Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Relatório final para o programa IC (voluntário) – Edital 01/2019

Problema do Caixeiro Viajante

Aluno:
Marcelo Tranche de Souza Junior

Supervisor:
Carla Negri Lintzmayer

Setembro/2020

Resumo

No problema do Caixeiro Viajante temos um conjunto de cidades e queremos encontrar uma rota de comprimento mínimo que passa por todas elas exatamente uma vez. Esse é um problema clássico e central em otimização combinatória, com diversas aplicações práticas. Este projeto teve por objetivo a introdução do candidato à pesquisa científica por meio da implementação de diferentes algoritmos para o problema mencionado. Cada algoritmo envolve técnicas e conceitos diferentes, possibilitando assim ao aluno complementar sua formação em Ciência da Computação. Esse relatório descreve os detalhes de cada algoritmo estudado e de suas implementações, bem como de testes que foram realizados sobre eles para fins de comparações.

Sumário

1 Introdução

Otimização combinatória é uma área da ciência da computação que estuda problemas nos quais o objetivo é encontrar a melhor solução, ou a que mais se aproxima da melhor, dentro de um conjunto de domínio finito de soluções.

Problemas de otimização combinatória em geral são NP-difíceis, ou seja, impossíveis de serem resolvidos em tempo polinomial, a menos que P = NP, fazendo com que seja necessário utilizar outras técnicas para encontrar soluções razoáveis para eles, como por exemplo utilizar algoritmos de aproximação, que nos dão a garantia de encontrar soluções com custos próximos da solução ótima. Outras técnicas consistem no desenvolvimento de heurísticas, que dão uma solução para qualquer instância em tempo razoável porém sem garantias no custo da mesma, e no desenvolvimento de bons algoritmos exatos, que dão a solução ótima porém apenas para instâncias de tamanho pequeno.

Esta iniciação científica teve por objetivo a introdução do aluno à pesquisa científica, bem como a complementação de sua formação em Ciência da Computação, iniciando seu conhecimento na área de otimização combinatória por meio do estudo e implementação de diversos algoritmos para o problema do Caixeiro Viajante.

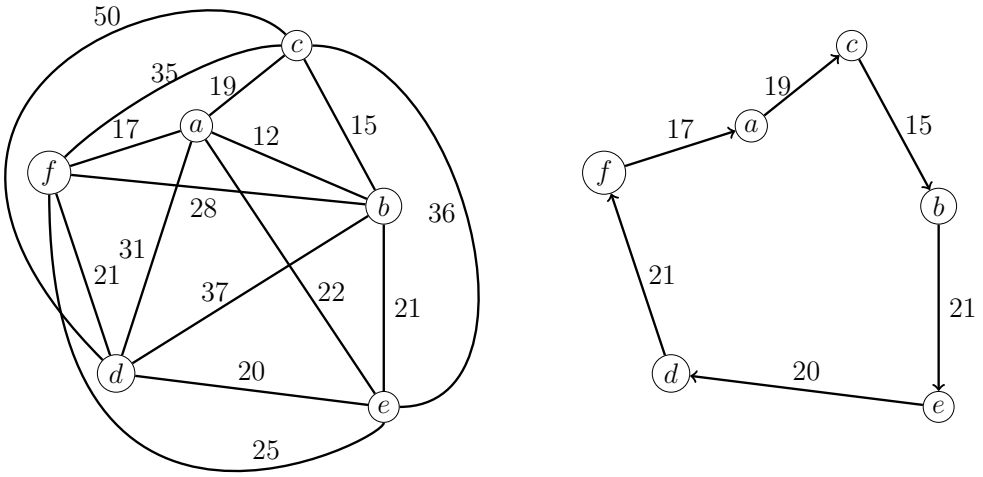
O restante desse relatório encontra-se dividido da seguinte forma. Na Seção 2 descrevemos formalmente o problema estudado. Na Seção 3 explicamos quais as técnicas e ferramentas que utilizamos para realizar o projeto. Na Seção 4 apresentamos os algoritmos utilizados para resolver o problema estudado. Na Seção 5 mostramos nossas implementações dos algoritmos estudados na linguagem de programação Python. Na Seção 6 mostramos os testes realizados com os algoritmos, e comparamos seus resultados em relação aos custos das soluções dadas, e com os tempos de execução de cada um. E por fim, na Seção 7 damos as considerações finais sobre o projeto.

2 O problema

Traveling salesman problem, ou o **problema do caixeiro viajante**, é um famoso problema NP-difícil [3, p. 799], que tem como objetivo achar o percurso de menor custo que um vendedor deve viajar, de maneira que ele visite todas as cidades uma única vez e depois volte para sua cidade de partida.

Formalmente, a entrada do problema pode ser modelada por um grafo não orientado completo $G = (V, E)$, onde cada vértice representa uma cidade e cada aresta ij possui um peso $c(ij)$, que representa o custo de viajar da cidade i à cidade j. Deseja-se encontrar um **ciclo hamiltoniano de custo mínimo** em G, ou seja, um percurso que começa em um vértice r e passa por cada um dos vértices do grafo uma única vez retornando ao vértice r e cuja soma das arestas percorridas seja a menor possível.

A Figura 1 mostra um exemplo de um possível grafo de entrada e uma solução ótima do



(a) Grafo G.

(b) Solução ótima do TSP para G, (a, c, b, e, d, f, a).

Figura 1: Exemplo de grafo de entrada e uma solução ótima.

TSP para o mesmo. Neste trabalho que os grafos são sempre grafos completos, isto é, que todo par de vértices possui aresta entre si.

3 Metodologia

Por ser um problema clássico e bem conhecido em otimização combinatória, o problema do caixeiro viajante possui muitas técnicas para se encontrar suas soluções, de forma que é possível encontrar diversas dessas técnicas em muitos artigos e livros [1, 2, 13, 15, 3]. Para esse projeto, buscou-se cobrir as soluções mais relevantes para o problema que pudessem introduzir o aluno à técnicas básicas de algoritmos. Por isso, escolhemos um algoritmo de cada técnica: gulosa, força bruta, programação dinâmica, *branch and bound*, aproximação, busca local e programação linear.

Além de estudar os algoritmos e resultados teóricos envolvidos, também decidimos por implementar cada um deles, para comparação prática. Para isso, a linguagem escolhida foi Python, devido à sua simplicidade e familiaridade do aluno. Python possui compatibilidade com as bibliotecas Networkx [9], própria para se trabalhar com grafos, e Gurobi [8], que fornece uma solução bastante completa para a implementação de soluções para problemas de otimização.

4 Algoritmos estudados

Nesta seção falaremos sobre os algoritmos que foram estudados. As Seções 4.1, 4.2, 4.3 e 4.4 apresentam algoritmos que devolvem soluções ótimas para o problema. Nota-se, portanto, que eles devem abrir mão de fazer isso em um tempo razoável, conforme a

entrada cresce. As Seções 4.5, 4.6, 4.7 e 4.8 apresentam algoritmos de aproximação e heurísticas para o problema. Uma x-aproximação é um algoritmo cuja solução tem custo no máximo x vezes o custo da solução ótima. Algoritmos heurísticos em geral não têm essa garantia de custo. Ambos, no entanto, têm tempo de execução polinomial e por isso executam para qualquer instância.

4.1 Força bruta

Carvalhaes e Rodrigues [12] dizem que um algoritmo de força bruta é feito testando e enumerando todas as possibilidades de soluções possíveis para um problema, ou seja, aplicando para o problema do caixeiro viajante, devemos testar todas as possíveis combinações de passeios, sendo essas combinações formadas por permutações do conjunto de vértices que resultam em um ciclo hamiltoniano válido, e escolher a de menor custo.

A ideia do algoritmo para o TSP é gerar todos os possíveis ciclos válidos sobre o grafo G através de permutações sobre seus vértices Note que essa técnica de solução só funciona se G for um grafo completo, senão ainda teríamos que verificar quais permutações formariam ciclos. Ele é formalizado no Algoritmo 1.

Algoritmo 1: FORCA-BRUTA

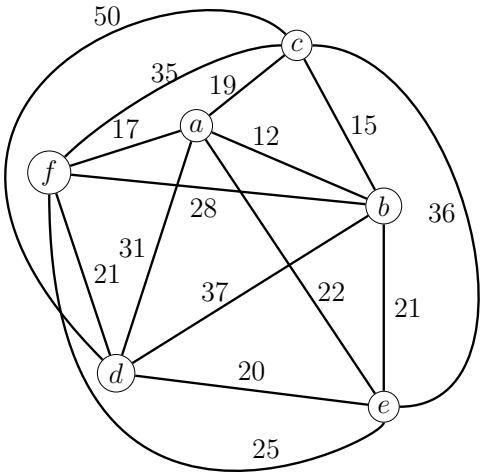
```
   Entrada: G = (V, E), c
1  início
2  |  OPT ← (0, 1, 2, ..., n − 1, 0)
3  |  para cada passeio ∈ PERMUTAÇÕES(V) faça
4  |  |  se c(passeio) ≤ c(OPT) então
5  |  |  |  OPT ← passeio
6  |  retorna OPT
```

Na linha 2 cria-se um passeio inicial para servir como base de comparação para as próximas soluções criadas pelas permutações. O laço da linha 3 utiliza a função PERMUTAÇÕES(V), que gera todas as possíveis permutações sobre V, para que depois na linha 4 se compare o custo dessa nova permutação com a solução de menor custo até o momento. Caso essa nova permutação tenha um custo menor, na linha 5 essa permutação se torna a nova solução de menor custo para ser comparada com as permutações subsequentes. Como iteramos por todas as permutações dos vértices e verificamos o custo de cada uma delas, esse algoritmo tem um tempo de execução $O(|V|!|V|)$.

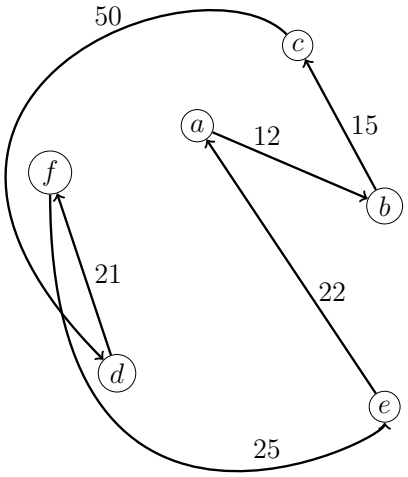
Um exemplo de execução desse algoritmo encontra-se na Figura 2.

4.2 Programação dinâmica

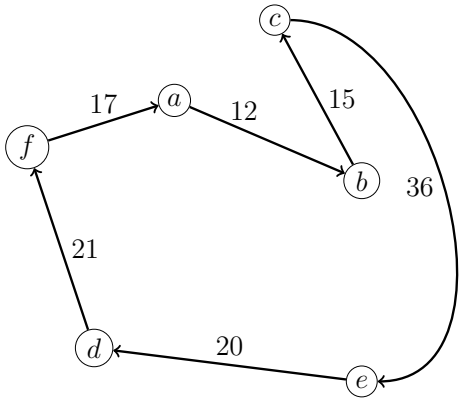
Bellman [1] apresentou o primeiro algoritmo de programação dinâmica para o TSP. Supondo que nosso grafo G tenha n vértices e que $V(G) = \{0, 1, \ldots, n-1\}$, a ideia é a seguinte. Dado um passeio ótimo sobre um grafo, considere que ele começa no vértice 0.



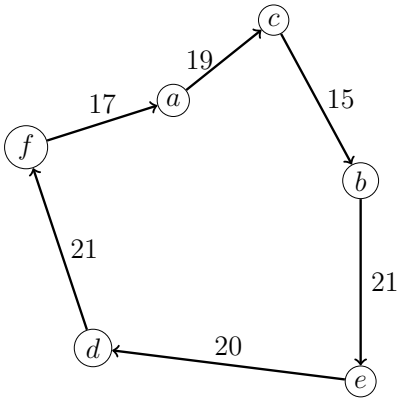
(a) Grafo G.



(b) Primeira permutação, (a, b, c, d, f, e, a).



(c) Segunda permutação, (a, b, c, e, d, f, a).



(d) Solução final, (a, c, b, e, d, f, a).

Figura 2: Exemplo da execução do algoritmo de força bruta.

Observando esse passeio até o vértice i, suponha que faltam k vértices $j_1, j_2, \ldots, j_k$ até retornar ao vértice de partida 0. Veja que o caminho de i até 0 passando por $j_1, \ldots, j_k, 0$ deve ser mínimo, uma vez que o passeio é ótimo. Vamos denotar então por $f(i; j_1, \ldots, j_k)$ o comprimento de tal caminho. Com isso temos que $f(0; j_1, \ldots, j_n)$ é o custo da solução ótima do TSP.

Considerando $c(i, j)$ como a distância entre os vértices i e j, e sendo $f(i; j) = c(i, j) + c(j, 0)$, a equação a seguir define a relação necessária para o algoritmo de programação dinâmica:

$$f(i; j_1, \ldots, j_k) = \min_{1 \leq m \leq k} \{c(i, j_m) + f(i; j_1, j_2, \ldots, j_{m-1}, j_{m+1} \ldots, j_k)\}, \tag{1}$$

sendo que a sequência de valores escolhidos de m que minimizam a equação nos dá o passeio ótimo em si.

A ideia do algoritmo, portanto, como todo algoritmo de programação dinâmica, é guardar esses valores da função f em memória para reutilizá-los rapidamente ao invés de recalculá-los.

4.3 *Branch and bound*

O algoritmo *branch and bound* consiste em testar todas as possíveis combinações válidas de passeios, porém durante a construção de cada nova combinação acontece uma checagem a cada adição de vértice, que compara o custo da nova combinação até o momento com o custo da melhor solução já completada anteriormente. Caso seja menor, guardamos essa nova combinação para comparar com as combinações subsequentes. Caso seja maior, não continuaremos a tentar construir o passeio atual. No final, teremos uma solução ótima por estarmos analisando todo o espaço de busca.

O Algoritmo 2 formaliza o procedimento. Ele recebe o grafo G sobre o qual queremos calcular o ciclo hamiltoniano de menor custo possível, uma função c de custo das arestas de G, uma variável *solAtual* que contém o passeio que está sendo construído no momento, uma variável *solOpt* que contém o ciclo completo de menor peso que foi construído até o momento, e recebe um vértice q que será o vértice inicial para se criar os caminhos. Na primeira chamada à essa função, enviamos *solAtual* contendo um único vértice inicial qualquer e *solOpt* contendo um passeio qualquer, só que considerando que todas as arestas desse passeio tenham o peso da maior aresta de G. Isso porque qualquer solução do problema terá que ter custo menor do que este.

A condição da linha 2 verifica se *solAtual* é hamiltoniano e se esse ciclo tem custo menor que o custo do *solOpt* e, caso sim, *solAtual* passa a ser a melhor solução, *solOpt*, e é devolvida na linha 4.

Na linha 7 temos uma condição que verifica se o custo de *solAtual* mais o custo da menor aresta de G vezes o número de arestas que falte a para a solução atual é menor que o

Algoritmo 2: BRANCH-AND-BOUND

Entrada: $G = (V, E)$, c, $solAtual$, $solOpt$

```
1  início
2      se solAtual tem todos os vértices e c(solAtual) < c(solOpt) então
3          solOpt ← solAtual
4          retorna solOpt
5      Seja e_min a aresta de menor custo em G
6      Seja x a quantidade de vértices de G que ainda não estão em solAtual
7      se c(solAtual) + xc(e_mmin) < c(solOpt) então
8          para cada vértice v que ainda não está em solAtual faça
9              se c(solAtual + v) < c(solOpt) e c(solAtual) + xc(e_min) < (solOpt)
                então
10                 solOpt ← BRANCH-AND-BOUND(G, c, solAtual + v, solOpt)
11     retorna solOpt
```

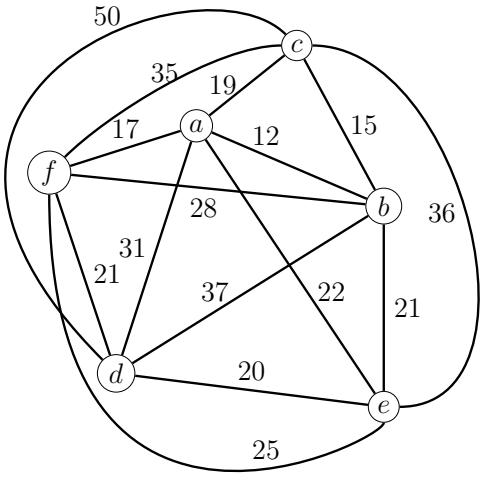
custo de $solOpt$, pois caso fosse maior, não seria necessário continuar a testar adicionar vértices a $solAtual$. Caso essa condição seja verdadeira, entramos no laço da linha 8 que é responsável por verificar todos os vértices que não estão em $solAtual$, para que na condição da linha 9 seja verificado quais desses vértices, se adicionados à $solAtual$, ainda deixam o custo menor que o custo de $solOpt$. Se o custo for menor, então na linha 10 faz-se uma chamada recursiva com os parâmetros G, c, $solAtual + v$, que é $solAtual$ com a adição de v ao fim, e $solOpt$. Um exemplo de execução desse algoritmo encontra-se na Figura 3.

Veja que o tempo de pior caso do *branch and bound* vai ser o mesmo do algoritmo de força bruta, pois ainda temos a chance de ter que percorrer todas as soluções possíveis.

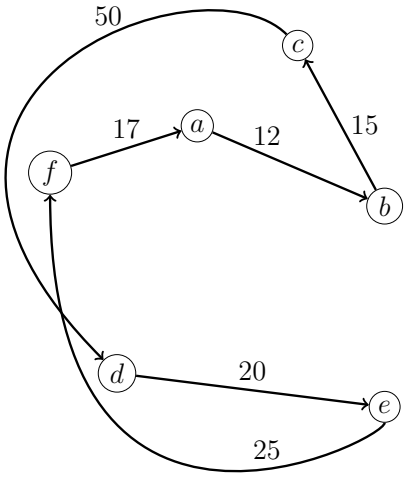
4.4 Programação linear

Programação linear é uma maneira formal de descrever um problema de otimização, seja ele de maximização (*Programa linear de maximização*) ou de minimização (*Programa linear de minimização*). De forma geral, todo programa linear tem um conjunto de **variáveis de decisão**, que indicam as escolhas feitas para chegar a uma solução, de **restrições**, que descrevem as características do problema, e uma **função de objetivo**, que indica o custo das soluções e se o programa linear é de maximização ou de minimização. A função de objetivo e as restrições de um programa linear são representadas por equações ou inequações lineares.

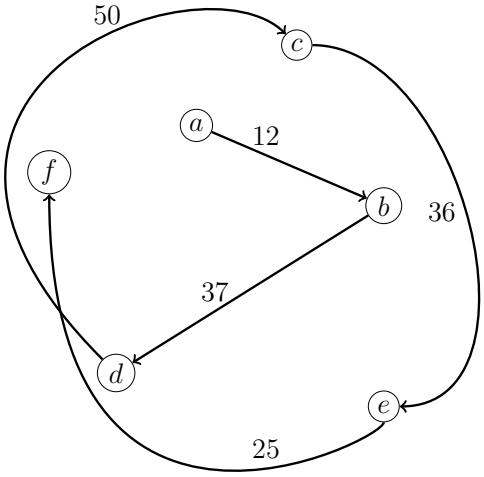
Para o problema do caixeiro viajante definimos seu programa linear da seguinte forma. Seja x_{ij} uma variável cujo valor é 1, se a aresta $ij \in E$ for escolhida para fazer parte do



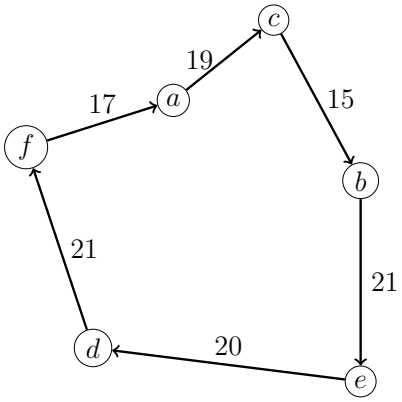
(a) Grafo G.



(b) Primeira solução completa, (a, b, c, d, e, f, a).



(c) Corte antes de completar o ciclo, (a, b, d, c, e, f).



(d) Solução final, (a, c, b, e, d, f, a).

Figura 3: Exemplo de execução do algoritmo *branch and bound*.

ciclo, ou é 0, caso contrário. Considere o seguinte programa linear inteiro [6]:

$$\text{minimizar} \quad \sum_{j=1}^{n}\sum_{i=1}^{n} c(ij)x_{ij} \tag{2}$$

$$\text{sujeito a} \quad \sum_{i=1}^{n} x_{ij} = 2 \qquad \forall j \in V \tag{3}$$

$$\sum_{i,j\in S} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq \emptyset \tag{4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V, i < j \tag{5}$$

Note que ele formaliza o problema do caixeiro viajante: a função objetivo em (2) indica que queremos minimizar o custo das arestas escolhidas, a restrição (3) garante que cada vértice será visitado uma vez pelo ciclo (pois duas arestas devem sair desse vértice), e a restrição (4) garante que não serão formados ciclos que não sejam hamiltonianos (todo subconjunto de vértices de G que não seja o próprio V deve ter no máximo $|S| - 1$ arestas ligando seus vértices – dessa forma, não é possível S conter um ciclo).

Infelizmente, note que existem $O(2^{|V|})$ restrições do tipo (4), uma para cada possível subconjunto de vértices, fazendo que essa formulação seja inviável conforme o tamanho do grafo aumenta. Existem técnicas de implementação que permitem contornar essa situação.

Programas lineares cujas variáveis que formulam o problema são contínuas pode ser resolvido em tempo polinomial, enquanto programas lineares inteiros (PLI), que é uma classe dos programas lineares cujas variáveis podem apenas assumir valores inteiros, são NP-difíceis [14], ou seja, são problemas cuja a solução ótima provavelmente não pode ser encontrada em tempo polinomial. Como visto, as variáveis que definem o problema do caixeiro viajante só podem assumir valores binários, por isso sua formulação está dentro da classe do PLI, o que é condizente com o fato de este ser um problema NP-difícil.

4.5 Algoritmo guloso

Parberry [10] define um algoritmo guloso como, dada uma solução parcial para um subproblema, deve-se aumentá-la sucessivamente para solucionar o problema completo, sendo que esse aumento deve ser feito de maneira gulosa, ou seja, sempre escolhendo a opção que no momento seja mais vantajosa. Para o problema do caixeiro viajante, a ideia é construir um passeio fazendo escolhas gulosas pelo próximo vértice.

Inicialmente temos um vértice inicial qualquer no passeio. O próximo vértice será o vizinho de menor peso do último vértice que foi adicionado ao passeio e que ainda não esteja presente no passeio. Isso é repetido até que todos os vértices do grafo estejam presentes na solução. O Algoritmo 3 formaliza essa ideia.

Na linha 2 cria-se a variável *verticeAtual*, que começa primeiramente como um vértice

Algoritmo 3: GULOSO

Entrada: $G = (V, E)$, c

```
1  início
2      verticeAtual ← vértice qualquer
3      passeio ← (verticeAtual)
4      repita
5          novoVertice ← qualquer vizinho de verticeAtual
6          para cada vizinho u de verticeAtual que não está em passeio faça
7              se c(u, verticeAtual) < c(novoVertice, verticeAtual) então
8                  novoVertice ← u
9          passeio ← passeio + novoVertice
10         verticeAtual ← novoVertice
11     até V \ passeio = ∅;
12     retorna passeio
```

qualquer e, conforme a execução do algoritmo, vai sendo atualizada para ser o último vértice adicionado ao passeio. Esse passeio será armazenado em *passeio*, que é inicializado com *verticeAtual*.

No laço da linha 4 acontece a construção do passeio de forma gulosa, que só termina quando o passeio conter todos os vértices de G. Na linha 6 pegamos o vizinho que ainda não está no passeio e é mais próximo do último vértice adicionado, para que na linha 9 adicionemos esse vizinho ao passeio. Como mostrado na linha 10 esse vizinho passa a ser o novo *verticeAtual*, portanto. Um exemplo de execução desse algoritmo encontra-se na Figura 4.

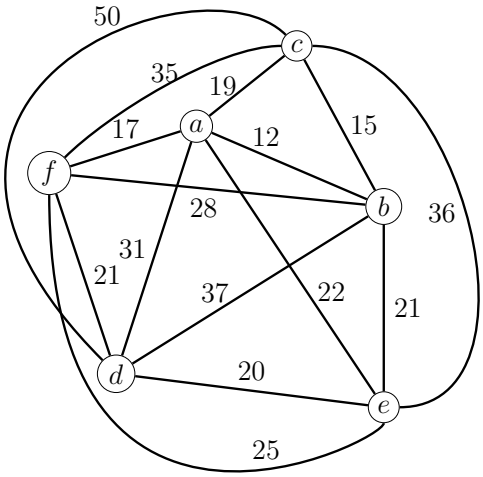
Como iteramos por todos os vértices de G para adicionarmos eles no caminho, e em cada iteração checamos todos os possíveis vizinhos de um vértice, no pior caso iteramos por $|V| - 1$ vértices, esse algoritmo tem um tempo de execução $O(|V|^2)$.

4.6 Algoritmo *Cheapest insertion*

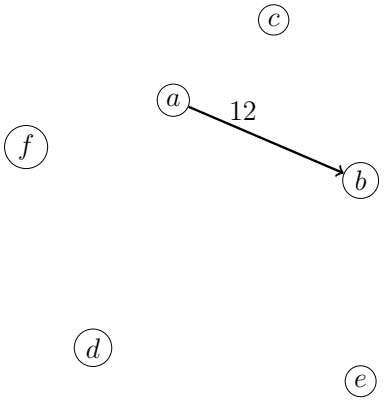
O algoritmo *Cheapest insertion* é uma 2-aproximação que se baseia em um método apresentado por Nicholson [7] e analisado por Rosenkrantz *et al.* [13]. Ele recebe o grafo G, uma função de custo c sobre as arestas e um vértice inicial q (pode ser escolhido aleatoriamente).

Esse algoritmo é uma 2-aproximação para casos do problema do caixeiro viajante métrico, isso é, grafos em que o custo de ir de um vértice u a um vértice w não é maior do que o custo de um caminho alternativo que, antes de chegar a w, passa por um vértice intermediário v. Formalmente, dizemos que G é métrico se para todo trio de vértices u, v, w temos que $c(u, w) \leq c(u, v) + c(v, w)$.

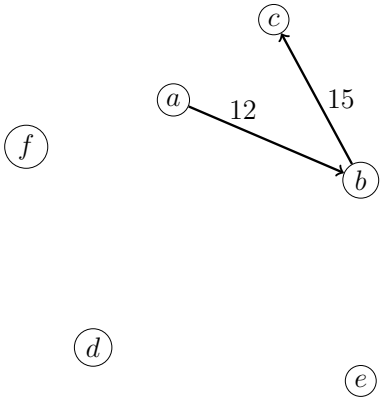
Mesmo assim, o algoritmo *Cheapest insertion* ainda pode ser usado como heurística para resolver o problema para casos não métricos, porém não será garantido que ainda será uma 2-aproximação.



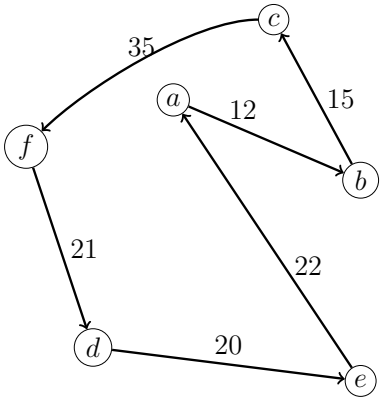
(a) Grafo G.



(b) Primeira aresta adicionada, (a, b).



(c) Segunda aresta adicionada, (a, b, c).



(d) Solução final, (a, b, c, f, d, e, a).

Figura 4: Exemplo de execução do algoritmo guloso

.

A ideia do algoritmo é manter um ciclo T sobre um conjunto de vértices e aumentar T, para que eventualmente ele vire um ciclo hamiltoniano, escolhendo um vértice que não está em T da seguinte forma. Seja ij uma aresta do ciclo T atual e x um vértice que não está em T. O ciclo T pode ser aumentado trocando a aresta ij pelas arestas ix e xj. Esse ciclo T é chamado *subtour*. Ele está formalizado no Algoritmo 4.

Algoritmo 4: CHEAPEST-INSERTION

Entrada: $G = (V, E)$, c, q

1 **início**
2 Seja r um vértice de G que minimize $c(qr)$
3 $T \leftarrow (q, rq)$
4 **para cada** $x \in V \setminus V(T)$ **faça**
5 Seja $(i, j) \in T$
6 Insira x em T, entre i e j, que minimize $c(ix) + c(xj) - c(ij)$
7 **retorna** T

Na linha 3 inicia-se um *subtour* que começa e termina no vértice inicial q e passa pelo vértice r, sendo esse o vértice que proporcionará o menor valor $c(qr)$. O laço da linha 4 seleciona vértices x que estão no grafo de entrada G mas que ainda não estão em T, e escolhe arestas ij de T de maneira que inserindo x entre i e j minimize $c(ix)+c(xj)-c(ij)$. Assim, os vértices inseridos em T tentam aumentar seu custo da menor maneira possível. Cada repetição do laço cria um novo *subtour* de acordo com o método de inserção descrito por Nicholson [7], até que todos os vértices de V estejam dentro do *tour* T. Um exemplo de execução do algoritmo pode ser visto na Figura 5.

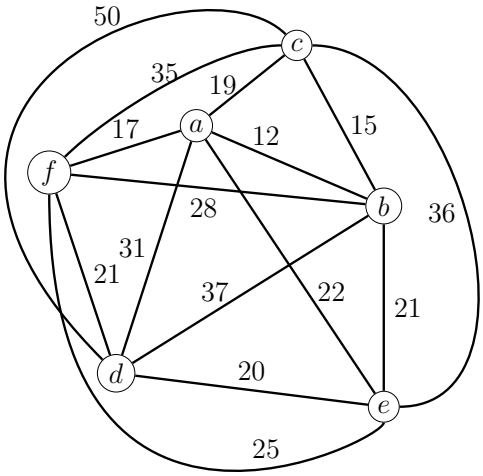
Como iteramos por todos os vértices de G para irmos inserindo eles no ciclo, e para cada inserção nós ainda iteramos entre todos os pares vértices fora do ciclo para encontrar a menor aresta para que temos a melhor inserção, cada inserção tem um tempo de $O(|V|^2)$, de forma que esse algoritmo tem um tempo de execução $O(|V|^3)$.

4.7 Algoritmo de Christofides

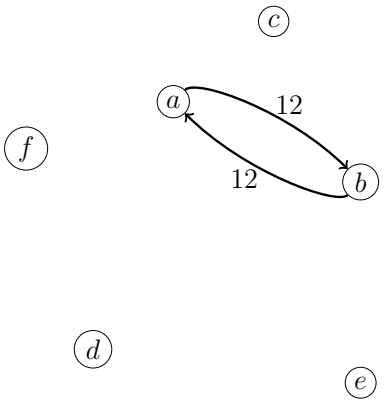
O algoritmo de Christofides [2] é uma $\frac{3}{2}$-aproximação, sendo até o momento o algoritmo com melhor fator de aproximação para o problema do caixeiro viajante.

Note que, como o algoritmo mostrado na Seção 4.6, a aproximação do algoritmo de Christofides só vale para grafos métricos, sendo ainda possível usá-lo como heurística para casos não métricos, porém sem a garantia da $\frac{3}{2}$-aproximação.

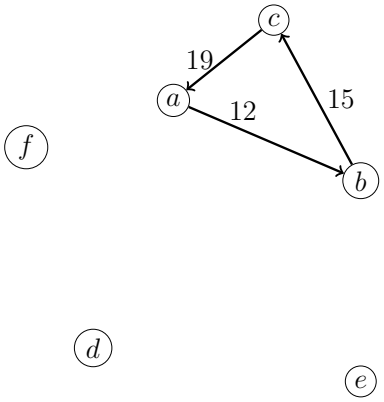
Ele recebe o grafo G e uma função de custo c sobre as arestas. A ideia é construir um ciclo hamiltoniano por meio de outro subgrafo hamiltoniano: uma árvore geradora, que é um subgrafo cuja as arestas tocam todos os vértices do grafo que o gerou, sendo então uma árvore geradora mínima um subgrafo que consegue ter as arestas tocando todos os vértices, mas com o menor custo possível. Existem algumas formulações de algoritmos para realizar essa tarefa, nesse projeto utilizou-se o algoritmo de Kruskal.



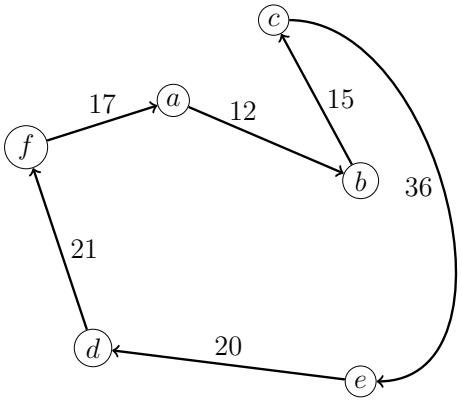
(a) Grafo G.



(b) Ciclo inicial, (a, b, a).



(c) Primeira inserção, (a, b, c, a).



(d) Solução final, (a, b, c, e, d, f, a)

Figura 5: Exemplo de execução do algoritmo *Cheapest insertion*.

Partindo da árvore geradora mínima, encontramos um emparelhamento perfeito de custo mínimo sobre os vértices de grau impar. Um emparelhamento pode ser definido como a seleção de arestas cujos extremos não se tocam, ou seja, arestas que não possuem vértices compartilhados entre si, sendo que o emparelhamento é dado como perfeito quando todos os vértices dados têm arestas que os tocam. Como o grafo é completo e existe um número par de vértices de grau ímpar na árvore, esse emparelhamento realmente existe.

A partir da adição da árvore geradora mínima com o emparelhamento perfeito, deve-se encontrar uma trilha Euleriana sobre esse novo grafo. Uma trilha Euleriana é um passeio que irá visitar todas as arestas sem repetir nenhuma delas, porém, muito provavelmente, vértices serão repetidos.

A partir da trilha feita, devemos fazer *shortcuts* sobre ela para no fim gerar um ciclo Hamiltoniano. *Shortcuts* são úteis para remover trechos já visitados em *passeios*. Dado um passeio $(a_1, a_2, \ldots, a_k)$ em um grafo, onde cada a_i é um vértice e todo $a_i a_{i+1}$ é aresta do grafo, definimos um *shortcut* como a substituição de um trecho $a_i, a_{i+1}, \ldots, a_j$ pela aresta direta $a_i a_j$.

O algoritmo de Christofides é formalizado no Algoritmo 5.

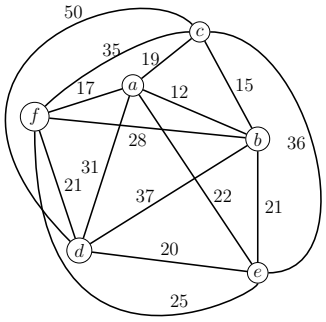
Algoritmo 5: CHRISTOFIDES

Entrada: $G = (V, E), c$

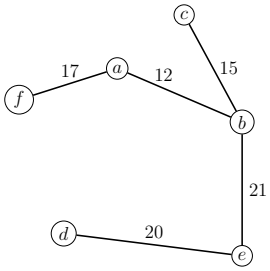
1 **início**
2 $T \leftarrow$ ÁRVOREGERADORAMÍNIMA(G, c)
3 $M \leftarrow$ Emparelhamento perfeito de custo mínimo sobre os vértices de grau ímpar de T
4 $F \leftarrow$ Trilha Euleriana sobre $T + M$
5 $H \leftarrow$ Ciclo Hamiltoniano fazendo *shortcuts* em F
6 **retorna** H

Na linha 2 o algoritmo chama uma função que retorna uma árvore geradora mínima do grafo G e armazena essa árvore em T. Na linha 3 é criado um emparelhamento apenas sobre os vértices de grau ímpar da árvore T. Isso é feito pois no grafo $T + M$ todos os vértices possuem grau par. Assim, na linha 4 podemos criar uma trilha Euleriana F sobre $T + M$. A partir da trilha Euleriana formada, criamos um ciclo Hamiltoniano H eliminando vértices que se repetem em F, da mesma forma do algoritmo anterior. A Figura 6 dá um exemplo de como o algoritmo funciona.

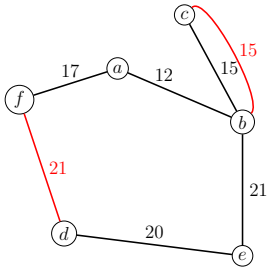
Sabemos que o tempo de execução do algoritmo de Kruskal para encontrar a árvore geradora mínima de G é $O(|E| \log |V|)$ [3], do algoritmo de emparelhamento perfeito é $O(|V|^2|E|)$ [5], para encontrar a trilha Euleriana é $O(|E|)$ e, por fim, para realizar todos os *shortcuts* leva-se o tempo $O(|V|)$. Com isso, o tempo total do algoritmo é $O(|V|^2|E|)$.



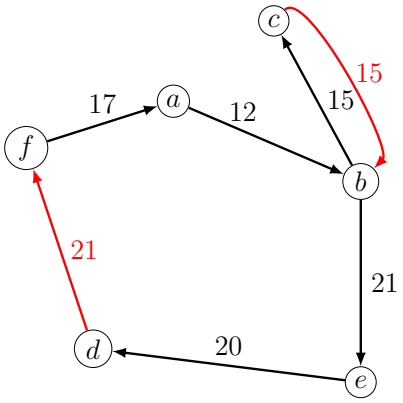
(a) Grafo G.



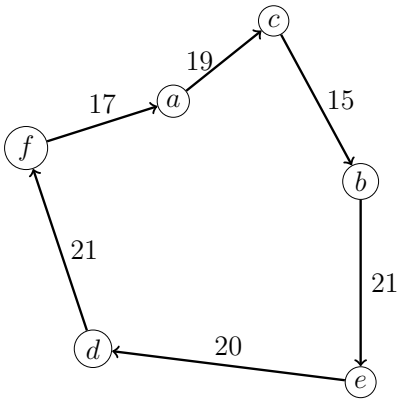
(b) Árvore geradora mínima T.



(c) T+M.

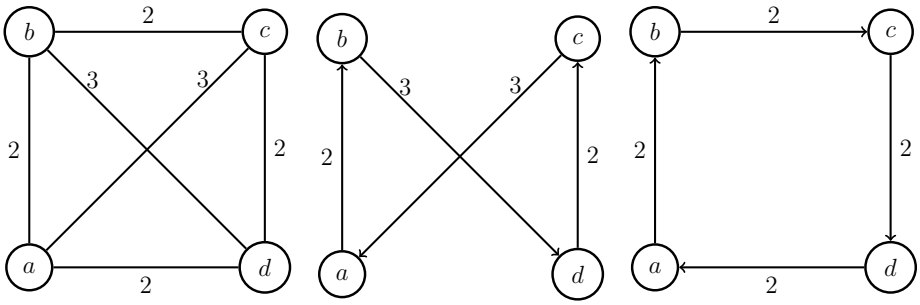


(d) Trilha euleriana $F = (a, b, c, b, e, d, f, a)$.



(e) Ciclo final $H = (a, c, b, e, d, f, a)$.

Figura 6: Exemplo de execução do algoritmo de Christofides.



(a) Grafo G.

(b) Passeio inicial em G, (a, b, d, c, a).

(c) Troca das arestas bd e ac por ad e bc, (a, b, c, d, a).

Figura 7: Exemplo de como acontece a troca de arestas do algoritmo 2-Opt.

4.8 Algoritmo 2-Opt

O algoritmo 2-Opt for proposto por Croes [4] e ele é uma heurística cujo objetivo é melhorar o custo de uma solução não ótima através de trocas de duas arestas do passeio por outras duas, de maneira que reduza ao máximo o peso da solução atual.

A Figura 7 mostra um exemplo de como essa troca de arestas acontece. Vale ressaltar que no exemplo dado não há outra possível troca de arestas que diminua o peso da solução depois da primeira troca, por isso o algoritmo terminaria com essa solução, porém, caso houvesse uma troca que diminuísse o custo desse passeio final, o algoritmo continuaria até que não houvesse mais trocas que abaixassem o seu custo.

O Algoritmo 6 mostra como fazer a troca de duas arestas. Ele recebe o grafo G, a função de custo c, um ciclo hamiltoniano sobre G e duas arestas e_1 e e_2 que estão no ciclo. A ideia é realizar as duas possíveis trocas das arestas e_1 e e_2 que ainda resulte em um ciclo completo, e devolver um ciclo válido gerado pela troca que seja de menor custo.

Algoritmo 6: Troca-Duas-Arestas

Entrada: $G = (V, E)$, c, $ciclo$, e_1, e_2

```
1 início
2 |  Sejam a_{e_i} e b_{e_i} os extremos da aresta e_i
3 |  troca_1 ← ciclo − e_1 − e_2 + a_{e_1}a_{e_2} + be_1b_{e_2}
4 |  troca_2 ← ciclo − e_1 − e_2 + a_{e_1}b_{e_2} + ae_2b_{e_1}
5 |  se c(troca1) ≤ c(troca2) então
6 |  |  retorna troca1
7 |  retorna troca2
```

Nas linhas 3 e 4 realizamos as trocas de arestas, primeiro removendo do ciclo as duas arestas de entrada do algoritmo, para depois adicionar arestas com os extremos das duas arestas removidas.

Após isso, na linha 5 vemos qual das trocas nos dá o ciclo de menor peso para o retornamos depois.

O algoritmo completo, 2-Opt, é mostrado no Algoritmo 7. Ele recebe o grafo G de entrada, a função c de custo nas arestas de G e um ciclo já G como o grafo que iremos calcular o ciclo de menor custo possível, c que é a função de custo das arestas de G, e por fim um *ciclo* não ótimo que tentaremos melhorar com a função de troca.

Algoritmo 7: 2-Opt

```
   Entrada: G = (V, E), c, ciclo
1  início
2  |  trocou ← 1
3  |  repita
4  |  |  trocou ← 0
5  |  |  para cada aresta e1 ∈ ciclo faça
6  |  |  |  para cada aresta e2 ∈ ciclo − e1 faça
7  |  |  |  |  cicloTroca ← Troca-Duas-Arestas(G, c, ciclo, e1, e2)
8  |  |  |  |  se c(cicloTroca) < c(ciclo) então
9  |  |  |  |  |  trocou ← 1
10 |  |  |  |  |  ciclo ← cicloTroca
11 |  até trocou = 0;
12 |  retorna ciclo
```

Na linha 2 se inicia a variável *trocou*, que será utilizada no laço da linha 3 para sinalizar quando não for possível melhorar a solução atual com a função de troca. Na linha 9 mudamos o valor da variável *trocou* para 1, caso exista uma troca no passeio atual que diminua seu custo. Assim, se essa linha não for executada, a variável segue com o valor 0 e o laço é interrompido.

Nos dois laços das linhas 5 e 6, realizamos todas as possíveis combinações de arestas do *ciclo*, para que na linha 7 possamos testar todas as possíveis trocas de arestas feitas pela função Troca-Duas-Arestas.

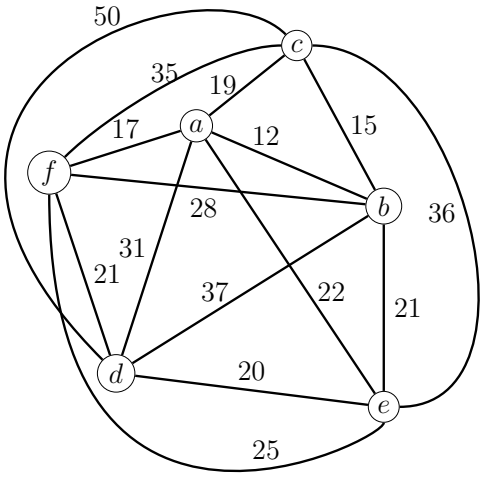
Na condição da linha 8 verificamos se a troca feita resultou em uma solução de custo melhor do que o custo da solução atual e, caso sim, essa se torna a nova solução atual, e atualizamos a variável *trocou*.

Um exemplo de execução desse algoritmo é dado na Figura 8.

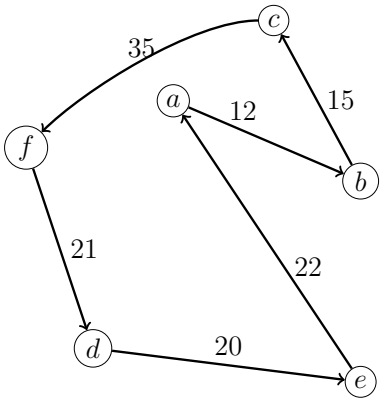
Para compararmos todos os pares de aresta do ciclo inicial e fazer a troca leva o tempo de $O(|V|^2)$, porém, como não sabemos quantas vezes essa troca deve ser feita até chegar na solução final, aquela que não é possível melhorar com uma troca, não é possível determinar o tempo do algoritmo completo.

5 Implementações

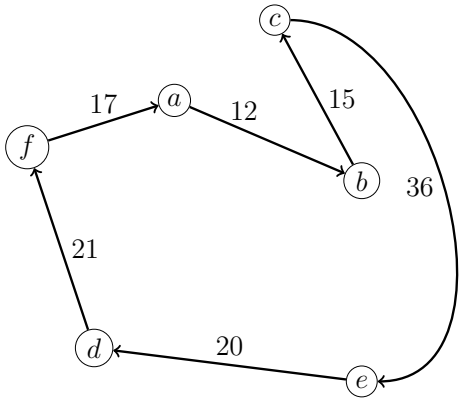
Como dito na introdução, todos os algoritmos citados anteriormente foram implementados na linguagem de programação Python, sendo usada principalmente o Networkx [9], uma biblioteca que fornece recursos que facilitam a criação e manipulação de grafos.



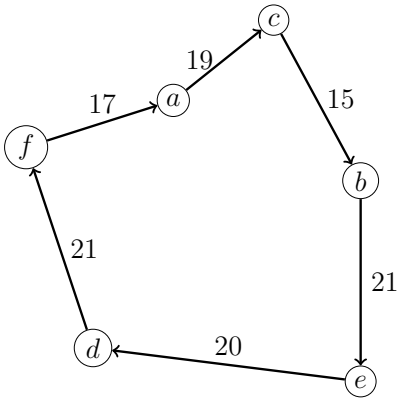
(a) Grafo G.



(b) Solução final do algoritmo guloso, (a, b, c, f, d, e, a).



(c) Troca $(a - e)$ com $(c - f)$, ficando (a, b, c, e, d, f, a).



(d) Troca $(a - b)$ com $(c - e)$, ficando (a, c, b, e, d, f, a) (solução final).

Figura 8: Exemplo de execução do algoritmo 2-Opt.

O Código 1 apresenta nossa implementação do algoritmo de força bruta, descrito na Seção 4.1. Ele faz uso da função `gen_tsp`, que recebe um objeto genérico que representa um ciclo Hamiltoniano, e retorna um ciclo formatado para o TSP, ajudando assim na padronização das soluções dos algoritmos.

Veja que para implementar a geração de todas as permutações do conjunto de vértices usamos uma função `permutacao`, que recebe uma sequência numerada de vértices, e de maneira sequencial realiza uma troca entre o primeiro valor, da direita para esquerda, da posição $i-1$ que seja maior que o valor da posição i, com o primeiro valor na posição j, também da direita para esquerda, que seja menor que $i-1$, e após isso, invertemos essa lista da posição i para frente. Isso faz com que cada vez que a função seja chamada com uma permutação qualquer, ela devolva a próxima permutação, em ordem lexicográfica.

Código 1: Implementação do algoritmo de força bruta.

```
import networkx as nx

# permutacao():
# recebe sequencia qualquer (lista)
# devolve proxima sequencia da permutacao (lista)
def permutacao(p):

    # achando o valor da posição i-1 que seja maior que o valor da posição i
    i = len(p) - 1
    while p[i-1] > p[i]:
        i -= 1

    # achando o valor da posição j que seja menor que i-1
    j = len(p) - 1
    while p[j] < p[i-1]:
        j -= 1

    # trocando valores das posicoes i-1 e j
    aux = p[i-1]
    p[i-1] = p[j]
    p[j] = aux

    # invertendo a list de i para frente
    aux_list = p[i:]
    aux_list.reverse()
    p[i:] = aux_list

    return p

def forca_bruta(G):
    vertices = list(G.nodes)
    # vértice 0 é o ponto de inicio, e não fará parte da permutação
    vertices.remove(0)

    # gera ciclo inicial
    opt_tsp = gen_tsp(G, vertices)
    ciclo_opt = vertices

    perm = permutacao(vertices.copy())

    # laço que termina quando a permutacao gerada for igual à solução inicial
    while perm != vertices:
        tsp = gen_tsp(G, perm)

        if tsp.size(weight='weight') < opt_tsp.size(weight='weight'):
            opt_tsp = tsp
            ciclo_opt = perm.copy()

        perm = permutacao(perm)

    return opt_tsp
```

O Código 2 apresenta nossa implementação do algoritmo *branch and bound*, descrito na Seção 4.3. Ele utiliza uma estrutura recursiva, que a cada chamada de recursão passa um novo passeio atual, e verifica se esse novo passeio é completo e menor que a melhor solução até o momento, e caso seja, atualiza a melhor solução. Caso não esteja completo ainda, verifica quais vértices ainda podem ser adicionados a ele, contanto que haja chances de melhorar o custo da melhor solução.

Código 2: Implementação do algoritmo de *branch and bound*.

```
import networkx as nx

def bbtsp(G, sol=[], custo_sol=0, melhor_custo=0, melhor_sol=[], menor_aresta=-1):
    # caso ainda não exista solução, inicia uma
    if sol == []:
        arestas = sorted(g.edges(data = True),
                         key = lambda t: t[2].get('weight', 1))

        maior_aresta = arestas[-1][2]['weight']
        menor_aresta = arestas[0][2]['weight']

        melhor_custo = maior_aresta*(G.number_of_nodes() - 1)
        sol = [0]

    # se a solução atual está completa, comparar com a melhor solução até o momento
    if G.number_of_nodes() == len(sol):
        if custo_sol + G[sol[-1]][0]['weight'] < melhor_custo:
            melhor_custo = custo_sol + G[sol[-1]][0]['weight']
            melhor_sol = sol + [0]

            return melhor_sol, melhor_custo

    # tenta adicionar vértices à solução atual
    for v in G.nodes:
        if v not in sol:
            # só gera soluções cujo custo seja menor que o melhor custo até o momento
            if custo_sol + G[sol[-1]][v]['weight'] < melhor_custo and
              custo_sol + menor_aresta*(G.number_of_nodes()-len(sol)) < melhor_custo:
                nova_solucao = bbtsp(G,
                                     sol + [v],
                                     custo_sol + G[sol[-1]][v]['weight'],
                                     melhor_custo,
                                     melhor_sol,
                                     menor_aresta)
                melhor_sol, melhor_custo = nova_solucao

    return melhor_sol, melhor_custo
```

O Código 3 apresenta nossa implementação do programa linear, descrito na Seção 4.4, com o Gurobi [8]. Como mencionamos, implementar o programa linear que foi apresentado na Seção 4.4 de forma direta não seria viável, pois existem $O(2^{|V|})$ restrições nele. Ao invés disso, a implementação usa restrições *lazy* que são restrições que muito provavelmente não serão violadas, e que só serão aplicadas quando for realmente necessário, sendo que não são especificadas no inicio do programa, sendo apenas utilizadas quando forem requiridas. Para o problema do caixeiro viajante utilizamos uma restrição *lazy* para a restrição que impede que se formem subciclos, e assim, ela não é executada a todo momento, economizando um tempo considerável de execução, tornando a implementação da programação linear viável. No código, utilizamos as funções já prontas `subtourelim` e `subtour`[1].

Código 3: Implementação do algoritmo de programação dinâmica.

```
import networkx as nx
```

[1] https://www.gurobi.com/documentation/9.0/examples/tsp_py.html

```
from gurobipy import GRB
from gurobipy import tuplelist
from gurobipy import quicksum
from gurobipy import Model

from itertools import combinations

# Copyright 2020, Gurobi Optimization, LLC

# https://www.gurobi.com/documentation/9.0/examples/tsp_py.html
def subtourelim(model, where):
    if where == GRB.Callback.MIPSOL:
        # verifica as arestas selecionadas pela solução
        vals = model.cbGetSolution(model._vars)
        selected = tuplelist((i, j) for i, j in model._vars.keys() if vals[i, j] > 0.5)

        # atraves das arestas selecionadas, geramos um ciclo
        tour = subtour(selected)
        if len(tour) < n:
            # restricao lazy para nao formar subciclos
            model.cbLazy(quicksum(model._vars[i, j] for i, j in combinations(tour, 2)) <= len(tour)-1)

# subtour():
# recebe arestas selecionadas (gurobipy.tuplelist)
# devolve melhor subciclo sobre as arestas selecionadas (lista)
def subtour(edges):
    # lista de vertices que ainda nao foram incluidos no ciclo
    unvisited = list(range(n))
    # iniciando ciclo generico para comparacao inicial
    cycle = range(n+1)

    # sai do laco quando a lista de vertices nao incluidos no ciclo estiver vazia
    while unvisited:
        # inicia ciclo auxiliar
        thiscycle = []

        # copiamos a lista de vertices fora do ciclo como vizinhos
        neighbors = unvisited

        # sai do laco quando a lista de vizinhos estiver vazia
        while neighbors:
            # definimos o vertice atual do ciclo
            current = neighbors[0]

            # adicionamos o vertice atual ao ciclo auxiliar
            thiscycle.append(current)

            # removemos o vertice atual da lista de vertices nao incluidos no ciclo
            unvisited.remove(current)

            # atualizamos os vizinhos como todos os vertices vizinhos do vertice atual
            # que ainda nao foram incluidos no ciclo auxilias
            neighbors = [j for i, j in edges.select(current, '*') if j in unvisited]

        # se o ciclo auxiliar for tiver custo menor que o ciclo principal, ele se torna
        # o ciclo principal
        if len(cycle) > len(thiscycle):
            cycle = thiscycle
    return cycle

def tsp_programacao_linear(G):
    tsp = nx.Graph()

    # tranformando grafo em uma matriz de adjacências
    mat = {(e[0], e[1]): e[2]['weight'] for e in G.edges(data=True)}

    # criando modelo de programacao linear
    model = Model()

    # adicionando uma variável para cada aresta do grafo (binárias, valendo 1
    # caso a aresta seja escolhida e 0 caso contrário)
    edges = model.addVars(mat.keys(), obj=mat, vtype=GRB.BINARY, name='edges')

    # como não é um grafo orientado, garantimos que a aresta é a mesma, independente
    # da ordem dos vértices
```

```
    for i, j in edges.keys():
        edges[j, i] = edges[i, j]

    # restrição para que cada vértice tenha duas arestas ligadas a ele
    model.addConstrs(edges.sum(i, '*') == 2 for i in range(n))

    # dizendo ao modelo que iremos utilizar lazy constraints
    model.Params.lazyConstraints = 1

    # iniciando otimização
    model._vars = edges
    model.optimize(subtourelim)

    vals = model.getAttr('x', edges)
    selected = tuplelist((i, j) for i, j in vals.keys() if vals[i, j] > 0.5)

    # transformando as variáveis selecionadas em um grafo com a solução
    for e in selected:
        tsp.add_edge(e[0], e[1], weight = G[e[0]][e[1]]['weight'])

    return tsp
```

O Código 4 apresenta nossa implementação do algoritmo guloso, descrito na Seção 4.5. Sua implementação é bem direta quando comparada ao pseudocódigo apresentado.

Código 4: Implementação do algoritmo guloso.

```
import networkx as nx

def guloso(G):
    tsp = nx.Graph()

    vertices_visitados = [0]*G.number_of_nodes()
    vertice_atual = 0

    # adicionando novas arestas até que todos os vertices sejam visitados
    vertices_visitados[vertice_atual] = 1
    while 0 in vertices_visitados:
        # iremos adicionar o vizinho de menor custo do vertice atual
        vizinhos_atual = [(vertice_atual, i) for i in G.neighbors(vertice_atual)
                                             if vertices_visitados[i] != 1]
        w = [G[i[0]][i[1]]['weight'] for i in vizinhos_atual]
        menor_aresta = sorted(zip(w, vizinhos_atual))[0][1]

        tsp.add_edge(vertice_atual, menor_aresta[1],
                     weight=G[vertice_atual][menor_aresta[1]]['weight'])

        vertice_atual = menor_aresta[1]

        # atualiza lista de vertices visitados
        vertices_visitados[vertice_atual] = 1

    tsp.add_edge(vertice_atual, 0, weight=G[vertice_atual][0]['weight'])

    return tsp
```

O Código 5 apresenta nossa implementação do algoritmo *Cheapest insertion*, descrito na Seção 4.6. Ele faz uso da estrutura `heap`, uma estrutura de dados baseada em árvore que armazena valores, e associa a cada um deles um peso, sendo que a operação de remoção sempre removerá o elemento de menor peso associado. Ela é usada para definir a ordem de inserção do algoritmo, pois sempre queremos inserir um novo vértice que esteja mais próximo dos vértices atuais no ciclo que está sendo construído.

Código 5: Implementação do algoritmo *Cheapest insertion*.

```
import networkx as nx
from heapq import heappush, heappop, heapify

# inserir_novas_arestas():
```

```
# recebe grafo base (networkx.Graph),
#        passeio (networkx.Graph),
#        aresta a ser inserita no passeio (tuple)
# devolve passeio com nova aresta inserida (lista)
def inserir_novas_arestas(G, tsp, insercao):
    # remove aresta para inserção do novo vértice
    tsp.remove_edge(*insercao[0])

    # adiciona arestas do novo vértice para os vértices da aresta excluida
    tsp.add_edge(insercao[0][0], insercao[1], weight=G[insercao[0][0]][insercao[1]]['weight'])
    tsp.add_edge(insercao[1], insercao[0][1], weight=G[insercao[1]][insercao[0][1]]['weight'])

    return nx.Graph(tsp).edge_subgraph([(insercao[0][0], insercao[1]),
           (insercao[1], insercao[0][1])]).edges(data=True)

# remove_dados_heap():
# recebe heap com vertices que ainda nao foram inseridos no passeio (heap),
#        aresta que devemos remover da heap os vertices que a compoe (tuple)
# devolve nova heap sem os vertices removidos (heap)
def remove_dados_heap(heap, aresta_remover):
    # exclui da heap as arestas que têm extremos na aresta "insercao"
    new_heap = [item for item in heap
                if item[1][0] != aresta_remover[0] and item[1][1] != aresta_remover[1]]
    heapify(new_heap)

    return new_heap


def tsp_cheapest_insertion(G):
    # criando heap
    heap = []

    # criando MultiGraph pois precisaremos de arestas paralelas para essa resolução
    tsp = nx.MultiGraph()
    vertices_visitados = [0]*G.number_of_nodes()

    # ordenado arestas por custos
    c = sorted(G.edges(data=True), key=lambda x: x[2]['weight'])

    # criando ciclo inicial
    tsp.add_edge(c[0][0], c[0][1], weight=G[c[0][0]][c[0][1]]['weight'])
    vertices_visitados[c[0][0]] = 1

    tsp.add_edge(c[0][1], c[0][0], weight=G[c[0][1]][c[0][0]]['weight'])
    vertices_visitados[c[0][1]] = 1

    novas_arestas = tsp.edges(data=True)

    # adicionando novas arestas até que todos os vértices sejam visitados
    i = 0
    while 0 in vertices_visitados:
        # procurando vértices nao visitados
        for i in range(len(vertices_visitados)):
            if vertices_visitados[i] == 0:
                # iterando pelas arestas já presentes na solução
                for e in novas_arestas:

                    # calculando peso do grafo
                    peso_insercao = G[e[0]][i]['weight'] + G[i][e[1]]['weight'] - e[2]['weight']

                    # colocando o peso da inserção calculada na heap
                    heappush(heap, (peso_insercao, (e[:2], i)))

        # a melhor inserção possível será o primeiro elemento da heap
        insercao = heappop(heap)[1]

        # atualiza solução com a nova inserção
        novas_arestas = inserir_novas_arestas(G, tsp, insercao)

        # deleta a aresta que a inserção substitui
        heap = remove_dados_heap(heap, insercao)

        vertices_visitados[insercao[1]] = 1

    return nx.Graph(tsp)
```

O Código 6 apresenta nossa implementação do algoritmo de Christofides, descrito na Seção 4.7. Utilizamos nele as seguintes funções oferecidas pela NetworkX: `mst_kruskal`, para calcular a árvore geradora mínima, e `algorithms.matching.max_weight_matching`, para calcular o emparelhamento entre vértices de grau ímpar.

Código 6: Implementação do algoritmo de Christofides.

```
import networkx as nx
from mst_kruskal import mst_kruskal

# trilha_maximal():
# recebe grafo (T + M) (networkx.MultiGraph),
#        vertice atual da trilha (int),
#        pilha que armazenara os vertices para a trilha (list)
# devolve proximo vertice da trilha maximal (int)
def trilha_maximal(G, v, STACK):
    # funcao que busca as arestas de G que tocam v
    arestas_do_vertice = lambda v: [e for e in G.edges() if v in e]

    # laco termina quando nao houverem mais arestas que toquem v
    while len(arestas_do_vertice(v)) > 0:
        # adiciona v a pilha
        STACK.append(v)

        # arestas que tocam v
        arestas = arestas_do_vertice(v)

        # remove uma das arestas que tocam v
        arestas_remover = arestas[0]
        G.remove_edge(arestas_remover[0], arestas_remover[1])

        # atualiza v com o outro vertice da aresta que foi removida
        v = arestas_remover[0] if arestas_remover[0] != v else arestas_remover[1]

    return v

# trilha_euleriana():
# recebe grafo (T + M) (networkx.MultiGraph),
#        vertice atual da trilha (int)
# devolve trilha euleriana sobre o grafo base (list)
def trilha_euleriana(G, v):
    # cria pilha
    STACK = []

    # inicia trilha
    trilha = [v]

    # busca o vertice da pilha que forme uma trilha maximal com o vertice atual
    # o laco termina quando nao houver mais vertices para formar a trilha maximal
    # com v, e nesse caso a funcao retorna o v, ou quando a pilha estiver vazia
    while (trilha_maximal(G, v, STACK) == v) and (STACK != []):
        v = STACK.pop()
        trilha.append(v)

    return trilha


# emparelhamento_vert_grau_impar():
# recebe grafo base (networkx.Graph),
#        arvore geradora do grafo base (networkx.MultiGraph)
# devolve arestas para realizar o emparelhamento perfeito sobre os vertices de grau impar (list)
def emparelhamento_vert_grau_impar(G, T):
    # lista vertices de grau impar
    vertices_impares = []
    for v in list(T.nodes):
        if len(list(T.neighbors(v)))\%2 != 0:
            vertices_impares.append(v)

    # gera subgrafo sobre os vertices de grau impar
    H = G.subgraph(vertices_impares)

    # o networkx so oferece uma funcao de emparelhamento maximo
    # para ainda utilizarmos ela, iremos multiplicar os pesos do subgrafo com -1,
    # e somar cada peso com o maior peso do subgrafo
```

```
    # isso fara com que a funcao de emparelhamento maximo na verdade retorne
    # as arestas do emparelhamento perfeito

    # achando o maior peso de aresta no subgrafo
    maior_peso = 0
    for _, _, d in H.edges(data=True):
        if d['weight'] > maior_peso:
            maior_peso = d['weight']

    # multiplicando os pesos das arestas por -1 e somando com o maior peso
    for _, _, d in H.edges(data=True):
        d['weight']  = -1*d['weight'] + maior_peso

    # conseguindo as arestas do emparelhamento perfeito
    m = nx.algorithms.matching.max_weight_matching(H, weight='weight')

    return m


def tsp_christofides(G):
    # cria arvore geradora minima
    T = nx.MultiGraph(mst_kruskal(G.copy()))

    # gera o emparelhamento perfeito sobre os vertices de grau impar
    M = emparelhamento_vert_grau_impar(G.copy(), T.copy())

    # somando as arestas do emparelhamento com a arvore geradora minima
    for e in M:
        T.add_edge(e[0], e[1], weight = G[e[0]][e[1]]['weight'])

    # gerando a trilha euleriana
    trilha_euler = trilha_euleriana(T, 0)

    # gerando o ciclo hamiltoniano
    ciclo_hamiltoniano = list(dict.fromkeys(trilha_euler)) + [0]

    tsp = gen_tsp(G, ciclo_hamiltoniano)

    return tsp
```

O Código 7 apresenta nossa implementação do algoritmo 2-Opt, descrito na Seção 4.8. Essa implementação também é bem direta quando comparada ao pseudocódigo que apresentamos.

Código 7: Implementação do algoritmo 2-Opt.

```
def troca_2opt(grafo_original, tsp_inicial):
    tsp = tsp_inicial.copy()
    custo_total = tsp_inicial.size(weight='weight')

    # iteração pelas arestas fazendo as trocas que diminuem o custo do passeio
    # caso não seja feita nenhuma troca, o laço é interrompido
    trocou = True
    while trocou:
        trocou = False
        for i, e_princ in enumerate(tsp.edges(data=True)):
            for e_sec in list(tsp.edges(data=True))[i+1:]:
                if e_princ[0] not in e_sec[:2] and e_princ[1] not in e_sec[:2]:
                    # criando a primeira opcao de troca
                    aux1 = tsp.copy()
                    aux1.remove_edge(e_princ[0], e_princ[1])
                    aux1.remove_edge(e_sec[0], e_sec[1])

                    aux1.add_edge(e_princ[0], e_sec[0],\
                                  weight = grafo_original[e_princ[0]][e_sec[0]]['weight'])
                    aux1.add_edge(e_princ[1], e_sec[1],\
                                  weight = grafo_original[e_princ[1]][e_sec[1]]['weight'])

                    # criando a segunda opcao de troca
                    aux2 = tsp.copy()
                    aux2.remove_edge(e_princ[0], e_princ[1])
                    aux2.remove_edge(e_sec[0], e_sec[1])

                    aux2.add_edge(e_princ[0], e_sec[1],\
```

```
                    weight = grafo_original[e_princ[0]][e_sec[1]]['weight'])
                aux2.add_edge(e_princ[1], e_sec[0],\
                    weight = grafo_original[e_princ[1]][e_sec[0]]['weight'])

                # a primeira troca gera um passeio?
                # diminui o custo do grafo atual?
                # diminui mais do que a segunda troca?
                if len(nx.cycle_basis(aux1)) == 1 and\
                  aux1.size(weight='weight') < custo_total and\
                  len(nx.cycle_basis(aux1)[0]) == tsp.number_of_nodes():

                    if len(nx.cycle_basis(aux2)) != 1 or\
                      len(nx.cycle_basis(aux2)[0]) != tsp.number_of_nodes() or\
                      aux1.size(weight='weight') < aux2.size(weight='weight'):

                        tsp = aux1
                        custo_total = aux1.size(weight='weight')
                        trocou = True
                        break

                # a segunda troca gera um passeio?
                # diminui o custo do grafo atual?
                if len(nx.cycle_basis(aux2)) == 1 and\
                  aux2.size(weight='weight') < custo_total and\
                  len(nx.cycle_basis(aux2)[0]) == tsp.number_of_nodes():

                    tsp = aux2
                    custo_total = aux2.size(weight='weight')
                    trocou = True
                    break

    return tsp
```

6 Resultados

Para testarmos as execuções dos algoritmos, utilizamos alguns grafos fornecidos pela TSPLIB [11], e também criamos alguns novos de tamanhos menores, para testarmos os algoritmos de solução ótima, sendo eles: g5, g6, g7, g8, g9, g10, g11 e g14, com respectivamente 5, 6, 7, 8, 9, 10, 11 e 14 vértices, sendo os pesos gerados aleatoriamente.

A Tabela 1 mostra o tempo de execução de cada um dos algoritmos que retornam uma solução ótima, mostrados nas Seções 4.1 4.2 4.3 e 4.4, através das implementações mostradas na Seção 5, para alguns dos grafos citados, sendo os que não aparecem marcados com o tempo, quer dizer que o algoritmo não foi executado em um tempo viável para aquele grafo.

Tabela 1: Tempo de execução dos algoritmos de solução ótima.

Grafo	Força bruta (s)	Branch and bound (s)	Programação linear (s)	Peso da solução ótima
g5	0,000907	0,000376	0,004901	19
g6	0,005341	0,000620	0,005338	12
g7	0,035863	0,009054	0,005996	2443
g8	0,268058	0,017654	0,006381	2447
g9	2,559686	0,10729	0,007049	2692
g10	22,444072	0,455042	0,00766	4389

g11	256,37232	2,575201	0,008553	3182
g14	-	778,987134	0,037047	3454
att48	-	-	0,224656	10628
eil51	-	-	0,086607	426
berlin52	-	-	0,05076	7542
st70	-	-	0,426418	675
pr76	-	-	2,92887	108159
eil76	-	-	0,127222	538
rat99	-	-	0,643524	1211
rd100	-	-	1,474202	7910
kroA100	-	-	1,335066	21282
kroB100	-	-	5,225921	22141
kroC100	-	-	1,679684	20749
kroD100	-	-	2,130524	21294
kroE100	-	-	2,212337	22068
eil101	-	-	0,421943	629
lin105	-	-	2,177114	14379
pr107	-	-	2,141814	44303
pr124	-	-	18,081187	59030
bier127	-	-	3,01635	118282
ch130	-	-	3,062976	6110
pr136	-	-	5,089435	96772
pr144	-	-	30,126296	58537
ch150	-	-	4,643567	6528
kroA150	-	-	19,955093	26524
kroB150	-	-	20,083801	26130
pr152	-	-	14,064846	73682
u159	-	-	4,862736	42080
rat195	-	-	59,291057	2323
d198	-	-	226,90502	15780
kroA200	-	-	172,783273	29368
kroB200	-	-	16,421794	29437
gil262	-	-	501,06653	2378
pr264	-	-	1394,41241	49135
a280	-	-	81,236346	2579
pr299	-	-	1026,40048	48191
lin318	-	-	1796,670429	42029
rd400	-	-	1880,611283	15281

Com relação aos algoritmos não ótimos, para todos eles fizemos uma comparação do peso que o algoritmo conseguiu com o peso da solução ótima, quando fornecido, ou com um

limitante inferior da solução ótima (LB). A divisão desses dois valores é o fator de aproximação daquela instância e encontra-se nas tabelas a seguir, sob as colunas "Comparação do peso (Solução/Ótima)". Assim, quanto mais próximo de 1, melhor é o resultado do algoritmo.

A Tabela 2 mostra o tempo e o peso das soluções dadas pelas implementações do algoritmo guloso mostrado na Seção 4.5. Veja que o algoritmo tem uma média de aproximação de 1,2355.

Tabela 2: Tempo e peso das soluções do algoritmo guloso.

Grafo	Tempo (s)	Peso da solução ótima	Peso da solução	Comparação do peso (Solução/Ótima)
g5	0,000046	19	21	1,1053
g6	0,000061	12	12	1,0000
g7	0,000083	2443	2841	1,1629
g8	0,000103	2447	2955	1,2076
g9	0,000124	2692	3225	1,1980
g10	0,002196	4389	4758	1,0841
g11	0,000180	3182	3671	1,1537
g14	0,000463	3454	4501	1,3031
att48	0,002197	10628	12861	1,2101
eil51	0,002601	426	511	1,1995
berlin52	0,002514	7542	8980	1,1907
st70	0,004837	675	830	1,2296
pr76	0,005550	108159	153462	1,4189
eil76	0,005471	538	642	1,1933
rat99	0,009267	1211	1554	1,2832
rd100	0,009559	7910	9938	1,2564
kroA100	0,009545	21.282	27807	1,3066
kroB100	0,011027	22141	29158	1,3169
kroC100	0,010767	20749	26227	1,2640
kroD100	0,012517	21294	26947	1,2655
kroE100	0,009617	22068	27460	1,2443
eil101	0,010460	629	803	1,2766
lin105	0,011863	14379	20356	1,4157
pr107	0,010274	44303	46680	1,0537
pr124	0,014863	59030	69297	1,1739
bier127	0,015729	118282	135737	1,1476
ch130	0,016063	6110	7579	1,2404
pr136	0,017598	96772	120769	1,2480
pr144	0,023046	58537	61652	1,0532

ch150	0,030571	6528	8191	1,2547
kroA150	0,022317	26524	33633	1,2680
kroB150	0,022219	26130	34499	1,3203
pr152	0,022441	73682	85699	1,1631
u159	0,024533	42080	54675	1,2993
rat195	0,033711	2323	2752	1,1847
d198	0,035691	15780	18240	1,1559
kroA200	0,040491	29368	35859	1,2210
kroB200	0,041675	29437	36980	1,2562
gil262	0,072513	2378	3208	1,3490
pr264	0,067063	49135	58023	1,1809
a280	0,080299	2579	3157	1,2241
pr299	0,090118	48191	59890	1,2428
lin318	0,107127	42029	54019	1,2853
rd400	0,170658	15281	19183	1,2553
fl417	0,180266	11861	15013	1,2657
pr439	0,188402	107217	131281	1,2244
pcb442	0,193573	50778	61979	1,2206
d493	0,249512	35002	41660	1,1902
att532	0,283901	27686	35516	1,2828
ali535	0,298467	202310	265952	1,3146
u574	0,320706	36905	50459	1,3673
rat575	0,311885	6773	8605	1,2705
p654	0,409402	34643	43457	1,2544
u724	0,600550	41910	52943	1,2633
rat783	0,636725	8806	11054	1,2553
pr1002	1,054897	259045	331103	1,2782
u1060	1,096500	224094	308980	1,3788
vm1084	1,317227	239297	301476	1,2598
pcb1173	1,405930	56892	71978	1,2652
d1291	1,800917	50801	60214	1,1853
rl1304	1,881862	252948	335779	1,3275
rl1323	1,928193	270199	332103	1,2291
nrw1379	2,014883	56638	68964	1,2176
fl1400	2,350065	20127	27447	1,3637
u1432	2,303238	152970	188807	1,2343
fl1577 (LB)	2,496180	22204	27996	1,2609
d1655	2,998490	62128	74032	1,1916
vm1748	3,614899	336556	408101	1,2126
u1817	3,708572	57201	72030	1,2592
rl1889	4,225599	316536	389270	1,2298

d2103 (LB)	4,579056	79952	86652	1,0838
u2152	6,007409	64253	79260	1,2336
u2319	5,649987	234256	278765	1,1900
pr2392	6,055991	378032	461170	1,2199
pcb3038	11,491976	137694	176310	1,2804
fl3795 (LB)	17,863013	28723	35285	1,2285
fnl4461	24,754531	182566	229963	1,2596
rl5915 (LB)	210,500905	565040	695602	1,2311
rl5934 (LB)	117,324818	554070	672412	1,2136

A Tabela 3 mostra o tempo e o peso das soluções dadas pelas implementações do algoritmo *Cheapest insertion* mostrado na Seção 4.6. Veja que o algoritmo tem uma média de aproximação de 1,1611.

Tabela 3: Tempo e peso das soluções do algoritmo *Cheapest insertion*.

Grafo	Tempo (s)	Peso da solução ótima	Peso da solução	Comparação do peso (Solução/Ótima)
g5	0,000498	19	19	1,0000
g6	0,000766	12	12	1,0000
g7	0,001120	2443	2443	1,0000
g8	0,001416	2447	2447	1,0000
g9	0,001748	2692	2695	1,0011
g10	0,002200	4389	4389	1,0000
g11	0,002608	3182	3182	1,0000
g14	0,011768	3454	3673	1,0634
att48	0,049098	10628	11488	1,0809
eil51	0,052715	426	467	1,0962
berlin52	0,054556	7542	8980	1,1907
st70	0,104668	675	776	1,1496
pr76	0,133348	108159	125272	1,1582
eil76	0,122088	538	599	1,1134
rat99	0,234594	1211	1384	1,1429
rd100	0,233309	7910	9117	1,1526
kroA100	0,233336	21.282	25303	1,1889
kroB100	0,242905	22141	25198	1,1381
kroC100	0,236728	20749	25408	1,2245
kroD100	0,230461	21294	25414	1,1935
kroE100	0,236550	22068	25359	1,1491
eil101	0,248543	629	721	1,1463

lin105	0,262338	14379	16905	1,1757
pr107	0,300957	44303	52544	1,1860
pr124	0,381023	59030	65934	1,1170
bier127	0,404369	118282	140716	1,1897
ch130	0,418207	6110	7092	1,1607
pr136	0,463708	96772	110321	1,1400
pr144	0,545182	58537	73032	1,2476
ch150	0,672988	6528	7769	1,1901
kroA150	0,612922	26524	30179	1,1378
kroB150	0,607164	26130	31323	1,1987
pr152	0,603960	73682	88879	1,2063
u159	0,702978	42080	50541	1,2011
rat195	1,236967	2323	2726	1,1735
d198	1,183413	15780	17681	1,1205
kroA200	1,373534	29368	35121	1,1959
kroB200	1,243369	29437	35883	1,2190
gil262	2,669317	2378	2752	1,1573
pr264	2,582861	49135	57864	1,1777
a280	3,061615	2579	3152	1,2222
pr299	3,843596	48191	58002	1,2036
lin318	4,643873	42029	49470	1,1770
rd400	10,125448	15281	18439	1,2067
fl417	10,917965	11861	14336	1,2087
pr439	12,268624	107217	131149	1,2232
pcb442	12,842792	50778	61081	1,2029
d493	19,874876	35002	39868	1,1390
att532	23,409908	27686	32364	1,1690
ali535	26,792040	202310	238099	1,1769
u574	29,957332	36905	43647	1,1827
rat575	28,791080	6773	7987	1,1792
p654	42,976049	34643	40461	1,1679
u724	61,556341	41910	50865	1,2137
rat783	80,878755	8806	10401	1,1811
pr1002	152,635727	259045	302003	1,1658
u1060	177,128006	224094	271326	1,2108
vm1084	235,126046	239297	278219	1,1627
pcb1173	255,843638	56892	70476	1,2388
d1291	354,487179	50801	59470	1,1706
rl1304	373,884945	252948	316641	1,2518
rl1323	427,093890	270199	339152	1,2552
nrw1379	469,123617	56638	66068	1,1665

fl1400	447,106280	20127	23707	1,1779
u1432	463,180445	152970	175625	1,1481
fl1577 (LB)	600,414791	22204	27335	1,2311
d1655	814,036476	62128	73225	1,1786
vm1748	904,934842	336556	402958	1,1973
u1817	1.071,947773	57201	67379	1,1779
rl1889	1.197,513541	316536	394969	1,2478
d2103 (LB)	1.676,483223	79952	88859	1,1114
u2152	1.728,954008	64253	75483	1,1748
u2319	1.958,518887	234256	271406	1,1586
pr2392	2.203,856457	378032	458832	1,2137
pcb3038	4.758,177250	137694	162299	1,1787
fl3795 (LB)	9.749,124437	28723	34085	1,1867

A Tabela 4 mostra o tempo e o peso das soluções dadas pelas implementações do algoritmo mostrado na Seção 4.7. Veja que o algoritmo tem uma média de aproximação de 1,1124, sendo essa a menor média entre os algoritmos de aproximação.

Tabela 4: Tempo e peso das soluções do algoritmo de Christofides.

Grafo	Tempo (s)	Peso da solução ótima	Peso da solução	Comparação do peso (Solução/Ótima)
g5	0,001002	19	21	1,1053
g6	0,000895	12	12	1,0000
g7	0,001204	2443	2443	1,0000
g8	0,001323	2447	2567	1,0490
g9	0,002488	2692	2812	1,0446
g10	0,001893	4389	4663	1,0624
g11	0,002585	3182	3343	1,0506
g14	0,019969	3454	3524	1,0203
att48	0,112064	10628	11899	1,1196
eil51	0,051292	426	477	1,1197
berlin52	0,059268	7542	8582	1,1379
st70	0,121378	675	779	1,1541
pr76	0,100573	108159	117862	1,0897
eil76	0,172690	538	602	1,1190
rat99	0,283734	1211	1363	1,1255
rd100	0,565961	7910	8685	1,0980
kroA100	0,330121	21.282	23468	1,1027
kroB100	0,239143	22141	24463	1,1049

kroC100	0,295118	20749	22973	1,1072
kroD100	0,326494	21294	23673	1,1117
kroE100	0,388768	22068	24739	1,1210
eil101	0,377677	629	707	1,1240
lin105	0,313034	14379	16608	1,1550
pr107	0,210791	44303	47906	1,0813
pr124	0,196318	59030	63614	1,0777
bier127	0,825811	118282	131078	1,1082
ch130	0,460384	6110	6752	1,1051
pr136	0,173249	96772	103502	1,0695
pr144	0,184947	58537	68392	1,1684
ch150	0,516273	6528	7252	1,1109
kroA150	1,196586	26524	29546	1,1139
kroB150	1,191251	26130	30130	1,1531
pr152	0,179125	73682	79113	1,0737
u159	0,845914	42080	47622	1,1317
rat195	1,265483	2323	2712	1,1675
d198	1,593524	15780	17335	1,0985
kroA200	2,296489	29368	33545	1,1422
kroB200	1,909289	29437	33191	1,1275
gil262	3,133878	2378	2729	1,1476
pr264	1,305065	49135	55396	1,1274
a280	3,092406	2579	2923	1,1334
pr299	4,395013	48191	53002	1,0998
lin318	5,402051	42029	48136	1,1453
rd400	13,864237	15281	17456	1,1423
fl417	5,598953	11861	13050	1,1002
pr439	7,275703	107217	119864	1,1180
pcb442	12,963700	50778	55372	1,0905
d493	21,706841	35002	39064	1,1161
att532	28,206267	27686	30923	1,1169
ali535	33,753166	202310	230575	1,1397
u574	28,364833	36905	41590	1,1269
rat575	28,529482	6773	7737	1,1423
p654	6,342735	34643	39341	1,1356
u724	51,749076	41910	47578	1,1352
rat783	68,371303	8806	10086	1,1454
pr1002	151,496340	259045	288752	1,1147
u1060	154,223089	224094	250078	1,1160
vm1084	101,192326	239297	266756	1,1147
pcb1173	144,295233	56892	63848	1,1223

d1291	51,042224	50801	57377	1,1294
rl1304	57,534469	252948	277816	1,0983
rl1323	56,995372	270199	299786	1,1095
nrw1379	410,352023	56638	64552	1,1397
fl1400	436,376246	20127	22972	1,1414
u1432	209,942230	152970	171597	1,1218
fl1577 (LB)	140,670952	22204	24574	1,1067
d1655	264,331405	62128	70666	1,1374
vm1748	422,465904	336556	379059	1,1263
u1817	258,391154	57201	66097	1,1555
rl1889	168,155168	316536	344694	1,0890
d2103 (LB)	46,422128	79952	84159	1,0526
u2152	457,470814	64253	72557	1,1292
u2319	1.816,239017	234256	273007	1,1654
pr2392	1.631,525669	378032	422642	1,1180
pcb3038	2.572,387007	137694	155195	1,1271
fl3795 (LB)	3.854,503772	28723	32056	1,1160

Como o algoritmo 2-Opt, apresentado na Seção 4.8, executa a partir de uma solução prévia, utilizamos as soluções dos algoritmos não ótimos como base para testarmos a sua implementação. Os resultados do tempo e do peso da sua execução sobre cada algoritmo estão na Tabela 5. Não apresentamos os fatores aqui, mas é possível calcular que o algoritmo tem uma média de aproximação de 1,0711 quando parte da solução do algoritmo *Cheapest insertion*, 1,0351 quando parte da solução do algoritmo de Christofides, e 1,0444 quando parte do algoritmo guloso.

Tabela 5: Tempo e peso das novas soluções do algoritmo 2-Opt.

Grafo	Tempo Cheapest insertion (s)	Tempo Christofides (s)	Tempo Guloso (s)	Peso Cheapest insertion (s)	Peso Christofides (s)	Peso Guloso (s)
g5	0,000879	0,001246	0,001379	19	19	19
g6	0,001698	0,001543	0,001901	12	12	12
g7	0,002525	0,002509	0,005539	2443	2443	2443
g8	0,003665	0,007286	0,010310	2447	2447	2447
g9	0,006169	0,010134	0,014547	2692	2692	2692
g10	0,007366	0,013241	0,009826	4389	4389	4414
g11	0,009669	0,018526	0,021365	3182	3212	3212
g14	0,070163	0,028858	0,143569	3489	3514	3629
att48	3,472982	5,767234	9,893354	11058	11228	10791

eil51	1,922467	6,556842	7,099367	455	444	450
berlin52	2,731795	6,613014	8,359332	8825	8120	8056
st70	20,136714	29,308328	24,710145	729	691	722
pr76	28,868219	20,620786	42,597133	114698	114975	115219
eil76	8,108390	17,143944	23,691985	588	568	561
rat99	9,995486	79,456855	94,552347	1376	1265	1269
rd100	34,177484	64,666926	180,903344	8629	8346	8642
kroA100	77,336264	79,405908	215,539964	22963	22735	22139
kroB100	80,876725	67,146558	87,184439	23361	23147	22475
kroC100	100,202708	105,316499	122,793918	23486	21411	21952
kroD100	112,542969	64,753902	143,022224	24159	21852	22407
kroE100	78,596942	60,839797	100,661892	23607	23256	24253
eil101	77,874993	76,349059	95,772071	663	671	648
lin105	103,489291	87,661403	408,555164	15139	15532	14883
pr107	142,451658	165,555913	77,510412	48671	45666	44613
pr124	116,397779	150,918569	170,541544	61936	59578	60469
bier127	108,528478	208,195619	205,037165	135293	124718	122103
ch130	209,913044	368,318343	249,326884	6794	6273	6626
pr136	378,933293	280,045760	278,669491	102250	99829	106479
pr144	465,504079	368,600874	116,619626	59889	59021	61244
ch150	417,516751	574,130513	426,675449	7169	6694	6807
kroA150	309,771196	497,147672	758,410360	29211	27652	28251
kroB150	398,092707	761,900399	569,918046	29296	27548	27570
pr152	382,294396	213,793682	337,995328	84397	75734	75242
u159	355,916251	531,926701	687,160490	47686	43995	47933
rat195	1.130,059736	1.043,060889	1.124,653125	2508	2459	2530
d198	1.452,096721	1.366,872603	1.150,448865	16916	16295	16386

7 Conclusão

Durante a realização desse projeto foram estudados ao todo oito algoritmos para o problema do caixeiro viajante.

Os de força bruta, programação dinâmica, *branch and bound* e programação linear são algoritmos que retornam uma solução ótima para o problema, e conseguimos implementar todos esses, exceto, por falta de tempo hábil, o de programação dinâmica. Percebemos que de fato são algoritmos que não são nem um pouco viáveis para grafos com um grande número de vértices, sendo a implementação da programação linear, com o Gurobi, o único que ainda consegue resolver grafos com 15 ou mais vértices, mas ainda sim demorando muito tempo.

Já para os algoritmos guloso, *cheapest insertion* e Christofides, que são algoritmos de aproximação e heurísticas, vimos que eles lidam bem com grafos de tamanhos maiores, e que o guloso é o que tem melhor desempenho em relação ao tempo, porém é o que em média retornou as piores soluções. Entre *Cheapest insertion* e Christofides, o tempo de execução é bastante parecido até quando os grafos têm menos de 1000 vértices, porém depois disso o tempo de execução do algoritmo de Christofides consegue ser melhor, além de apresentar uma melhor média de aproximação do que o algoritmo de *Cheapest insertion*.

Por fim, o algoritmo 2-Opt, dos algoritmos que não retornam uma solução ótima, é o que tem o pior desempenho em relação a tempo, o que já era esperado já que ele verifica todos os pares de arestas em todas as iterações e só termina quando chega na melhor solução possível de alcançar com as trocas.

Concluimos que o problema do caixeiro viajante é um problema que tem muitas aplicações no mundo real, onde dificilmente haverá situações em que o grafo que representa o problema terá um número baixo de vértices, então é inviável aplicar algoritmos de solução ótima devido aos seus tempos de execução. Então algoritmos de aproximação e as heurísticas se tornam fundamentais para se trabalhar com o TSP.

Referências

[1] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[2] N. Christofides. Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. Technical report, Carnegie Mellon University, 02 1976.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Algoritmos*. Campus Ltda., second edition, 2002.

[4] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[5] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[6] M. C. Goldbarg and H. P. L. Luna. *Otimização combinatória e programação linear: modelos e algoritmos*, pages 332–333. Elsevier Ltda, second edition, 2005.

[7] T. A. J. Nicholson. *A sequential method for discrete optimization problems and its application to the assignment, travelling salesman, and three machine scheduling problems*. J. Inst. Math Appl., 1967.

[8] Gurobi Optimization. Gurobi – The Fastest Solver. https://www.gurobi.com/. Acesso em 29 de setembro de 2020.

[9] NetworkX package. NetworkX – Network Analysis in Python. https://networkx.github.io/. Acesso em 29 de setembro de 2020.

[10] Ian Parberry. *Problems on Algorithms*, page 101. Prentice Hall, second edition, 2002.

[11] Gerhard Reinelt. Tsplib. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Acesso em 29 de setembro de 2020.

[12] Raphael Augusto Nascimento Rodrigues and Millys Fabrielle Araújo Carvalhaes. Análise e complexidade de algoritmos: Backtracking e força bruta. *Revista Ada Lovelace*, 1:45–48, 2017.

[13] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *Society for Industrial and Applied Mathematics*, 1977.

[14] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[15] V. V. Vazirani. *Approximation Algorithms*, pages 28–29. Springer Science and Business Media, 2001.