Comparação de Algoritmos de Caminho Mínimo

Felipe Augusto Morais Silva

[1]ICEI – Instituto de Ciências Exatas e Informática (PUC-MG)

1. Introdução

No âmbito da teoria dos grafos e otimização, o estudo dos algoritmos de caminho mínimo é de suma importância, sendo essencial para uma ampla gama de aplicações práticas. A resolução eficiente desse problema complexo é crucial em diversas áreas, como redes de computadores, logística, transporte e design de circuitos, destacando a relevância destes algoritmos no contexto da pesquisa em ciência da computação.

Este artigo visa fornecer uma análise abrangente de quatro proeminentes algoritmos de caminho mínimo e um algoritmo de segmentação de imagens, explorando suas características distintas, aplicabilidades e desempenho relativo. Os algoritmos discutidos incluem o clássico algoritmo de Dijkstra, conhecido por sua eficiência em grafos com pesos não negativos; o versátil algoritmo de Bellman-Ford, capaz de lidar com arestas com pesos negativos; a abordagem dinâmica do algoritmo de Floyd-Warshall, adequada para grafos densos; a inovadora Floresta de Caminho Ótimo, que destaca-se pela sua adaptabilidade a diferentes cenários; e o Algoritmo de Johnson, que combina técnicas diversas para superar limitações de outros métodos.

2. Referencial Teórico

2.1. Grafos e Definições Básicas

Um grafo, representado por $G = (V, E)$, é uma estrutura matemática composta por um conjunto de vértices (V) e um conjunto de arestas (E), onde cada aresta conecta dois vértices. Dependendo das características das arestas, o grafo pode ser classificado como ponderado ou não ponderado. Em grafos ponderados, as arestas têm atributos numéricos associados, como pesos.

2.2. Caminho Mínimo

O problema do caminho mínimo é uma questão fundamental em teoria dos grafos, envolvendo a busca pelo caminho de menor custo entre dois vértices em um grafo ponderado. O custo de um caminho é geralmente determinado pela soma dos pesos das arestas ao longo do caminho. Vários algoritmos foram desenvolvidos para resolver esse problema, cada um com suas características distintas.

2.3. Algoritmo de Dijkstra

O algoritmo de Dijkstra, proposto por Edsger W. Dijkstra em 1956, é um método eficiente para encontrar o caminho mais curto entre um vértice de origem e todos os outros vértices em um grafo com arestas de peso não negativo. A abordagem baseia-se na seleção iterativa dos vértices mais próximos à origem, atualizando os caminhos mínimos conhecidos até o momento.

Algorithm 1 Algoritmo de Dijkstra

```
1: procedure DIJKSTRA(G, s)                          ▷ G: Grafo, s: Vértice de origem
2:     Inicializar conjuntos de distâncias dist e predecessores prev
3:     dist[s] ← 0
4:     Q ← PRIORITYQUEUE()                                        ▷ Fila de prioridade
5:     ENQUEUE(Q, (s, 0))
6:     while Q não está vazia do
7:         (u, d) ← DEQUEUE(Q)
8:         for cada vértice v adjacente a u do
9:             alt ← dist[u] + PESO(u, v)
10:            if alt < dist[v] then
11:                dist[v] ← alt
12:                prev[v] ← u
13:                ENQUEUE(Q, (v, alt))
14:            end if
15:        end for
16:    end while
17:    return dist, prev
18: end procedure
```

2.4. Algoritmo de Bellman-Ford

Desenvolvido por Richard Bellman e Lester Ford em 1958, o algoritmo de Bellman-Ford é projetado para lidar com grafos que podem conter arestas de peso negativo, embora não permita ciclos negativos alcançáveis a partir do vértice de origem. O método utiliza uma abordagem de relaxamento iterativo para determinar os caminhos mínimos.

Algorithm 2 Algoritmo de Bellman-Ford

```
1: procedure BELLMANFORD(G, s)                       ▷ G: Grafo, s: Vértice de origem
2:     Inicializar conjunto de distâncias dist e predecessores prev
3:     dist[s] ← 0
4:     for i ← 1 até |V| − 1 do                    ▷ |V| é o número de vértices em G
5:         for cada aresta (u, v) em G do
6:             alt ← dist[u] + PESO(u, v)
7:             if alt < dist[v] then
8:                 dist[v] ← alt
9:                 prev[v] ← u
10:            end if
11:        end for
12:    end for
13:    return dist, prev
14: end procedure
```

2.5. Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é uma técnica de programação dinâmica para encontrar os caminhos mínimos entre todos os pares de vértices em um grafo direcionado ponderado.

Proposto por Robert Floyd e Stephen Warshall em 1962, o método é eficiente apenas para grafos pequenos devido à sua complexidade temporal cubica.

Algorithm 3 Algoritmo de Floyd-Warshall

```
1:  procedure FLOYDWARSHALL(G)                          ▷ G: Grafo ponderado
2:      Inicializar matriz de distâncias dist com infinito
3:      for cada vértice v em G do
4:          dist[v][v] ← 0
5:      end for
6:      for cada aresta (u, v) em G do
7:          dist[u][v] ← PESO(u, v)
8:      end for
9:      for cada vértice k em G do
10:         for cada vértice i em G do
11:             for cada vértice j em G do
12:                 if dist[i][j] > dist[i][k] + dist[k][j] then
13:                     dist[i][j] ← dist[i][k] + dist[k][j]
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return dist
19: end procedure
```

2.6. Optimum Path Forest

O OPF (Optimum Path Forest) é um algoritmo de aprendizado de máquina supervisionado usado principalmente para classificação em problemas de reconhecimento de padrões. Ele é baseado em árvores de caminho ótimo, onde cada nó representa uma amostra do conjunto de dados e as arestas representam a similaridade entre as amostras.

Algorithm 4 OPF

```
procedure DIJKSTRA(G, s)                          ▷ G: Grafo, s: Vértice de origem
    Inicializar conjuntos de distâncias dist e predecessores prev
    dist[s] ← 0
    Q ← PRIORITYQUEUE()                                   ▷ Fila de prioridade
    ENQUEUE(Q, (s, 0))
    while Q não está vazia do
        x ← DEQUEUE(Q)
        for cada vértice v adjacente a u do
            alt ← dist[v] + PESOx
            if alt < dist[u] then
                dist[v] ← alt
                prev[v] ← u
                ENQUEUE(Q, (v, alt))
            end if
        end for
    end while
    return dist, prev
end procedure
```

2.7. Algoritmo de Johnson

O Algoritmo de Johnson, proposto por Donald B. Johnson em 1977, combina técnicas de remoção de ciclos negativos e reatribuição de pesos para lidar com grafos que contenham arestas de peso negativo. Esta abordagem inovadora permite encontrar caminhos mínimos em grafos que podem incluir arestas com valores negativos, contornando as limitações de outros métodos.

Algorithm 5 Algoritmo de Johnson

```
procedure JOHNSON(G)                              ▷ G: Grafo ponderado
    Adicionar um novo vértice s ao grafo G
    for cada vértice v em G do
        Adicionar uma aresta ponderada de s para v com peso 0
    end for
    h ← BELLMANFORD(G, s)        ▷ Aplicar Bellman-Ford para calcular potenciais
    for cada aresta (u, v) em G do
        Atualizar peso da aresta: PESO′(u, v) ← PESO(u, v) + h[u] − h[v]
    end for
    Inicializar matriz de distâncias dist usando Dijkstra para cada vértice como fonte
    for cada vértice u em G do
        Aplicar Dijkstra com fonte em u para obter distâncias d_u
        for cada vértice v em G do
            dist[u][v] ← d_u[v] + h[v] − h[u]
        end for
    end for
    return dist
end procedure
```

3. Metodologia

3.1. Implementação em C++

Cada algoritmo foi implementado em C++ para garantir uniformidade na análise de desempenho. A escolha da linguagem C++ foi motivada pela eficiência na execução de algoritmos e pela capacidade de controle próximo do hardware.

3.2. Compilação com GCC

Utilizou-se o compilador GCC como padrão para compilar os algoritmos. A decisão de não incluir flags extras de otimização foi tomada para avaliar o desempenho bruto dos algoritmos sem intervenções específicas do compilador.

3.3. Coleta de Dados

Os algoritmos foram executados no intervalo proposto, e a partir disso, a principal informação coletada foi o tempo de execução de cada algoritmo.

3.4. Organização dos Resultados em Jupyter Notebook

Utilizou-se Jupyter Notebook como ambiente de desenvolvimento interativo para a análise e visualização dos resultados. A flexibilidade do Jupyter Notebook permitiu uma exploração interativa e eficaz dos dados.

3.5. Visualização com Matplotlib

A biblioteca Matplotlib do Python foi empregada para criar visualizações gráficas dos resultados. Gráficos, como gráficos de barras e gráficos de linhas, foram escolhidos de acordo com a natureza dos dados e os objetivos de comunicação visual.

3.6. Avaliação de Consistência

Foram realizadas execuções repetidas dos algoritmos para verificar a consistência dos resultados.

4. Resultados

Ao longo da seção de ”Resultados”, apresentaremos experimentos e comparações práticas, utilizando o tempo de execução de cada algoritmo para avaliar o desempenho. Essas análises práticas visam iniciar uma discussão sobre o comportamento de cada algoritmo e a escolha adequada de algoritmos em situações específicas, considerando as características do grafo e outros fatores relevantes.

4.1. Objetivos

O principal objetivo deste artigo é avaliar o desempenho relativo desses algoritmos em diferentes contextos, identificando suas vantagens e limitações em cenários específicos. Além disso, discutiremos possíveis extensões e modificações dos algoritmos apresentados, visando melhorar seu desempenho ou adaptá-los a requisitos específicos.

4.2. Organização dos resultados

Os resultados obtidos foram organizados em graficos, utilizando Jupyter Notebook e a Matplotlib do python. Todos os algoritmos foram implementados em C++, utilizando GCC como o compilador padrão, sem nenhuma flag extra de otimização.

Foram gerados:

- 1 gráfico para cada algoritmo - comparando o número de arestas com o runtime (em segundos);
- 2 gráficos comparando todos os algorítmos - comparando o runtime de cada um deles;
- 1 gráfico comparando o Dijkstra com o OPF;
- 1 gráfico comparando o Bellman-ford, Johnson, e Floyd-Warshall;

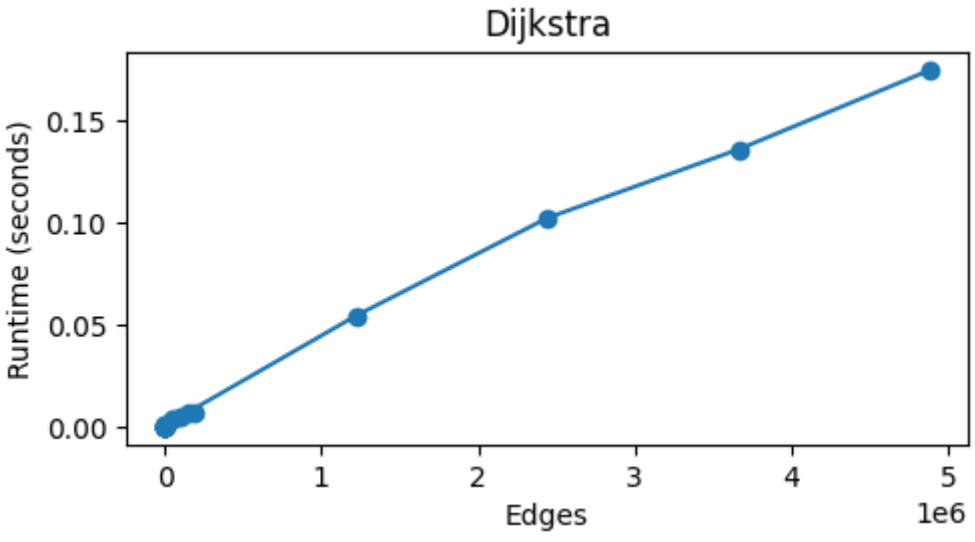
4.3. Resultados Brutos

Ao final do experimento, foram obtidos os seguintes resultados:

Algorithm	Média (segundos)	Desvio Padrão
Bellman-Ford	74.40404270800002	187.96468173090474
Dijkstra	0.01969078	0.04586702881413619
Johnson	76.61738192	151.64151466956298
OPF	0.004662648	0.010115243579118399
Floyd-Warshall	74.24657166	146.91219580449305

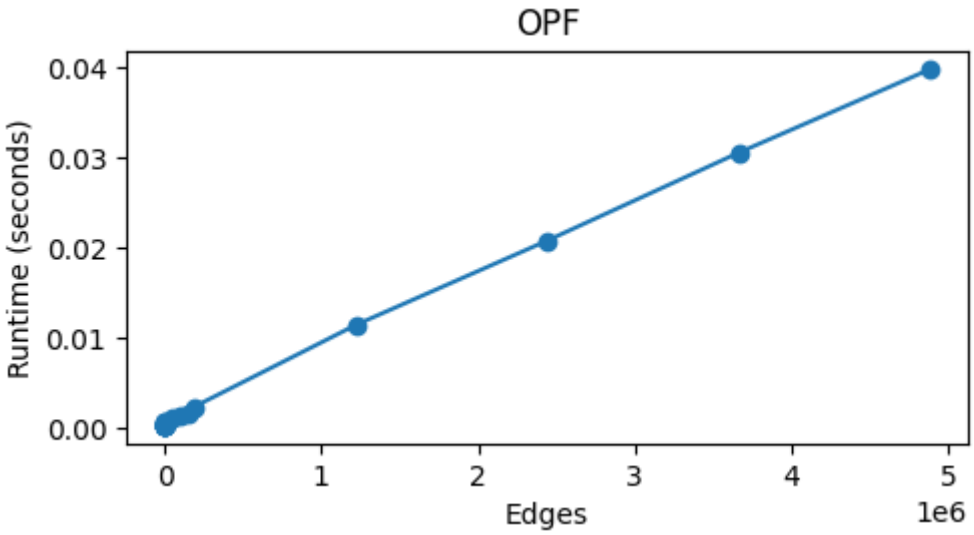
Table 1. Algorithm Performance Metrics

E a partir dos mesmos dados, foram gerados os seguintes gráficos:



a

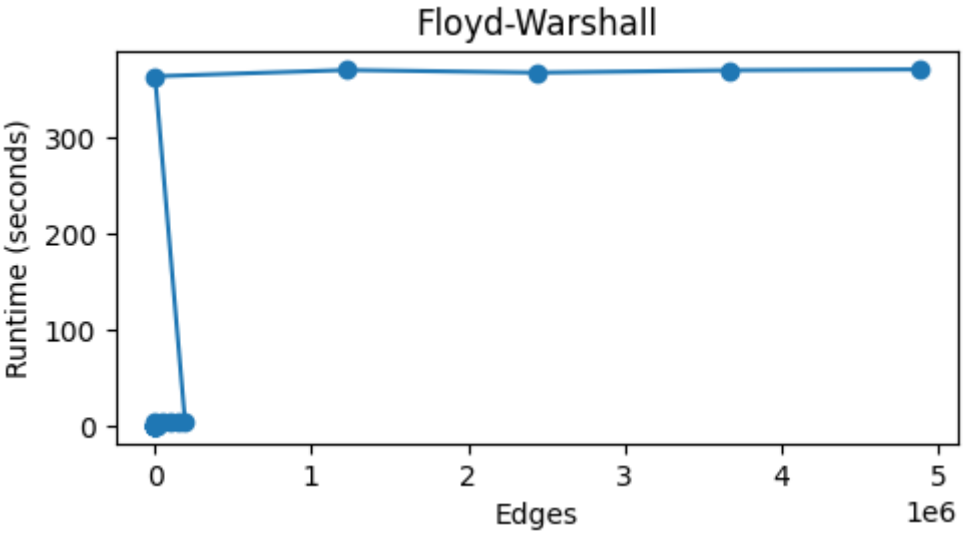
Figure 1. Dijkstra - Número de Arestas x Runtime



b

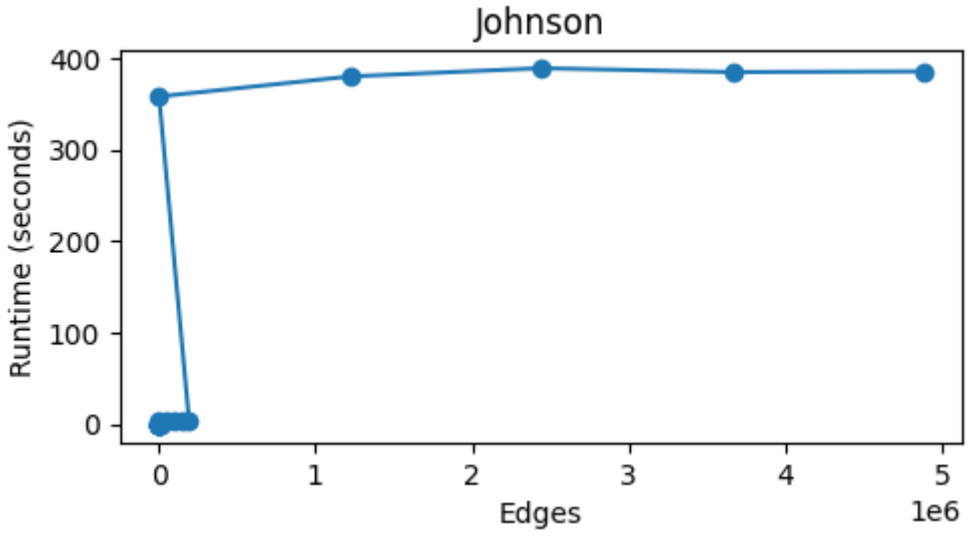
Figure 2. Opf - Número de Arestas x Runtime

Percebe-se, que os graficos obtidos pelo dijkstra e OPF são razoavelmente semelhantes, além de serem, dentre os algoritmos analisados, os com menor tempo de execução.



c

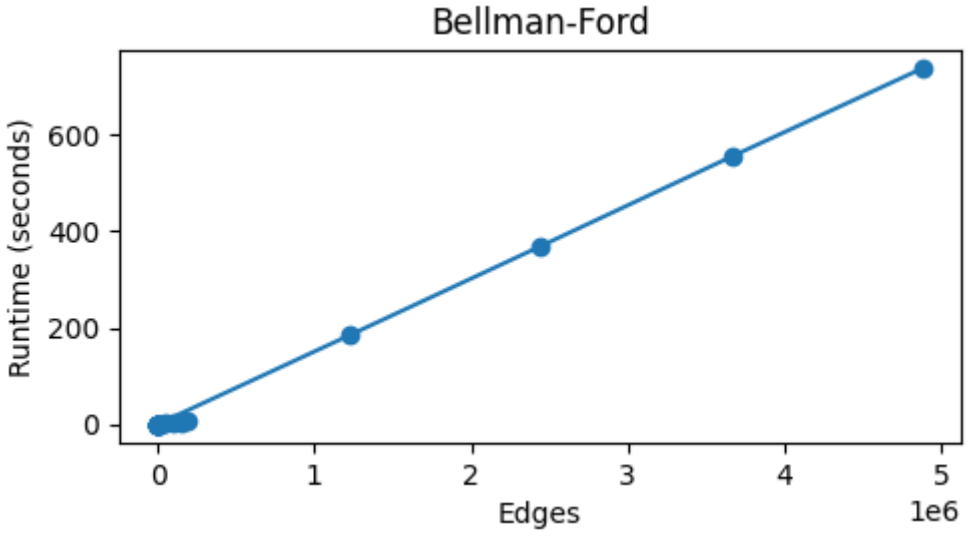
Figure 3. Floyd-Warshall - Número de Arestas x Runtime



d

Figure 4. Johnson - Número de Arestas x Runtime

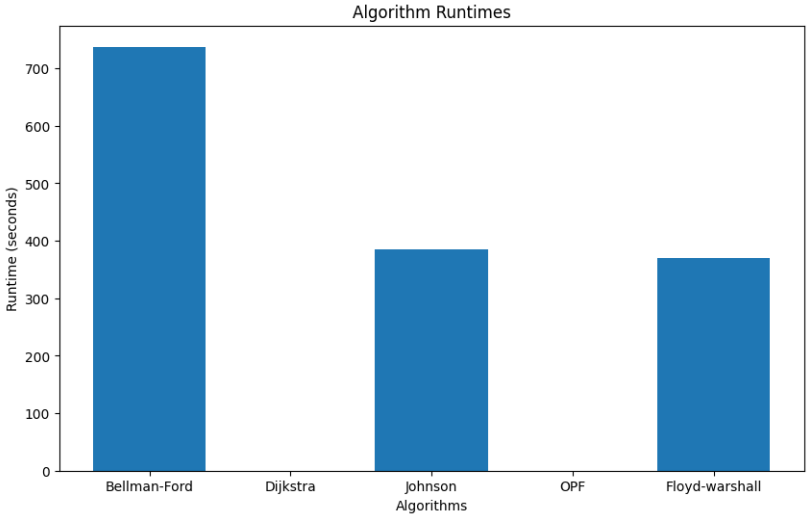
Percebe-se que tanto o johnson quanto o floyd-warshall se estabilizaram, e apesar do aumento de arestas, o tempo de execução se manteve em ambos os algoritmos.



e

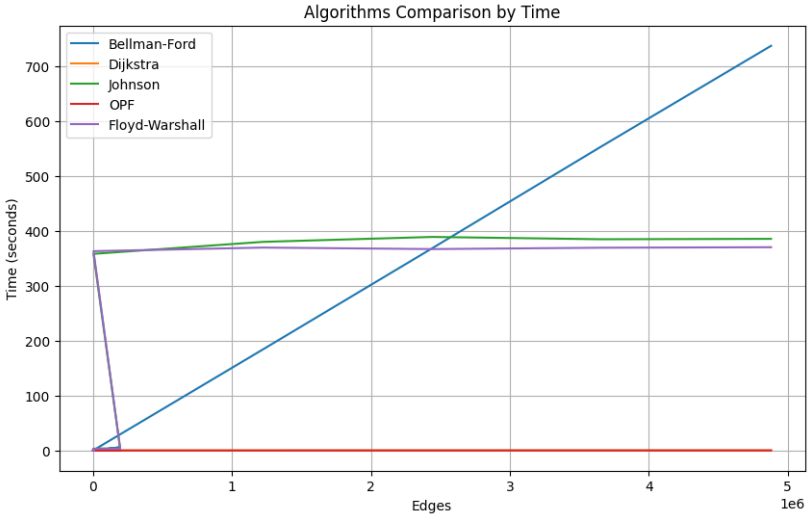
Figure 5. Bellman-Ford - Número de Arestas x Runtime

Observa-se que o tempo de execução do bellman-ford cresceu junto com o número de arestas. Além disso, o bellman-ford foi algoritmo o com maior tempo de execução.



f

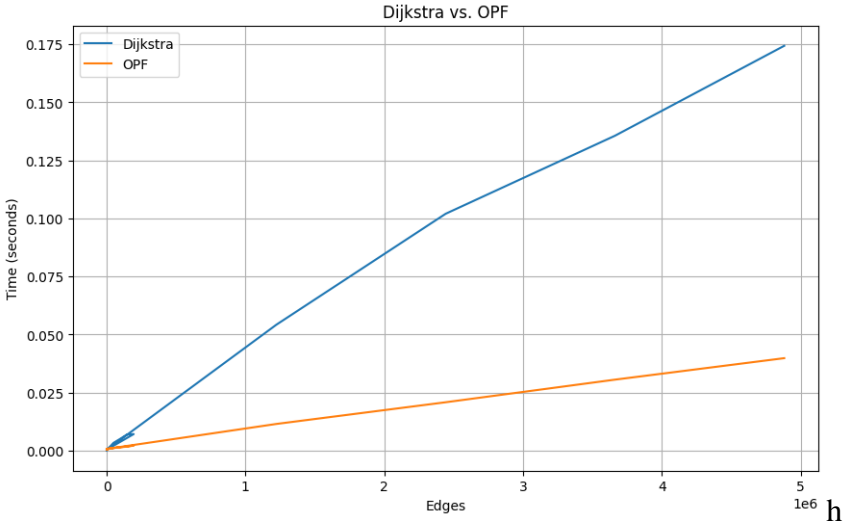
Figure 6. All algorithms - runtime (bars)



g

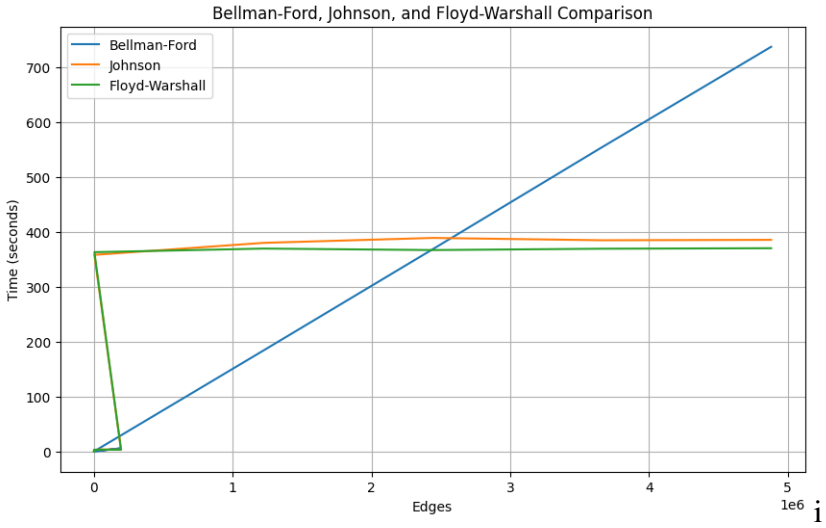
Figure 7. All algorithms - runtime (lines)

Ao comparar todos os algoritmos em um unico gráfico, é possível perceber com mais clareza que a diferença dos tempos de execução do Dijkstra e OPF para os outros algoritmos.



h

Figure 8. Dijkstra x OPF - runtime



i

Figure 9. Bellman-Ford x Floyd-Warshall x Johnson - runtime (lines)

Ao separar os gráficos em grupos, percebe-se como Bellman-Ford, Floyd-Warshall e Johnson estão em uma ordem de grandeza completamente diferente do Dijkstra e OPF

5. Discussão dos resultados

Na análise dos resultados obtidos, fica evidente que os cinco algoritmos - Dijkstra, OPF (*Optimal Path Forest*), Algoritmo de Johnson, Bellman-Ford e Floyd-Warshall - apresentam diferentes desempenhos em relação ao tempo de execução, conforme ilustrado nos gráficos da seção anterior.

5.1. Dijkstra e OPF

Os gráficos revelam uma notável semelhança nos resultados entre os algoritmos Dijkstra e OPF. Além disso, destaca-se que ambos são os mais eficientes em termos de tempo

de execução quando comparados aos demais algoritmos analisados. Esta similaridade não é surpreendente, dado que o OPF é uma variação do Dijkstra. Creio que a pequena diferença de desempenho observada pode ser atribuída à otimização do OPF, que, ao ajustar a condição de verificação, evita a entrada em determinadas condições, resultando em um tempo de execução algumas vezes mais rápido em comparação ao Dijkstra. Com essa abordagem, nem todos os vértices acabam sendo visitados, resultando em menos iterações do algoritmo. Isso contribui para um desempenho ligeiramente superior em comparação com o algoritmo de Dijkstra. A complexidade final de ambos os algoritmos foi de $O(v \log v)$, pois foi usada uma fila de prioridade para ordenação dos vértices.

5.2. Johnson e Floyd-Warshall

Os resultados para o Algoritmo de Johnson e Floyd-Warshall indicam uma estabilização do tempo de execução, mesmo com o aumento do número de arestas nos grafos. Essa estabilidade pode ser explicada pela natureza da implementação desses algoritmos, que dependem principalmente do número de vértices do grafo e não do número de arestas. Portanto, mesmo com variações nas condições do grafo, os tempos de execução permaneceram consistentes. A complexidade final do Floyd-Warshall foi $O(v^3)$, e a do algoritmo de johnson tambem foi de $O(v^3)$, mais especificamente $O(v^3 + v^2)$.

5.3. Bellman-Ford

Observa-se, por outro lado, que o tempo de execução do Bellman-Ford cresce proporcionalmente ao aumento do número de arestas nos grafos analisados. Isso se alinha com a expectativa, uma vez que a complexidade deste algoritmo depende do número de arestas. Notavelmente, o Bellman-Ford apresentou o maior tempo de execução entre os algoritmos considerados, o que destaca sua sensibilidade ao aumento da densidade de arestas nos grafos. A complexidade final do Bellman-Ford foi de $O(nm)$.

6. Conclusão

6.1. Dijkstra e OPF

No contexto do Dijkstra e OPF, a semelhança nos tempos de execução sugere que o OPF, como uma variação do Dijkstra, é capaz de preservar a eficiência fundamental deste último, enquanto em alguns casos demonstra vantagens decorrentes de otimizações específicas. Outro ponto a se dizer, é que apesar de servirem propósitos diferentes, é interessante perceber como algoritmos clássicos da literatura ajudam no desenvolvimento de novas soluções.

6.2. Johnson e Floyd-Warshall

Os Algoritmos de Johnson e Floyd-Warshall, por outro lado, exibiram estabilidade em seus tempos de execução, independentemente do aumento no número de arestas. Essa estabilidade, atribuída à dependência da complexidade em relação ao número de vértices, destaca a robustez desses algoritmos em cenários variáveis.

6.3. bellman-ford

O Bellman-Ford, embora tenha se mostrado sensível ao crescimento do número de arestas, desempenhou um papel crucial ao revelar uma relação direta entre sua complexidade e a densidade de arestas nos grafos analisados.

6.4. Considerações Finais

Em síntese, a escolha de um algoritmo específico deve ser pautada nas características particulares do problema em questão. Considerações como a preservação da eficiência, a estabilidade diante de variações e a sensibilidade a determinadas condições devem ser ponderadas.

7. Referências

1. Antti Laaksonen. (2023). *Competitive Programmer's Handbook*. Recuperado de: https://cses.fi/book/book.pdf
2. *Competitive Programming Algorithms*. Recuperado de: https://cp-algorithms.com/
3. Alexandre Falcão. (Ano não especificado). *Slides da disciplina MO443 - Processamento de Imagens*. Universidade Estadual de Campinas. Recuperado de: https://ic.unicamp.br/~afalcao/mo443/slides-aula30.pdf
4. Reinhard Diestel. (Ano não especificado). *Graph Theory*. Livro disponível em formato físico.