Um Estudo Comparativo entre um Algoritmo Exato e uma Metaheurística para Resolução do Problema do Caixeiro Viajante

Projeto Final de CIC111

Grupo:
Josias Vieira Varela - 2017011470
Érick de Oliveira Teixeira - 2017001437
Bruno Fernando Lopes - 2017014669

1. O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante é um problema de otimização da classe NP-Difícil, que consiste, basicamente, em encontrar o menor caminho que passe por uma série de pontos uma vez e retorne à origem inicial.

De forma ilustrativa, o problema propõe que existe um vendedor que está inicialmente em uma cidade A, e precisa visitar n cidades distintas e em seguida voltar para A. Não importa a ordem em que as cidades são visitadas, o importante é que ele inicie e comece sua viagem em uma mesma cidade, passando apenas uma vez em cada uma das outras cidades e, principalmente, tomando o menor caminho possível (Nilson, 1982).

Supondo um caso para n = 4, por exemplo, temos as cidades A, B, C e D e o caixeiro viajante está na cidade A. As rotas {A,B,C,D,A} e {A,B,D,C,A} são plausíveis, porque iniciam em A, terminam em A e passam por todos os pontos. Porém, pode acontecer de nenhuma delas ser a resposta que estamos procurando, isto é, a menor rota que atende esses requisitos.

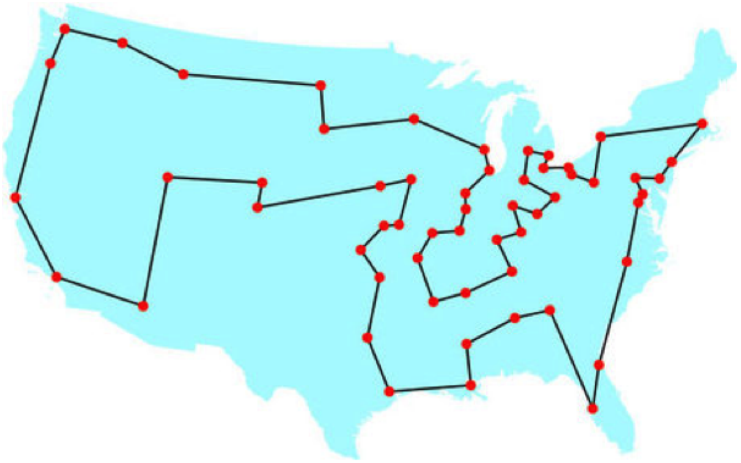


Figura 1 - Ilustração do Problema do Caixeiro Viajante, com as cidades identificadas pelos pontos vermelhos

Fonte:https://web.colby.edu/thegeometricviewpoint/2015/03/09/delauney-triangulations-and-the-traveling-salesman/

2. Formalização

A estrutura comumente utilizada para representar e trabalhar com o PCV são os grafos. Nesse contexto, cada cidade é representada por um vértice e as rotas que ligam as cidades umas às outras são representadas pelas arestas. Cada aresta possui um valor que representa o custo, que de forma ilustrativa é o comprimento da estrada que liga uma cidade a outra. A Figura 2 ilustra um grafo que representa uma instância desse problema:

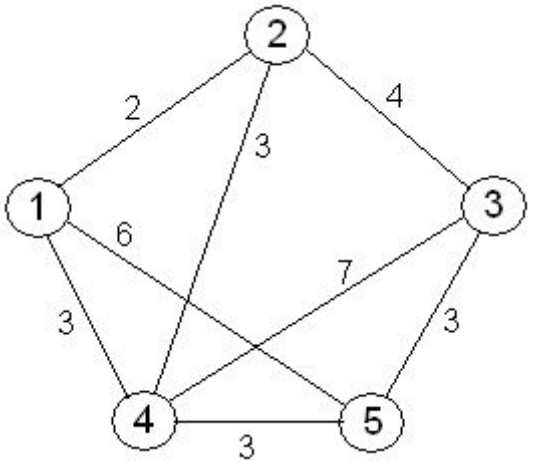


Figura 2 - Grafo representando o PCV
Fonte: https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante

Se baseando na Teoria de Grafos, podemos afirmar que estamos em busca de um **ciclo hamiltoniano** ótimo. A distância total será a soma do custo de todas as arestas.

Sendo assim, representamos o problema por um grafo G(V, E) onde $|V| \geq 3$ com custos $C_{i,j}$ (i, j) $\in$ E. Se o grafo é completo e possui n vértices, então existem [(n-1)! / 2] circuitos possíveis.

Se calculássemos todas as rotas possíveis para compará-las e ver qual é a menor, teríamos que decidir entre (n-1)! rotas. A princípio, pode parecer fácil, pois para n = 4 teríamos que decidir entre 6 rotas, e para n = 5, entre 24 rotas. Porém, a função fatorial cresce com uma velocidade muito alta e para n = 20, um número razoavelmente pequeno, isso já se torna inviável. Com essa solução ingênua, a complexidade do problema é O(n!).

3. Prova de que o Problema do Caixeiro Viajante é NP-Difícil

Pode-se provar que o problema do caixeiro viajante é NP-difícil através do ciclo hamiltoniano, sendo o problema do caixeiro viajante definido como um grafo Kn com uma função de custo c : E -> R+. Podemos reduzir o problema do ciclo hamiltoniano para o Problema do Caixeiro Viajante: CH ≤p PCV.
Dado G(V, E) com |V| = n definimos c(e) = 1 para todos os e ∈ E. Então se adicionarmos os vértices E' ao G para torná-lo G um grafo completo a assumimos c(e) = 2 para todos os e ∈ E.

Dado esse custo (c(e) = 2), se existir uma resposta do PCV de custo máximo n, então nós sabemos que há um ciclo que visita todos os nós exatamente uma vez e também temos uma resposta para o problema do ciclo hamiltoniano. Se o problema do ciclo hamiltoniano é NP-Completo, podemos afirmar que o PCV é NP-Difícil.

4. Formulação Exata

Para o caso de estudo contido neste documento, formula-se o PCV como um problema de Programação Linear Inteira (PLI) de acordo com as seguintes etapas e restrições:

Rotule as cidades com os números 1 ... n, e defina:

$x_{ij} = 1,$ se o caminho for da cidade *i* para a cidade *j*, ou
$x_{ij} = 0,$ caso contrário

Tome c_{ij} como a distância entre a cidade i e j. Então o TSP pode ser escrito como o seguinte problema de programação linear inteira:

$$min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}$$

Sujeito à:

$$0 \leq x_{ij} \leq 1 \qquad i, j = 1, 2, ..., n$$

$$\sum_{i \neq j, i=1}^{n} x_{ij} = 1 \qquad j = 1, 2, ..., n$$

$$\sum_{i \neq j, j=1}^{n} x_{ij} = 1 \qquad i = 1, 2, ..., n$$

$$\sum_{i \varepsilon Q} \sum_{j \neq i, j \varepsilon Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subseteq \{1, 2, ..., n\},\ |Q| \geq 2$$

A última restrição da formulação DFJ (Dantzig–Fulkerson–Johnson) garante que não haja sub-roteiros entre os vértices não iniciais, portanto a solução retornada é um único roteiro e não a união de roteiros menores. Como isso leva a um número exponencial de possíveis restrições, na prática ele é resolvido com o algoritmo delayed column generation.

4.1 Pyomo e IBM Cplex

O Pyomo é uma biblioteca de modelagem open-source em Python que oferece ferramentas para formular, analisar e resolver modelos de otimização, podendo ser usada para definir problemas gerais simbólicos, criar instâncias específicas e resolvê-las utilizando resolvedores comerciais open-source.

Para estudar e analisar o PCV, a formulação do problema foi implementada em Python utilizando o Pyomo como ferramenta para otimizar a modelagem em PLI. Como a biblioteca não possui um resolvedor próprio, foi utilizado IBM Cplex.

O IBM Cplex é um resolvedor de programação matemática de alto desempenho para programação linear, programação inteira mista, programação quadrática e problemas de programação quadraticamente restritos.

5. Metaheurística

Metaheurísticas são métodos de solução que coordenam procedimentos de busca locais com estratégias de mais alto nível, de modo a criar um processo capaz de escapar de mínimos locais e realizar uma busca robusta no espaço de soluções de um problema (Glover). De modo geral elas conseguem encontrar uma solução satisfatória, mas não garantem que a mesma seja ótima.

A metaheurística empregada neste trabalho foi a **2-opt swap**, sendo ela um algoritmo de busca local que consiste em: dado uma rota inicial que satisfaça o PCV, recombine a ordem das cidades visitadas duas a duas a fim de achar um melhor caminho. Faça novamente o procedimento para cada novo caminho possível.

A fim de encontrar uma solução inicial para aplicar o 2-opt swap, foi utilizado o algoritmo guloso do vizinho mais próximo, onde dado o nó presente, o caixeiro viaja para o nó mais próximo que ainda não foi visitado.

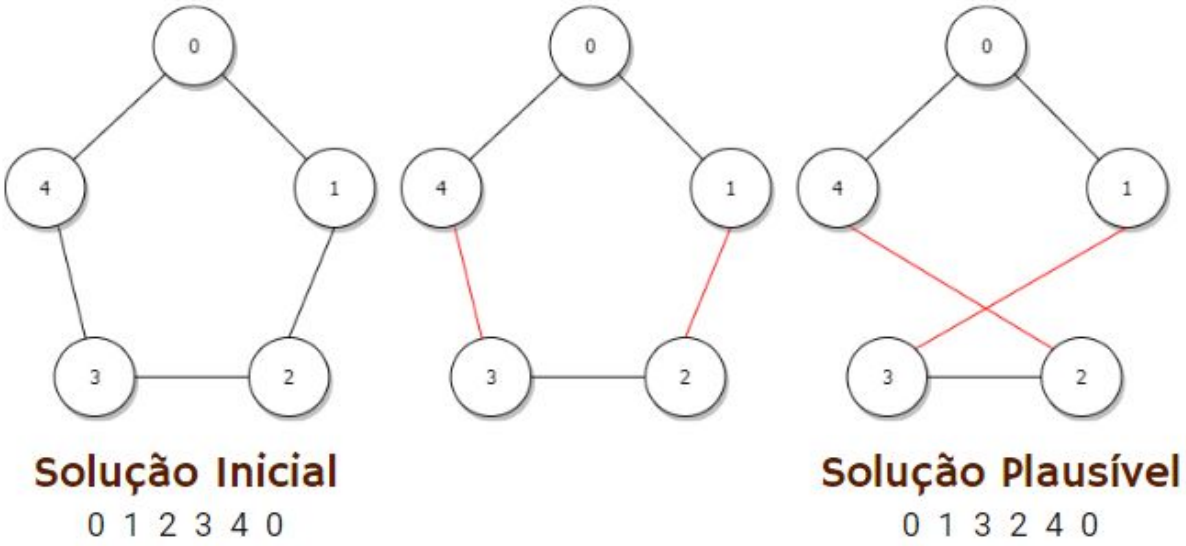


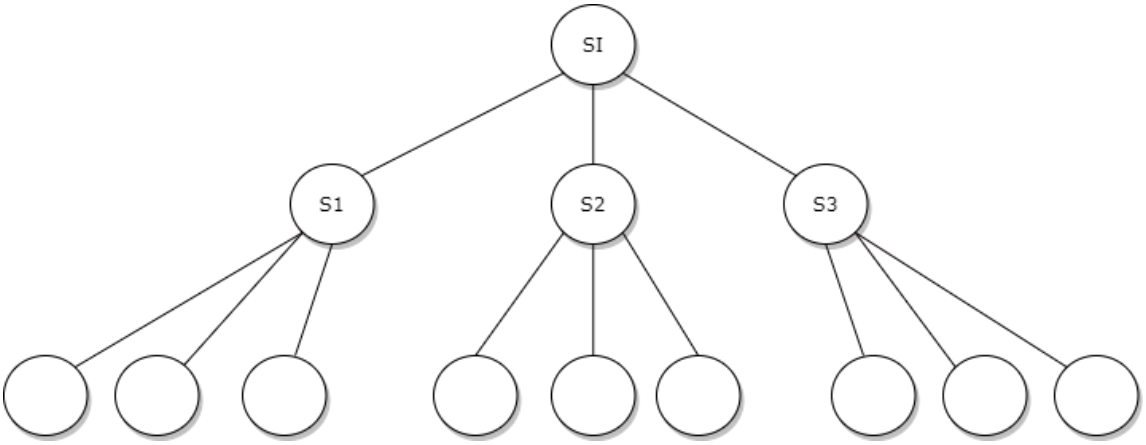
Figura 3 - Exemplo de aplicação do algoritmo 2-opt swap

Na prática, quando observamos a diferença entre uma solução inicial e solução plausível obtida após a execução da metaheurística, a mudança está na inversão de posições entre 2 cidades. Como se pode constatar na Figura 3, as cidades 2 e 3 trocaram de posição entre si.

Na nossa implementação, utilizamos uma matriz de adjacência para representar o grafo da instância executada. Logo, o que o programa desenvolvido faz, é calcular todas as possíveis trocas selecionando 2 cidades sem considerar a cidade inicial e final (que por definição devem permanecer as mesmas) e as trocas que são inversamente iguais (não é necessário trocar a cidade 1 com a 4 e depois a 4 com a 1, pois a solução teria o mesmo custo, gerando desperdício de tempo).

Para cada nova solução gerada, seu custo é comparado com o custo da melhor solução conhecida até então. Caso o novo custo seja menor, o custo mínimo conhecido é atualizado, bem como o melhor caminho que o obteve. A partir de cada nova solução gerada, esse processo de gerar trocas e compará-las se repete.

A resposta da metaheurística será o melhor custo/caminho entre todas as soluções testadas. Uma árvore representada a seguir facilita o entendimento desse processo como um todo. Vale lembrar que o melhor custo não necessariamente estará no fundo dessa árvore.



Um critério de parada é necessário, visto que a execução será infinita e as comparações continuarão sendo feitas mesmo que não haja mais melhora na solução. No nosso caso, escolhemos como critério escolhemos a altura da árvore igual a 2. Ainda que essa árvore não tenho sido implementada de fato, ela ajuda na compreensão do método. Nesse caso, comparamos as trocas possíveis a partir da solução inicial gulosa e a partir de cada nova solução gerada. Não continuamos depois desse ponto.

Quanto mais a fundo formos nessa busca, melhores e mais próximos da solução exata serão os resultados, mas temos que estar dispostos a oferecer memória, processamento e principalmente tempo à execução do problema.

6. Resultados

Foram geradas um total de 30 instâncias de forma aleatória (programa disponível no repositório) em formato .txt que representam a matriz de adjacência que representa o grafo da instância. Existem 3 instâncias para cada configuração de 10, 20, 30, …, 100 cidades, e essas que foram utilizadas para todos os testes cujos resultados estão descritos a seguir.

6.1 Método Exato

A tabela a seguir mostra os resultados encontrados empregando o método exato em instâncias de 10, 20 e 30 cidades.

Tamanho da Instância	Custo encontrado	Tempo de execução (s)
10	23	0,08
10	21	0,15
10	29	0,23
20	123	0,4
20	110	0,38
20	129	0,42
30	142	1,35
30	95	1,27
30	107	1,32

Tabela 1 - Resultados do Método Exato

6.2 Metaheurística

A tabela a seguir mostra os da execução da metaheurística para cada uma das instâncias geradas, evidenciando a melhoria em porcentagem que foi obtida em cima da solução inicial após o processo de otimização.

Instância	Nº de Cidades	Custo Inicial	Custo Final	Melhoria (%)	Tempo (seg)
instance_10_1	10	33	29	12,12	0
instance_10_2	10	22	22	0,00	0,02
instance_10_3	10	36	30	16,67	0
instance_20_1	20	195	157	19,49	0,09
instance_20_2	20	204	155	24,02	0,09
instance_20_3	20	204	161	21,08	0,11
instance_30_1	30	231	215	6,93	0,73
instance_30_2	30	159	136	14,47	0,7
instance_30_3	30	196	150	23,47	0,7
instance_40_1	40	439	324	26,20	4,33
instance_40_2	40	310	263	15,16	3,96
instance_40_3	40	422	323	23,46	3,76
instance_50_1	50	316	295	6,65	12,35
instance_50_2	50	542	456	15,87	13,43
instance_50_3	50	520	380	26,92	12,12
instance_60_1	60	556	419	24,64	30,77
instance_60_2	60	567	444	21,69	29,69
instance_60_3	60	460	400	13,04	29,41
instance_70_1	70	355	323	9,01	62,71
instance_70_2	70	424	380	10,38	67,4
instance_70_3	70	483	432	10,56	62,66
instance_80_1	80	536	469	12,50	125,19
instance_80_2	80	506	422	16,60	126,23
instance_80_3	80	559	474	15,21	125,77
instance_90_1	90	525	487	7,24	232,36
instance_90_2	90	516	438	15,12	226,17
instance_90_3	90	458	391	14,63	229,64
instance_100_1	100	555	470	15,32	427,32
instance_100_2	100	459	381	16,99	410,52
instance_100_3	100	503	426	15,31	411,9

Tabela 2 - Resultados da Metaheurística

O gráfico a seguir mostra o crescimento do tempo médio de execução do programa conforme se aumenta o tamanho da instância.

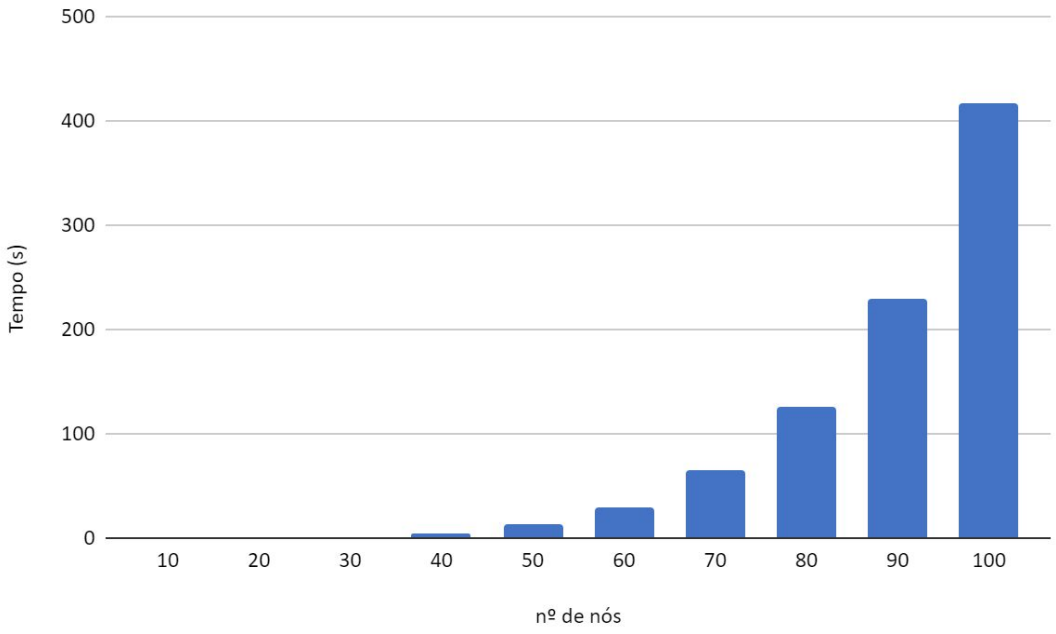


Gráfico 1 - Tempo médio de execução da Metaheurística por tamanho da instância

6.3 Método Exato vs. Metaheurística

Os gráficos abaixo apresentam um comparativo de custo e tempo de execução entre o método exato e a metaheurística a partir das instâncias testadas.

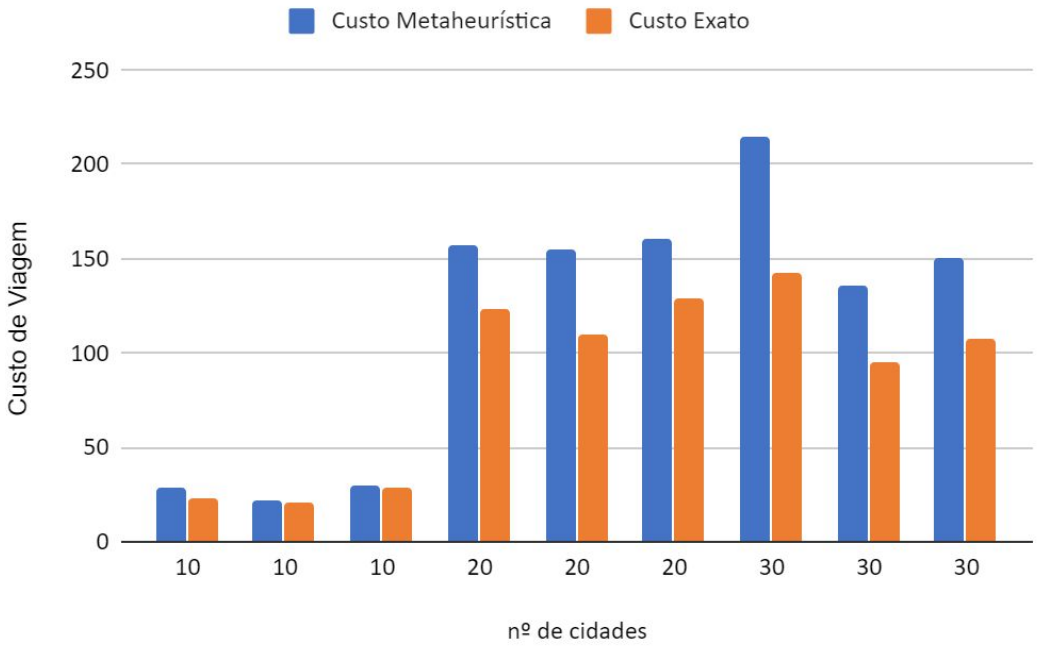


Gráfico 2 - Comparação do custo obtido pela Metaheurística e pelo Método Exato

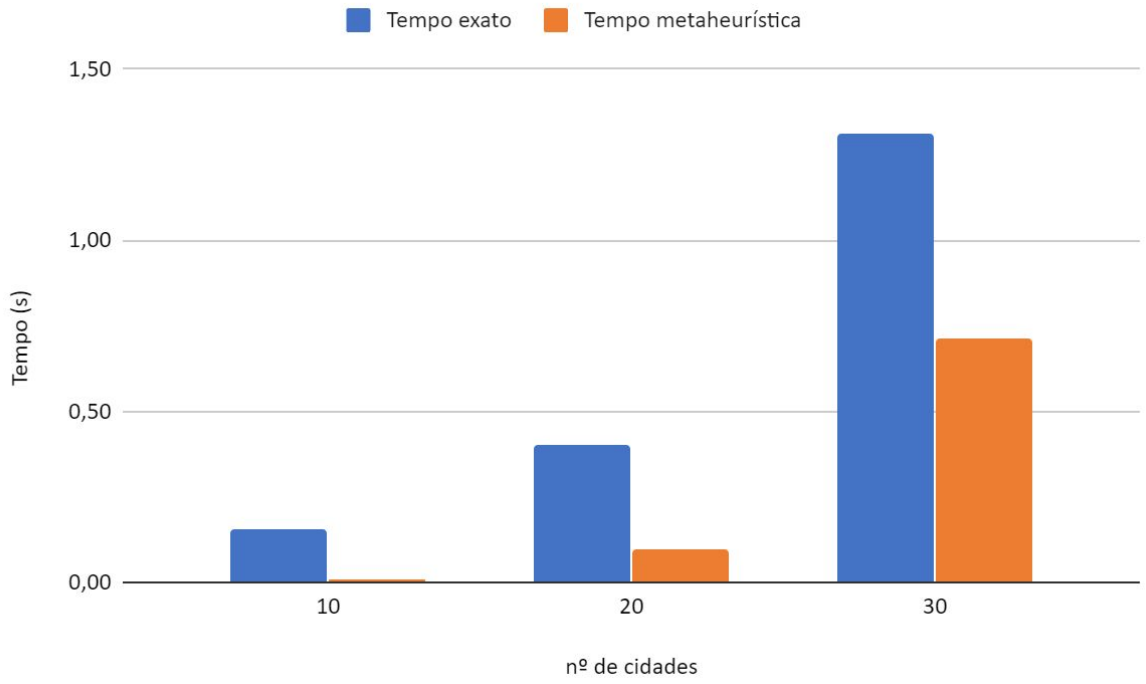


Gráfico 3 - Comparação do custo entre a metaheurística e o método exato.

7. Conclusões

De acordo com as instâncias testadas de até 30 nós cada, o algoritmo exato encontrou soluções em média 20% melhores do que a metaheurística, que por sua vez possui um tempo de execução, em média, 57% menor. Mostrando que a metaheurística empregada pode ser muito eficiente para situações que não exijam a solução ótima para o problema, ou em casos que o tempo de execução seja um fator determinante.

Para casos em que o tamanho da instância não seja muito grande, pode-se aumentar o critério de parada da metaheurística para que a solução se aproxime da exata, mas para instâncias maiores, é necessário se contentar com uma solução aproximada a menos que o tempo de execução não seja um problema.

8. Repositório

Todos os códigos do trabalho podem ser encontrados em:
https://github.com/ErickOliveiraT/Problema-do-Caixeiro-Viajante.git

9. Referências Bibliográficas

- NILSSON, Nils J. – **Principals of artificial intelligence**. New York: edição de Birkhauser, 1982. ISBN 978-3-540-11340-9.

- J.F. Porto da Silveira, **O Problema do Caixeiro Viajante**. Disponível em: <http://www.mat.ufrgs.br/~portosil/caixeiro.html>. Acesso em: 03 de jun. de 2020.

- Aaron Liu, **The Traveling Salesman and Delauney Triangulation**. Disponível em: <https://web.colby.edu/thegeometricviewpoint/2015/03/09/delauney-triangulations-and-the-traveling-salesman/>. Acesso em: 03 de jun. de 2020.

- **Problema do caixeiro-viajante**. Disponível em: <https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante>. Acesso em: 03 de jun. de 2020.

- **CME305 Sample Midterm II**. Disponível em: <http://stanford.edu/~rezab/discrete/Midterm/pmidtermIIsoln.pdf>. Acesso em: 03 de jun. de 2020

- **Traveling Salesman Problem**. Disponível em: <https://en.wikipedia.org/wiki/Travelling_salesman_problem>. Acesso em: 10 de jun. de 2020.

- Dantzig, G. B.; Fulkerson, R.; Johnson, S. M. (1954), **Solution of a large-scale traveling salesman problem**. Disponível em: <https://pdfs.semanticscholar.org/0fa3/7740fe865bcac0d9687c0e5f65131f4492c8.pdf?_ga=2.195352236.1690283454.1591820123-1593276478.1591820123>. Acesso em: 10 de jun. de 2020.

- GLOVER, F. e KOCHENBERGER, G. A. (2003). Handbook of Metaheuristics. Kluwer Academic Publishers, Boston

- Alves, Rui; Delgado, Catarina. (1997), **Programação Linear Inteira.** Disponível em: <https://repositorio-aberto.up.pt/bitstream/10216/74369/2/40539.pdf>. Acesso em: 17 de jun. de 2020

- Maciel, André; Martinhon, Carlos; Och, Luis. (2005), **Heurísticas e Metaheurísticas para o Problema do Caixeiro Viajante Branco e Preto.** Disponível em: <http://www.decom.ufop.br/prof/marcone/Disciplinas/InteligenciaComputacional/CaixeiroViajanteBrancoPreto.pdf > Acesso em: 1 de jul. de 2020 .

- Wilhelm, Volmir. **Problema do Caixeiro Viajante.** Disponível em: <https://docs.ufpr.br/~volmir/PO_II_12_TSP.pdf> Acesso em: 1 de jul. de 2020 .