruskal, entre outras, que foram úteis na hora de escrever os algoritmos aproximativos.

Além de usar essa estrutura pronta, criei uma classe própria denominada Node, para ser usada no algoritmo de branch and bound. Ela será descrita em detalhes na próxima subseção.

3.3 Branch and bound

3.3.1 Descrição teórica

O algoritmo de Branch and Bound, aplicado especificamente ao Problema do Caixeiro Viajante, é uma técnica de otimização que busca encontrar a menor rota possível que passa por todas as cidades e retorna à cidade de origem. Ele funciona da seguinte maneira:

- **Estruturação da Árvore de Busca:** O algoritmo inicia com a criação de uma árvore de busca, onde cada nó representa um caminho parcial. No PCV, cada nó corresponde a um percurso parcial pelas cidades.
- **Branching (Ramificação):** A etapa de ramificação envolve a expansão de um nó da árvore. Para cada nó, geram-se nós "filhos", cada um representando uma extensão do caminho do nó pai com uma cidade adicional.
- **Bounding (Limitação):** Para cada nó, o algoritmo calcula um limite inferior do custo mínimo possível para completar a rota a partir desse nó. Esta estimativa é feita somando as duas menores arestas conectadas a cada vértice não visitado, dividindo essa soma por dois e arredondando para cima. Se o limite inferior de um nó é maior que o custo da melhor solução até o momento, esse ramo pode ser descartado.
- **Seleção de Nós para Expansão:** O algoritmo seleciona o próximo nó a ser expandido com base no menor limite inferior, direcionando a busca para regiões mais promissoras da árvore. Caso o limite inferior do nó seja maior que a melhor solução já encontrada, o nó é diretamente descartado.
- **Iteração e Convergência:** O processo de ramificação e limitação continua até que todos os nós viáveis sejam explorados ou descartados. O algoritmo converge para a solução ótima quando não há mais nós a serem expandidos.
- **Finalização:** Após a convergência, o algoritmo retorna o caminho com o menor custo encontrado, representando a solução ótima para o TSP.

A forma que o algoritmo de branch and bound percorre a árvore de soluções é denominada Best-First-Search. Em vez de explorar os filhos usando uma ordem arbitrária, como no Depth-

First-Seach, ela os explora conforme suas prioridades. Para tal, ela usa uma fila de prioridades para armazenar os nós que serão visitados.

3.3.2 Classe Node

Continuando, para implementar o branch and bound, criei a classe `Node`, que representa um nó na árvore de busca do algoritmo. Ela contém:

- `bound`: Limite inferior do custo do circuito a partir deste nó.
- `caminho`: Caminho percorrido até o momento.
- `nivel`: Nível do nó na árvore de busca.

A função `__lt__` define a comparação entre dois nós baseada em seus `bound`.

3.3.3 Funções auxiliares

Ademais, as seguinte funções foram criadas para auxiliar na implementação do algoritmo:

- `no_caminho(c, g, j)`: Verifica se um vértice `j` está no caminho `c`.
- `bound_raiz(arestas, n)`: Calcula o limite inferior do custo na raiz, usando a lista arestas que contém as menores arestas de cada vértice.
- `bound_ultimo`: Calcula o limite inferior para o último vértice adicionado ao caminho. Foi implementada separadamente pois é similar a função custo_caminho.
- `menores_arestas`: Encontra as duas menores arestas para cada vértice, e retorna um array com elas.

3.3.4 Limite inferior

Agora, sobre a função de custo:

- `bound(c, j, menores_arestas, n, g)`: Calcula o limite inferior do custo para um determinado caminho `c`, ao adicionar o vértice `j`. Essa função percorre o caminho dado por `c`. Para um caminho c com k posições já prenchidas, para o vértice 2 até o vértice $k-1$, a função soma para cada vértice os pesos das arestas usadas no caminho. Já para o primeiro e últimos vértices no caminho, que tem só uma aresta definida por ele, ela verifica se alguma das arestas definidas por c é a menor aresta para algum desses vértices. Se for, ela soma as duas menores arestas como contribuintes do vértice na soma do custo do caminho, senão, ela soma a menor aresta ligada ao vértice, mais o respectivo vértice definido pelo caminho. Por fim, a função retorna o teto da metade da soma total.

Foi feita a escolha de implementar a função de custo vista em sala porque ela cria um limite inferior "apertado". Existem outras funções de estimativa de limite inferior menos apertadas, porém mais fáceis de calcular que esta. Existe um trade-off em funções de estimativa que se resume em: quanto mais certeira a estimativa, maior é o custo de calculá-la. Ao optar em calcular o lower bound dessa forma, é possível fazer cortes na árvore mais cedo do que calculando de formas mais "soltas", então se por um lado o algoritmo fica mais devagar com essa função de custo, uma economia de espaço é feita na quantidade de nós gerados.

3.3.5 Função principal

branch_and_bound(g, d): Recebe um grafo `g` e o seu número de vértices `d`. Para a fila de prioridades, usei um objeto do tipo heap da biblioteca heapq de Python. Funciona de seguinte forma:

1. Inicializa o caminho, calcula as arestas menores e o limite inferior da raiz.
2. Usa uma fila de prioridades para armazenar os nós, começando pela raiz.
3. Executa um loop enquanto a fila não estiver vazia:
   (a) Retira o nó de menor `bound` da fila.
   (b) Se o nó representa um circuito completo, verifica se é uma solução melhor.
   (c) Caso contrário, expande o nó gerando novos nós filhos e calcula seus limites inferiores.
   (d) Adiciona os nós filhos à fila se seus limites inferiores forem menores que o melhor custo encontrado até então.
4. Retorna o melhor custo e o circuito associado.

3.4 Twice around the tree

O algoritmo Twice-aroud-the-tree é um método heurístico 2-aproximativo para resolver o Problema do Caixeiro Viajante. Ele funciona da seguinte maneira:

1. Constrói uma Árvore Geradora Mínima (AGM) do grafo fornecido.
2. Realiza uma busca em profundidade (DFS) na AGM, começando do vértice 2, para obter um caminho.
3. Calcula o custo do circuito formado por este caminho.
4. Retorna o custo e o caminho.

Sua implementação é feita na função `twice_around_the_tree`. Dado que a biblioteca Networkx fornece funções prontas para obter a AGM(`minimum_spanning_tree`) e para percorrer a árvore em pré-ordem(`dfs_preorder_nodes`), sua implementação seguiu exatamente a sequência de passos acima, chamando essas funções e por último a função `custo_caminho` para retornar o custo da solução.

4 Algoritmo Christofides

4.1 Descrição

O algoritmo de Christofides é uma abordagem heurística mais sofisticada(1.5 aproximativo) para o TSP, que tende a fornecer soluções próximas do ótimo. Ele procede da seguinte forma:

1. Calcula a Árvore Geradora Mínima (AGM) do grafo.
2. Identifica todos os vértices de grau ímpar na AGM.
3. Cria um subgrafo induzido pelos vértices de grau ímpar.
4. Encontra um emparelhamento mínimo de peso no subgrafo induzido, formando um multigrafo que é a união da AGM e do emparelhamento.

5. Realiza uma busca em profundidade (DFS) neste multigrafo, começando do vértice 1, para obter um caminho.
6. Calcula o custo deste circuito.
7. Retorna o custo e o caminho.

Novamente, as funções já prontas do Networkx vieram a calhar. Além de usar as funções da biblioteca citadas na subseção anterior, foram usadas também as funções `induced_subgraph` para gerar o subgrafo induzido e `min_weight_matching` calcular o matching máximo de peso mínimo. Dessa forma, a função `christofides` foi implementada exatamente como descrito no algortimo acima, chamando cada função necessária por vez, para no final ober um caminho c e retornar seu custo usando a função `custo_caminho`.

4.2 Função `experimento`

Foi implementada a função `experimento` para automatizar os experimentos do trabalho. Essa função lê todos arquivos de instâncias um a um, gera seu respectivo grafo, e roda os algortimos aproximativos nele. Por fim ela registra as soluções e os tempos de cada algoritmo em um documento denominado `resultados.txt`.

5 Experimentos

5.1 Branch and bound

Por ser um algoritmo exato, o branch and bound é especialmente custoso, e apesar de fazer podas na árvore do espaço de busca, o algoritmo ainda assim no pior caso tem um custo fatorial de tempo $O(|V|!)$ e de espaço também, pois no pior caso ele precisa representar todos nós da árvore de soluções. Quando, podas não são possíveis.

Resolvi testar o algoritmo primeiro para a menor instância fornecida, com 51 vértices. Infelizmente ela não conseguiu rodar em menos de 30 minutos. Tendo constatado isso, fiz diversas alterações no algoritmo buscando aumentar sua eficiência. Inicialmente, para calcular o limite inferior, toda vez que ele era calculado a função estava calculando as menores arestas de cada vértice. Assim, resolvi guardar as duas menores arestas de cada vértice em uma tabela, e essa mudança aumentou bastante a velocidade do algoritmo, porém ele ainda não rodava dentro do limite de tempo para menor instância dada. Fiz trocas nas estruturas usadas no algoritmo, testei usar listas e sets, o que mostrou não fazer muita diferença.

Além disso, testei usar uma função de custo menos "apertada", mas mais rápida de calcular, o que piorou o tempo de execução do algoritmo por aumentar o número de nós percorridos. Sendo $51! = 1.5511188e{+}66$, penso que a instância mínima fornecida era muito grande. Tentei implementar toda forma de otimização que me veio em mente. Mesmo assim, o algoritmo não melhorou o suficiente. Sendo assim, testei o branch and bound com instâncias menores próprias(com até 23 vértices ele rodava em menos de meia hora), e ele funcionou perfeitamente. Porém, por não serem oficiais não colocarei os resultados aqui. Foi bastante desafiador implementar esse algoritmo.

5.2 Twice-around-the-tree e Christofides

Para realizar os experimentos, executei a função `experimento` em partes. Primeiro para os grafos com até 1500 vértices, depois para os grafos com 1500 até 3000 vértices e por fim para o grafos entre

3000 e 4000 vértices. A partir de 4000 vértices, os algoritmos demoraram mais que meia hora para a execução, e portanto não colocarei eles nos resultados. Na tabela a seguir estão os resultados.

Instância	Vértices	Christofides	Twice-around-the-tree	Ótimo
a280.tsp	280	3569	3531	2579
berlin52.tsp	52	8610	9443	7542
bier127.tsp	127	139450	157911	118282
ch130.tsp	130	7777	8138	6110
ch150.tsp	150	8170	8596	6528
d1291.tsp	1291	68427	72373	50801
d198.tsp	198	19718	17297	15780
d493.tsp	493	45625	43420	35002
d657.tsp	657	64159	63989	48912
eil101.tsp	101	820	841	629
eil51.tsp	51	530	605	426
eil76.tsp	76	682	693	538
fl1400.tsp	1400	32583	27455	20127
fl417.tsp	417	18156	16217	11861
gil262.tsp	262	3219	3366	2378
kroA100.tsp	100	29961	27238	21282
kroA150.tsp	150	37348	35689	26524
kroA200.tsp	200	41496	39845	29368
kroB100.tsp	100	26796	26642	22141
kroB150.tsp	150	34824	36524	26130
kroB200.tsp	200	38459	39980	29437
kroC100.tsp	100	26870	27809	20749
kroD100.tsp	100	28106	27679	21294
kroE100.tsp	100	29261	28162	22068
lin105.tsp	105	19688	19438	14379
lin318.tsp	318	58776	58098	42029
linhp318.tsp	318	58776	58098	41345
nrw1379.tsp	1379	79011	78835	56638
p654.tsp	654	50427	49098	34643
pcb1173.tsp	1173	81654	80540	56892
pcb442.tsp	442	67855	69509	50778
pr1002.tsp	1002	351108	339114	259045
pr107.tsp	107	57221	53613	44303
pr124.tsp	124	78769	73044	59030

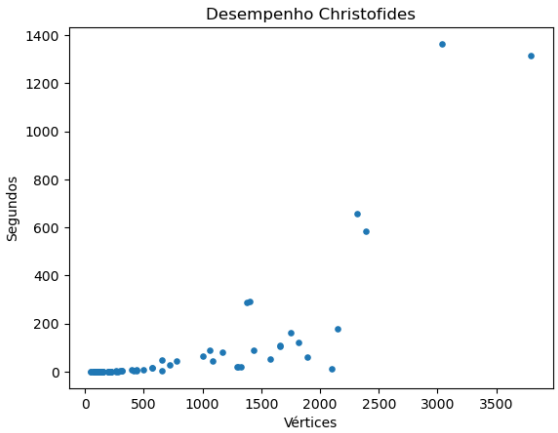
Instância	Vértices	Christofides	Twice-around-the-tree	Ótimo
pr136.tsp	136	126470	150828	96772
pr144.tsp	144	66810	80215	58537
pr152.tsp	152	96461	87344	73682
pr226.tsp	226	119829	116593	80369
pr264.tsp	264	70464	65971	49135
pr299.tsp	299	64360	64763	48191
pr439.tsp	439	143596	143110	107217
pr76.tsp	76	129386	145228	108159
rat195.tsp	195	2897	3282	2323
rat575.tsp	575	9253	9393	6773
rat783.tsp	783	12175	12045	8806
rat99.tsp	99	1489	1707	1211
rd100.tsp	100	10775	10357	7910
rd400.tsp	400	20717	20271	15281
rl1304.tsp	1304	365428	347720	252948
rl1323.tsp	1323	370204	380481	270199
st70.tsp	70	868	875	675
ts225.tsp	225	167439	187145	126643
tsp225.tsp	225	5461	5082	3919
u1060.tsp	1060	297125	302405	224094
u1432.tsp	1432	207767	214342	152970
u159.tsp	159	55510	57455	42080
u574.tsp	574	49810	49016	36905
u724.tsp	724	58134	57851	41910
vm1084.tsp	1084	331311	314871	239297
d1655.tsp	1655	84561	85714	62128
d1655.tsp	1655	84561	85714	62128
d2103.tsp	2103	93206	123322	[79952,80450]
fl1577.tsp	1577	30857	31218	[22204,22249]
pr2392.tsp	2392	527409	523731	378032
rl1889.tsp	1889	435267	450339	316536
u1817.tsp	1817	78960	83575	57201
u2152.tsp	2152	88581	95067	64253
u2319.tsp	2319	318100	320510	234256
vm1748.tsp	1748	474190	444439	336556
fl3795.tsp	3795	42831	39105	[28723,28772]
pcb3038.tsp	3038	186072	196398	137694

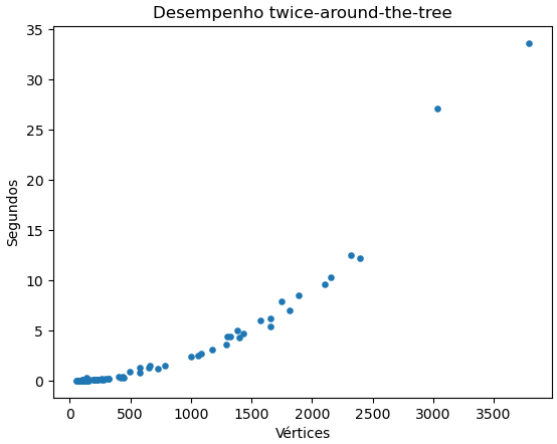
6 Discutindo os resultados

Em todas as intâncias, verificou-se que o algoritmo de Christofides respeitou o limite de custo de $1.5 \times c$, sendo c o custo da solução ótima. Da mesma forma o Twice-around-the-tree também respeitou o limite de custo de $2 \times c$, sendo c o custo da solução ótima. No total, 71 instâncias foram testadas em comum para ambos algoritmos.

A seguir, estão gráficos mostrando a relação de tempo de acordo com o número de cidades para

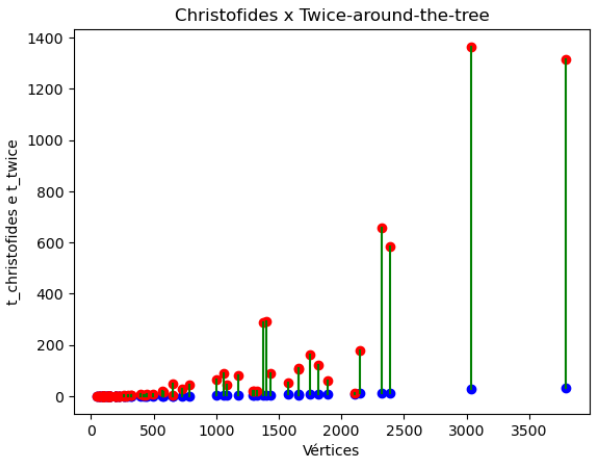
cada algoritmo.





Vê-se que o algoritmo de Christofides mostra uma relação menos contínua entre o número de vértices e o tempo, e isso se deve as características próprias de cada grafo, que variam no número de vértices impares na AGM gerada.

Podemos observar uma clara diferença no desempenho dos algoritmos, que fica ainda mais explicitada no gráfico a seguir. Eles são muito similares, visto que ambos retornam uma solução a partir da árvore geradora mínima. Ela é calculada usando a função `minimum_spanning_tree`, que por default usa o algoritmo de Kruskal e tem custo $O|E|log(|V|)$, e como o grafo é completo, $|E| = |V|^2$ e o custo é $O|V|^2log(|V|)$. O twice around the tree é dominado por essa complexidade.



Já o algoritmo de Christofides, além calcular a AGM usando Kruskal, ele calcula também o matching máximo de peso mínimo do subgrafo induzido pelos vértices de grau ímpar da AGM. A função utilizada para obter esse matching é `minimum_weight_matching`, da biblioteca, e usa o algoritmo de Blossom. Esse custo domina a complexidade de Christofides, e justifica seu tempo de execução superior. Além disso, ela tem complexidade espacial $O(|V|^2)$. Ou seja, também é espacialmente inferior. Por outro lado, isso que a permite gerar soluções mais próximas do ótimo. O gráfico ao lado explicita a diferença de desempenho dos algoritmos.

Devido a sua menor complexidade, foi possível testar o Twice-around-the-tree para mais instâncias. Indo até $|V| < 12.000$ Os resultados desse experimentos estão a seguir.

Instância	Vértices	Twice-around-the-tree	Ótimo	Tempo
fnl4461.tsp	4461	254116	182566	101
rl5915.tsp	5915	839645	[565040,565530]	160
rl5934.tsp	5934	822355	[554070,556045]	361

Calculando a relação entre os resultados e a solução ótima s^*, observou-se que o algoritmo Twice-around-the-tree retornou uma solução s, tal que $2s^* > s > 1,3s*$, e em média $1,7$ vezes mais cara que a solução ótima. Já o algortimo de Christofides ficou em uma média de $1,3$ vezes mais cara, com soluções no seguinte intervalo: $1,5s^* > s > 1,1s*$.

Dado tudo que foi observado, nos experimentos, percebemos claramente o trade-off entre soluções rápidas vs soluções ótimas. Em termos de qualidade, o branch and bound supera definitivamente as soluções aproximativas, mas seu custo é carísissmo, e portanto vale ser implementada em situções de maior abundância de recursos financeiros e/ou que requerem soluções exatas, especificamente.

Por outro lado as soluções aproximadas oferecem um equilíbrio entre tempo de execução e qualidade da solução, sendo mais adequadas para instâncias de maior tamanho. E comparando elas entre si, vemos que, em uma situação que se exige respostas o mais rápido possível, e em que a precisão pode ser pior, o Twice-around-the-tree se mostrou a melhor opção. Ademais, observei que em muitas instâncias, principalmente as menores, que ele retornou resultados pouco mais custosos que o Christofides. Porém, em instâncias maiores, a superioridade do Christofides ficou mais clara. Portanto, o Christofides é uma solução média, entre o branch and bound e o Twice around the tree.

7 Conclusão

Este trabalho ofereceu uma experiência desafiadora na implementação e análise de algoritmos complexos para o problema do caixeiro viajante. Ao desenvolver soluções para o TSP geométrico, utilizando tanto uma abordagem exata (branch-and-bound) quanto métodos aproximados (twice-around-the-tree e algoritmo de Christofides), foi possível vivenciar diretamente as dificuldades práticas e teóricas associadas à otimização em algoritmos. A escolha criteriosa de estruturas de dados, juntamente com a avaliação cuidadosa dos custos estimados, se mostrou importante para o desempenho dos algoritmos implementados.

A análise comparativa dos algoritmos, focando em tempo, espaço e qualidade das soluções, trouxe insights sobre as compensações entre precisão e eficiência computacional. Este estudo não apenas reforçou o entendimento teórico dos algoritmos do TSP, mas também sublinhou a importância de considerar aspectos práticos, como a eficiência computacional e limitações de recursos, ao se deparar com problemas de otimização na vida real. Assim, este trabalho contribui não só para o desenvolvimento técnico, mas também para a formação de uma visão crítica e aplicada na ciência da computação.