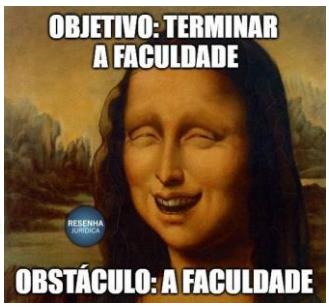
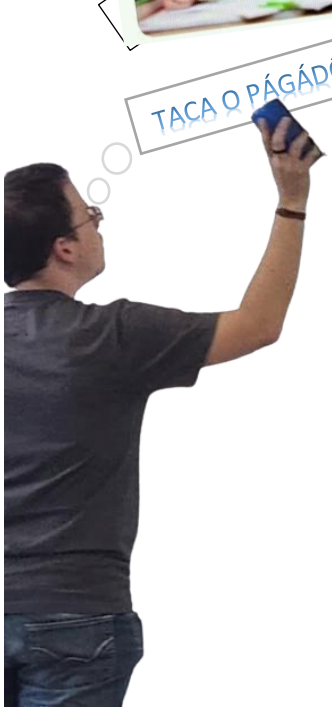
TEORIA DOS GRAFOS

LUCAS AUGUSTO AZEVEDO





PROFESSOR FELIPE BELÉM

Sumário

1. INTRODUÇÃO

Os algoritmos de caminhos mínimos em grafos ponderados representam um campo essencial na teoria dos grafos, desempenhando um papel fundamental em diversas áreas, desde logística e transporte até redes de computadores. Este documento apresenta uma análise aprofundada e comparativa de cinco proeminentes algoritmos: Bellman-Ford, Dijkstra, Floresta de Caminhos Ótimo, Floyd-Warshall e Johnson. Cada um desses algoritmos aborda o desafio de encontrar os caminhos mais curtos entre vértices em um grafo, considerando a ponderação associada às arestas.

2. REFERENCIAL TEÓRICO

2.1. Algoritmo de Bellman-Ford: é um método eficaz para encontrar os caminhos mínimos em grafos ponderados. Desenvolvido por Richard Bellman e Lester Ford Jr., o algoritmo foi proposto na década de 1950 e tem aplicações significativas em diversas áreas, desde redes de transporte até análise de circuitos elétricos.
O algoritmo de Bellman-Ford destaca-se por sua capacidade de lidar com arestas de peso negativo e pela detecção eficiente de ciclos negativos (se, após V-1 iterações (onde V é o número de vértices), ainda for possível relaxar uma aresta, há um ciclo negativo no grafo). Sua abordagem iterativa de relaxamento (processo de relaxamento compara as distâncias calculadas com as distâncias existentes e, se necessário, atualiza-as) permite que o algoritmo explore grafos complexos, embora sua complexidade de tempo seja O(VE). Essa flexibilidade torna o Bellman-Ford uma escolha valiosa em cenários onde arestas com pesos negativos são uma consideração.

2.1.1 Aplicações Práticas:

- O algoritmo é amplamente utilizado em redes de comunicação, onde as arestas podem representar atrasos ou custos associados às conexões.
- Na análise de sistemas elétricos, o Bellman-Ford é empregado para otimizar o roteamento de energia em circuitos.

2.1.2 Limitações e Considerações:

- Embora seja robusto, o Bellman-Ford pode não ser a escolha ideal em grafos densos devido à sua complexidade.
- Em grafos onde todas as arestas têm pesos não negativos, algoritmos como Dijkstra podem ser mais eficientes.

Compreender suas características e limitações é fundamental para a seleção adequada em diferentes contextos de aplicação.

2.2. Algoritmo de Dijkstra: foi proposto por Edsger W. Dijkstra em 1956, o algoritmo é amplamente utilizado em diversas áreas, como redes de computadores, planejamento de rotas, e otimização de sistemas logísticos.
Em contraste, o algoritmo de Dijkstra é conhecido por sua eficiência em grafos densos, especialmente quando não há arestas com pesos negativos. A utilização de uma fila de prioridade ordenada pelo peso das arestas contribui para sua complexidade de tempo, que é O((V + E) log V). O Dijkstra destaca-se em cenários onde o grafo é relativamente denso e não há considerações de ciclos negativos.

2.2.1. Aplicações Práticas:

- O algoritmo de Dijkstra é amplamente empregado em roteamento de redes, como na Internet, onde as arestas representam os custos associados à comunicação entre diferentes nodos.
- Em sistemas de transporte, o Dijkstra é utilizado para encontrar rotas otimizadas em mapas de ruas ou redes de transporte público.

2.2.2 Limitações e Considerações:

- O Dijkstra assume que todos os pesos das arestas são não negativos. Em cenários com arestas de peso negativo, o algoritmo de Bellman-Ford pode ser mais apropriado.
- A utilização de uma fila de prioridade, embora eficiente, pode tornar o algoritmo menos prático em cenários de grandes grafos com arestas dinâmicas.

Sua simplicidade conceitual e implementação tornaram-no uma ferramenta valiosa em muitas disciplinas, refletindo sua relevância contínua na solução de problemas práticos de otimização.

2.3. Algoritmo de Floresta de Caminhos Ótimo: Uma variante do Dijkstra (compartilha sua base conceitual com o algoritmo de Dijkstra), a Floresta de Caminhos Ótimo utiliza uma abordagem de fila simples para explorar grafos de forma eficiente. Este algoritmo é adequado para grafos ponderados, proporcionando resultados detalhados que são especialmente úteis em cenários onde a simplicidade da implementação é uma consideração. A principal inovação do algoritmo de Floresta de Caminhos Ótimo é a estratégia de explorar os caminhos mínimos através de uma floresta de árvores de busca, onde cada árvore representa uma busca por caminhos mínimos a partir de um vértice específico, otimizando o processo de exploração.

2.3.1. Aplicações Práticas:

- A otimização na exploração faz com que o algoritmo seja particularmente útil em ambientes de redes, como roteamento de pacotes, onde a rápida adaptação a mudanças na topologia é essencial.

2.3.2. Considerações e Limitações:

- Embora ofereça eficiência em certos cenários, a escolha entre o algoritmo de Dijkstra e a Floresta de Caminhos Ótimo depende das características específicas do grafo e dos requisitos da aplicação.

2.4. Algoritmo de Floyd-Warshall: foi proposto por Robert W. Floyd e Stephen Warshall na década de 1960, o algoritmo é conhecido por sua simplicidade conceitual e pela capacidade de lidar com grafos que podem conter arestas de peso negativo.

O algoritmo de Floyd-Warshall é conhecido por sua capacidade de encontrar caminhos mínimos entre todos os pares de vértices em um grafo ponderado. Sua complexidade de tempo, O(V^3), torna-o prático para grafos de tamanho moderado. O algoritmo utiliza uma matriz de adjacência para representar o grafo, onde cada entrada (i, j) armazena o peso da aresta entre os vértices i e j. Tabelas auxiliares são utilizadas para armazenar os caminhos mínimos intermédios durante a execução do algoritmo. Uma das vantagens do Floyd-Warshall é sua capacidade de detectar ciclos negativos no grafo, na qual, se houver um ciclo negativo, a matriz de distâncias mínimas terá valores negativos ao longo da diagonal principal.

2.4.1. Aplicações Práticas:

- O algoritmo é utilizado em sistemas de navegação para calcular rotas otimizadas entre diferentes localizações.
- Em redes de comunicação, o Floyd-Warshall é aplicado para analisar a conectividade eficiente entre diferentes pontos.

2.4.2. Considerações e Limitações:

- Enquanto o Floyd-Warshall é robusto e eficiente para grafos pequenos a médios, sua complexidade o torna menos prático para grafos muito grandes.
- Em grafos densos, onde há muitas arestas, a eficiência do algoritmo pode ser prejudicada.

2.5. Algoritmo de Johnson: Desenvolvido por Donald B. Johnson, este algoritmo é uma extensão dos algoritmos de Bellman-Ford e Dijkstra, combinando suas características para superar limitações individuais e oferecer uma solução abrangente para o problema de caminho mínimo.

Destaca-se pela capacidade de lidar com grafos que podem conter ciclos negativos. Sua eficiência é evidente na complexidade total, geralmente dominada pelo passo modificado do Bellman-Ford, resultando em O (VE + V^2 * log(V)). Esta adaptação torna o Johnson uma escolha eficaz em cenários onde ciclos negativos são uma consideração. O algoritmo de Johnson inicia adicionando um nó fictício ao grafo e conectando-o a todos os outros vértices com peso zero, em seguida, executa uma versão modificada do algoritmo de Bellman-Ford para encontrar os menores caminhos do nó fictício para todos os outros nós.

2.5.1. Aplicações Práticas e Uso Generalizado:

- O algoritmo de Johnson é amplamente utilizado em diversas aplicações práticas, como sistemas de navegação, otimização de redes, e análise de rotas em logística.
- Sua adaptabilidade a diferentes contextos torna-o uma escolha versátil em problemas de caminho mínimo.

2.5.2. Considerações e Limitações:

- A estratégia de adicionar um nó fictício ao grafo simplifica o tratamento de arestas de peso negativo.
- Introduzir um nó fictício e ajustar os pesos pode aumentar a complexidade do algoritmo, tornando-o menos prático para grafos muito grandes.

Ao longo deste documento, serão apresentadas implementações eficientes desses algoritmos em diferentes cenários, proporcionando uma compreensão aprofundada de suas aplicações e desempenho. Os resultados obtidos e apresentados em arquivos complementarão a análise, permitindo comparações valiosas sobre o comportamento desses algoritmos em grafos diversos.

3. ALGORITMO DE BELLMAN-FORD

O código apresentado implementa o algoritmo de Bellman-Ford para encontrar os caminhos mínimos em um grafo ponderado. Abaixo, apresento uma análise detalhada do código e uma explicação do algoritmo.

3.1. Estrutura do Grafo (Classe Graph):

- A classe Graph representa um grafo, contendo o número de vértices e uma lista de adjacências.
- O construtor inicializa a lista de adjacências.
- O método addEdge adiciona uma aresta ao grafo, garantindo que a lista de adjacências tenha capacidade suficiente para acomodar os vértices especificados.

3.2. Estrutura de Pares (Classe Pair):

- A classe **Pair** é uma simples estrutura de dados que armazena um vértice e um peso, representar as arestas do grafo.

3.3. Algoritmo de Bellman-Ford (AlgoritmoBellmanFord):

- O código implementa a classe principal **AlgoritmoBellmanFord**, que contém o algoritmo de Bellman-Ford.
- A função **bellmanFordAlgorithm** recebe um grafo, um vértice de origem e um array de pais, e retorna um array de distâncias mínimas.
- O algoritmo funciona relaxando as arestas repetidamente para encontrar as distâncias mínimas de um vértice de origem a todos os outros vértices.

```java
// Implementação do algoritmo de Bellman-Ford
public static int[] bellmanFordAlgorithm(Graph graph, int startVertex, int[] parents) {
    int[] distances = new int[graph.getNumVertices()];
    for (int i = 0; i < graph.getNumVertices(); i++) {
        distances[i] = Integer.MAX_VALUE;
    }
    distances[startVertex] = 0;

    // Relaxamento de arestas
    for (int i = 0; i < graph.getNumVertices() - 1; i++) {
        for (int u = 0; u < graph.getNumVertices(); u++) {
            for (Pair neighbor : graph.getAdjList().get(u)) {
                int v = neighbor.getVertex();
                int weightUV = neighbor.getWeight();

                // Relaxamento da aresta
                if (distances[u] != Integer.MAX_VALUE && distances[v] > distances[u] + weightUV) {
                    distances[v] = distances[u] + weightUV;
                    parents[v] = u;  // Atualiza o pai para o caminho mínimo
                }
            }
        }
    }

    return distances;
}
```

Passos do Algoritmo:

1. **Inicialização:** Inicializa um array de distâncias com valores infinitos, exceto para a origem que é zero. Inicializa o array de pais como -1.
2. **Relaxamento:** Realiza relaxamento de arestas repetidamente, comparando as distâncias calculadas com as distâncias existentes e atualizando conforme necessário.
3. **Detecção de Ciclo Negativo:** O algoritmo é capaz de detectar ciclos negativos. Se, após todas as iterações, ainda for possível relaxar uma aresta, há um ciclo negativo no grafo.

3.4. Execução para Vários Grafos:

- O código executa o algoritmo em vinte e cinco grafos diferentes, variando o número de vértices e arestas.
- Para cada execução, registra o tempo de execução, as distâncias mínimas e os caminhos mínimos a partir de um vértice de origem.
- Os resultados são armazenados em strings e também escritos em arquivos de saída (**BellmanFord.txt** e **ResulTime_BellmanFord.txt**).

```
Grafo 1 - Iteração 1 - Arestas 4
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 16, Caminho - 0 -> 3 -> 4 -> 1
Para o vértice 2: Distância - 11, Caminho - 0 -> 3 -> 4 -> 2
Para o vértice 3: Distância - 1, Caminho - 0 -> 3
Para o vértice 4: Distância - 8, Caminho - 0 -> 3 -> 4

Grafo 1 - Iteração 2 - Arestas 6
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 3, Caminho - 0 -> 2 -> 1
Para o vértice 2: Distância - 1, Caminho - 0 -> 2
Para o vértice 3: Distância - 1, Caminho - 0 -> 3
Para o vértice 4: Distância - 4, Caminho - 0 -> 2 -> 4

Grafo 1 - Iteração 3 - Arestas 7
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 3, Caminho - 0 -> 2 -> 1
Para o vértice 2: Distância - 1, Caminho - 0 -> 2
Para o vértice 3: Distância - 1, Caminho - 0 -> 3
Para o vértice 4: Distância - 4, Caminho - 0 -> 2 -> 4
```

Exemplo do arquivo BellmanFord.txt

```
[[0, 0, 0, 0, 0, 0, 1, 0, 0, 4, 4, 5, 0, 4, 16, 5, 27,
36, 97, 825, 150, 1391, 1371, 7138, 114687], [4, 6, 7, 9,
10, 24, 72, 104, 152, 300, 124, 558, 831, 1242, 7750,
624, 3744, 5608, 8404, 195000, 3124, 23430, 35135, 52695,
4881250]]
```

Exemplo do arquivo ResulTime_BellmanFord.txt

3.5. Leitura de Grafos a Partir de Arquivos:

- A função **lerGrafo** lê um grafo de um arquivo, onde o primeiro número é o número de vértices e os subsequentes são triplas (origem, destino, peso) representando as arestas.

grafo_1.txt

```
5
3 4 7
4 2 3
0 3 1
4 1 8
2 0 1
1 2 2
3 2 5
1 3 7
1 0 8
0 4 7
```

3.6. Escrita em Arquivos:

- A função **criarArquivoTexto** é responsável por escrever strings em arquivos.

3.7. Loop Principal (main):

- O loop principal executa o algoritmo em cinco grafos diferentes, medindo o tempo de execução e registrando os resultados.

3.8. Resultados:

- Os resultados incluem distâncias mínimas, caminhos mínimos, e tempos de execução para cada grafo e iteração.
- Os tempos de execução e informações sobre as arestas são armazenados em arquivos.

3.9. Conclusão: O código implementa com sucesso o algoritmo de Bellman-Ford e fornece uma análise detalhada dos caminhos mínimos em vários grafos. O algoritmo é útil para encontrar os caminhos mais curtos em grafos ponderados, mesmo na presença de arestas com pesos negativos, embora ciclos negativos possam ser detectados. O Algoritmo de Bellman-Ford possui uma **complexidade de tempo de O(VE),** onde *V* é o número de vértices e *E* é o número de arestas no grafo.

5. ALGORITMO DE DIJKSTRA

O código implementa o algoritmo de Dijkstra para encontrar os caminhos mínimos em um grafo ponderado. Vamos analisar detalhadamente cada parte do código.

5.1. Estrutura do Grafo (Graph):

- A classe **Graph** representa um grafo, contendo o número de vértices e uma lista de adjacências.
- O construtor inicializa a lista de adjacências.
- O método **addEdge** adiciona uma aresta ao grafo, garantindo que a lista de adjacências tenha capacidade suficiente para acomodar os vértices especificados.

5.2. Estrutura de Pares (Pair):

- A classe **Pair** é uma simples estrutura de dados que armazena um vértice e um peso, representar as arestas do grafo.

5.3. Algoritmo de Dijkstra (AlgoritmoDijkstra):

- A classe principal contém a implementação do algoritmo de Dijkstra.
- O método **dijkstraAlgorithm** recebe o grafo, o vértice de origem e um array de pais, retornando um array de distâncias mínimas.

```java
// Implementação do algoritmo de Dijkstra
public static int[] dijkstraAlgorithm(Graph graph, int startVertex, int[] parents) {
    int[] distances = new int[graph.getNumVertices()];
    for (int i = 0; i < graph.getNumVertices(); i++) {
        distances[i] = Integer.MAX_VALUE;
    }
    distances[startVertex] = 0;

    PriorityQueue<Pair> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(Pair::getWeight));
    priorityQueue.add(new Pair(startVertex, weight:0));

    while (!priorityQueue.isEmpty()) {
        int u = priorityQueue.poll().getVertex();

        for (Pair neighbor : graph.getAdjList().get(u)) {
            int v = neighbor.getVertex();
            int weightUV = neighbor.getWeight();

            if (distances[v] > distances[u] + weightUV) {
                distances[v] = distances[u] + weightUV;
                priorityQueue.add(new Pair(v, distances[v]));
                parents[v] = u;  // Atualiza o pai para o caminho mínimo
            }
        }
    }

    return distances;
}
```

Passos do Algoritmo de Dijkstra:

1. **Inicialização:** Inicializa um array de distâncias com valores infinitos, exceto para a origem que é zero. Utiliza uma fila de prioridade (**PriorityQueue**) ordenada pelo peso da aresta.
2. **Exploração:** Enquanto a fila de prioridade não estiver vazia, extrai o vértice com menor distância (**u**) e relaxa todas as arestas adjacentes.
3. **Relaxamento:** Se a distância até **v** passando por **u** for menor que a distância atualmente conhecida até **v**, atualiza a distância e adiciona **v** à fila de prioridade.
4. **Armazenamento de Caminhos Mínimos:** Mantém um array de pais para reconstruir os caminhos mínimos após a execução.

5.4. Execução para Vários Grafos:

- O código executa o algoritmo em vinte e cinco grafos diferentes, variando o número de vértices e arestas.
- Para cada execução, registra o tempo de execução, as distâncias mínimas e os caminhos mínimos a partir de um vértice de origem.
- Os resultados são armazenados em strings e também escritos em arquivos de saída (**Dijkstra.txt** e **ResulTime_Dijkstra.txt**).

```
Grafo 2 - Iteração 3 - Arestas 52
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 13, Caminho - 0 -> 19 -> 9 -> 23 -> 5 -> 14 -> 1
Para o vértice 2: Distância - 9, Caminho - 0 -> 19 -> 9 -> 11 -> 2
Para o vértice 3: Distância - 8, Caminho - 0 -> 19 -> 3
Para o vértice 4: Distância - 5, Caminho - 0 -> 19 -> 4
Para o vértice 5: Distância - 9, Caminho - 0 -> 19 -> 9 -> 23 -> 5
Para o vértice 6: Distância - 6, Caminho - 0 -> 6
Para o vértice 7: Distância - 14, Caminho - 0 -> 19 -> 9 -> 11 -> 2 -> 7
Para o vértice 8: Distância - 9, Caminho - 0 -> 19 -> 9 -> 8
Para o vértice 9: Distância - 5, Caminho - 0 -> 19 -> 9
Para o vértice 10: Distância - 10, Caminho - 0 -> 19 -> 9 -> 23 -> 5 -> 10
Para o vértice 11: Distância - 7, Caminho - 0 -> 19 -> 9 -> 11
Para o vértice 12: Distância - 12, Caminho - 0 -> 19 -> 9 -> 11 -> 2 -> 12
Para o vértice 13: Distância - 9, Caminho - 0 -> 19 -> 3 -> 13
Para o vértice 14: Distância - 10, Caminho - 0 -> 19 -> 9 -> 23 -> 5 -> 14
Para o vértice 15: Distância - 13, Caminho - 0 -> 19 -> 3 -> 17 -> 15
Para o vértice 16: Distância - 14, Caminho - 0 -> 19 -> 9 -> 23 -> 16
Para o vértice 17: Distância - 10, Caminho - 0 -> 19 -> 3 -> 17
Para o vértice 18: Distância - 18, Caminho - 0 -> 19 -> 3 -> 17 -> 15 -> 18
Para o vértice 19: Distância - 3, Caminho - 0 -> 19
Para o vértice 20: Distância - 11, Caminho - 0 -> 19 -> 4 -> 20
Para o vértice 21: Distância - 11, Caminho - 0 -> 19 -> 3 -> 17 -> 21
Para o vértice 22: Distância - 13, Caminho - 0 -> 19 -> 4 -> 20 -> 22
Para o vértice 23: Distância - 6, Caminho - 0 -> 19 -> 9 -> 23
Para o vértice 24: Distância - 14, Caminho - 0 -> 19 -> 9 -> 11 -> 2 -> 12 -> 24
```

Exemplo do arquivo Dijkstra.txt

```
[[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 4, 9,
0, 16, 0, 16, 8, 0, 165], [4, 6, 7, 9, 10, 24, 72, 104,
152, 300, 124, 558, 831, 1242, 7750, 624, 3744, 5608,
8404, 195000, 3124, 23430, 35135, 52695, 4881250]]
```

Exemplo do arquivo ResulTime_Dijkstra.txt

5.5. Leitura de Grafos a Partir de Arquivos:

- A função **lerGrafo** lê um grafo de um arquivo, onde o primeiro número é o número de vértices e os subsequentes são triplas (origem, destino, peso) representando as arestas.

grafo_2.txt

```
25
23 3 7
17 14 9
6 4 3
19 9 2
20 22 2
5 22 6
1 24 2
24 12 2
24 21 6
5 8 4
```

5.6. Escrita em Arquivos:

- A função **criarArquivoTexto** é responsável por escrever strings em arquivos.

5.7. Loop Principal (main):

- O loop principal executa o algoritmo em cinco grafos diferentes, medindo o tempo de execução e registrando os resultados.

5.8. Resultados:

- Os resultados incluem distâncias mínimas, caminhos mínimos e tempos de execução para cada grafo e iteração.
- Os tempos de execução e informações sobre as arestas são armazenados em arquivos.

5.9. Conclusão: O código implementa com sucesso o algoritmo de Dijkstra, um algoritmo eficiente para encontrar os caminhos mínimos em grafos ponderados. O uso de uma fila de prioridade garante uma execução eficiente, especialmente em grafos densos. O código fornece análises detalhadas dos caminhos mínimos em diferentes grafos, incluindo o tempo de execução e os resultados escritos em arquivos. O algoritmo de Dijkstra possui uma **complexidade de tempo de $O((V+E)\log V)$,** onde V é o número de vértices e E é o número de arestas no grafo.

6. ALGORITMO DE FLOYD-WARSHALL

O código implementa o algoritmo de Floyd-Warshall para encontrar os caminhos mínimos entre todos os pares de vértices em um grafo ponderado. Vamos analisar detalhadamente cada parte do código.

6.1. Algoritmo de Floyd-Warshall (AlgoritmoFloydWarshall):

- A classe principal contém a implementação do algoritmo de Floyd-Warshall para encontrar caminhos mínimos em grafos ponderados.
- A constante **INFINITY** é usada para representar a ausência de uma aresta entre dois vértices.

6.2. Função Principal (main):

- O método **main** é responsável por executar o algoritmo em vinte e cinco grafos diferentes e registrar os resultados.
- O loop principal itera sobre os grafos, ajustando o número de vértices e arestas com base na variável **y**.
- Para cada arquivo, o algoritmo é executado cinco vezes, repetindo a leitura do arquivo e o processamento.

6.3. Algoritmo de Floyd-Warshall (floydWarshall):

- A função recebe a matriz de adjacência do grafo, o número de vértices, uma instância de **StringBuilder** para construir o resultado, o número do grafo, a iteração e o número de linhas do grafo atual.
- Inicializa as matrizes de distâncias e caminhos, preenchendo-as com valores iniciais.
- Executa o algoritmo de Floyd-Warshall para calcular as distâncias mínimas e os caminhos mínimos entre todos os pares de vértices.

```java
// Implementação do algoritmo de Floyd-Warshall
public static void floydWarshall(int[][] matrizAdj, int numVertices, StringBuilder resultado, int grafoNum, int iteracao, int numLinhas) {
    int[][] distancias = new int[numVertices][numVertices];
    int[][] caminhos = new int[numVertices][numVertices];

    // Inicialização das matrizes de distâncias e caminhos
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            distancias[i][j] = matrizAdj[i][j];
            if (i != j && matrizAdj[i][j] != INFINITY) {
                caminhos[i][j] = i;
            } else {
                caminhos[i][j] = -1;
            }
        }
    }

    // Algoritmo de Floyd-Warshall
    for (int k = 0; k < numVertices; k++) {
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                if (distancias[i][k] != INFINITY && distancias[k][j] != INFINITY &&
                        distancias[i][k] + distancias[k][j] < distancias[i][j]) {
                    distancias[i][j] = distancias[i][k] + distancias[k][j];
                    caminhos[i][j] = caminhos[k][j];
                }
            }
        }
    }
```

Passos do Algoritmo de Floyd-Warshall:

1. **Inicialização:** Inicializa as matrizes de distâncias e caminhos com base na matriz de adjacência do grafo.
2. **Iteração:** Para cada vértice intermediário (**k**), verifica se há um caminho mais curto entre os pares de vértices **i** e **j** passando por **k**.
3. **Atualização:** Se o caminho passando por **k** é mais curto, atualiza as matrizes de distâncias e caminhos.

6.4. Construção de Caminho Mínimo (construirCaminho):

- A função **construirCaminho** recebe as matrizes de caminhos, a origem e o destino para construir o caminho mínimo em forma de string.

```
Grafo 4 - Iteração 2 - Arestas 936
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 25, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 525 -> 17 -> 244 -> 249 -> 1
Para o vértice 2: Distância - 20, Caminho - 0 -> 458 -> 83 -> 325 -> 419 -> 473 -> 2
Para o vértice 3: Distância - 27, Caminho - 0 -> 458 -> 269 -> 301 -> 313 -> 454 -> 3
Para o vértice 4: Distância - 23, Caminho - 0 -> 458 -> 269 -> 372 -> 133 -> 4
Para o vértice 5: Distância - 33, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 525 -> 357 -> 466 -> 120 -> 98 -> 5
Para o vértice 6: Distância - 48, Caminho - 0 -> 458 -> 83 -> 325 -> 419 -> 130 -> 438 -> 258 -> 254 -> 378 -> 493 -> 177 -> 6
Para o vértice 7: Distância - 36, Caminho - 0 -> 458 -> 269 -> 372 -> 298 -> 169 -> 550 -> 74 -> 593 -> 7
Para o vértice 8: Distância - 31, Caminho - 0 -> 458 -> 269 -> 301 -> 313 -> 454 -> 397 -> 589 -> 8
Para o vértice 9: Distância - 27, Caminho - 0 -> 306 -> 278 -> 620 -> 60 -> 237 -> 353 -> 9
Para o vértice 10: Distância - 33, Caminho - 0 -> 29 -> 371 -> 207 -> 605 -> 316 -> 225 -> 72 -> 291 -> 10
Para o vértice 11: Distância - 2147483647, Caminho - Destino inválido.
Para o vértice 12: Distância - 29, Caminho - 0 -> 458 -> 269 -> 372 -> 298 -> 169 -> 523 -> 12
Para o vértice 13: Distância - 28, Caminho - 0 -> 458 -> 83 -> 325 -> 419 -> 130 -> 336 -> 13
Para o vértice 14: Distância - 32, Caminho - 0 -> 458 -> 269 -> 437 -> 575 -> 231 -> 395 -> 14
Para o vértice 15: Distância - 32, Caminho - 0 -> 29 -> 371 -> 207 -> 605 -> 316 -> 433 -> 267 -> 15
Para o vértice 16: Distância - 25, Caminho - 0 -> 29 -> 371 -> 207 -> 605 -> 134 -> 162 -> 16
Para o vértice 17: Distância - 17, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 525 -> 17
Para o vértice 18: Distância - 38, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 525 -> 17 -> 244 -> 249 -> 1 -> 155 -> 215 -> 442 -> 18
Para o vértice 19: Distância - 36, Caminho - 0 -> 458 -> 486 -> 489 -> 233 -> 148 -> 519 -> 80 -> 300 -> 19
Para o vértice 20: Distância - 30, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 184 -> 494 -> 84 -> 292 -> 20
Para o vértice 21: Distância - 28, Caminho - 0 -> 29 -> 371 -> 282 -> 178 -> 21
Para o vértice 22: Distância - 31, Caminho - 0 -> 458 -> 486 -> 489 -> 77 -> 200 -> 314 -> 22
Para o vértice 23: Distância - 28, Caminho - 0 -> 458 -> 269 -> 372 -> 133 -> 159 -> 23
Para o vértice 24: Distância - 40, Caminho - 0 -> 458 -> 83 -> 325 -> 419 -> 130 -> 438 -> 258 -> 254 -> 378 -> 493 -> 177 -> 24
Para o vértice 25: Distância - 22, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 434 -> 25
Para o vértice 26: Distância - 34, Caminho - 0 -> 458 -> 83 -> 325 -> 202 -> 516 -> 564 -> 557 -> 26
Para o vértice 27: Distância - 26, Caminho - 0 -> 306 -> 278 -> 620 -> 406 -> 184 -> 494 -> 211 -> 94 -> 27
Para o vértice 28: Distância - 26, Caminho - 0 -> 458 -> 269 -> 301 -> 341 -> 295 -> 35 -> 387 -> 28
Para o vértice 29: Distância - 4, Caminho - 0 -> 29
Para o vértice 30: Distância - 29, Caminho - 0 -> 29 -> 371 -> 207 -> 605 -> 134 -> 162 -> 30
```

Exemplo do arquivo FloydWarshall.txt

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 2,
2, 15, 0, 0, 0, 14, 356], [4, 6, 7, 9, 10, 24, 72, 104,
152, 300, 124, 558, 831, 1242, 7750, 624, 3744, 5608,
8404, 195000, 3124, 23430, 35135, 52695, 4881250]]
```

Exemplo do arquivo ResulTime_FloydWarshall.txt

6.5. Escrita em Arquivos (criarArquivoTexto):

- A função **criarArquivoTexto** é responsável por escrever strings em arquivos.

6.6. Leitura de Grafos a Partir de Arquivos (lerGrafo):

- A função **lerGrafo** lê um grafo de um arquivo, onde o primeiro número é o número de vértices e os subsequentes são triplas (origem, destino, peso) representando as arestas.

grafo_4.txt

```
625
84 258 2
452 573 9
418 148 4
168 153 9
15 260 3
387 624 5
225 104 1
602 328 9
350 487 4
313 358 5
```

6.7. Resultados:

- Os resultados incluem distâncias mínimas e caminhos mínimos a partir do vértice 0 para todos os outros vértices.
- Os tempos de execução e informações sobre as arestas são armazenados em arquivos.

6.8. Conclusão: O código implementa com sucesso o algoritmo de Floyd-Warshall, que encontra todos os caminhos mínimos em um grafo ponderado. Os resultados são detalhados e escritos em arquivos para análise posterior. O algoritmo possui uma **complexidade de tempo de O(V^3)**, onde *V* é o número de vértices no grafo, tornando-o prático para grafos pequenos a médios.

7. ALGORITMO DE FLORESTA DE CAMINHOS ÓTIMO

O código implementa o algoritmo de Floresta de Caminhos Ótimo para encontrar os caminhos mínimos em um grafo ponderado. Vamos analisar detalhadamente cada parte do código.

7.1. Estrutura do Grafo (Graph):

- A classe **Graph** representa um grafo, contendo o número de vértices e uma lista de adjacências.
- O construtor inicializa a lista de adjacências.
- O método **addEdge** adiciona uma aresta ao grafo, garantindo que a lista de adjacências tenha capacidade suficiente para acomodar os vértices especificados.

7.2. Estrutura de Pares (Pair):

- A classe **Pair** é uma simples estrutura de dados que armazena um vértice e um peso, representar as arestas do grafo.

7.3. Algoritmo de Floresta de Caminhos Ótimo (AlgoritmoFlorestaCaminhosOtimo):

A classe principal contém a implementação do algoritmo de Floresta de Caminhos Ótimo.

- O método **forestPathsAlgorithm** recebe o grafo, o vértice de origem e um array de pais, retornando um array de distâncias mínimas.

```java
// Implementação do algoritmo de Floresta de Caminhos Otimo
public static int[] forestPathsAlgorithm(Graph graph, int startVertex, int[] parents) {
    int[] distances = new int[graph.getNumVertices()];
    for (int i = 0; i < graph.getNumVertices(); i++) {
        distances[i] = Integer.MAX_VALUE;
    }
    distances[startVertex] = 0;

    Queue<Integer> queue = new LinkedList<>();
    queue.offer(startVertex);

    while (!queue.isEmpty()) {
        int u = queue.poll();

        for (Pair neighbor : graph.getAdjList().get(u)) {
            int v = neighbor.getVertex();
            int weightUV = neighbor.getWeight();

            if (distances[v] > distances[u] + weightUV) {
                distances[v] = distances[u] + weightUV;
                parents[v] = u;  // Atualiza o pai para o caminho mínimo
                queue.offer(v);
            }
        }
    }

    return distances;
}
```

Passos do Algoritmo de Floresta de Caminhos Ótimo:

1. **Inicialização:** Inicializa um array de distâncias com valores infinitos, exceto para a origem que é zero. Utiliza uma fila (**Queue**) para realizar a exploração.
2. **Exploração:** Enquanto a fila não estiver vazia, extrai o vértice com menor distância (**u**) e relaxa todas as arestas adjacentes.
3. **Relaxamento:** Se a distância até **v** passando por **u** for menor que a distância atualmente conhecida até **v**, atualiza a distância e adiciona **v** à fila.
4. **Armazenamento de Caminhos Mínimos:** Mantém um array de pais para reconstruir os caminhos mínimos após a execução.

7.4. Execução para Vários Grafos:

- O código executa o algoritmo em vinte e cinco grafos diferentes, variando o número de vértices e arestas.
- Para cada execução, registra o tempo de execução, as distâncias mínimas e os caminhos mínimos a partir de um vértice de origem.
- Os resultados são armazenados em strings e também escritos em arquivos de saída (**ForestPaths.txt** e **ResulTime_ForestPaths.txt**).

```
Grafo 4 - Iteração 4 - Arestas 2101
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 13, Caminho - 0 -> 306 -> 278 -> 249 -> 1
Para o vértice 2: Distância - 12, Caminho - 0 -> 306 -> 351 -> 243 -> 248 -> 2
Para o vértice 3: Distância - 11, Caminho - 0 -> 29 -> 185 -> 335 -> 105 -> 219 -> 3
Para o vértice 4: Distância - 12, Caminho - 0 -> 98 -> 531 -> 532 -> 4
Para o vértice 5: Distância - 8, Caminho - 0 -> 98 -> 5
Para o vértice 6: Distância - 16, Caminho - 0 -> 306 -> 351 -> 93 -> 547 -> 386 -> 6
Para o vértice 7: Distância - 17, Caminho - 0 -> 306 -> 351 -> 121 -> 227 -> 345 -> 7
Para o vértice 8: Distância - 15, Caminho - 0 -> 306 -> 351 -> 93 -> 152 -> 589 -> 8
Para o vértice 9: Distância - 18, Caminho - 0 -> 98 -> 531 -> 558 -> 64 -> 9
Para o vértice 10: Distância - 16, Caminho - 0 -> 123 -> 246 -> 566 -> 253 -> 291 -> 10
Para o vértice 11: Distância - 13, Caminho - 0 -> 98 -> 531 -> 532 -> 11
Para o vértice 12: Distância - 10, Caminho - 0 -> 123 -> 12
Para o vértice 13: Distância - 10, Caminho - 0 -> 29 -> 185 -> 379 -> 13
Para o vértice 14: Distância - 14, Caminho - 0 -> 98 -> 624 -> 14
Para o vértice 15: Distância - 13, Caminho - 0 -> 123 -> 12 -> 143 -> 15
Para o vértice 16: Distância - 13, Caminho - 0 -> 123 -> 501 -> 16
Para o vértice 17: Distância - 14, Caminho - 0 -> 29 -> 185 -> 379 -> 102 -> 429 -> 17
Para o vértice 18: Distância - 14, Caminho - 0 -> 29 -> 185 -> 18
Para o vértice 19: Distância - 17, Caminho - 0 -> 306 -> 351 -> 93 -> 152 -> 589 -> 19
Para o vértice 20: Distância - 17, Caminho - 0 -> 98 -> 120 -> 466 -> 192 -> 20
Para o vértice 21: Distância - 14, Caminho - 0 -> 98 -> 120 -> 466 -> 464 -> 21
Para o vértice 22: Distância - 17, Caminho - 0 -> 123 -> 12 -> 143 -> 15 -> 298 -> 65 -> 22
Para o vértice 23: Distância - 12, Caminho - 0 -> 306 -> 278 -> 591 -> 180 -> 23
Para o vértice 24: Distância - 15, Caminho - 0 -> 123 -> 501 -> 452 -> 24
Para o vértice 25: Distância - 15, Caminho - 0 -> 611 -> 25
Para o vértice 26: Distância - 19, Caminho - 0 -> 29 -> 185 -> 56 -> 41 -> 557 -> 26
Para o vértice 27: Distância - 16, Caminho - 0 -> 29 -> 185 -> 379 -> 43 -> 135 -> 27
Para o vértice 28: Distância - 14, Caminho - 0 -> 29 -> 185 -> 56 -> 41 -> 28
Para o vértice 29: Distância - 4, Caminho - 0 -> 29
Para o vértice 30: Distância - 12, Caminho - 0 -> 98 -> 531 -> 30
```

Exemplo do arquivo ForestPaths.txt

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 2,
2, 15, 0, 0, 0, 14, 356], [4, 6, 7, 9, 10, 24, 72, 104,
152, 300, 124, 558, 831, 1242, 7750, 624, 3744, 5608,
8404, 195000, 3124, 23430, 35135, 52695, 4881250]]
```

Exemplo do arquivo ResulTime_ForestPaths.txt

7.5. Leitura de Grafos a Partir de Arquivos:

- A função **lerGrafo** lê um grafo de um arquivo, onde o primeiro número é o número de vértices e os subsequentes são triplas (origem, destino, peso) representando as arestas.

grafo_3.txt

```
125
106 111 7
24 21 6
81 71 9
6 96 9
46 17 5
39 78 5
59 80 1
10 98 8
47 19 9
108 8 5
```

7.6. Escrita em Arquivos:

- A função **criarArquivoTexto** é responsável por escrever strings em arquivos.

7.7. Loop Principal (main):

- O loop principal executa o algoritmo em cinco grafos diferentes, medindo o tempo de execução e registrando os resultados.

7.8. Resultados:

- Os resultados incluem distâncias mínimas, caminhos mínimos e tempos de execução para cada grafo e iteração.
- Os tempos de execução e informações sobre as arestas são armazenados em arquivos.

7.9. Conclusão: O código implementa com sucesso o algoritmo de Floresta de Caminhos Ótimo, uma variação do algoritmo de Dijkstra. Ele encontra os caminhos mínimos em grafos ponderados, utilizando uma fila para a exploração eficiente dos vértices. Os resultados são detalhados e escritos em arquivos para análise posterior.

8. ALGORITMO DE JOHNSON

O código implementa o Algoritmo de Johnson para encontrar todos os caminhos mínimos em um grafo ponderado. Vamos analisar detalhadamente cada parte do código.

8.1. Estrutura do Grafo (Graph):

- A classe **Graph** representa um grafo, contendo o número de vértices e uma lista de adjacências.
- O construtor inicializa a lista de adjacências.
- O método **addEdge** adiciona uma aresta ao grafo, garantindo que a lista de adjacências tenha capacidade suficiente para acomodar os vértices especificados.

8.2. Estrutura de Pares (Pair):

- A classe **Pair** é uma simples estrutura de dados que armazena um vértice e um peso, representar as arestas do grafo.

8.3. Algoritmo de Johnson (AlgoritmoJohnson):

- O método **johnson** executa o Algoritmo de Johnson para encontrar todos os caminhos mínimos em um grafo ponderado.

```java
// Algoritmo de Johnson
public static int[] johnson(Graph graph, int startVertex, int[] parents) {
    int numVertices = graph.getNumVertices();

    // Adição do nó fictício
    graph.addEdge(numVertices, destination:0, weight:0);

    // Execução do Bellman-Ford para encontrar os menores caminhos do nó fictício para todos os outros nós
    int[] hValues = bellmanFord(graph, numVertices);

    // Verificação de ciclo negativo
    if (hValues == null) {
        System.out.println(x:"O grafo contém um ciclo negativo. O algoritmo de Johnson não pode ser aplicado.");
        return null;
    }

    // Ajuste dos pesos das arestas originais
    for (int u = 0; u < numVertices; u++) {
        for (Pair neighbor : graph.getAdjList().get(u)) {
            int v = neighbor.getVertex();
            int weightUV = neighbor.getWeight();
            neighbor.setWeight(weightUV + hValues[u] - hValues[v]);
        }
    }

    // Remoção do nó fictício
    graph.getAdjList().remove(numVertices);

    // Inicialização do array de resultados
    int[] result = new int[numVertices * numVertices];
    int index = 0;
    for (int i = 0; i < numVertices; i++) {
        int[] distances = dijkstra(graph, i);
        for (int j = 0; j < numVertices; j++) {
            result[index++] = distances[j] - hValues[i] + hValues[j];
        }
    }

    return result;
}
```

Passos do Algoritmo de Johnson:

1. **Adição do Nó Fictício:**
   - Adiciona um nó fictício ao grafo e o conecta a todos os outros vértices com peso zero.
2. **Bellman-Ford Modificado:**
   - Executa o algoritmo de Bellman-Ford modificado para encontrar os menores caminhos do nó fictício para todos os outros nós.
   - Verifica a existência de ciclos negativos.
3. **Ajuste dos Pesos:**
   - Ajusta os pesos das arestas originais com base nos resultados do Bellman-Ford.
4. **Remoção do Nó Fictício:**
   - Remove o nó fictício adicionado anteriormente.
5. **Dijkstra Modificado:**
   - Executa o algoritmo de Dijkstra modificado para cada vértice do grafo ajustado.
   - Calcula as distâncias mínimas e os caminhos mínimos.

8.4. Algoritmo de Bellman-Ford Modificado (bellmanFord):

- O método **bellmanFord** é uma versão modificada do algoritmo de Bellman-Ford para lidar com grafos com pesos negativos.

8.5. Algoritmo de Dijkstra Modificado (dijkstra):

- O método **dijkstra** é uma versão modificada do algoritmo de Dijkstra para lidar com grafos com pesos ajustados.

8.6. Execução para Vários Grafos:

- O código executa o algoritmo em vinte e cinco grafos diferentes, variando o número de vértices e arestas.
- Para cada execução, registra o tempo de execução, as distâncias mínimas e os caminhos mínimos a partir de um vértice de origem.
- Os resultados são armazenados em strings e também escritos em arquivos de saída (**Jhonson.txt** e **ResulTime_Jhonson.txt**).

```
Grafo 5 - Iteração 2 - Arestas 4686
Distâncias mínimas e caminhos mínimos a partir do vértice 0:
Para o vértice 0: Distância - 0, Caminho - 0
Para o vértice 1: Distância - 37, Caminho - 1
Para o vértice 2: Distância - 36, Caminho - 2
Para o vértice 3: Distância - 38, Caminho - 3
Para o vértice 4: Distância - 31, Caminho - 4
Para o vértice 5: Distância - 32, Caminho - 5
Para o vértice 6: Distância - 29, Caminho - 6
Para o vértice 7: Distância - 33, Caminho - 7
Para o vértice 8: Distância - 35, Caminho - 8
Para o vértice 9: Distância - 39, Caminho - 9
Para o vértice 10: Distância - 36, Caminho - 10
Para o vértice 11: Distância - 37, Caminho - 11
Para o vértice 12: Distância - 40, Caminho - 12
Para o vértice 13: Distância - 33, Caminho - 13
Para o vértice 14: Distância - 32, Caminho - 14
Para o vértice 15: Distância - 38, Caminho - 15
Para o vértice 16: Distância - 28, Caminho - 16
Para o vértice 17: Distância - 30, Caminho - 17
Para o vértice 18: Distância - 31, Caminho - 18
Para o vértice 19: Distância - 29, Caminho - 19
Para o vértice 20: Distância - 40, Caminho - 20
Para o vértice 21: Distância - 32, Caminho - 21
Para o vértice 22: Distância - 27, Caminho - 22
Para o vértice 23: Distância - 30, Caminho - 23
Para o vértice 24: Distância - 30, Caminho - 24
Para o vértice 25: Distância - 49, Caminho - 25
Para o vértice 26: Distância - 31, Caminho - 26
Para o vértice 27: Distância - 35, Caminho - 27
Para o vértice 28: Distância - 28, Caminho - 28
Para o vértice 29: Distância - 27, Caminho - 29
Para o vértice 30: Distância - 40, Caminho - 30
```

Exemplo do arquivo Jhonson.txt

```
[[12, 0, 0, 0, 0, 2, 1, 0, 7, 4, 14, 21, 20, 48, 95, 34,
191, 290, 425, 11390, 522, 8431, 17794, 22578, 852500],
[4, 6, 7, 9, 10, 24, 72, 104, 152, 300, 124, 558, 831,
1242, 7750, 624, 3744, 5608, 8404, 195000, 3124, 23430,
35135, 52695, 4881250]]
```

Exemplo do arquivo ResulTime_Jhonson.txt

8.7. Leitura de Grafos a Partir de Arquivos (lerGrafo):

- O método **lerGrafo** lê um grafo de um arquivo, onde o primeiro número é o número de vértices e os subsequentes são triplas (origem, destino, peso) representando as arestas.

grafo_5.txt

```
1   3125
2   2978 1923 6
3   303 2700 1
4   272 729 5
5   1800 248 6
6   2105 661 3
7   1954 644 4
8   2781 1027 1
9   1975 1106 5
10  2690 466 9
11  1886 2850 4
```

8.8. Escrita em Arquivos (criarArquivoTexto):

- O método **criarArquivoTexto** é responsável por escrever strings em arquivos.

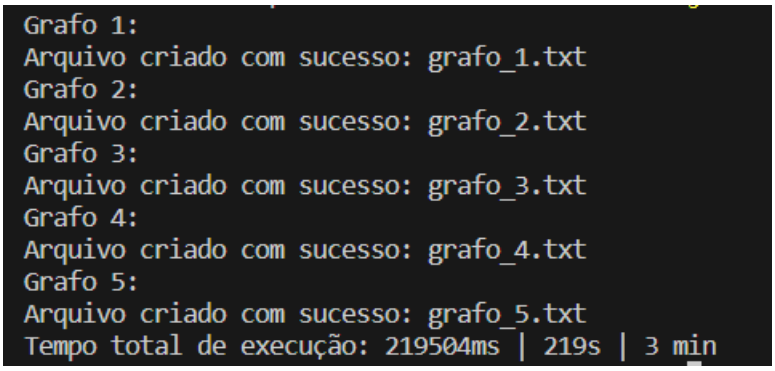
8.9. Resultados:

- O código itera sobre cinco grafos diferentes e realiza cinco iterações para cada grafo.
- Para cada iteração, são registrados os tempos de execução, o número de arestas e os resultados dos caminhos mínimos.

8.10. Conclusão: O código implementa com sucesso o Algoritmo de Johnson para encontrar todos os caminhos mínimos em grafos ponderados. A manipulação de grafos é feita de maneira eficiente usando listas de adjacência. O código é estruturado e fornece resultados detalhados, incluindo tempos de execução e caminhos mínimos. O Algoritmo de Johnson é uma extensão eficiente dos algoritmos de Bellman-Ford e Dijkstra para lidar com grafos que podem conter ciclos negativos. A complexidade total do Algoritmo de Johnson é geralmente dominada pelo passo do Bellman-Ford modificado, **resultando em O (VE + V^2 * log(V)).**

9. ANÁLISE QUANTITATIVA E COMPARATIVA DE TODOS ALGORITMOS E DISCUSSÃO DOS RESULTADOS

9.1. De início vou apresentar o tempo gasto para gerar os grafos:



Como podemos observar, foram necessários 219.504 milissegundos, o que equivale a 219 segundos ou 3 minutos, para a criação dos grafos. Cada grafo foi criado e armazenado em um arquivo separado. Esses cinco grafos foram gerados com o maior número possível de arestas, seguindo a fórmula |V| (|V| - 1) /2. O número de vértices foi definido como 5^x, onde x pertence ao conjunto {1, ..., 5}. Isso permitiu determinar o número máximo de arestas a serem utilizadas. Não sendo necessário criar vários grafos com quantidades diferentes de arestas. Dessa forma, com o número máximo de arestas criado, é possível extrair a quantidade desejada de arestas.

9.2. Agora vamos analisar o tempo gasto por cada algoritmo:

Antes da análise, esses são os números de arestas que foram criados, 25 grafos diferentes com 25 tamanhos de arestas diferentes, respeitando a formula na qual dizia: "Para as arestas, deve-se selecioná-las aleatoriamente de tal forma a obter cada uma das quantias equidistantes descritas no intervalo [|V| - 1, |V| (|V| - 1) /2]. Ou seja, o primeiro valor seria |V| - 1, o último valor, |V| (|V| - 1) /2, e os 3 valores restantes devem estar espaçados de maneira equidistante no intervalo descrito pelos dois anteriores."

Arestas: [4, 6, 7, 9, 10, 24, 72, 104, 152, 300, 124, 558, 831, 1242, 7750, 624, 3744, 5608, 8404, 195000, 3124, 23430, 35135, 52695, 4881250]

O algoritmo de **Bellman-Ford** teve os seguintes resultados em milissegundos:

1° 25°

[0, 0, 0, 0, 0, 0, 1, 0, 0,,4, 4, 5, 0, 4, 16, 5, 27, 36, 97, 825, 150, 1391, 1371, 7138, 114687]

O algoritmo de **Dijkstra** teve os seguintes resultados em milissegundos:

1° 25°

[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4 , 0, 4, 9, 0, 16, 0, 16, 8, 0, 165]

O algoritmo de **Floresta de Caminhos Ótimos** teve os seguintes resultados em milissegundos:

1° 25°

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 2, 2, 15, 0, 0, 0, 15, 356]

O algoritmo de **Floyd Warshall** teve os seguintes resultados em milissegundos:

1° 25°

[3, 0, 0, 0, 0, 0, 0, 0, 0, 5, 19, 16, 21, 16, 14, 502, 631, 229, 300, 275, 24591, 66817, 60033, 49261, 49200]

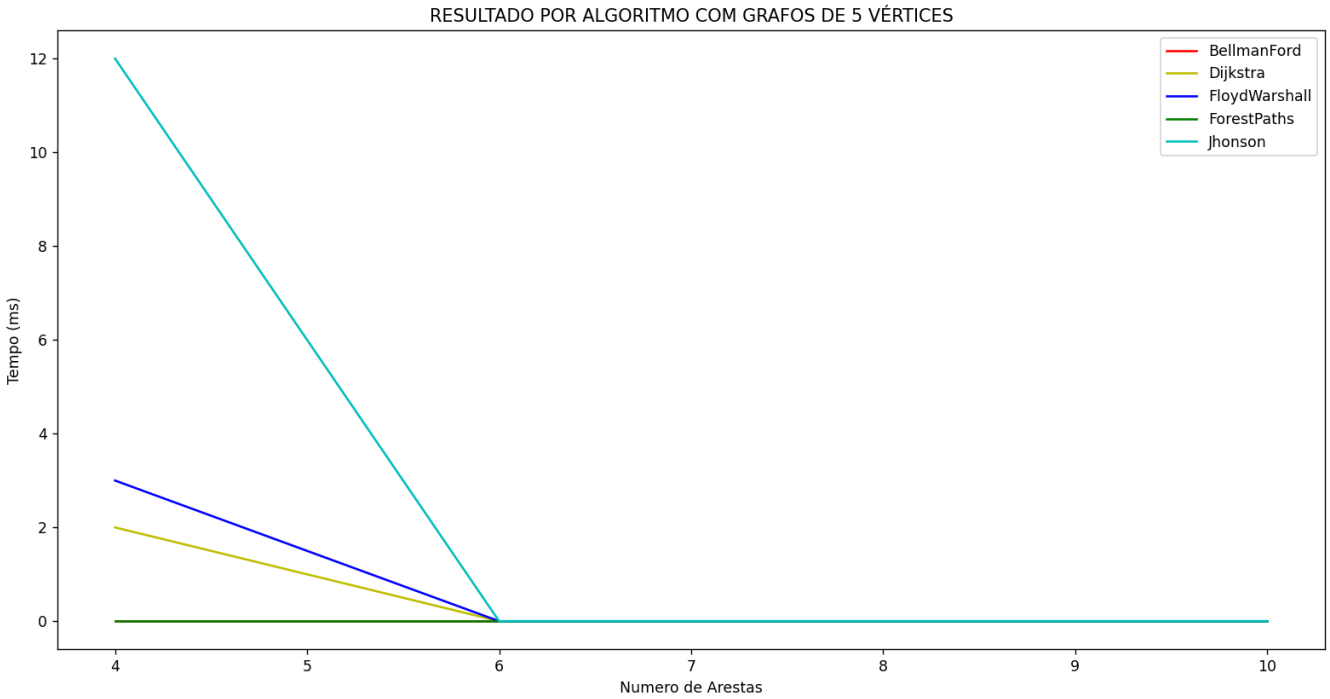
O algoritmo de **Johnson** teve os seguintes resultados em milissegundos:

1° 25°

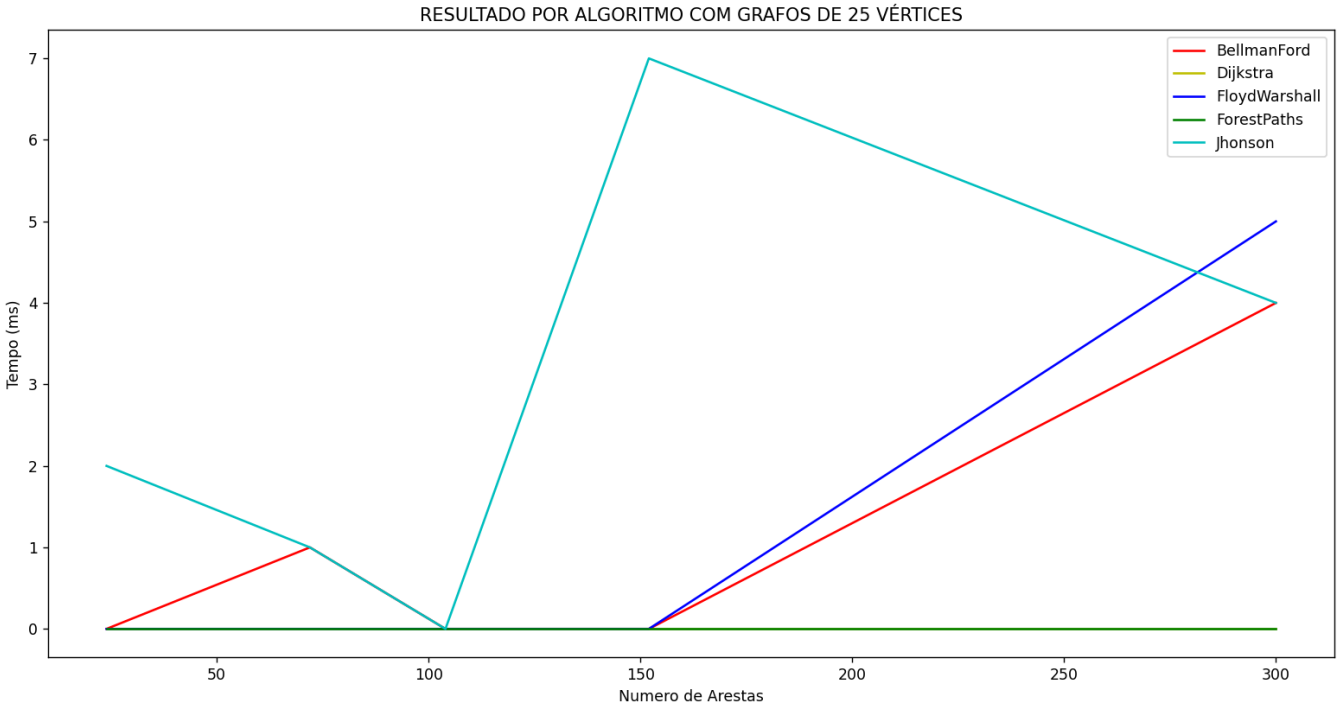
[12, 0, 0, 0, 0, 0, 2, 1, 0, 7, 4, 14, 21, 20, 48, 95, 34, 191, 290, 425, 11390, 522, 8431, 17794, 22578, 852500]

Grafo_1.txt = _5 vértices_ | Grafo_2.txt = _25 vértices _ | Grafo_3.txt = _125 vértices _ | Grafo_4.txt = _625 vértices _ | Grafo_5.txt = _3125 vértices _

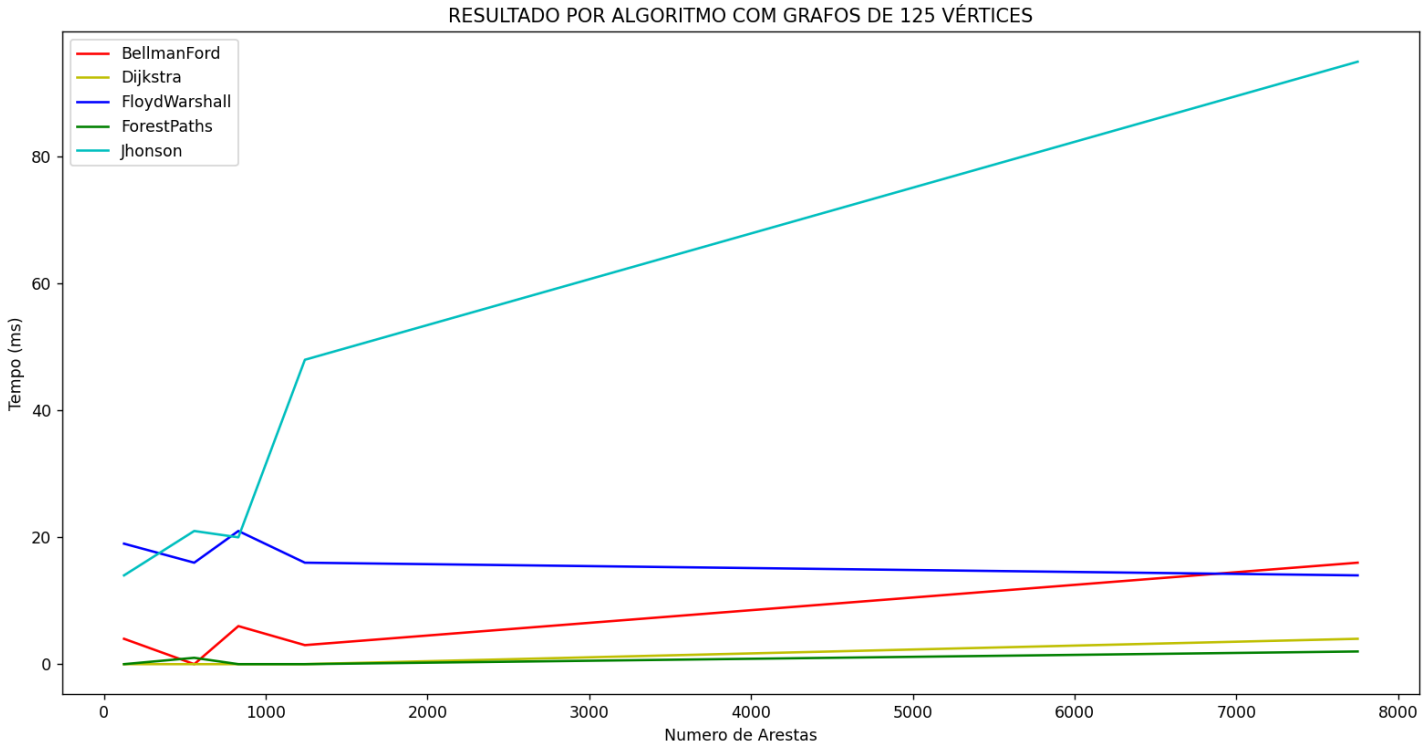
10. ANALISANDO OS GRÁFICOS



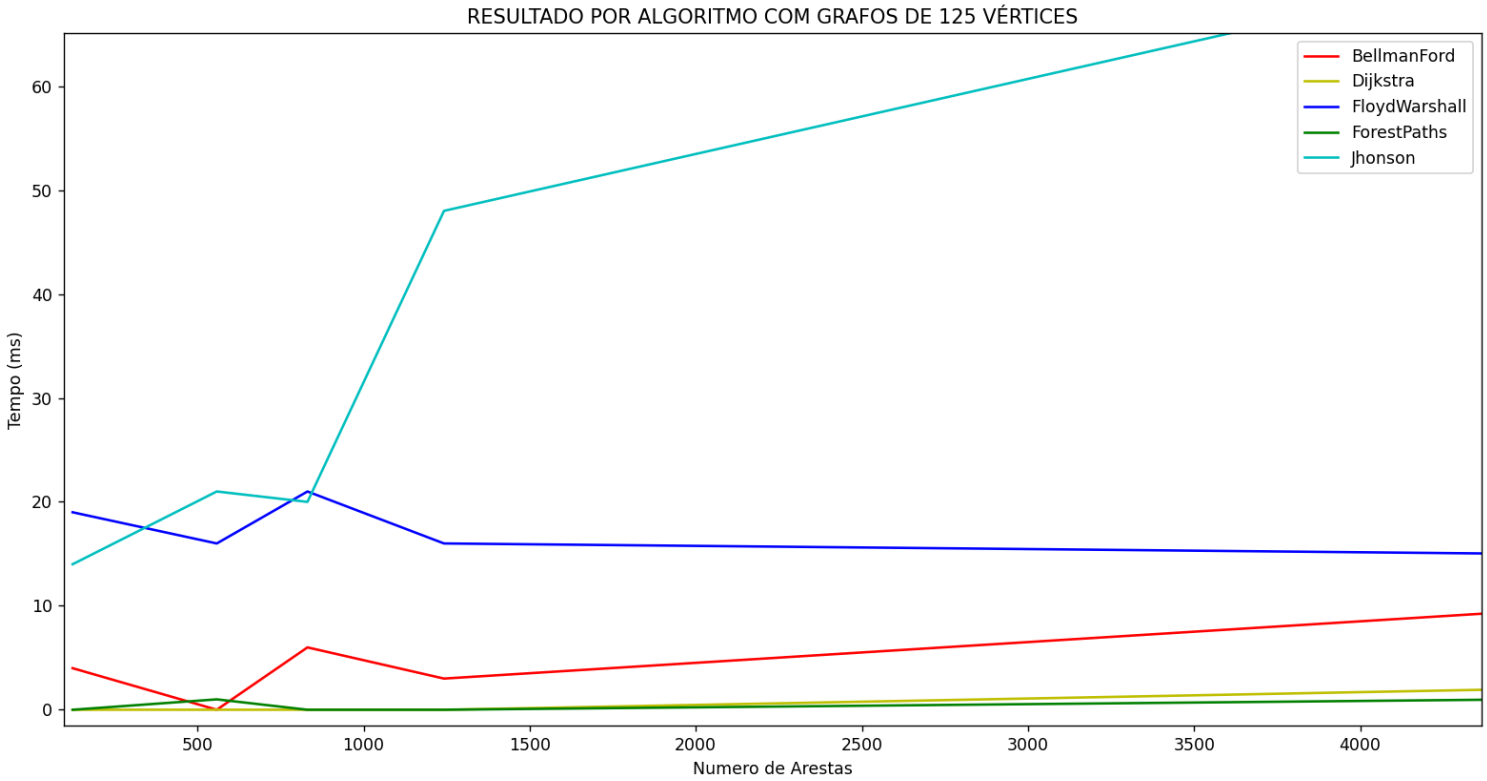
Ao analisarmos o gráfico acima, notamos que todo os algoritmos estiverem bem próximos. Mas, o algoritmo de **Johnson** foi que levou mais tempo, 12ms, na primeira análise do grafo com 4 arestas e 5 vértices. Logo após foi o algoritmo **Floyd Warshall** que levou o tempo de 3ms para analisar o grafo com 4 arestas e 5 vértices. O **Dijkstra** também teve um tempo gasto, 2ms, nas mesmas condições dos algoritmos anteriores. **Forest Path e Bellman Ford**, tiveram excelência nos testes. Nos demais grafos, após os de 4 arestas, os resultados foram iguais a todos, com tempo de 0ms.



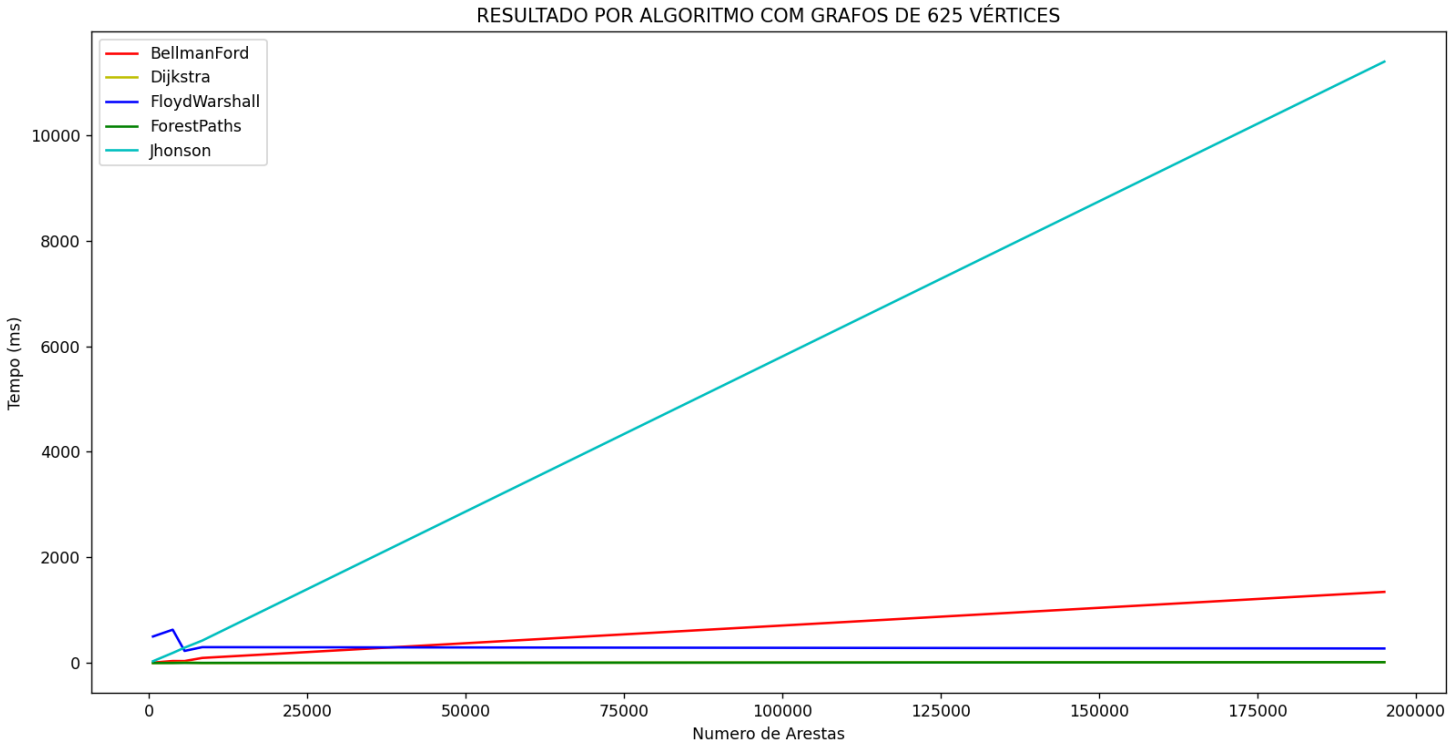
Nesta análise, vale ressaltar o algoritmo de **Bellman Ford** que não manteve os resultados excelentes, tendo oscilado com resultados como 1ms e 4ms, terminando igual ao algoritmo de Johnson que tem os piores resultado. Já o **Johnson** foi o único que teve gasto de tempo no primeiro grafo (24 arestas e 25 vértices) de 2ms, e um bom resultado nas 104 arestas, mas voltando a ter ganho e ficando na penúltima posição em desempenho, ficando à frente apenas do Floyd. O algoritmo **Floyd Warshall** apareceu, mas teve um bom desempenho no começo, depois tendo ganho de tempo, de 5ms, apenas no ultimo grafo desse arquivo (300 arestas e 25 vértices). **Forest Path** continua tendo o melhor desempenho até o momento, e nesses grafos teve a companhia do **Dijkstra** que também teve excelência nesses grafos com 25 vértices, ambos com 0ms.



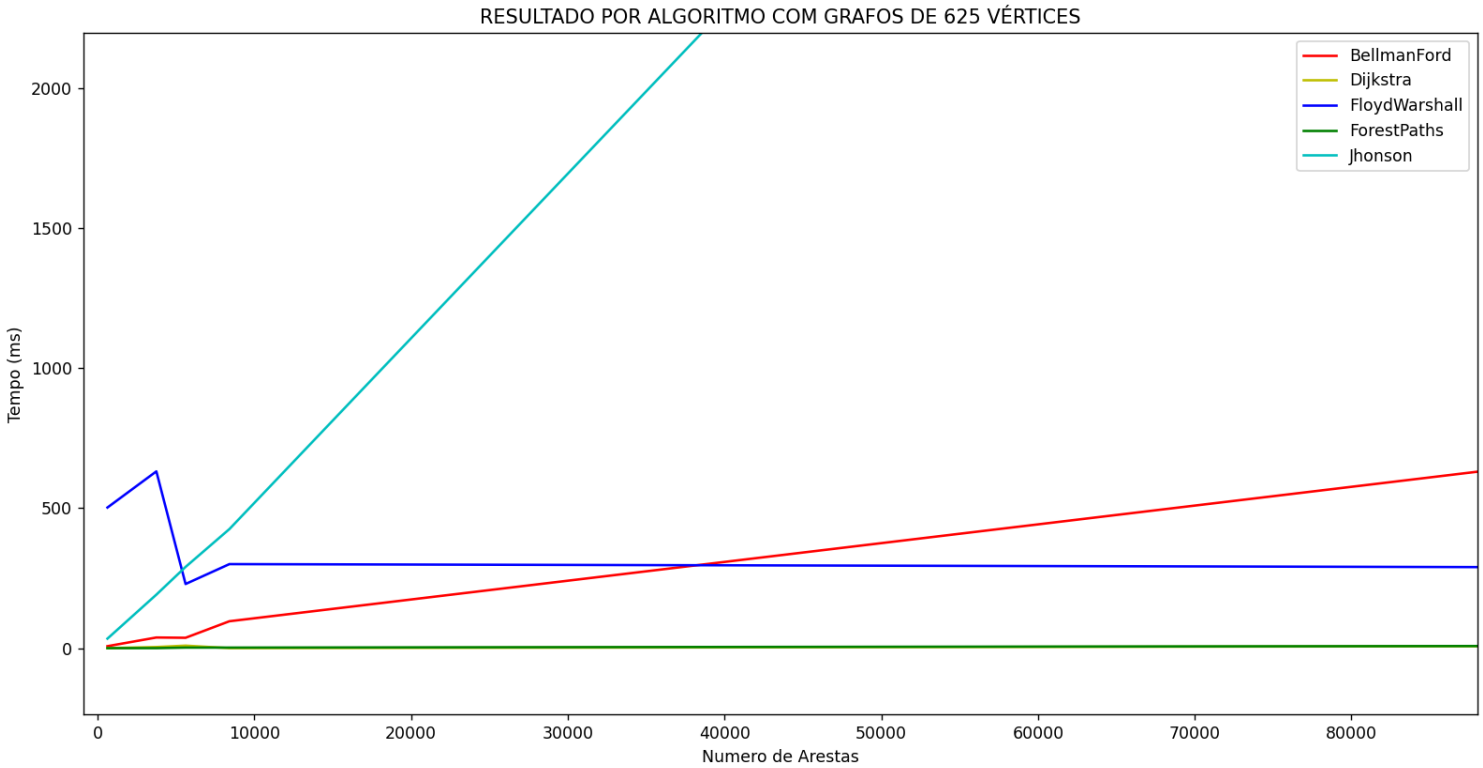
Neste gráfico fica bem nítido as três divisões. Estando sozinho e com um gasto de tempo muito alto o algoritmo de **Johnson** começou com o tempo de 14ms, ficando longe dos 0ms em todos os grafos de 125 vértice, tendo no último grafo desse arquivo (7750 arestas e 125 vértices) um tempo de 95ms, ficando sozinho no alto do gráfico com um desempenho ruim. O algoritmo **Floyd Warshall** começou com maior tempo no primeiro grafo (124 arestas e 125 vértices) com 19ms, mas foi ultrapassado pelo Johnson e ficando no meio do gráfico com desempenho regular, junto com o **Bellman** que teve um bom desempenho no início, mas não manteve e terminou ultrapassando o Floyd, (Bellman: 16ms e Floyd: 14ms).



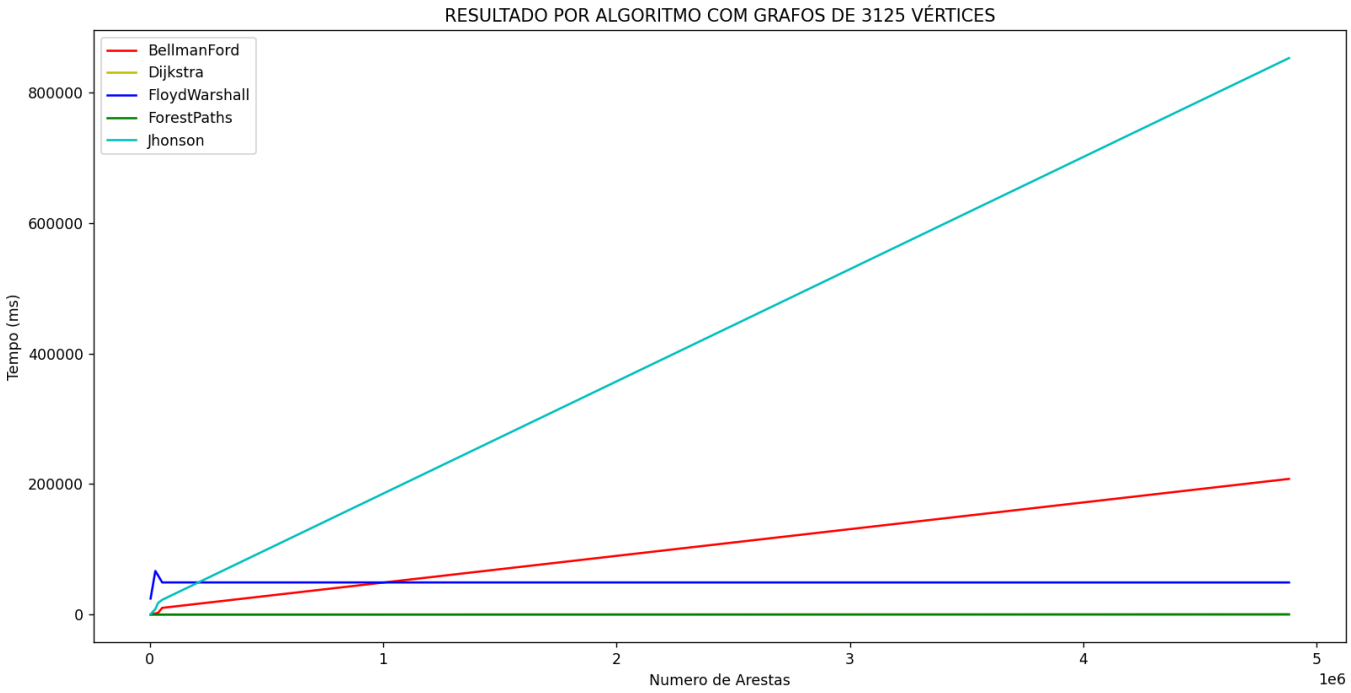
Ampliando mais o gráfico, podemos analisar um pouco melhor os restantes dos algoritmos. O **Dijkstra e Florest Path** mantiveram como os melhores algoritmos com excelentes tempos. E observasse a grande arrancada do Bellman para ultrapassar o Floyd e o Johnson literalmente isolado.



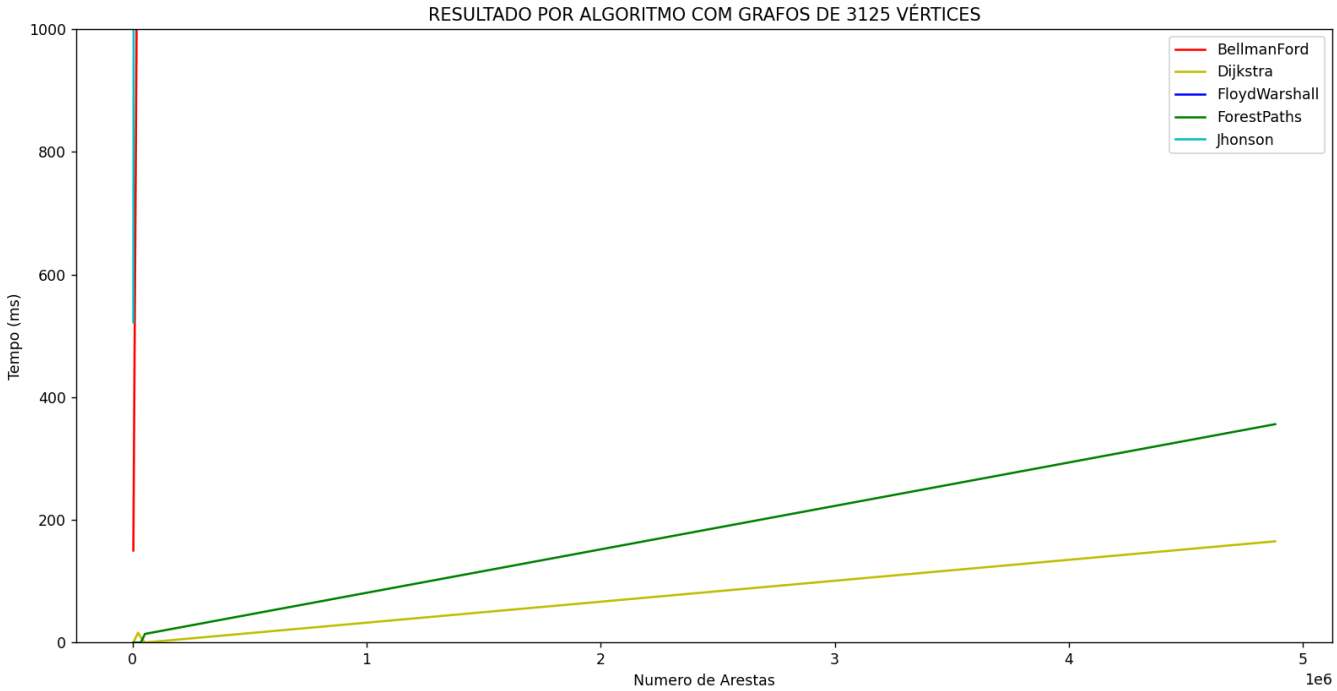
Mais uma vez o algoritmo de **Johnson** teve um desempenho ruim, apesar de ter começado com um tempo bom de 34ms, não conseguiu manter, e disparou a cada grafo, com resultados de 191ms, 290ms, 425ms e finalizando com 11390ms. Já o algoritmo de **Floyd Warshall** surpreendeu, pois teve um começo ruim com 502ms no primeiro grafo (624 arestas e 625 vértices) e 631ms no segundo grafo (3744 arestas e 625 vértices), mas recuperou com uma sequência de desempenho regular com 229ms, 300ms e 275ms. O **Bellman Ford** novamente tem um bom começo com 5ms, 27ms, 36ms e 97ms, mas no último grafo (195000 arestas e 625 vértices) um tempo de 825ms, resultado final esse que afetou seu desempenho, e acabou ultrapassando o Floyd e novamente terminado apenas melhor que o Johnson.



Nem mesmo ampliando o gráfico é possível analisar o desempenho do Dijkstra e Florest Path, ambos com desempenhos muito bom. O algoritmo de **Dijkstra** teve esse resultado 0ms, 4ms, 9ms, 0ms, 16ms. O que está bem próximo do resultado do **Florest Path** com teve esse resultado 0ms, 0ms, 2ms, 2ms, 15ms, porém, sendo melhor que o Dijkstra nesses grafos com 625 vértices.

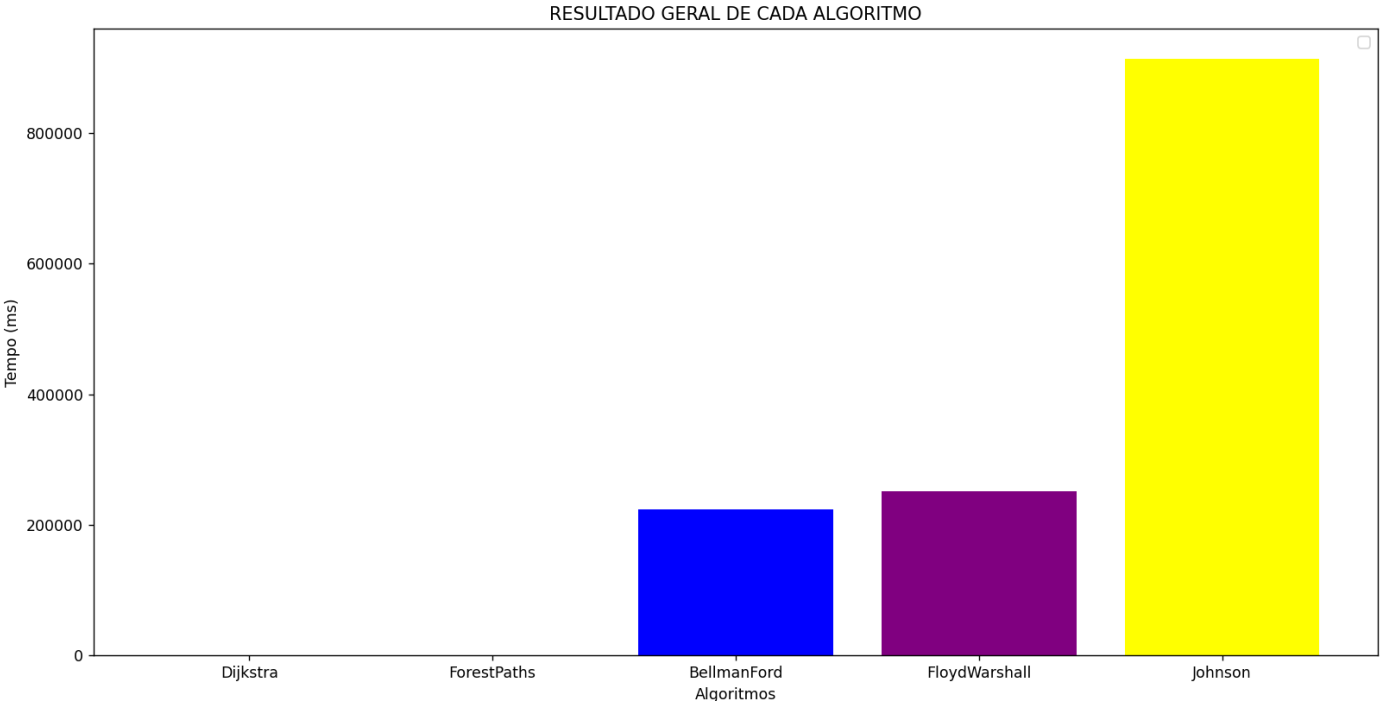


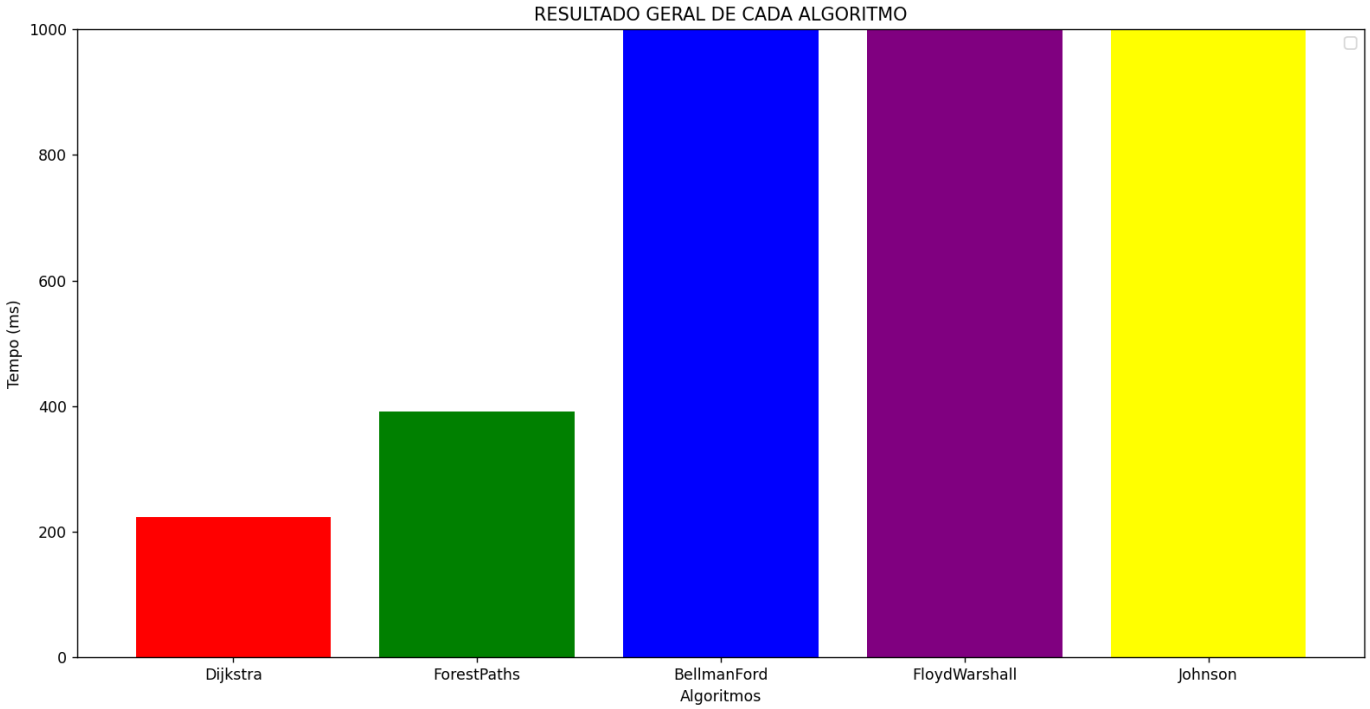
O algoritmo de **Johnson** confirmou novamente que tem um começo razoável com 522ms no primeiro grafo (3124 arestas e 3125 vértices) e após esse grafo, a situação sai do controle com grandes tempos, como 8431ms (8s), 17794ms (17s), 22578ms (22s) e 852500ms (852s ou 14min), sendo o último grafo (4881250 arestas e 3125 vértices) um tempo muito astronômico, com um tempo de 14min o Johnson ficando muito à frente de todos os outros. O **Floyd Warshall** tem um começo razoável com 24591ms (24s), mas do segundo grafo (23430 arestas e 3125 vértices) até o último grafo a uma pequena constância no tempo com os seguintes valores 66817ms (66s), 60033ms (60s), 49261ms (50s), 49200ms (50s). O algoritmo **Bellman Ford** teve um bom desempenho do primeiro grafo até o penúltimo grafo (52695 arestas e 3125 vértices) com esses resultados 150ms, 1391ms, 1371ms, 7138ms (7s), havendo uma crescente pequena até então, mas novamente no último grafo a um grande disparo com um tempo de 114687ms (114s ou 2min), afetando seu desempenho e o deixando na penúltima posição.



Mais uma vez teve que ampliar o gráfico, para ver os algoritmos de Dijkstra e Florest Path, na qual ambos têm um desempenho impecável quando comparados com os outros. O **Florest Path** tem os seguintes resultados 0ms, 0ms, 0ms, 14ms, 356ms, tempos bons e parecidos com o do **Dijkstra**. Que por sua vez, teve excelentes resultado, com 0ms, 16ms, 8ms, 0ms, 165ms, chegando a ter um resultado melhor que o Florest Path. E ambos se consolidando com os melhores algoritmos para caminhos mínimos.

11. RESULTADOS FINAIS





O resultado final, após a análise dos 25 grafos gerados, revela que o algoritmo de **Johnson** consistentemente apresenta o pior desempenho. Enquanto o **Bellman-Ford** e o algoritmo de **Floyd-Warshall** mantêm desempenhos regulares ao longo de todo o processo, a conclusão final destaca que, ao considerar todos os grafos em conjunto, ambos os algoritmos demonstram tempos de execução razoáveis. Em contrapartida, os algoritmos de **Dijkstra** e **Forest Path** destacam-se com desempenhos excepcionais, consolidando-se como opções eficientes para a resolução desse conjunto específico de problemas.

12. REFERÊNCIAS

BONDY, J. A. e MURTY, U.S.R. (1979). ``Graph Theory with Applications''. 5 edição. 271 páginas. ISBN: 0-444-19451-7. Disponível:https://www.zib.de/groetschel/teaching/WS1314/BondyMurtyGTWA.pdf

WEST, D. B. (2001). ``Introduction to Graph Theory''. 2 edição. 871 páginas. ISBN: 81-7808-830-4.

Disponível: https://athena.nitc.ac.in/summerschool/Files/West.pdf

DIESTEL, R. (2017). ``Graph Theory''. 5 edição. ISBN: 978-3-662-53621-6.

Disponível https://www.emis.de/monographs/Diestel/en/GraphTheoryII.pdf