Universidade Federal Fluminense

Instituto de Computação
Coordenação de Pós-Graduação em Computação

Disciplina: Analise e Síntese de Algoritmos

Professor: Dr. Luís Antônio Brasil Kowada

Aluno: Flávio Miranda de Farias

CAMINHO MAIS CURTO (SHORTEST PATH)

Este trabalho visa abordar o problema do Caminho Mais Curto (*shortest path*). A problemática vem sendo atacada a muitas décadas e tem muitas aplicações práticas, como em rodas de mapas e principalmente em roteadores e controladores de redes de computadores. Este tema é bem trabalhado nos livros do (ANDREW S. TANENBAUM E DAVID WETHERALL, 2011; CORMEN, 2012; KUROSE; ROSS, 2010).

A família de algoritmos que abordam esse assunto é dividida segundo duas técnicas, sendo, Métodos Gulosos e Programação Dinâmica (metodologias *Bottom/Up* e *Top/Down*) (encontrado nos capítulos 15 e 16 do livro do Comen, 2012), estas metodologia trabalham com maximização ou minimização e o objetivo deste trabalho busca apresentar, dentre de algoritmos clássicos, os custos de minimização através do menor caminho para, por exemplo, um melhor fluxo de rede.

Para analise serão apresentados brevemente o Método Guloso de Dijkstra e os Métodos de Programação Dinâmica Bellman-Ford e Floyd-Warshall, assim como uma implementação dos mesmos.

1. MOTIVAÇÃO

Esta pesquisa foi estimulada em sala de aula na disciplina de Analise e Síntese de Algoritmos da Pós-graduação em Computação da UFF em 2019-2 pelo professor Dr. Luís Antônio Brasil Kowada. Que instigou aos alunos praticarem uma análise de algoritmos para problema de maximização ou minimização, nas quais utilizassem metodologias *Bottom/Up* e *Top/Down*. A seguir a atividade proposta na integra.

> **"Problema de Otimização**
>
> Escolha de um problema de otimização (maximização ou minimização).
>
> Implemente uma abordagem top-down e outra bottom-up.
>
> A abordagem top-down deve ser baseada em decisões gulosas inteligentes, ou seja, deve buscar uma melhor decisão local baseada nas diversas possibilidades de dividir o problema.
>
> A abordagem bottom-up deve ir dividindo o problema em sub-problemas pequenos e ir montando as soluções baseadas nas soluções dos problemas menores, como ocorre com programação dinâmica.
>
> Se o problema for polinomial, a abordagem bottom-up deve retornar a solução exata, caso contrário pode retornar uma solução aproximada ou pseudo-polinomial.
>
> No relatório, deve-se contextualizar o problema, indicar corretude e complexidade dos algoritmos apresentados e comparação dos resultados para alguns exemplos.
>
> Junto com o relatório, deve-se enviar os arquivos do programa, num arquivo único compactado."

2. ALGORITMOS

A seguir, serão apresentadas as três abordagens clássicas escolhidas para o problema, sendo que a codificação foi baseada e adaptada dos códigos e teoria de (MARTIN BROADHURST, 2019) e a teoria em (ANDREW S. TANENBAUM E DAVID WETHERALL, 2011; CORMEN, 2012; DE SOUZA et al., 2012; GAGNON, 2011; KUROSE; ROSS, 2010; VASCONCELOS; GRAMACHO, 2019).

2.1. Dijkstra

Algoritmo muito utilizado em roteamento de estado de enlace que leva o nome de seu criador em 1959, calcula o caminho de menor custo entre um nó de redes (aresta) e todos os outros nós da rede. É um algoritmo iterativo, usualmente usado em roteadores de redes menores e tem a propriedade de, através de uma abordagem *top-down*, após a k-ésima iteração, conhecer os caminhos de menor custo para k nós de destino e, dentre os caminhos de menor custo até todos os nós de destino, esses k caminhos terão os k menores custos (KUROSE; ROSS, 2010), este algoritmo tem uma desvantagem em relação a outros, que é o fato de não trabalhar com valores negativos no peso das vértices.

Agora suponhamos que um grafo é representado por uma matriz de adjacência onde temos o valor se não existe aresta entre dois vértices. Suponhamos também que os vértices do grafo são enumerados de 1 até n, isto é, o conjunto de vértices é N = {1, 2, ..., n}. Utilizaremos também um vetor D[2..n] que conterá a distância que separa todo vértice do vértice 1 (o vértice do grafo que é o vértice 1 é escolhido arbitrariamente). Eis o algoritmo:

função Dijkstra (L = [1..n, 1..n]: grafo): vetor[2..n]

    C ← {2,3,...,n} {Implicitamente S = N - C}

    Para i ← 2 **até** n:

        D[i] ← L[1,i]

    Repetir n-2 vezes:

        v ← Elemento de C que minimiza D[v]

        C ← C - {v}

        Para cada elemento w de C:

            D[w] ← min(D[w],D[v]+ L[v,w])

    Retornar D

2.1.1. Análise do algoritmo Dijkstra e comparação com dinâmico

A inicialização do algoritmo exige um tempo em O(n). O loop é executado n-2 vezes e a cada iteração todos os vértices não atualmente no conjunto S são visitados. Na primeira iteração, visitaremos n-1 vértices, na segunda, n-2 vértices, e assim por diante. Podemos então deduzir que o tempo de execução é em $O(n^2)$.

Da para implementar uma versão em Programação Dinâmica que obtenha o caminho mais curto entre qualquer par de vértices. Lembramos que a programação dinâmica consiste essencialmente em produzir soluções intermediárias que serão utilizadas incrementalmente para obter a solução final. Seja um grafo direcionado representado por uma matriz de adjacência L[1..n,1..n]. Calculamos a cada etapa do algoritmo uma matriz D onde cada elemento D[i,j] representa o valor do caminho mais curto de k arestas entre i e j. Na etapa seguinte os valores das matrizes D e L

são utilizadas para calcular a nova matriz D que contém os valores para k+1 arestas. Para obter o novo valor D[i,j], consideramos o valor D[i,u] + L[u,j] para todo vértice u e escolhemos o menor valor. Isto é, consideramos os caminhos de k vértices a partir de i, acrescentamos, para cada um desses caminhos, a arestas que falta para alcançar j e selecionamos o mais curto (GAGNON, 2011), a seguir o pseudocódigo:

```
função CamMaisCurtos(L[1..n, 1..n]: grafo): matriz [1..n,1..n]
    D ← L
    Repetir n-2 vezes:
        Para i ← 1 até n:
            Para j ← 1 até n:
                min ←
                Para k ← 1 até n:
                    Se D[i,k]+L[k,j] < min então min ← D[i,k]+L[k,j]
                D'[i,j]← min
        D ← D'
    Retornar D
```

É possível observar no pseudocódigo que o tempo de execução desse algoritmo está em $O(n^4)$, em comparação ao Dijkstra que é **$O(n^2)$**, é mais custoso. Porém dependendo da necessidade de uso da aplicação e tamanho do grafo (ou rede por exemplo), pode ser mais interessante um código que apesar de mais custoso, entregue um melhor resultado. Com esta finalidade serão apresentados outros algoritmos que são aplicados comercialmente em alguns cenários de roteamento de grandes redes.

2.1.2. Corretude de Dijkstra

Como qualquer avaliação de corretude, não é uma tarefa fácil, porém, um caminho para obtermos neste caso é provar que quando o algoritmo para, temos que d[v] = dist(s, v) para todo v ∈ V[G] e π [ ] define uma Arvore de Caminhos Mínimos. Mais precisamente, o conjunto {(π [x], x) : x pertence a V[G] - {s}} forma uma Arvore de Caminhos Mínimos (DE SOUZA et al., 2012).

Outra forma de obtermos, é através de indução, A demonstração de que o algoritmo de Dijkstra está correto será feita por indução em F, considerando que, se o vértice i está em F, então dt[i] é o caminho mais curto da origem até o vértice i.
Como base da indução, a demonstração de que o algoritmo de Dijkstra está correto será feita por indução em F, considerando que, se o vértice i está em F, então dt[i] é o caminho mais curto da origem até o vértice i e a hipótese é que vale para todos os vértices de F até imediatamente antes da inserção de um vértice i.

Constatação da indução:

Se o vértice i foi escolhido pelo algoritmo, então dt[i] é o menor dentre todos os vértices em A. Deve-se mostrar que dt[i] é o comprimento do caminho mais curto entre a origem e i. Supomos o contrário, ou seja, que existe pelo menos um vértice x no menor caminho entre a origem e i, que não pertence ao caminho atual, de comprimento dt[i], tal que dt[x]< dt[i] e x ∈ A. Neste caso, o algoritmo deveria ter escolhido x ao invés de i. Mas escolheu i, significando que este nó x não existe. Portanto, quando i é adicionado a F, o caminho mais curto entre a origem e i foi encontrado.

2.2. Floyd-Warshall

O algoritmo de Floyd-Warshall calcula as distâncias entre todos os pares de vértices em um gráfico ponderado. É um algoritmo de programação dinâmica que utiliza a abordagem *botton-up*, além de poder trabalhar com pesos negativos.

De forma geral no início o algoritmo possui variações de implementações neste caso, efetua n^2 iterações. Na iteração k do algoritmo obteremos uma matriz D cujos elementos na posição D[i,j] representam o caminho mais curto de i até j, entre todos que passam somente por vértices do conjunto C = {1,2,...,k}, isto é, um caminho que pode conter somente os vértices i e j e qualquer outro do conjunto C = {1,2,...,k}. Quando acrescentamos o vértice k+1, temos que atualizar todas as distâncias para ver se agora não temos um caminho mais curto passando pelo vértice k. O valor na posição D[i,j] muda somente se o valor D[i,k] + D[k,j] é menor. Nesse caso colocamos em D[i,j] o valor D[i,k] + D[k,j]. Quando o algoritmo pára, isto é, quando k = n, temos na matriz os valores dos caminhos mais curtos entre qualquer par de vértices.

```
função Floyd(L[1..n, 1..n]: grafo): matriz [1..n,1..n]
    D ← L
    Para i ← 1 até n:
        Para j ← 1 até n:
            P[i,j] ←  0
    Para k ← 1 até n:
        Para i ← 1 até n:
            Para j ← 1 até n:
                Se D[i,k]+D[k,j] < D[i,j]
                    D[i,j] ← D[i,k]+D[k,j]
                    P[i,j] ← k
    Retornar D
```

2.2.1. Análise do algoritmo Floyd-Warshall dinâmico em comparação ao Dijkstra guloso

É muito fácil ver que o algoritmo tem um tempo de execução em $\mathbf{O(n^3)}$. Note que o mesmo problema pode ser resolvido aplicando n vezes o algoritmo de Dijkstra, começando cada vez com um vértice diferente. Nesse caso, teríamos também um tempo de execução em $O(n^3)$. Mas isso não significa que ambos têm a mesma eficiência. O algoritmo de Floyd tem que inicializar uma matriz de tamanho n x n. O algoritmo de Dijktra tem, dentro do primeiro loop, dois loops onde cada uma exige um tempo de execução na ordem de n. Também ele deve efetuar uma inicialização que exige um tempo em O(n) (isso repetido n vezes vai dar um total em $O(n^2)$). Lembramos que usando uma estrutura de heap, podemos obter uma implementação do algoritmo de Dijkstra que está em O((a+n) log n), onde a é o número de arestas. Aplicando n vezes o algoritmo, obtemos um tempo de execução de n x O((a+n) log n) = O((an + n^2) log n). Se o grafo é esparso, pode ser melhor utilizar o algoritmo de Dijkstra. Se o número de arestas se aproxima do grafo completo, provavelmente o algoritmo de Floyd será melhor.

2.2.2.Corretude de Floyd-Warshall

Utilizando indução para provar a corretude. Consideramos um grafo G, com número de vértices V numerados de 1 a N. Consideramos o menor caminho p dos caminhos entre dois vértices i e j que usam os vértices intermediários $\{1, 2, ..., k\}$, dado um k qualquer. A questão aqui é saber se $k \in p$, ou seja, se k é um vértice intermediário de p ou não. Seja $menorCaminho(i, j, k)$ uma função que retorna o menor caminho entre os vértices i e j, usando os vértices intermediários de 1 até k, temos que $menorCaminho(i, j, k)$ (VASCONCELOS; GRAMACHO, 2019):

- um caminho que não passa por k se $k \notin p$. Então todos os vértices intermediários de p estão no subconjunto $\{1, 2, ..., k - 1\}$
- um caminho que passa por k se $k \in p$. Logo o caminho pode ser divido em ir de i até k e de k até j (ambos caminhos usando os vértices intermediários do subconjunto $\{1, 2, ..., k - 1\}$)

A partir da nossa hipótese de indução de que o menor caminho de i até j que só usa vértices de 1 até $k - 1$ é definido por $menorCaminho(i, j, k - 1)$ e que se existe um caminho menor usando o vértice k esse caminho será certamente a concatenação dos caminhos de i até k e de k até j.

Sendo $w(i, j)$ o peso entre dá aresta entre i e j, temos que o caso base é: $menorCaminho(i, j, 0) = w(i, j)$. Com isso, definimos:

$mC(i, j, k) = min(mC(i, j, k - 1), mC(i, k, k - 1) + mC(k, j, k - 1))$.

2.3. Bellman-Ford

Os primeiros protocolos de roteamento intradomínio usavam um algoritmo por vetor de distância, baseado no algoritmo de Bellman-Ford distribuído, herdado da ARPANET, algoritmo de Bellman-Ford é outro que resolve o problema utilizando uma abordagem *bottom-up* através de programação dinâmica assim como o Floyd, ele encontra o caminho mais curto entre um único vértice e todos os outros vértices em um gráfico. Esse é o mesmo problema que o algoritmo de Dijkstra resolve, mas, diferentemente de Dijkstra, o algoritmo Bellman-Ford pode lidar com gráficos com pesos de borda negativos.

Uma consequência dos pesos negativos é que um gráfico pode conter um ciclo negativo e, se esse for o caso, o caminho mais curto para todos os vértices alcançáveis a partir do ciclo é o infinito negativo, porque é possível atravessar o ciclo várias vezes e continuar reduzindo o comprimento do caminho indefinidamente. O algoritmo Bellman-Ford detecta isso executando uma iteração extra sobre o gráfico e, se isso resultar em uma distância diminuindo, ele sabe que há um ciclo negativo.

função BellmanFord(G, s)

    para v ∈ vertices(G)

        se v = s v.distance ← 0

        senão

            v.distance ← ∞

            v.parent ← null

    para t ← 1 até n faça

        flag_stop ← 1

```
        para v ∈ vertices(G)
            para e ∈ incomingEdges(G, v)
                z ← opposite(G, v, e)
                se z.distance + e.weight < v.distance
                    v.distance ← z.distance + e.weight
                    v.parent ← z
                    flag_stop ← 0
        se flag_stop = 1 retorne G
    retorne nulo
```

2.3.1. Análise do algoritmo Bellman-Ford e comparativo com Dijkstra

Inicialmente com a atribuição de valores iniciais aos **n** vértices temos O(n), após temos a cada interação, cada uma das **m** arestas direcionadas é verificada uma vez (no vértice de entrada) para atualização de valores ao custo de O(m), portanto como são n interações temos O(nm+n), ou assintoticamente o custo é de **O(nm)**.

Assim como o algoritmo de Dijkstra, o algoritmo de Bellman-Ford utiliza a técnica de relaxamento, ou seja, realiza sucessivas aproximações das distâncias até finalmente chegar na solução. A principal diferença entre Dijkstra e Bellman-Ford é que no algoritmo de Dijkstra é utilizada uma fila de prioridades para selecionar os vértices a serem relaxados, enquanto o algoritmo de Bellman-Ford simplesmente relaxa todas as arestas.

2.3.2. Corretude de Bellman-Ford

Para obtermos a corretude temos de considerar que, Se (G, w) não contem ciclos negativos atingíveis por s, então no final o algoritmo devolve TRUE, d[v] = dist(s, v) para v ∈ V e π[ ] define uma Arvore de Caminhos Mínimos. Se (G, w) contem ciclos negativos atingíveis por s, então no final o algoritmo devolve FALSO (DE SOUZA et al., 2012).

Para provamos que devolve FALSO, vamos supor que o grafo não possui ciclos negativos atingíveis por s. Seja P = (v_0 = s, v_1,..., v_k) um caminho mínimo de v_1 a v_k e suponha que as arestas (v_0, v_1), (v_1, v_2),. . . ,(v_{k-1}, v_k) são relaxadas nesta ordem. Então d[v_k] = dist(s, v_k).

Seja v um vértice atingível por s e seja P = (v_0 = s, v_1,..., v_k = v) um caminho mínimo de s a v. Note que P tem no máximo |V| 1 arestas. Cada uma das |V| - 1 iterações do primeiro laço relaxa todas as |E| arestas. Na iteração i a aresta (v_{i-1}, v_i) e relaxada. Logo temo, d[v] = d[v_k] = dist(s, v).

Se v e um vértice não atingível por s pode-se mostrar que d[v] = ∞. Assim, no final do algoritmo d[v] = dist(s, v) para v € V. Pode-se verificar que π [ ] define uma Arvore de Caminhos Mínimos.

Agora falta mostrar que devolve VERDADE. Seja (u, v) uma aresta de G. Então:

d[v] = dist(s, v)

d[v] <= dist(s, u) + w(u, v)

d[v] = d[u] + w(u, v),

e assim nenhum dos testes faz com que o algoritmo devolva FALSO. Logo, ele devolve VERDADE.

Agora suponha que (G, w) contém um ciclo negativo atingível por s. Seja C = (v_0, v_1,..., v_k = v_0) um tal ciclo. Então $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$. Suponha por contradição que o algoritmo devolve VERDADE. Então d[v_i] <= d[v_{i-1}] + w(v_{i-1}, v_i) para i = 1, 2 ..., k.

Somando as desigualdades ao longo do ciclo temos:

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} \left(d[v_{i-1}] + w(v_{i-1}, v_i)\right)$$

$$= \sum_{i=1}^{k} (d[v_{i-1}]) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Como v_0 = v_k, temos que $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$. Mas então $\sum_{i=1}^{k} w[v_{i-1}, v_i] \geq 0$ o que contraria o fato de o ciclo ser negativo.

2.4. Comparativo

A seguir, o comparativo entre os algoritmos apresentados na Tabela 1:

Tabela 1- Comparativo entre as abordagens.

Algoritmo	Custo de tempo	Abordagem
Dijkstra	O(n^2)	Gulosa (*top-down*)
Dijkstra (modificado)	O(n^3)	P. Dinâmica (*Bottom-up*)
Floyd-Warshall	O(n^3)	P. Dinâmica (*Bottom-up*)
Bellman-Ford	O(nm)	P. Dinâmica (*Bottom-up*)

3. IMPLEMENTAÇÃO

Para implementação foi decidido usar a linguagem de programação C, em todos os métodos, os três algoritmos são executados em sequência e trabalho com assinatura de entrada semelhante, sendo uma lista de estruturas que possui primeiro vértice, segundo vértice e peso da aresta.

O método *main* é responsável por alimentar a lista de estruturas, após inserir nas funções de cada um dos algoritmos e após imprimir todos os custos a partir da origem (ver Figura 1).

```
UFF - Pos-graduacao em Computacao
Analise e Sintese de Algoritmos - 2019-2.
Professor Luis Antonio Brasil Kowada
Aluno Flavio Miranda de Farias
Analise de metodos de algotritmos de maximinizacao e minimizacao custo.

Distancias DIJKSTRA:
0: 0
1: 2
2: 3
3: 6
4: 4
5: 6
Tempo gasto pelo DIJKSTRA = 0.000000 segundos.

Distancias BELLMAN-FORD:
0: 0
1: 2
2: 3
3: 6
4: 4
5: 6
Tempo gasto pelo BELLMAN-FORD = 0.000000 segundos.

Distancias FLOYD-WARSHALL:
0: 0
1: 2
2: 3
3: 6
4: 4
5: 6

Tempo gasto pelo FLOYD-WARSHALL = 0.000000 segundos.

Process returned 0 (0x0)   execution time : 0.165 s
Press any key to continue.
```

Figura 1- Saída.

Para exemplificar foi utilizado o grafo [1] a baixo (ver Figura 2) segundo os dados da tabela a seguir (ver Tabela 2), mas é possível a implementação de grafos aleatórios, apesar de Dijkstra não suportar valores negativos como os outros.

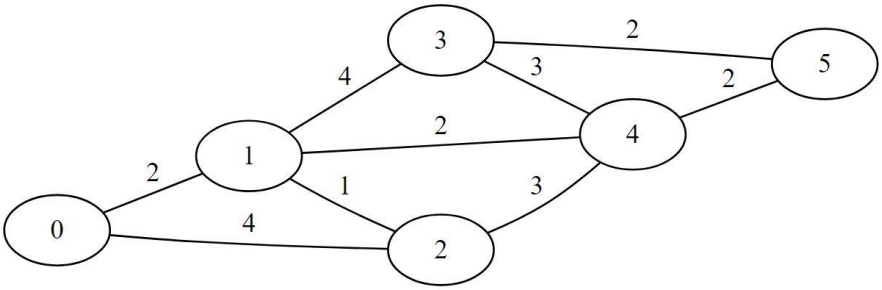

Figura 2- Grafo de exemplo.

Tabela 2- Dados do Grafo

Vértice	Vértice	Aresta (Peso)
0	1	2
0	2	4
1	2	1
1	3	4
1	4	2
2	4	3
3	4	3
3	5	2
4	5	2

Na saída, é cronometrado e exibido o tempo de duração de cada método.

[1] Para geração visual dos grafos foi utilizado a ferramenta (GRAPHVIZ, 2019).

4. RESULTADOS

Foram realizados dois tipos de experimentos, sendo o primeiro com um com um gráfico pequeno e fácil de ser reproduzido e visualizado graficamente e posteriormente, uma segunda bateria de experimentos automatizados e com dados randômicos.

Os experimentos foram realizados em um notebook com sistema operacional Windows 10, processador i5-7200 com 2,71GHz e 8 Gb RAM e SSD e todo este projeto como código fonte, tabulação de dados, entre outras, então disponíveis na pasta do GitHub (FARIAS, 2019) ou pelo link. A seguir como foi realizado os experimentos.

4.1. Primero experimento

Para os testes foram utilizados os dados da Tabela 2. Por se tratar de um grafo pequeno e facilitar a depuração humana dos caminhos. Como visto na Figura 1, todos os três métodos trouxeram resultados idênticos, considerando o vértice de origem 0 e com custo 0 para ele mesmo, temos para o vértice de destino 1 a Figura 3, para 2 a Figura 4, para 3 a **Erro! Fonte de referência não encontrada.**, para 4 a Figura 5 e para 5 a Figura 6 a seguir.

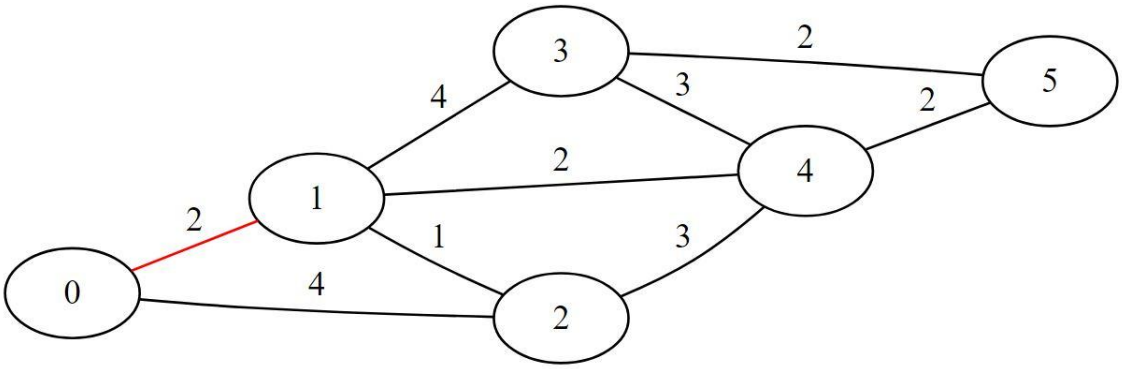

Figura 3- Caminho mais curto para 1.

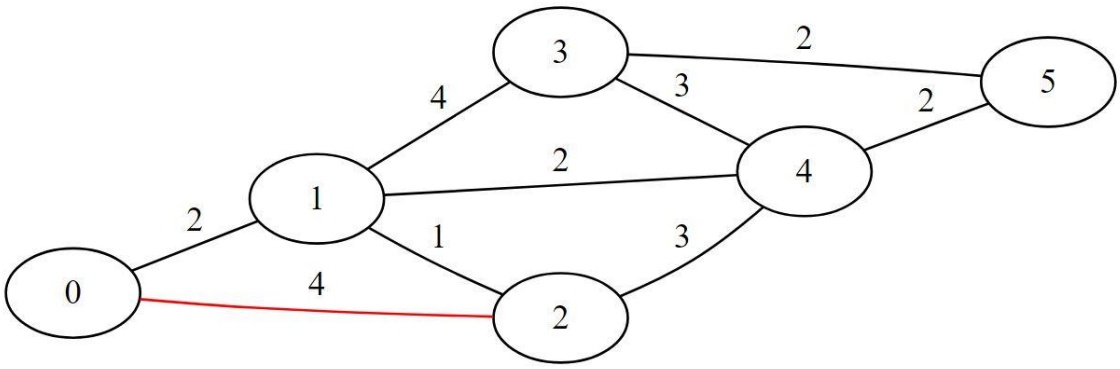

Figura 4- Caminho mais curto para 2.

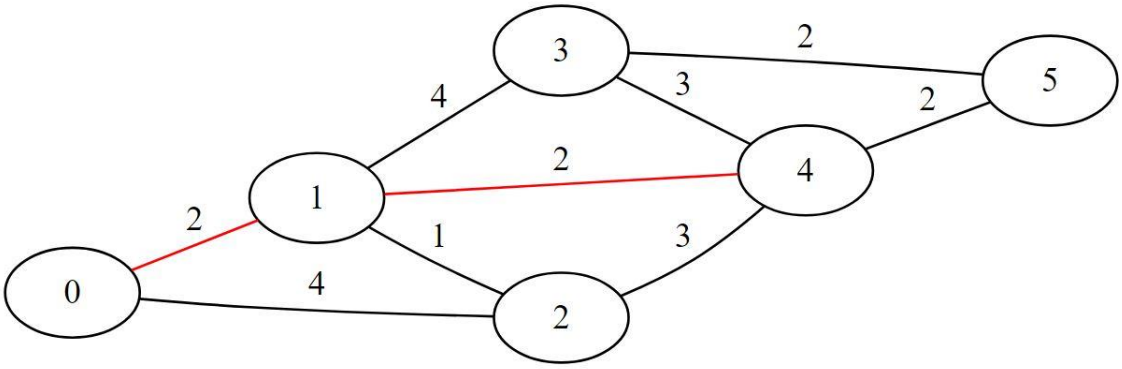

Figura 5- Caminho mais curto para 4.

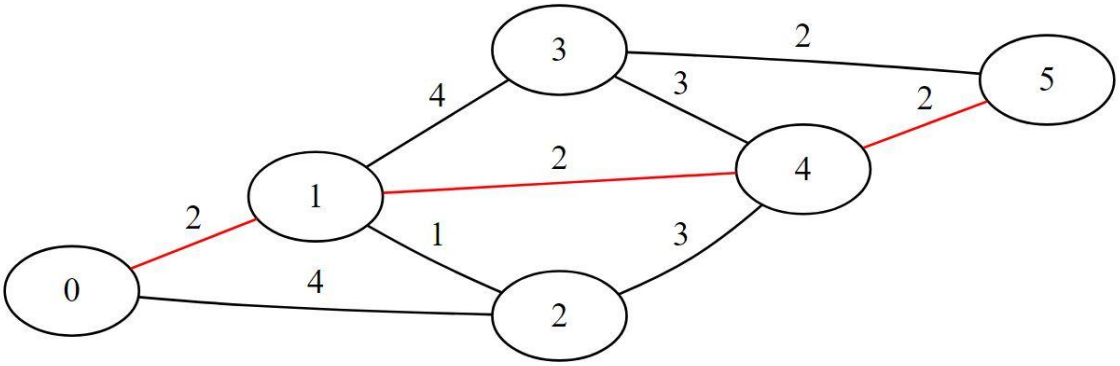

Figura 6- Caminho mais curto para 5.

Neste exemplo todos os códigos foram executados em tempo insignificante e equivalente como pode ser visto na Figura 1.

4.2. Experimento automatizado

Para esta segunda fase, foram implementadas algumas mudanças na função *main* do programa a fim de automatizar o processo:

- Criação de uma função randômica responsável por receber como entrada a quantidade de vértices e tamanho limite do peso dos vértices. Como saída retorna um vetor único com os dados do grafo, que será usado nos três algoritmos de caminho mais curto;
- Criação de laço de repetição responsável por gerar grafos com 2^2 a 2^{28} vértices e executar nos algoritmos;
- Criação de arquivo com os dados gerado durante o teste; e
- Otimização da exibição dos dados (ver Figura 8).

Após rodar o experimento, foi obtido os dados da Tabela 3, os dados eram compostos horizontalmente pela quantidade de vértices e os tempos de execução dos três algoritmos.

Tabela 3- Dados do experimento.

VERTICES	DIJKSTRA	BELLMAN	FLOYD
4	0.000000	0.000000	0.000000
8	0.000000	0.000000	0.000000
16	0.000000	0.000000	0.000000
32	0.000000	0.000000	0.000000

64	0.000000	0.000000	0.000000
128	0.000000	0.000000	0.000000
256	0.000000	0.000000	0.000000
512	0.000000	0.000000	0.000000
1024	0.000000	0.000000	0.000000
2048	0.000000	0.000000	0.000000
4096	0.000000	0.000000	0.000000
8192	0.000000	0.001000	0.000000
16384	0.000000	0.000000	0.000000
32768	0.001000	0.001000	0.000000
65536	0.000000	0.001000	0.000000
131072	0.000000	0.002000	0.002000
262144	0.000000	0.002000	0.001000
524288	0.000000	0.006000	0.002000
1048576	0.000000	0.004000	0.001000
2097152	0.017000	0.012000	0.005000
4194304	0.016000	0.032000	0.005000
8388608	0.049000	0.091000	0.030000
16777216	0.090000	0.195000	0.073000
33554432	0.211000	0.331000	0.127000
67108864	0.408000	0.715000	0.310000
134217728	2.547.000	3.259.000	2.521.000
268435456	12.507.000	15.917.000	16.209.999

Com a depuração dos dados, foi obtido o gráfico da Figura 7 que representa o crescimento do tamanho pelo tempo entre os três algoritmos.

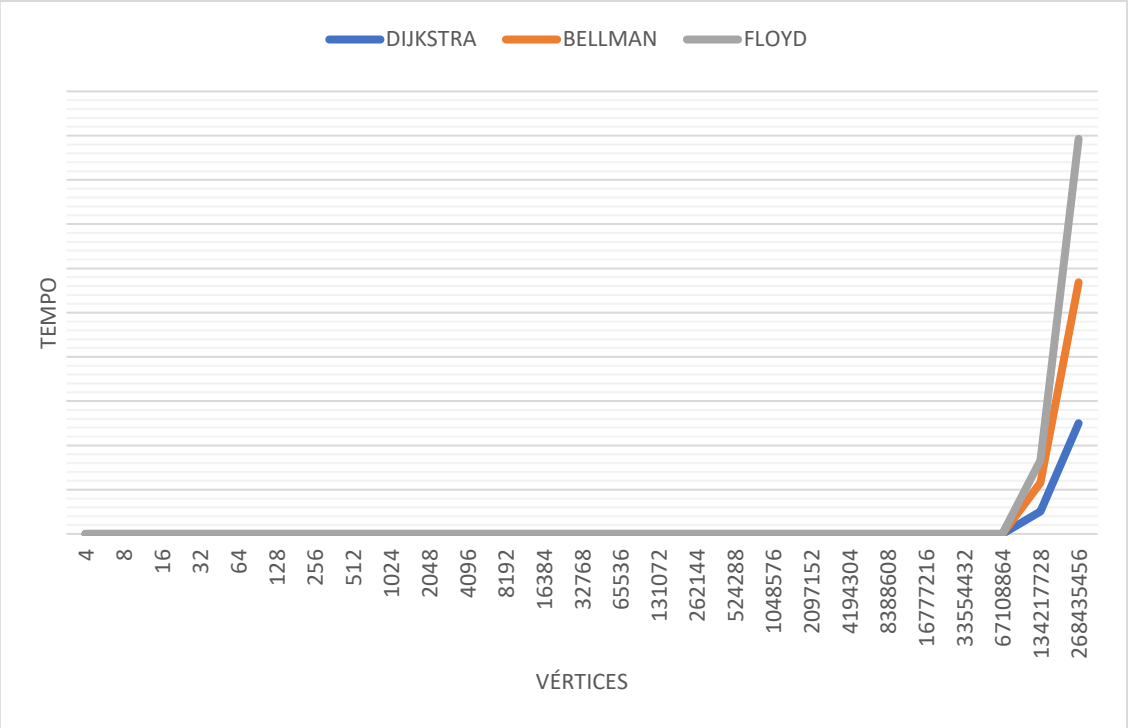

Figura 7- Tamanho por tempo.

```
UFF - Pos-graduacao em Computacao
Analise e Sintese de Algoritmos - 2019-2.
Professor Luis Antonio Brasil Kowada
Aluno Flavio Miranda de Farias
Analise de metodos de algotritmos de maximinizacao e minimizacao custo.

Para Grafo de 4 Vertices = 2^2

Distancias DIJKSTRA:
Tempo gasto pelo DIJKSTRA = 0.000000 segundos.

Distancias BELLMAN-FORD:
Tempo gasto pelo BELLMAN-FORD = 0.000000 segundos.

Distancias FLOYD-WARSHALL:
Tempo gasto pelo FLOYD-WARSHALL = 0.000000 segundos.
-------------------------------------------------------

Para Grafo de 8 Vertices = 2^3

Distancias DIJKSTRA:
Tempo gasto pelo DIJKSTRA = 0.000000 segundos.

Distancias BELLMAN-FORD:
Tempo gasto pelo BELLMAN-FORD = 0.000000 segundos.

Distancias FLOYD-WARSHALL:
Tempo gasto pelo FLOYD-WARSHALL = 0.000000 segundos.
-------------------------------------------------------

Para Grafo de 16 Vertices = 2^4

Distancias DIJKSTRA:
Tempo gasto pelo DIJKSTRA = 0.000000 segundos.

Distancias BELLMAN-FORD:
Tempo gasto pelo BELLMAN-FORD = 0.000000 segundos.

Distancias FLOYD-WARSHALL:
Tempo gasto pelo FLOYD-WARSHALL = 0.000000 segundos.
```

Figura 8-Tela inicial do sistema.

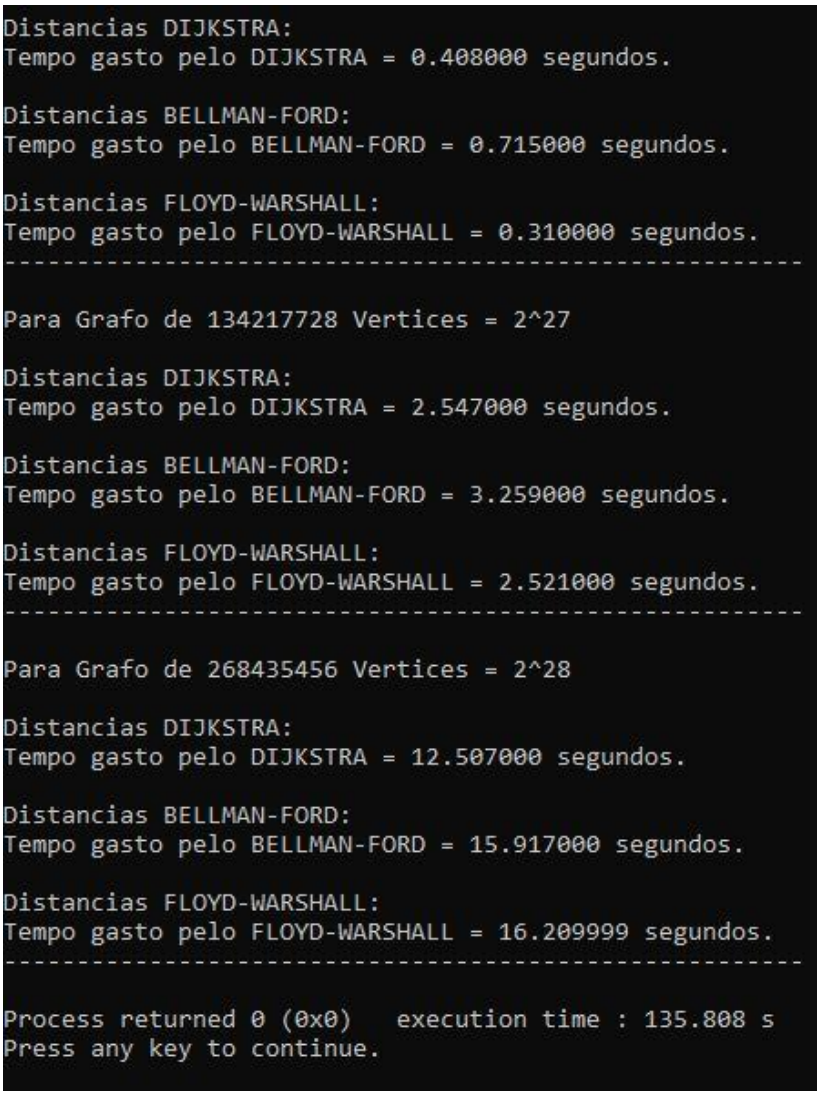

Figura 9- Tela final do sistema.

Com a analise dos dados, é possível observar que até o tamanho 2^{26} (onde representa o n_0) (ver Figura 9) praticamente não há alteração significativa no tempo dos três algoritmos e a partir daí, de então, há um crescimento o algoritmo de Dijkstra possui apresenta melhor desempenho, apesar de muito pouco, confirmando os dados apresentados na Tabela 1 que mostra a complexidade dos algoritmos.

5. CONCLUSÕES

As três soluções apresentadas para o problema clássico do caminho mais curto (*shortest path*), são bastante divulgadas e com o passar dos anos sofreram diversas melhorias, adaptando-se a diversas aplicações, como gerando rotas de trajetos em mapas ou gerenciamento de fluxo de redes. Temos o algoritmo de Dijkstra como o mais rápido em relação ao custo com $O(n^2)$ e o Floyd-Warshall como o menos rápido com $O(n^3)$, porem, juntamente com o Bellman-Ford, por serem baseados em programação dinâmica, podem até entregar melhores resultados, normalmente em grafos relevantemente maiores. O que realmente acontece na pratica com gerentes de rede, que possuem roteadores que necessitam estar atualizando suas tabelas de rotas para si e para enviar para outros roteadores em tempos curtos. Em redes grandes o custo de buscar a rota ótima através de um algoritmo dinâmico pode extrapolar a janela de atualização, sendo assim, mais útil ter um resultado bom com o algoritmo guloso.

6. REFERÊNCIAS

ANDREW S. TANENBAUM E DAVID WETHERALL. **Redes de Computadores**. São Paulo: Pearson Prentice Hall, 2011.

CORMEN, Thomas H. **Algoritmos : teoria e prática**. [s.l.] : Campus, 2012. Disponível em: <https://books.google.com.br/books?id=6iA4LgEACAAJ&dq=cormen+algoritmos+teoria+pratica&hl=pt-BR&sa=X&ved=0ahUKEwjHspTOutPkAhW3H7kGHfacDlMQ6AEIKTAA>. Acesso em: 15 set. 2019.

DE SOUZA, C. C. et al. **MO417-Complexidade de Algoritmos I**. São Paulo.

FARIAS, Flávio Miranda De. **fmflavio/Caminho-de-Custo-Minimo: DIJKSTRA BELLMAN-FORD FLOYD-WARSHALL**. 2019. Disponível em: <https://github.com/fmflavio/Caminho-de-Custo-Minimo>. Acesso em: 16 nov. 2019.

GAGNON, Michel. **Algoritmos e teoria dos grafos**. 2011. Disponível em: <http://www.inf.ufpr.br/andre/Disciplinas/BSc/CI065/michel/CaminhoMin/caminho_min.html>. Acesso em: 15 nov. 2019.

GRAPHVIZ. **Webgraphviz**. 2019. Disponível em: <http://www.webgraphviz.com/>. Acesso em: 15 nov. 2019.

KUROSE, Jim.; ROSS, Keith. **Redes de Computadores e a Internet**. 6. ed. São Paulo: Pearson, 2010. Disponível em: <http://loja.pearson.com.br/redes-de-computadores-e-a-internet-uma-abordagem-top-down-9788581436777/p>

MARTIN BROADHURST. **Martin Broadhurst**. 2019. Disponível em: <http://www.martinbroadhurst.com/>. Acesso em: 15 nov. 2019.

VASCONCELOS, Gabriel; GRAMACHO, Wladimir. **Projeto e Análise de Algoritmos Algoritmo de Floyd-Warshall**. Brasilia.