Teoria dos Grafos

Por mago Felipe Belém

De Wanderson de Souza Pessoa

1. Introdução

A pesquisa em caminhos mínimos em grafos ponderados tem implicações profundas em diversas áreas, abrangendo desde logística e transporte até a estruturação eficiente de redes de computadores. Este projeto tem como objetivo a implementação e avaliação de vários métodos para o cálculo de caminhos mínimos em um grafo não direcionado com pesos positivos. Os algoritmos abordados incluem o renomado Algoritmo de Dijkstra, o versátil Algoritmo de Bellman-Ford, o eficiente Algoritmo de Floyd-Warshall, a inovadora Floresta de Caminhos Ótimos com a função f_{max} e o estratégico Algoritmo de Johnson.

O foco principal está na análise quantitativa e comparativa desses algoritmos, com ênfase no tempo médio de execução e desvio padrão. A variação nos resultados será investigada à medida que a quantidade de vértices do grafo varia, seguindo uma progressão determinada por 5x, onde x pertence ao conjunto {1, ..., 5}. No caso das arestas, a abordagem seleciona aleatoriamente quantidades específicas, distribuídas de maneira equidistante no intervalo definido por [|V| - 1, |V|(|V| - 1)/2], abrangendo desde o número mínimo até o máximo possível de arestas no grafo.

Essa abordagem meticulosa permitirá uma compreensão profunda do desempenho relativo desses algoritmos em diferentes configurações de grafos, proporcionando insights valiosos sobre suas eficácias e eficiências em cenários diversos. A análise minuciosa dos resultados contribuirá para o avanço do conhecimento na resolução eficaz de problemas relacionados a caminhos mínimos em grafos ponderados.

2. Referencial Teórico

2.1. Grafo

Um grafo é uma estrutura matemática que consiste em um conjunto de vértices (ou nós) e um conjunto de arestas que conectam pares de vértices. Formalmente, um grafo é definido como um par ordenado $G = (V, E)$, onde:

1. V é um conjunto finito de vértices, representando entidades distintas dentro do grafo.
2. E é um conjunto de arestas, cada uma conectando um par de vértices. As arestas podem ser direcionadas ou não direcionadas, dependendo se a ordem dos vértices importa ou não.

2.2. Grafo Ponderado

Um grafo ponderado é uma extensão do conceito básico de um grafo, no qual cada aresta possui um peso associado. Formalmente, um grafo ponderado pode ser definido como um par ordenado $G = (V, E)$, onde:

1. V é um conjunto finito de vértices, representando entidades distintas dentro do grafo.
2. E é um conjunto de arestas, cada uma conectando um par de vértices, assim como em um grafo não ponderado.
3. Cada aresta e em E é associada a um peso $\omega(e)$, que é um valor numérico que representa a medida ou custo da relação representada pela aresta.

2.3. Grafo Direcionado

Um grafo direcionado, também conhecido como digrafo, é uma estrutura matemática que consiste em um conjunto de vértices (ou nós) é um conjunto de arestas direcionadas que conectam pares de vértices. A direção das arestas indica a relação de origem e destino entre os vértices. Formalmente, um grafo direcionado é definido como um par ordenado $G = (V, E)$, onde:

1. V é um conjunto finito de vértices, representando entidades distintas dentro do grafo.
2. E é um conjunto de arestas direcionadas, cada uma representando uma relação direcional entre dois vértices. Formalmente, cada aresta é um par ordenado (u, v), onde u é o vértice de origem e v é o vértice de destino.

Um grafo direcionado pode ser representado de maneira visual por meio de setas que indicam a direção das relações entre os vértices. A ausência de uma seta entre dois vértices não implica automaticamente a ausência da relação; a direção das arestas é crucial na interpretação da conectividade.

2.4. Relaxamento

Suponha-se que tenhamos uma vértice u e um vértice v conectados por uma aresta (u, v) no grafo, e $dist(u)$ seja a estimativa atual da distância mais curta do vértice de origem até u. O relaxamento é então definido da seguinte forma:

Se a distância estimada de u mais o peso da aresta (u, v) for menor do que a distância estimada atual de v (ou se $(d(u) + \omega(u, v) < dist(v))$), então a distância estimada de v é atualizada para $(dist(u) + \omega(u, v)$.

Em termos matemáticos, o relaxamento pode ser expresso como:

$$d(v) \leftarrow min(d(v), (d(u) + \omega(u, v))$$

2.5. Arestas de peso negativo

Arestas de peso negativo referem-se a arestas em um grafo ponderado cujos pesos associados são valores numéricos negativos. Formalmente, em um grafo ponderado $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, uma aresta (u, v) com peso negativo seria representada como $\omega(u, v) < 0$.

A existência de arestas de peso negativo pode ter implicações significativas em problemas relacionados a caminhos mais curtos, como no algoritmo de Dijkstra. A presença de arestas de peso negativo pode tornar o problema mais complexo, uma vez que caminhos aparentemente mais longos podem se tornar mais curtos quando incluem arestas de peso negativo.

Algoritmos como o algoritmo de Bellman-Ford são projetados para lidar com arestas de peso negativo, pois podem detectar ciclos de peso negativo no grafo. Um ciclo de peso negativo é um ciclo no qual a soma dos pesos das arestas é negativa, e em tal caso, o algoritmo indica a presença de caminhos de comprimento infinitamente negativo.

Em resumo, arestas de peso negativo são aquelas em que o valor associado à aresta é menor que zero em um grafo ponderado, e sua presença pode afetar a solução de problemas específicos, exigindo algoritmos adequados para lidar com essas condições.

2.6. Detecção de Ciclos negativos

A detecção de ciclos negativos em um grafo ponderado é um procedimento que identifica se o grafo possui um ciclo cuja soma dos pesos das arestas é negativa. Isso é particularmente relevante quando se trabalha com algoritmos que envolvem caminhos mais curtos, pois ciclos negativos podem levar a problemas de definição de distância e caminho mínimo. Formalmente, a detecção de ciclos negativos pode ser definida da seguinte maneira:

Dado um grafo ponderado, a detecção de ciclos negativos pode ser realizada por meio de um algoritmo. Este capaz de identificar a presença de ciclos negativos no grafo ao percorrer todas as arestas repetidamente e atualizar as distâncias estimadas dos vértices. Se, em qualquer iteração, uma atualização de distância ocorrer, isso indica a presença de um ciclo negativo.

Formalmente, um ciclo negativo é detectado se, após $|V| - 1$ iterações do algoritmo de Bellman-Ford (onde $|V|$ é o número de vértices), ainda houver uma atualização de distância em pelo menos uma aresta no grafo. A existência de tal atualização indica que o grafo contém um ciclo negativo.

2.7. Complexidade Temporal

A complexidade temporal, também conhecida como complexidade de tempo, refere-se à medida do tempo de execução de um algoritmo em relação ao tamanho da entrada. Essa medida é utilizada para analisar o desempenho de algoritmos e compreender como o tempo de execução aumenta à medida que o tamanho do problema cresce.

A notação big-O (O) é comumente usada para descrever a complexidade temporal de um algoritmo.

Por exemplo:

- Se um algoritmo tem complexidade temporal $\theta(n^2)$, isso significa que o tempo de execução do algoritmo é, no máximo, proporcional a n^2, onde n é o tamanho da entrada.
- Se um algoritmo tem complexidade temporal $\theta(n * log\, n)$, isso significa que o tempo de execução do algoritmo é, no máximo, proporcional a $n * log\, n$.

É importante notar que a complexidade temporal não fornece uma medida exata do tempo de execução; em vez disso, ela descreve como o tempo de execução aumenta à medida que o tamanho da entrada cresce. A complexidade temporal é uma ferramenta valiosa para comparar algoritmos e avaliar seu desempenho em diferentes cenários.

2.8. Prioridade de Vértices

A prioridade de vértices refere-se à ordem em que os vértices são processados em algoritmos que envolvem grafos ou estruturas de dados que utilizam prioridades. Uma fila de prioridade é comumente usada para determinar a ordem em que os vértices são acessados ou removidos de uma estrutura de dados, como em algoritmos de busca em grafos ou algoritmos baseados em prioridade.

Formalmente, uma fila de prioridade é uma estrutura de dados que mantém uma coleção de elementos, cada um associado a uma prioridade ou chave. A prioridade é usada para determinar a ordem de acesso ou remoção dos elementos da fila. Em um contexto de grafos, os vértices podem ser os elementos da fila de prioridade, e a prioridade pode ser determinada por uma função associada a cada vértice.

Portanto, a prioridade de vértices refere-se à ordem ou sequência em que os vértices são tratados durante a execução de algoritmos, e a fila de prioridade é uma estrutura de dados fundamental para implementar essa ordenação com base em critérios específicos associados a cada vértice.

2.9. Algoritmo Guloso

Um algoritmo guloso (também conhecido como algoritmo ganancioso ou voraz) é um método de resolução de problemas que segue a abordagem de fazer a escolha localmente ótima em cada etapa com a esperança de alcançar uma solução globalmente ótima. Formalmente, um algoritmo guloso pode ser definido da seguinte maneira:

Dado um conjunto de escolhas em um determinado problema, um algoritmo guloso faz a escolha que parece ser a melhor no momento, sem se preocupar com as escolhas futuras. A estratégia geralmente envolve a seleção de uma opção que otimiza um critério específico localmente, esperando que essa escolha leve a uma solução globalmente otimizada.

A estrutura básica de um algoritmo guloso pode ser descrita através dos seguintes passos:

1. Inicialização: Inicializar uma solução parcial vazia ou completa, dependendo do problema.

2. Escolha Gulosa: Fazer a escolha localmente ótima com base em algum critério. Esta escolha é geralmente determinada por uma função de avaliação ou heurística.

3. Atualização da Solução: Atualizar a solução parcial com a escolha feita na etapa anterior.

4. Critério de Parada: Verificar se a solução alcançada é válida ou suficientemente boa. Se for o caso, o algoritmo termina; caso contrário, o algoritmo continua iterando.

É importante notar que a estratégia gulosa não garante sempre a obtenção da solução globalmente ótima para todos os problemas. No entanto, em muitos casos, ela fornece soluções suficientemente boas com um custo computacional menor em comparação com algoritmos exatos.

2.10. Matriz de Adjacência

A matriz de adjacência é uma representação tabular de um grafo, onde as relações entre os vértices são expressas por meio de uma matriz. Formalmente, para um grafo não direcionado com n vértices, a matriz de adjacência é uma matriz quadrada A de ordem $n * n$, onde n é o número de vértices no grafo. Se o grafo é ponderado, os elementos da matriz podem representar os pesos das arestas; caso contrário, a matriz pode ser binária, indicando apenas a presença ou ausência de uma aresta.

A representação formal de uma matriz de adjacência A é dada por $A = a_{i,j}$, onde $a_{i,j}$ é o elemento localizado na i-ésima linha e j-ésima coluna da matriz. A interpretação desse elemento depende do tipo de grafo e se é ponderado ou não. Em um grafo não ponderado, $a_{i,j}$ é 1 se existe uma aresta entre os vértices i e j, e é 0 caso contrário. Em um grafo ponderado, $a_{i,j}$ pode ser o peso da aresta entre i e j, ou uma marcação especial (como ∞) para indicar a ausência de uma aresta.

Se o grafo é direcionado, a matriz de adjacência pode não ser simétrica, pois a presença de uma aresta de i para j não implica automaticamente na presença da aresta de j para i. Em um grafo não direcionado, a matriz de adjacência é simétrica, e $a_{i,j} = a_{j,i}$.

A utilização da matriz de adjacência é conveniente para verificar rapidamente se há uma aresta entre dois vértices e para computar propriedades relacionadas à conectividade do grafo. No entanto, ela pode ser menos eficiente em termos de espaço para grafos esparsos (com poucas arestas) em comparação com outras representações como a lista de adjacência.

3. Descrição dos Algoritmos

3.1. Bellman-Ford

O Algoritmo de Bellman-Ford é especialmente projetado para operar em grafos ponderados, destacando-se pela eficiência na resolução de problemas que exigem a consideração de diversos custos ao percorrer arestas. Utilizando uma técnica de relaxamento iterativo, o algoritmo adapta-se de forma versátil a diferentes configurações de grafos, buscando encontrar as distâncias mais curtas entre os vértices.

Uma característica distintiva é a sua capacidade de lidar com arestas de peso negativo, tornando-o uma escolha apropriada para cenários em que outras abordagens podem falhar. Após $|V| - 1$ iterações, onde $|V|$ representa o número de vértices, o algoritmo eficientemente detecta a presença de ciclos negativos, proporcionando insights valiosos sobre a estrutura do grafo.

Embora ofereça flexibilidade, é importante considerar a complexidade de tempo $O(|V| * |E|)$, onde $|E|$ é o número de arestas, o que destaca a necessidade de avaliação cuidadosa, especialmente em grafos densos, onde outras abordagens podem apresentar maior eficiência. O Algoritmo de Bellman-Ford encontra diversas aplicações práticas em redes de comunicação, otimização de rotas e análise de circuitos elétricos, destacando-se como uma ferramenta crucial na determinação de caminhos mínimos em contextos fundamentais.

3.2. Dijkstra

O Algoritmo de Dijkstra foi meticulosamente projetado para operar em grafos ponderados, destacando-se por sua priorização na eficiência ao buscar caminhos mais curtos. Ao empregar uma fila de prioridade, o Dijkstra explora os vértices de maneira ordenada, assegurando eficiência ao selecionar os próximos vértices a serem considerados no processo.

Este algoritmo utiliza a técnica de relaxamento de forma iterativa para encontrar caminhos mais curtos, ajustando as estimativas de distância conforme necessário. A sua natureza gulosa, ao escolher sempre a aresta com a menor estimativa de distância, contribui significativamente para a eficácia na busca por caminhos mais curtos.

A complexidade temporal do Algoritmo de Dijkstra é geralmente expressa como $O((|V| + |E|) * log |V|)$,onde $|V|$ é o número de vértices e $|E|$ é o número de arestas, evidenciando sua eficiência, especialmente em grafos esparsos. Nas aplicações práticas, o Algoritmo de Dijkstra desempenha um papel fundamental em redes de computadores, otimização de rotas, logística e em qualquer contexto em que a determinação de caminhos mais curtos seja uma necessidade crucial.

3.3. Floyd-Warshall

O Algoritmo de Floyd-Warshall é projetado para ser aplicável a grafos ponderados, destacando-se pela eficácia em encontrar todos os caminhos mais curtos entre vértices em uma única execução. Utilizando a matriz de adjacência, o algoritmo armazena e calcula de maneira eficiente os caminhos mais curtos entre todos os pares de vértices.

Ao realizar iterações, o algoritmo progressivamente aprimora as estimativas de caminhos mais curtos, empregando a ideia de programação dinâmica. Explorando subestruturas ótimas durante o cálculo dos caminhos mais curtos entre todos os pares de vértices, o algoritmo assegura uma solução globalmente ótima.

A complexidade temporal do Floyd-Warshall, expressa como $O(|V|^3)$, onde $|V|$ é o número de vértices ,destaca sua eficiência, especialmente em grafos de tamanho moderado. Nas aplicações práticas, o Algoritmo de Floyd-Warshall é amplamente utilizado em redes de transporte, modelagem de sistemas de comunicação e em situações onde o cálculo de todos os caminhos mais curtos entre todos os pares de vértices é essencial para uma abordagem eficiente. Compreender esses conceitos é crucial para a aplicação eficaz desse algoritmo, destacando sua utilidade em contextos que demandam a determinação completa de caminhos mais curtos.

3.4. Floresta de Caminhos Ótimos* com a função f_{max}

A Floresta de Caminhos Ótimos com f_{max}, fundamentada no algoritmo de Dijkstra, compartilha sua estrutura conceitual para a busca de caminhos mínimos em grafos ponderados. Essa condição específica otimiza o processo de relaxamento, considerando o máximo entre a distância atual e o peso da aresta durante a atualização das distâncias estimadas. A estratégia da Floresta de Caminhos Ótimos é mantida, explorando caminhos mínimos por meio de uma floresta de árvores de busca, onde cada árvore representa a busca por caminhos mínimos a partir de um vértice específico.

A eficiência do algoritmo é evidenciada pela inclusão da função f_{max}, proporcionando uma resposta ágil em ambientes dinâmicos, como o roteamento de pacotes em redes. A complexidade temporal da Floresta de Caminhos Ótimos com f_{max} é influenciada pela combinação das operações do algoritmo de Dijkstra e das árvores de busca, resultando em uma complexidade aproximada de $O((|V| + |E|) * log |V|)$, onde $|V|$ é o número de vértices e $|E|$ é o número de arestas.

Essa melhoria na eficiência destaca a aplicabilidade do algoritmo em contextos nos quais a exploração eficiente de caminhos mínimos é crucial, como em sistemas de comunicação e roteamento de dados.

3.5. Jhonson

Esta abordagem é especialmente indicada para grafos ponderados, oferecendo uma solução eficaz na busca por caminhos mínimos. A técnica do Algoritmo de Johnson incorpora uma estratégia integrada que aborda desafios específicos relacionados a arestas de peso negativo, fornecendo uma solução robusta na determinação de caminhos mínimos.

A fase de conversão, realizada através do algoritmo de Bellman-Ford, desempenha um papel crucial na detecção de ciclos negativos, permitindo o tratamento eficiente de arestas com pesos negativos. O processo de comparar e, se necessário, atualizar as distâncias estimadas entre os vértices, utilizando potenciais para aprimorar a eficiência do relaxamento, combinado ao algoritmo de Dijkstra, possibilita o cálculo eficiente de caminhos mínimos em um grafo modificado.

A estratégia integrada do Algoritmo de Johnson, ao unir elementos do Bellman-Ford e Dijkstra, proporciona uma abordagem abrangente que supera obstáculos relacionados a arestas de peso negativo e busca por caminhos mínimos. Apesar da eficiência do algoritmo, sua complexidade temporal, influenciada por essa combinação sinérgica, torna-o particularmente adequado para grafos de tamanho moderado. A complexidade temporal aproximada é da ordem de $O(|V|^2 * log|V| + |V| * |E|)$, onde $|V|$ é o número de vértices e $|E|$ é o número de arestas.

Em aplicações práticas, o Algoritmo de Johnson desempenha um papel essencial em redes de comunicação, roteamento em sistemas de transporte e em cenários nos quais a manipulação de grafos ponderados, incluindo arestas de peso negativo, é crucial para alcançar uma eficiente determinação de caminhos mínimos.

4. Pseudocódigo

4.1. Bellman-Ford

1. $Para\ todo\ vértice\ v \in V(G)\ faça$:
   $dist(v) \leftarrow \infty$
2. $dist(inicial) \leftarrow 0$
3. $Para\ i = 1, \ldots, |V(G)| - 1\ faça$:
   $Para\ cada\ aresta\ (v, w) \in E(G)\ faça$:
   $Se\ dist(w) > dist(v) + \omega(v, w)\ então$:
   $dist(v) \leftarrow dist(w) + \omega\ (v, w)$
   $Se\ não\ houver\ alteração\ em\ dist\ então\ algoritmo\ finalizado$

O algoritmo estava preparado para lidar com ciclos negativos, o que não se aplica ao novo cenário do grafo. Portanto, certas partes do código foram ajustadas para refletir essa característica, eliminando verificações ou processos específicos relacionados a pesos negativos.

Além disso, foi incorporada uma otimização que verifica se houve alguma alteração nas distâncias durante uma iteração. Se nenhuma alteração for detectada, o algoritmo interrompe suas iterações restantes, economizando recursos computacionais. Essa otimização é especialmente útil em cenários onde o grafo não sofre mais alterações nas distâncias após um determinado ponto, melhorando a eficiência do algoritmo.

4.2. Dijkstra

1. $Para\ todo\ v \in V(G)\ faça$:
   $dist(v) \leftarrow \infty$
   $Adiciona\ v\ em\ não Visitados$
2. $dist(inicial) \leftarrow 0$
3. $Ordena\ não Visitados$
4. $Enquanto\ houver\ vértices\ em\ não Visitados\ faça$:
   $v \leftarrow Primeira\ posição\ de\ não\ Visitados$
   $Para\ cada\ aresta\ (atual, w) \in E(G)\ faça$:
   $Se\ dist(w) > dist(atual) + \omega\ (atual, w)\ então$:
   $dist(w) \leftarrow dist(atual) + \omega\ (atual, w)$
   $visita\ atual\ e\ o\ remove\ de\ não Visitados$
   $Ordena\ não Visitados$

4.3. Floyd-Warshall

1. $Para\ i = 0, \dots, |V| - 1\ faça$:
    $Para\ j = 0, \dots, |V| - 1\ faça$:
        $dist_{i,j} \leftarrow matrizAdj_{i,j}$
    $dist_{i,j} \leftarrow 0$
2. $Para\ k = 0, \dots, |V| - 1\ faça$:
    $Para\ i = 0, \dots, |V| - 1\ faça$:
        $Para\ j = 0, \dots, |V| - 1\ faça$:
            $Se\ dist_{i,j} > dist_{i,k} + dist_{k,j}\ então$:
                $dist_{i,j} \leftarrow dist_{i,k} + dist_{k,j}$

No código, uma matriz de adjacência é inicializada antes da execução do algoritmo. Apenas as distâncias em relação ao vértice inicial são gravadas no arquivo.

4.4. Floresta de Caminhos Ótimos* com a função f_{max}

1. $Para\ todo\ v \in V(G)\ faça$:
    $Dist(v) \leftarrow \infty$
    $Adiciona\ v\ em\ nãoVisitados$
2. $Dist(inicial) \leftarrow 0$
3. $Ordena\ nãoVisitados$
4. $Enquanto\ houver\ vértices\ em\ nãoVisitados\ faça$:
    $v \leftarrow Primeira\ posição\ de\ não\ Visitados$
    $Para\ cada\ aresta\ (atual, w) \in E(G)\ faça$:
        $Se\ dist(w) > Maior(dist(atual), \omega\ (atual, w))\ então$:
            $dist(w) \leftarrow Maior(dist(atual) + \omega\ (atual, w))$
    $visita\ atual\ e\ o\ remove\ de\ nãoVisitados$
    $Ordena\ nãoVisitados$

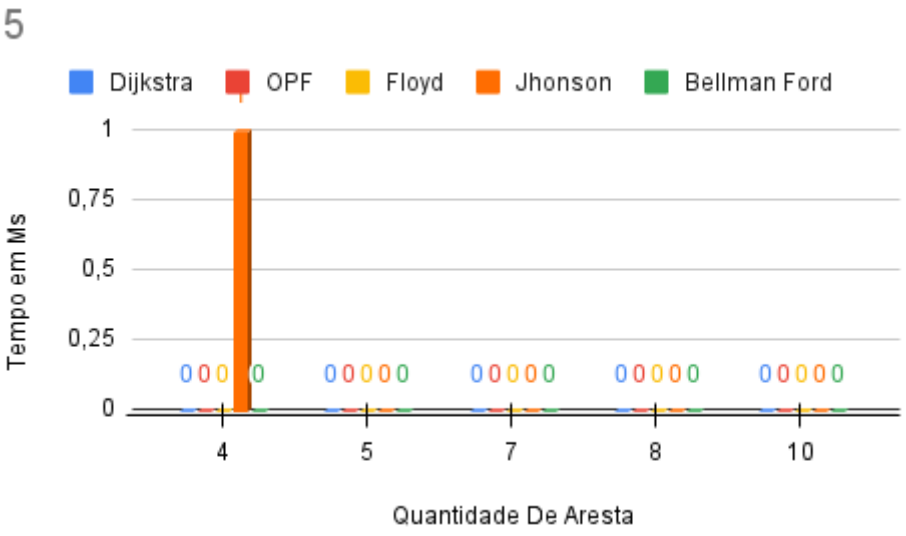
4.5. Jhonson

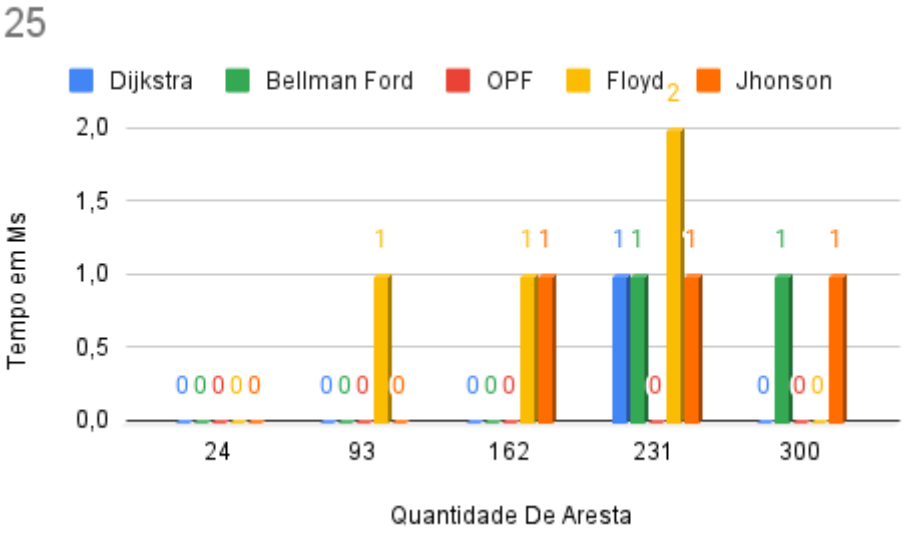
1. $Para\ todo\ v \in V(G)\ faça$:
   $Dist(v) \leftarrow \infty$
2. $Para\ todo\ v \in V(G)$:
   $\omega(aux, v) \leftarrow 0$
   $E(G)\ \cup\ (aux, v)$
3. $V(G)\ \cup\ aux$
4. $ListaB \leftarrow BellmanFord\ do\ grafo\ clone$
5. $V(G)/aux$
6. $Para\ (v, w) \in E(G)\ faça$:
   $\omega(v, w) \leftarrow \omega(v, w)\ +\ dist(v)\ -\ dist(w) \in ListaB)$
7. $ListaD \leftarrow Dijkstra\ do\ grafo\ original$
8. $Para\ todo\ v \in V(G)\ faça$:
   $Dist(v) \leftarrow (dist(v) \in ListaD)\ -\ (dist(inicial) \in ListaB)$
   $+\ (dist(v) \in ListaB)$

O algoritmo foi modificado em relação à versão original. Agora, a execução do Dijkstra é restrita ao vértice inicial, pois não há necessidade de calcular para todos os vértices. Além disso, dado que não há pesos negativos, não é necessário testar a presença de ciclos negativos, eliminando essa verificação do Bellman-Ford. Também é importante notar que as distâncias não são inicializadas com ∞ no Bellman-Ford e Dijkstra.

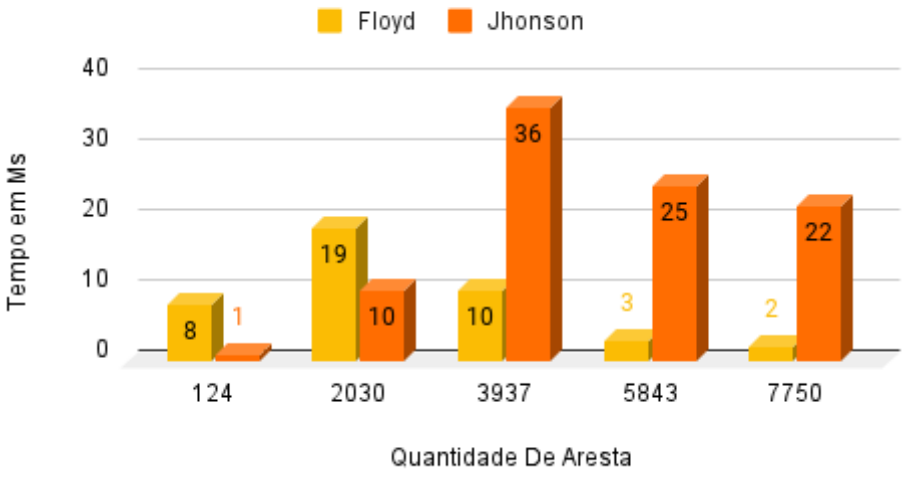
Essas modificações simplificam o processo, tornando o algoritmo mais eficiente e adequado ao contexto específico do grafo sem pesos negativos.

5. Análise comparativa

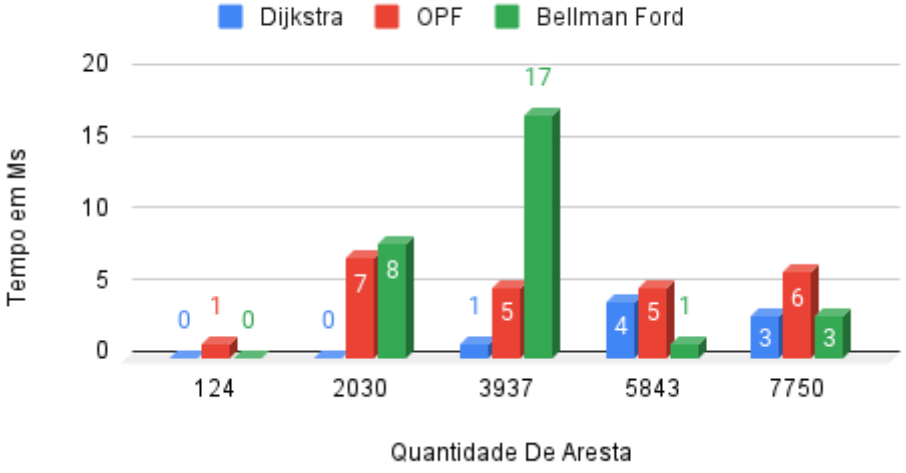




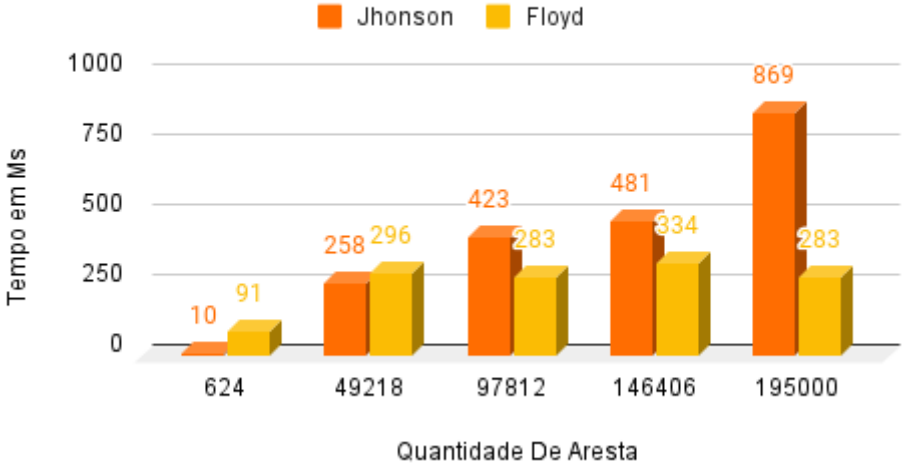
125


Floyd
Jhonson
40
30
20
10
0
Tempo em Ms
8
1
19
10
10
36
3
25
2
22
124
2030
3937
5843
7750
Quantidade De Aresta

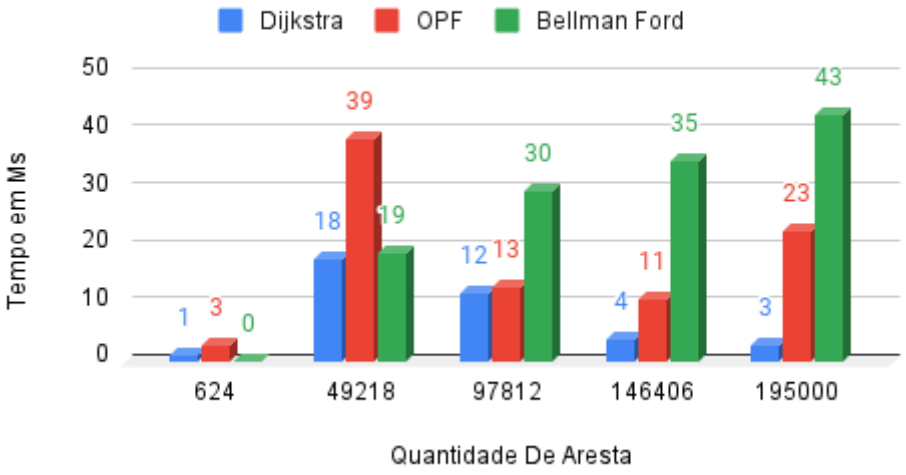
125


Dijkstra
OPF
Bellman Ford
20
15
10
5
0
Tempo em Ms
0
1
0
0
7
8
1
5
17
4
5
1
3
6
3
124
2030
3937
5843
7750
Quantidade De Aresta

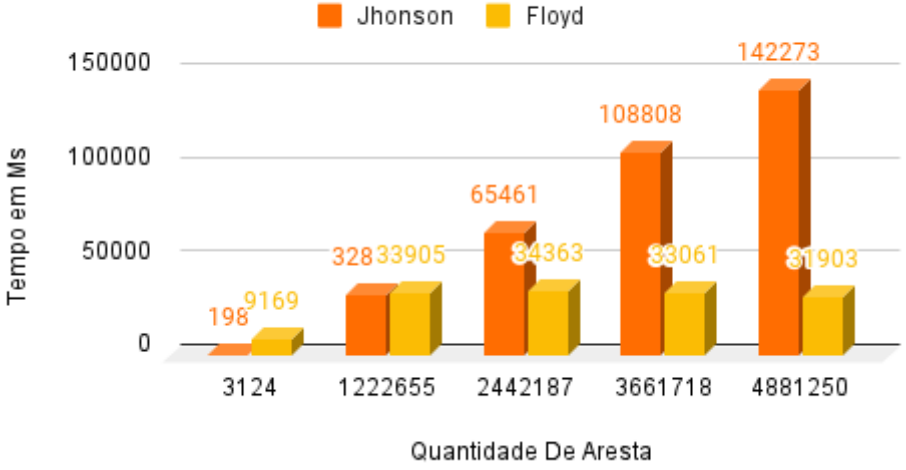
625


Jhonson
Floyd
1000
750
500
250
0
Tempo em Ms
10
91
258
296
423
283
481
334
869
283
624
49218
97812
146406
195000
Quantidade De Aresta

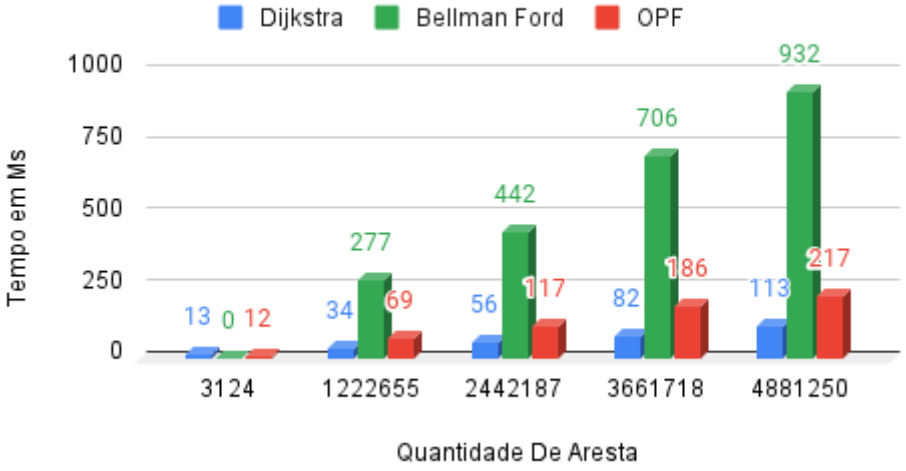
625



3125



3125



6. Conclusão

Com base nas análises realizadas em um sistema, equipado com um processador Ryzen 3 2200G e 16GB de RAM, e considerando um grafo sem arestas de peso negativo, constatou-se que o método de Dijkstra demonstrou uma eficiência consistentemente superior. Em seguida, a Floresta de Caminhos Ótimos* com a função f_{max} apresentou bom desempenho, seguida pelo algoritmo de Bellman-Ford. Posteriormente, o algoritmo de Floyd-Warshall demonstrou um desempenho inferior, e o algoritmo de Johnson foi o menos eficiente em termos de tempo de execução.

Vamos analisar a ordem de desempenho dos algoritmos com base nas características de implementação e nas complexidades temporais. O algoritmo de Dijkstra e a Floresta de Caminhos Ótimos* apresentam eficiência semelhante devido à implementação praticamente idêntica, com uma diferença notável na função f_{max}. A complexidade destes algoritmos é aproximadamente $O(|V| + |E| * log|V|)$, e o aumento no número de vértices afeta proporcionalmente o tempo de execução.

Quanto ao Bellman-Ford, observamos um pico no tempo de execução para grafos com 125 vértices. Isso está diretamente relacionado à complexidade do algoritmo, que é $O(|V| * |E|)$. A presença de mais arestas contribui para um aumento significativo no tempo de execução.

O algoritmo de Floyd-Warshall, cuja complexidade é $O(|V|^3)$, está mais vinculado à quantidade de vértices. Em nossa análise, não houve necessidade de testar ciclos negativos, mas o desempenho geral foi inferior devido à natureza cúbica do algoritmo.

Por fim, o Algoritmo de Johnson apresentou os piores resultados. Ele executa Bellman-Ford com um vértice adicional conectado a todos os outros com peso 0 e, em seguida, utiliza o Dijkstra. No entanto, o desempenho é prejudicado pela implementação menos otimizada, especialmente no que diz respeito à inserção e remoção de arestas, sua complexidade se é dada por $O(|V|^2 * log|V| + |V| * |E|)$.

7. Referências

BONDY, J. A. e MURTY, U.S.R. (1979). Graph Theory with Applications. 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'. edição. 871 páginas. ISBN: 81-7808-830-4. Disponível https://athena.nitc.ac.in/summerschool/Files/West.pdf

DIESTEL, R. (2017). Graph Theory'. edição. ISBN: 978-3-662-53621-6. Disponível https://www.emis.de/monographs/Diestel/en/GraphTheoryII.pdf