Grafos Aplicados ao Problema do Caixeiro Viajante

André Luiz Gonçalves Larrosa, Eduardo Henrique Molina da Cruz

[1]Instituto Federal do Paraná - Campus Paranavaí (IFPR)

aluiz8822@gmail.com, eduardo.cruz@ifpr.edu.br

Abstract. *The Traveling Salesman Problem is a classic combinatorial optimization problem, and as a result, several algorithms have been developed to solve it. Therefore, this study aims to perform a comparative analysis of some algorithms using graphs while observing the difference in time that each algorithm takes to solve the problem with the same graph input.*

Resumo. *O Problema do Caixeiro Viajante é um problema clássico de otimização combinatório. Tendo em vista isso, surgiram diversos algoritmos que visam solucionar esse problema. Assim, este estudo tem o objetivo de realizar uma análise comparativa entre alguns algoritmos utilizando grafos observando a diferença no tempo que cada algoritmo demora para solucionar o problema com a mesma entrada de Grafo.*

1. Introdução

O Problema do Caixeiro Viajante é um clássico problema de otimização combinatória que consiste em estabelecer uma única rota que passe em cada cidade uma vez, retornando à cidade de início ao final da rota. Uma das possíveis soluções estudadas para este problema é o uso da estrutura de Grafos, onde cada vértice representa uma cidade e cada aresta o caminho que o caixeiro viajante deverá percorrer.

Para a aplicação de grafo e possíveis soluções do Problema do Caixeiro Viajante, são utilizados algoritmos exatos, que trazem a solução correta, porém com uma limitação de desempenho dada a complexidade do problema, e algoritmos heurísticos, que, não garantem uma resposta nem a melhor resposta possível, mas não possuem os problemas de desempenho do algoritmo exato. Tendo em vista isso, o presente trabalho busca, analisar e comparar alguns algoritmos sendo um algoritmo exato e dois algoritmos heurísticos e ainda desenvolver uma biblioteca na linguagem de programação C++ que facilitará a utilização de grafos durante a programação de nossos algoritmos.

2. Fundamentação Teórica

Nesta seção, será apresentado o conceito de Grafo, que servirá para contextualizar este trabalho. Também introduzirá o Problema do Caixeiro Viajante.

2.1. Grafos

Um grafo simples consiste em um conjunto finito e não vazio de vértices, juntamente com um conjunto de pares de vértices, chamados arestas. Segundo [Burguetti 2022], matematicamente, pode-se dizer que em um dado grafo $G = (V, E)$, V é o conjunto de vértices, e E é o conjunto de arestas, no qual para cada aresta $e \in E$, tem-se que

$\exists u, v \in V$, de forma que $e = (u, v)$. Dois vértices u e v são adjacentes quando há uma aresta ligando eles, isto é, caso $(u, v) \in E$.

Pode-se fazer a representação de grafos por diagramas como demonstrado na Figura 1, onde os elementos de V correspondem a círculos no plano e as arestas de E correspondem a arcos, ligando os vértices correspondentes. A figura não terá significado geométrico, seu propósito será somente representar esquematicamente as relações de adjacência entre os vértices de G.

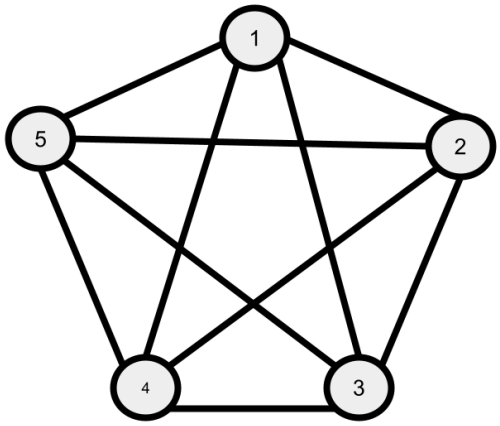


Figura 1. Exemplo de um Grafo.

Existem alguns tipos de grafos, entre eles tem-se o Grafo trivial, o qual a definição matemática é $G = (V, E)$, sendo $E = \emptyset$. Ou seja, um grafo que possui n vértices, mas nenhuma aresta. Ainda pode ter um Grafo Completo, que possui n vértices, tendo uma aresta conectando cada par de vértices, tendo um total de arestas dado pela Equação 1 afirmada por [Feofiloff et al. 2011][1].

$$\sum_{i=1}^{n-1} i = \frac{(1+n-1)(n-1)}{2} = \frac{(n)(n-1)}{2} = \frac{n^2-n}{2} \tag{1}$$

Para representar Grafos na computação, tem-se basicamente duas formas:

Lista de Adjacência Consiste em uma estrutura de dados de lista, onde cada vértice do grafo contém os seus vizinhos e cada elemento da lista contém o identificador deste vizinho. Essa estrutura é mais compacta e eficiente para grafos esparsos, aqueles que possuem poucas arestas em relação ao número total de vértices.

Matriz de Adjacência É uma matriz bidimensional que representa as conexões entre os vértices de um grafo. Se um grafo possui n vértices, a matriz terá n linhas e n colunas, onde o valor na posição (i, j) indica se há uma aresta conectando o vértice i ao vértice j. Se houver uma aresta, o valor sera 1 (verdadeiro), caso contrário, será 0 (falso).

[1] Considerando um grafo completo não dirigido, sem laços, e sem multi-arestas.

2.1.1. Grafos Dirigidos e não-dirigidos

Os grafos podem ser categorizados como **Dirigidos** e **Não-dirigidos**. Em um grafo dirigido, cada aresta é representada por uma seta que indica claramente o vértice de origem e o vértice de destino. No contexto de grafos dirigidos, a extremidade final de uma aresta é distinta de sua extremidade inicial. Tendo como exemplo o grafo da Figura 2, tem-se a aresta denotada como $2 - 1$, o que significa que a aresta se origina do vértice 2 e termina no vértice 1. Nesse cenário, a existência da aresta $2 - 1$ é independente da existência da aresta $1 - 2$. Portanto, o grafo pode conter ambas as arestas, apenas uma delas ou nenhuma delas, proporcionando uma ampla gama de possibilidades estruturais. Um exemplo de uso para grafos dirigidos, pode-se comentar por exemplo de ruas de mão única, onde só se pode ir em uma direção.

Figura 2. Exemplo de grafo dirigido.

No caso do grafo não dirigido, para cada uma de suas aresta tem outra aresta, não adjacente, que conecta os mesmos dois vértices. Ou seja, para uma aresta que liga o vértice a ao vértice b, tal aresta também liga b ao a. Tal tipo de aresta pode ser representada somente como uma ligação entre os vértices, como pode ser observado na Figura 3.

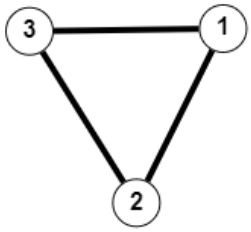


Figura 3. Exemplo de grafo não dirigido.

2.1.2. Rotulação de Grafos

A Rotulação de Grafos consiste em uma atribuição de valores para os vértices e/ou arestas de um Grafo sob determinadas condições. Tais valores podem ser números, ou, em outra abordagem, cores.

2.2. Problema do Caixeiro Viajante (PCV)

O Problema do Caixeiro viajante ou do inglês *Travelling Salesman Problem* é um problema de otimização combinatória. Segundo [Laporte et al. 1986], o PCV é um dos problemas mais estudados em otimização combinatória. Sua definição é simples, segundo

[Cunha et al. 2002], *"O PCV pode ser definido como o problema de encontrar o roteiro de menor distância ou custo que passa por um conjunto de cidades, sendo cada cidade visitada exatamente uma vez."*

O Problema do Caixeiro Viajante (PCV) pode ser analisado em duas perspectivas: simétrica, onde a distância entre cidades é independente do sentido percorrido, e assimétrica, onde essa distância varia conforme o sentido. Ao buscar soluções para o PCV, existem duas abordagens principais. Primeiro, os métodos exatos são algoritmos precisos que garantem a solução ótima, mas sua aplicação é limitada devido à complexidade combinatória do problema. Estes métodos, como os procedimentos de força bruta, exploram minuciosamente todas as rotas possíveis.

Por outro lado, os métodos heurísticos são técnicas aproximadas que buscam soluções aceitáveis de maneira mais eficiente, priorizando a rapidez computacional. Ambas as categorias de métodos desempenham um papel crucial na resolução do desafiador PCV, cada uma com suas vantagens e limitações.

Tendo em vista a limitação dos métodos exatos, o principal foco para a resolução do problema do caixeiro viajante são os métodos heurísticos, que, na maioria das vezes, se baseiam em uma abordagem intuitiva e não garantem uma resposta ótima. Na verdade, não há qualquer garantia sobre a qualidade da resposta. Assim, em geral, as propostas heurísticas não conseguem produzir boas soluções para problemas com características e condicionantes diferentes das quais foram desenvolvidas.

Como por exemplo nas imagens abaixo, consegue-se ver todas as rotas que o caixeiro pode fazer ao percorrer todas as cidades. Como demonstrado na imagem, o caixeiro estando na cidade 1, para percorrer as demais cidades ele pode fazer diversas rotas. Por exemplo, iniciando da 1, ele pode ir até a cidade 2, em seguida, 3, 4, 5 e retornar à cidade 1. Algumas possibilidades podem ser observadas nas Figuras 4 à 7.

2.3. Análise de Algoritmos e a Teoria da Complexidade Computacional

Diante da diversos algoritmos heurísticos e exatos existentes, definir se um determinado algoritmo resolve o problema para o qual foi proposto, de forma que seja mais eficiente que outros algoritmos, acaba se tornando uma tarefa complicada. Desta forma, uma maneira para calcular a eficiência de um algoritmo é medir sua complexidade através da quantidade necessária de processamento e armazenamento para sua execução [Bovet et al. 1994].

A análise de algoritmos possibilita identificar o algoritmo mais eficiente dentre vários candidatos a resolver um determinado problema. Muitas vezes, a análise pode indicar mais de um candidato viável, porém ajuda a descartar diversos candidatos menos eficientes [Cormen et al. 2022]. Tendo em vista isso, consegue-se quantificar e medir a complexidade de um algoritmo, não de forma prática, mas de forma teórica. A teoria da complexidade foi desenvolvida baseando-se principalmente em duas medidas de complexidade, chamadas de tempo e espaço [Bovet et al. 1994].

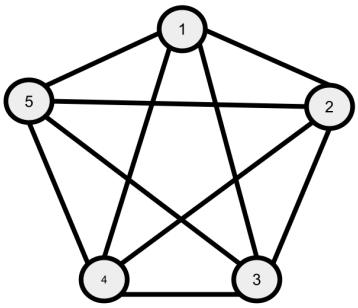


Figura 4. Grafo usado no exemplo do Caixeiro Viajante.

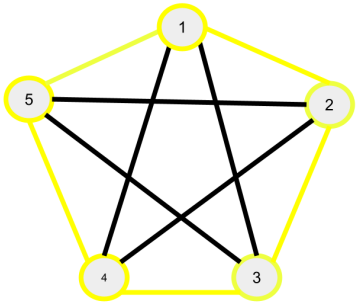


Figura 5. Primeiro Caminho.

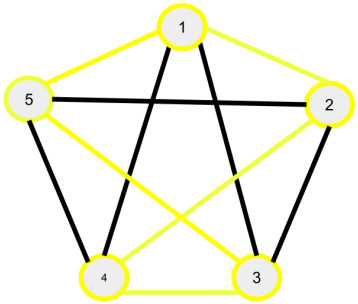


Figura 6. Segundo Caminho.

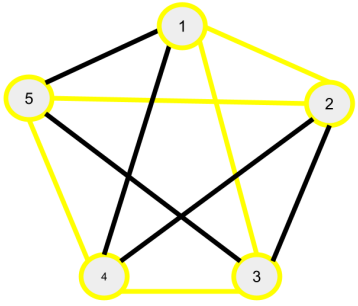


Figura 7. Terceiro Caminho.

2.3.1. Complexidade de tempo

A complexidade de tempo verifica quanto tempo um algoritmo determinado demora para completar sua tarefa, baseando-se na quantidade de operações durante a execução do mesmo. Assim, cada operação soma para a expressão da complexidade, até que um

código simples, começa a crescer significativamente seu tempo de execução, dependendo do tamanho da entrada que é posto. Segundo [Cormen et al. 2022], *"[...] o tempo de execução de um algoritmo é o número de operações primitivas ou passos executados em razão a determinada entrada."*

2.3.2. Complexidade de espaço

Complexidade de espaço é semelhante a complexidade de tempo, porém, é somado também à expressão da complexidade, as variáveis que são armazenadas em memória.

2.3.3. Complexidade assintótica

Como afirmado por [Lima 2014], a complexidade assintótica é uma medida que objetiva comparar a eficiência de um dado algoritmo, analisando tempo, memória e processamento, desenvolvida por Juris Hartmanis e Richard E. Stearns, em 1965. Não dependendo do sistema que está sendo executado e nem da linguagem em que foi desenvolvido, baseando-se em uma função que expressa uma relação entre a quantidade de dados e o tempo necessário para processá-los, calculando a complexidade assintótica de um código de forma teórica e experimental.

Para a complexidade assintótica, existem algumas notações, são elas:

Notação Big-O – Segundo [Martinez et al. 2019], representa o limite superior de uma função na complexidade assintótica, assim, dadas as funções $f(n)$ e $g(n)$, diz-se que $f(n) = O(g(n))$, se $0 \leq f(n) \leq c.g(n)$, sendo c uma constante positiva e n suficientemente grande. Sendo assim, por maior que seja o tamanho de n para $f(n)$, nunca ultrapassará $g(n)$ para o mesmo valor n.

Notação Θ (Theta) – De acordo com [Martinez et al. 2019], representa quando a entrada de dados similares a casos médios, ou seja, quando a entrada do algoritmo não coincide com o seu pior caso e nem com o seu melhor caso. Para isso, dize-se que $f(n) = \Theta(g(n))$, se $c1.g(n) \leq f(n) \leq c2.g(n)$, para as constantes $c1$ e $c2$ positivas e todo n suficientemente grande.

Notação Ω (Ômega) – [Martinez et al. 2019], em seu trabalho, afirma que representa o limite inferior. Assim, é dito que $f(n) = \Omega(g(n))$, se $0 \leq c.g(n) \leq f(n)$, para qualquer constante c e para todo n suficientemente grande.

2.4. Pesquisas e implementações similares

Existem outros trabalhos que trazem implementações de algoritmos que podem ser encontrados na literatura, sendo destacado o trabalho [Silva et al. 2013], o qual analisa alguns algoritmos heurísticos construtivos do Problema do Caixeiro Viajante. Dentre os algoritmos deste trabalho estão: "Vizinho Mais Próximo" e "Inserção do Mais Distante", que foram abordados no presente trabalho.

Os demais algoritmos abordados no trabalho de [Silva et al. 2013] foram: inserção do mais próximo e inserção do mais barato, porém, não foram abordados algoritmos exatos, como no trabalho presente. Portanto, o presente trabalho busca analisar os algoritmos citados acima e o algoritmo exato para o PCV.

Da mesma maneira que da [Silva et al. 2013], [Cunha et al. 2002], desenvolveu uma pesquisa que analisava estratégias alternativas para implementação de heurísticas relacionadas ao Problema do Caixeiro Viajante. Tais melhorias se davam por heurísticas do tipo k-opt, que removiam-se k arcos de um roteiro definido e substituia-se por outros k arcos, com a finalidade de diminuir a distância total percorrida. Neste trabalho, [Cunha et al. 2002], não analisava diretamente heurísticas, não analisando nenhum algoritmo exato para o Problema do Caixeiro Viajante.

Em relação ao desenvolvimento da biblioteca de grafos, existem diversas bibliotecas feitas para a linguagem de programação C++ que implementam a estrutura de Grafos. Dentre elas, pode-se citar a Boost Graph Library (BGL), uma biblioteca que fornece uma gama ampla para estrutura de dados e algoritmos para grafos, fornecendo suporte a grafos dirigidos e não dirigidos, alguns dos algoritmos implementados por essa biblioteca são busca em profundidade, busca em largura, algoritmos de árvore geradora mínima. Porém, foi decidido para este trabalho, desenvolver a própria biblioteca, tendo assim, uma biblioteca que atenderia uma análise comparativa de forma mais facilitada.

Diversos outros estudos sobre o problema do caixeiro viajante foram feitas, além de outras implementações de grafos em C++. Sendo assim, o presente trabalho propõe colaborar com a literatura oferecendo uma análise comparativa entre os algoritmos.

3. Algoritmos

No presente trabalho, serão estudados 3 algoritmos, sendo eles: O algoritmo ótimo para a resolução do problema do caixeiro viajante, o algoritmo heurístico Vizinho mais próximo e, por último, o algoritmo de Inserção do mais distante. Nesta seção, serão apresentados cada um deles.

3.1. Algoritmo Bruta Força (Algoritmo Exato)

O Algoritmo Exato tem uma solução ótima para o problema do caixeiro viajante, porém, é importante mencionar que embora garantidos de uma solução ótima, tal algoritmo pode se tornar impraticável dependendo da entrada, tendo em vista o seu alto custo computacional. Um entre os vários algoritmos exatos é o algoritmo Bruta força, que testa todas as possibilidades de rotas. O algoritmo ótimo está descrito no Algoritmo 1.

Algoritmo 1: Algoritmo Ótimo

```
Dados: Grafo;
início
    Função Caixeiro(grafo: Grafo):
        nosSolucao = lista vazia de inteiros;
        noVisitado = lista de booleanos com grafo.quantidadeNosGrafo() elementos, todos
          inicializados como falso;
        para i = 0 até grafo.quantidadeNosGrafo() faça
            noAtual = grafo.nos[i];
            pontoInicial = noAtual;
            noVisitado[noAtual.getId()] = verdadeiro;
            nosSolucao.adicionar(noAtual.getId());
            CaixeiroProximoNo(grafo, noAtual, noVisitado, nosSolucao, 0, 1);
            nosSolucao.removerUltimo();
            noVisitado[noAtual.getId()] = falso;
        fim
    Função CaixeiroProximoNo(grafo: Grafo, noAtual: No, nosVisitados: vetor de
      booleanos, nosSolucao: vetor de inteiros, custoAcumulado: real, quantidadeNos:
      inteiro):
        se quantidadeNos = grafo.quantidadeNosGrafo() então
            PesoArestaBuscar = -1.0;
            para cada aresta em noAtual.getArestas() faça
                se aresta.getNoFim() = pontoInicial então
                    PesoArestaBuscar = aresta.getPeso();
                    break;
                fim
            fim
            custoAcumulado = custoAcumulado + PesoArestaBuscar;
            se custoAcumulado < custoMelhorSolucao então
                melhorSolucao = nosSolucao;
                custoMelhorSolucao = custoAcumulado;
            fim
        fim
        senão
            para cada aresta em noAtual.getArestas() faça
                aresta = aresta;
                noDestino = aresta.getNoFim();
                se nosVisitados[noDestino.getId()] = falso então
                    nosVisitados[noDestino.getId()] = verdadeiro;
                    nosSolucao.adicionar(noDestino.getId());
                    CaixeiroProximoNo(grafo, noDestino, nosVisitados, nosSolucao,
                      custoAcumulado + aresta.getPeso(), quantidadeNos + 1);
                    nosVisitados[noDestino.getId()] = falso;
                    nosSolucao.pop_back();
                fim
            fim
        fim
fim
```

3.2. Vizinho mais próximo

Este algoritmo heurístico começa do vértice inicial e adiciona um vértice ainda não visitado, cuja a distância do último vértice visitado seja a mínima. O algoritmo é finalizado quando todos os vértices forem visitados. Ao final do algoritmo, é feita a ligação do

último vértice com o vértice inicial. O algoritmo está descrito no Algoritmo 2.

Algoritmo 2: Vizinho mais próximo

```
   Dados: Rota;
   Resultado: Rotaminimizada
 1 início
 2     resultado = Grafo();
 3     pontoInicial = Rota[0];
 4     pesoArestaABuscar = 0.0;
 5     nosVisitados = [false];
 6     noAtual = pontoInicial;
 7     resultadoAtual = resultado.adicionarNo(noAtual.Valor);
 8     resultadoNoInicio = resultadoNoAtual;
 9     noVisitado[noAtual.Id] = true;
10     para i = 0; i < grafo.quantidadeNos(); i++ faça
11         arestas = noAtual.arestas();
12         menorPeso = 100000000000000;
13         arestaSelecionada = null;
14         para j = 0; j < arestas.tamanho(); j++ faça
15             aresta = arestas[j]; if aresta.NoDestino.Id != noAtual.Id then
16                 if nosVisitados[aresta.NoDestino.Id] == false && aresta.Peso <
                     menorPeso then
17                     menorPeso = aresta.Peso;
18                     arestaSelecionada = aresta;
19             noSelecionado = arestaSelecionada.NoDestino;
20             resultadoNoSelecionado = resultado.adicionarNo(noSelecionado.Valor);
21             resultado.adicionarAresta(arestaSelecionada.Peso, resultadoNoAtual,
                 resultadoNoSelecionado);
22             noAtual = noSelecionado;
23             resultadoNoAtual = resultadoNoSelecionado;
24             nosVisitados[noAtual.Id] = true;
25         fim
26     fim
27     para j = 0; j < arestas.tamanho(); j++ faça
28         if aresta.NoDestino == pontoInicial then
29             PesoArestaABuscar = aresta.Peso;
30             break;
31     fim
32     resultado.adicionarAresta(PesoArestaABuscar, resultadoNoAtual, resultadoNoInicio);
         return resultado;
33 fim
```

3.3. Inserção do mais distante

A inserção do mais distante é outro algoritmo heurístico, porém ele contrapõe o algoritmo do vizinho mais próximo, pois nele, inicia-se com uma sub-rota que contém o vértice de origem e o vértice mais distante da origem e é feito isso para cada vértice, sempre escolhendo o vértice que esteja com a distância máxima entre eles. O algoritmo está descrito no Algoritmo 3.

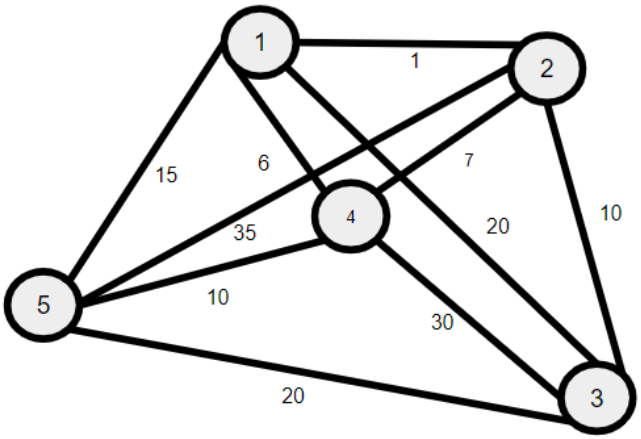
Algoritmo 3: Inserção do mais distante

```
   Dados: Grafo;
   Resultado: Grafo
1  início
2      resultado = Grafo();
3      pontoInicial = Rota[0];
4      pesoArestaABuscar = 0.0;
5      nosVisitados = [false];
6      noAtual = pontoInicial;
7      resultadoAtual = resultado.adicionarNo(noAtual.Valor);
8      resultadoNoInicio = resultadoNoAtual;
9      noVisitado[noAtual.Id] = true;
10     para i = 0; i < grafo.quantidadeNos(); i++ faça
11         arestas = noAtual.arestas();
12         maiorPeso = -1;
13         arestaSelecionada = null;
14         para j = 0; j < arestas.tamanho(); j++ faça
15             aresta = arestas[j];
16             if aresta.NoDestino.Id != noAtual.Id then
17                 if nosVisitados[aresta.NoDestino.Id] == false && aresta.Peso >
                     maiorPeso then
18                     maiorPeso = aresta.Peso;
19                     arestaSelecionada = aresta;
20             noSelecionado = arestaSelecionada.NoDestino;
21             resultadoNoSelecionado = resultado.adicionarNo(noSelecionado.Valor);
22             resultado.adicionarAresta(arestaSelecionada.Peso, resultadoNoAtual,
                 resultadoNoSelecionado);
23             noAtual = noSelecionado;
24             resultadoNoAtual = resultadoNoSelecionado;
25             nosVisitados[noAtual.Id] = true;
26         fim
27     fim
28     para j = 0; j < arestas.tamanho(); j++ faça
29         if aresta.NoDestino == pontoInicial then
30             PesoArestaABuscar = aresta.Peso;
31             break;
32     fim
33     resultado.adicionarAresta(PesoArestaABuscar, resultadoNoAtual, resultadoNoInicio);
34     return resultado;
35 fim
```

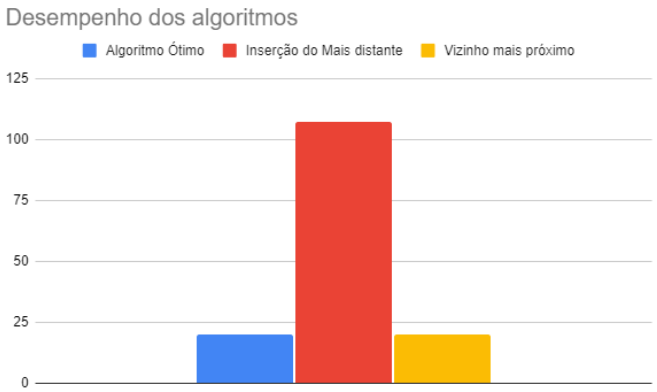
4. Resultados Experimentais

Os algoritmos foram implementados na linguagem C++. Para o teste, foi dado como entrada o grafo ilustrado na Figura 8a. O resultado contendo o custo de cada solução pode ser visualizado na Figura 8b, onde no eixo Y mostra a distância total percorrida.

Com o experimento realizado, consegue-se perceber que o algoritmo ótimo garantiu a solução correta para o problema do caixeiro viajante, juntamente com o algoritmo do vizinho mais próximo. Porém, ao analisar o algoritmo Inserção do Mais distante, pode-se perceber que a resposta informada para o problema não foi a de menor custo possível, já que os demais algoritmos trouxeram uma resposta com o custo mais otimizado.



(a) Grafo utilizado nos experimentos.



(b) Custo das soluções encontradas por cada algoritmo.

Figura 8. Resultado Experimental.

5. Conclusão

O Problema do Caixeiro Viajante, sendo um problema clássico de otimização combinatória, torna-se um desafio quando tenta-se uma solução para ele, tendo em vista a quantidade de algoritmos que existem com o objetivo de obter um resultado melhor para o mesmo. Nesse sentido, aplicar grafo para solucionar o problema é uma das abordagens que pode-se ter.

No presente trabalho, aborda-se alguns métodos de resolução para o PCV, sendo um deles o algoritmo exato, e os algoritmos heurísticos 'Vizinho mais próximo' e 'Inserção do mais distante'. Para o desenvolvimento desses algoritmos, foi feita uma biblioteca de grafos visando facilitar o desenvolvimento desses algoritmos, para ao final fazer-se uma análise comparativa entre os três. Nos testes, o algoritmo do 'Vizinho mais próximo' conseguiu bons resultados, sendo então considerado uma heurística melhor que o 'Inserção do mais distante'.

Referências

Bovet, D. P., Crescenzi, P., and Bovet, D. (1994). *Introduction to the Theory of Complexity*, volume 7. Prentice Hall London.

Burguetti, R. (2022). Alguns tipos de grafos e aplicações.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

Cunha, C. B., de Oliveira Bonasser, U., and Abrahão, F. T. M. (2002). Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. In *XVI Congresso da Anpet*.

Feofiloff, P., Kohayakawa, Y., and Wakabayashi, Y. (2011). Uma introdução sucinta à teoria dos grafos.

Laporte, G., Mercure, H., and Nobert, Y. (1986). An exact algorithm for the asymmetrical capacitated vehicle routing problem. *Networks*, 16(1):33–46.

Lima, A. d. S. C. (2014). Aproximação experimental da complexidade assintótica de shaders para dispositivos móveis utilizando opengl es.

Martinez, J. G. A. et al. (2019). Caracterização teórica e limites assintóticos para o problema da geração de redes candidatas na busca por palavras-chave em bancos de dados relacionais.

Silva, G. A. N. d., Silva, F. A. d., Russi, D. T. A., Pazoti, M. A., and Siscoutto, R. A. (2013). Algoritmos heurÍsticos construtivos aplicados ao problema do caixeiro viajante para a definiÇÃo de rotas otimizadas. *Colloquium Exactarum. ISSN: 2178-8332*, 5(2):30–46.