Projeto e Análise de Algoritmos – Caminhos Mínimos Utilizando Algoritmos de Dijkstra, Bellman-Ford, Floyd-Warshall, com detecção de Ciclos de Custo Negativo

Conrado C. Bicalho, Danilo S. Souza, Rodolfo L. M. Guimarães, Thiago Schons

[1]Departamento de Computação – Universidade Federal de Ouro Preto
35.400-000 – Ouro Preto - MG – Brasil

{conradobh, danilo.gdc, rodolfolabiapari, thiagoschons2}@gmail.com

Abstract. *This report aims to present the main algorithms for finding the shortest path between all pairs of nodes in a graph and these from all and all for everyone. The algorithms will be presented Dijkstra, Bellman-Ford, Floyd-Warshall with detection negative cost cycles, as well as their complexity, characteristics, implementation, results of experiments and final considerations.*

Resumo. *Este relatório tem como principal objetivo apresentar os principais algoritmos para encontrar o caminho de custo mínimo entre todos pares de nós de um grafo sendo estes de todos e de todos para todos. Serão apresentados os algoritmos de Dijkstra, Bellman-Ford, Floyd-Warshall com detecção de ciclos de custo negativo, assim como suas complexidades, características, implementação, os resultados obtidos dos experimentos realizados e considerações finais.*

1. Introdução à Teoria de Grafos

Segundo Netto [Netto 2003], a definição de um grafo pode ser dada por uma estrutura $G = (V, E)$ onde V é um conjunto discreto e não vazio de vértices e E é uma família de elementos não vazios chamado de arestas, definidos em função dos elementos em V. Cada aresta tem um ou dois nós associados a ela e faz o papel de interligar suas extremidades. A Figura 1 exibe um exemplo de grafo com seus vértices e arestas.

[Goldbarg 2012] define grafo como uma estrutura abstrata que representa um conjunto de elementos denominados **Vértices** e suas relações de interdependência denominadas **Arestas**. O termo Nó é empregado na literatura como sinônimo de vértice.

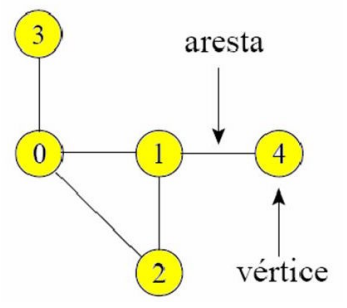

Figura 1. Grafo simples exibindo arestas e vértices. Fonte: Autor.

Um grafo orientado $G = (V, E)$, também descrito como grafo direcionado ou dígrafo, consiste em um conjunto não vazio de vértices V e um conjunto de arestas orientadas E. No grafo orientado o sentido das ligações entre os vértices é importante. Para esse tipo de grafo as arestas também são chamadas de **Arcos**. Cada aresta orientada está associada a um par ordenado de nós (u, v) onde tal arco começa em u e termina em v como é exibido na Figura 2.

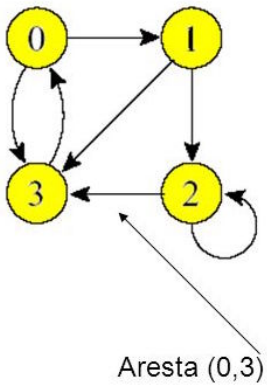

Figura 2. Exemplo simples de um grafo orientado. Fonte: Autor.

Tais arcos também podem ser ponderados sendo sua função peso $w(u, v) : E \to \mathbb{R}$ [Netto 2003].

1.1. Definição do Problema de Caminhos Mínimos de Única Origem

O problema de encontrar o caminho mínimo entre dois nós de um grafo é um dos problemas clássicos da ciência da computação. Este consiste, genericamente, em encontrar o caminho de menor custo entre dois nós de um grafo, considerando a soma dos custos associados aos arcos percorridos.[Hernandes et al. 2009]

Segundo Cormen [Cormen 2002], para definir o problema de caminho mínimo, deve considerar um grafo $G = (V, E)$ sem laços e valorado sobre os arcos por uma função peso $w : E \to \mathbb{R}$ mapeando arestas para pesos de valores reais sendo o peso $p = \langle v_1, v_2, \ldots, v_k \rangle$ é o somatório dos pesos de suas arestas constituintes

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Portanto, o peso do caminho mais curto de u até v é definida por

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{se existe um caminho de } u \text{ até } v, \\ \infty & \text{em caso contrário.} \end{cases}$$

Segundo Goldbard[Goldbarg 2012] a determinação do caminho mais curto entre qualquer par de vértice de um grafo é de ordem polinomial.

1.1.1. Prova da Subestrutura Ótima de um Caminho Mais Curto

Cormen [Cormen 2002] continua explicando claramente como é o lema e a prova deste problema:

Lema: Dado um grafo orientado ponderado $G = (V, E)$ com função peso $w : E \to \mathbb{R}$, seja $p = \langle v_1, v_2, \ldots, v_k \rangle$ um caminho mais curto do vértice v_1 até o vértice v_k e, para quaisquer i e j tais que $1 \leq i \leq j \leq k$, seja $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ o subcaminho p desde o vértice v_i até o vértice v_j. Então, p_{ij} é um caminho mais curto de v_i e v_j.

Prova: Quando realiza-se a decomposição do caminho p em $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, então teremos $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Supondo que existisse um caminho p'_{ij} de v_i até v_j com peso $w(p'_{ij}) < w(p_{ij})$. Então, $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ é um caminho de v_1 até v_k cujo peso $w(p) = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ é menor que $w(p)$, o que contradiz a hipótese de que p é um caminho mais curto de v_1 até v_k.

1.1.2. Arestas de Peso Negativo

Em algumas instâncias do problema, pode haver arestas cujo pesos são negativos. Deve-se ter em mente que se um grafo $G = (V, E)$ que não contenha nenhum ciclo de peso negativo acessível a partir da origem s, então para todo $v \in V$, o peso do caminho mais curto $\delta(s, v)$ permanece bem definido, mesmo tendo um valor negativo. Entretanto, caso exista um ciclo de peso negativo acessível, os pesos de caminhos mais curtos não serão bem definidos, permitindo uma série de caminhos mínimos entre s e v, pelo fato de sempre ser possível encontrar um caminho que tenha menor custo fazendo ciclos repetidos. Existindo o ciclo negativo no caminho de s até v, então $\delta(s, v) = -\infty$ [Cormen 2002].

Um exemplo é exibido na Figura 3. Tendo a premissa que o vértice de partida é o s, neste grafo, existe vértices que possuem caminho mínimo bem definido como os vértices b, d, h, i e j [1] e vértices que não possuem caminho mínimo bem definido como os vértices e, f, g no qual possuem pesos de caminhos mais curtos igual a $-\infty$.

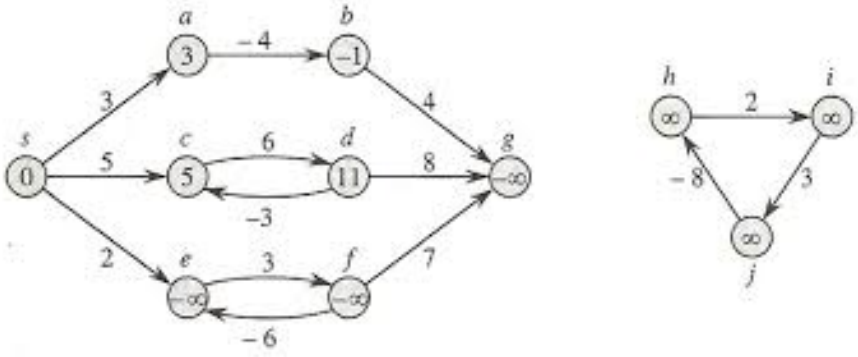

Figura 3. Exemplo simples de um grafo com arestas de peso negativo. Fonte: [Cormen 2002].

[1] Os vértices h, i e j são bem definidos pois todos são inacessíveis, tornando seu valor $+\infty$, ou seja, um valor bem definido segundo a definição do problema.

O Algoritmo de Dijkstra por exemplo, necessita que o peso de todos os arcos sejam positivos para determinar o caminho mínimo de forma correta. Por outro lado, os algoritmos de Bellman-Ford e de Floyd-Warshall permitem que as arestas possuam peso negativo no grafo de entrada e encontram o caminho corretamente, com exceção de quando houver um ciclo de peso negativo acessível a partir da origem. Vale ressaltar, que os algoritmos citados possuem detecção de ciclos negativos.

Os exemplos dados até agora está relacionado a apenas ao problema de menor caminho de uma única origem para todos os vértices. Mas além desta, existe uma variação deste problema que tem como propósito encontrar o caminho mais curto de todos os vértices para todos os vértices utilizando um único algoritmo sem utilizar repetições sucessivas dele. Ela é descrita a seguir.

1.2. Caminhos mais Curtos de Todos para Todos

Diferentemente do método citado anteriormente, que procura o caminho mínimo de todos os vértices a partir de uma única fonte, os algoritmos todos para todos, calculam a distância mínima de todos os pares de vértices de um grafo.

Tal como dito na Seção 1.1, tem-se um grafo orientado ponderado com sua função peso que mapeia as arestas como pesos de valores reais. Entretanto, deseja-se encontrar o caminho mais curto entre todos os pares de vértices $u, v \in V$ onde o peso de um caminho é a soma dos pesos de suas arestas constituintes. Este problema também pode ser resolvido utilizando um algoritmo de caminho mínimo um para todos $|V|$ vezes, uma para cada vértice de origem, mas existem algoritmos específicos para a resolução deste.

Naturalmente, sua estrutura de dados é lidada como uma matriz de adjacência onde é exibido os valores de todos para todos de acordo com sua linha e colina. Supondo que existem $|V|$ vértices e enumerados de forma crescente e contínua, uma matriz de entrada seria W $n \times n$ representando os peso das arestas do grafo orientado de n vértices. Isto é, $W = (w_{ij})$ onde

$$w_{ij} = \begin{cases} 0 & \text{se } i = j, \\ \text{o peso da aresta orientada } (i,j) & \text{se } i \neq j \text{ e } (i,j) \in E, \\ \infty & \text{se } i \neq j \text{ e } (i,j) \notin E. \end{cases}$$

Arestas negativas também são permitidas mas com a continuidade da restrição de ciclos negativos.

1.3. Variantes do Problema

Duas variantes do problema de caminho mínimo foram apresentadas nas subseções anteriores: Caminho mais curto de uma única origem; e de todos para todos. Outras variantes deste problema podem ser abordadas sem perder sua natureza, como por exemplo: o caminho mais curto de destino único; e de par único.

O problema de destino único busca encontrar as menores distâncias de todos os vértices para um determinado destino e de par único são problemas que desejam encontrar a menor distância entre dois pontos específicos do grafo.

1.4. Aplicações

O problema de caminhos mínimos se adapta a diversas situações práticas. Por exemplo, em roteamento pode-se modelar os vértices de um grafo como os cruzamentos, as arestas como vias, e os custo como distancia percorrida. Outra aplicação comumente utilizada é em redes de computadores, onde os nós representam o equipamentos, os arcos os trechos de cabeamento e os custos as taxas de transmissão, onde a solução seria a rota de transmissão mais rápida.

Estes problemas de uma única origem envolvem naturalmente problemas relacionados com sequências de decisões (escolhas de itinerários ao longo de uma viagem ou traçado de uma estratégia em um problema de investimentos). Sendo assim, trata-se de decisões envolvendo alguma forma de custo a ser minimizado. O problema de todos para todos seria útil numa possível elaboração de uma tabela de distância entre todos os pares de cidades de um certa região para um atlas rodoviário.

2. Os Algoritmos

Nesta seção, será descrito como a literatura aborda os algoritmos, suas principais características e a sua implementação junto com a análise de sua complexidade assintótica.

2.1. Considerações de Projeto e Análise

Para que o usuário fique ciente de algumas decisões utilizadas na implementação, esta seção tratará de alguns detalhes que são importantes para compreender o projeto dos algoritmos:

- Alguns algoritmos implementados tratam o 'infinito' como: o maior peso encontradas das arestas multiplicado por ele mesmo.
  Isso pois, ele será o maior valor e, comparado a todos os outros valores, ele é considerado infinito.
- Nenhum algoritmo faz teste de verificação de entradas inválidas.
  É necessário que todas as instâncias sejam compatíveis para cada um dos algoritmo, ou seja, não deve-se colocar uma instância com peso negativo no Algoritmo do Dijkstra pois ele não avisará erro e tentará executar normalmente.
- Foi executado em todos os algoritmos analisadores de código estáticos e dinâmicos. Executou-se primeiramente o Clang Static Analyzer e em seguida o Valgrind. Com exceção do Algoritmo de Bellman Ford, todos retornaram sucesso nas análises.
  O Algoritmo de Belmman Ford encontrou-se o seguinte *warning* que não fora resolvido, devido a não diminuir o desempenho de algoritmo:

```
scan-build: Using '/Users/pripyat/clang/bin/clang'
 for static analysis
main.c:147:18: warning: Potential memory leak
                          * maior_peso = -1;
                          ~~~~~~~~~~~~~~~~~

1 warning generated.
scan-build: 1 bug found.
```

2.2. Ambiente de *Hardware* e *Software* Utilizado

A descrição do ambiente de testes é descrito abaixo.

Tabela 1. Tabela com as informações de ambiente de execução do trabalho realizado.

Item	Descrição
Processador	1 Processador Intel Core i7 - 2,9 GHz
Núcleos	4 Núcleos
Cache L2 (por Núcleo)	256 KB
Cache L3	4 MB
Memória RAM	10 GB DDR3
Arquitetura	Arquitetura de von Neumann
Sistema Operacional	OS X 10.11.4 (15E65)
Versão do Kernel	Darwin 15.4.0
Compilador	Apple LLVM version 7.3.0 (clang-703.0.31)

2.3. Relaxamento

Segundo Cormen [Cormen 2002] os algoritmos possuem a técnica de relaxamento onde para cada vértice $v \in V$, mantém-se um atributo $d[v]$, chamado de estimativa de caminho mais curto, que é o limite superior sobre o peso do caminho mais curto entre s e v. Assim, como primeiro passo, a estima de distância de cada vértice é infinito, já que o algoritmo não sabe de antemão qual é o caminho mais curto até cada um deles[2]. Faz-se uma estimativa pessimista inicial para o caminho mínimo até cada vértice: $d(v) = \infty$.

O processo de relaxar uma aresta (u, v) consiste em testar alguma forma de melhorar o caminho mais curto para v encontrado até agora por outros caminhos intermediários que utilizem u. Um relaxamento realiza a alteração dos valores conhecidos das distâncias atuais atualizando-os para novos valores de acordo com os caminhos intermediários testados atualizando o novo predecessor de v como é exibido na Figura 4.

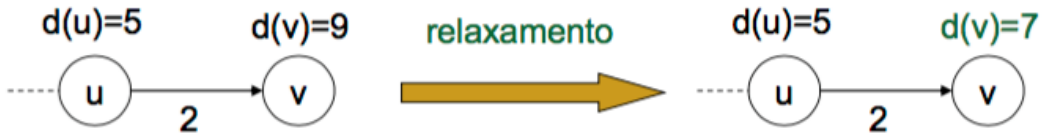

Figura 4. Exemplo de um relaxamento de uma aresta. Fonte: `http://wiki.icmc.usp.br/images/b/b4/7._1GrafosCaminhosLA(Graca).pdf`

Na Figura 4, existe um caminho já conhecido que utiliza a aresta (u, v) com custo de 9 unidades. O relaxamento consiste em encontrar outro caminho que utiliza um custo menor que o atual que no caso foi encontrado uma aresta com peso 2 partindo de $d[u] = 5$ totalizando $d[v] = 7$, ou seja, um caminho intermediário menor que o atual. O novo valor é atualizado e o processamento do algoritmo continua até que todos sejam analisados.

[2]Com exceção do no de partida s que a distância para ele mesmo é zero.

2.4. Algoritmo Bellman-Ford

De acordo com Cormen [Cormen 2002], tal algoritmo se baseia em algoritmos independentes criados por Bellman e Ford. Ele resolve o problema de caminhos mais curtos de uma única origem no caso mais geral[3]. Ao final, o algoritmo verifica se existe ou não um ciclo de peso negativo acessível a partir da origem. Não existe solução se e somente se existe um ciclo negativo a partir da origem. Caso contrário, retorna os caminhos mais curtos e seus pesos.

Utiliza a técnica de relaxamento já citada reduzindo progressivamente o peso do caminho da origem até $v \in V$ até alcançar o peso real do caminho mais curto.

Algorithm 1 Bellman-Ford

```
1:  procedure BELLMAN-FORD(G, w, s)
2:      INICIALIZA-UNICA-FONTE(G, s);
3:      for i ← 1, |V[G]| − 1 do
4:          for cada aresta (u, v) ∈ E[G] do
5:              RELAXA(u, v, w);
6:          end for
7:      end for
8:      for cada aresta (u, v) ∈ E[G] do
9:          if d[v] > d[u] + w(u, v) then
10:             return FALSE;
11:         end if
12:     end for
13:     return TRUE;
14: end procedure
```

Este pseudocódigo possui detector de ciclos negativos. Seu comando `for` situado na linha 9 realiza a verificação de valores de caminhos de menor custo que não serão bem definidos.

2.4.1. Análise do Algoritmo

Após a inicialização o algoritmo realiza $|V| - 1$ passagens sobre as arestas do grafo. Cada passagem consiste em relaxar cada aresta do grafo uma vez. Depois de fazer $|V| - 1$ passagens, é realizado a procura de um ciclo negativo.

Sendo assim, é executado no tempo $\mathcal{O}(VE)$ podendo então ser comparado a $\mathcal{O}(n^2)$ no pior caso. Isso pois a inicialização demora $\Theta(V)$, em seguida cada uma das $|V| - 1$ passagens sobre as arestas demoram $\Theta(E)$, e o loop final que demora $\mathcal{O}(E)$.

2.5. Algoritmo de Dijkstra

O algoritmo de Dijkstra surgiu em 1959 e resolve o problema de caminhos mais curtos de uma única origem em um grafo orientado de arestas com pesos não negativos. Em razão disso, deve-se supor sempre que $w(u, v) \geq 0$ para cada aresta $(u, v) \in E$.

[3]Caso que é permitido arestas com valores negativos.

O algoritmo mantém um conjunto S de vértices cujo pesos finais de caminhos mais curtos desde a origem já foram determinados. Seu diferencial é que ele seleciona repetidamente o vértice $u \geq V - S$ com a estimativa mínima de caminhos mais curtos, adiciona u a S e relaxa todas as arestas que saem de u [Cormen 2002].

Ele escolhe sempre o vértice 'mais leve' ou 'mais próximo' em $V - S$ para adicionar ao conjunto S e com isso utiliza-se a estratégia gulosa em sua execução.

Algorithm 2 Dijkstra

```
1:  procedure DIJKSTRA(G, w, s)
2:      INICIALIZA-UNICA-FONTE(G, s);
3:      S ← ∅;
4:      Q ← V[G];
5:      while Q ≠ ∅ do
6:          u ← RETIRA-MINIMO(Q);
7:          S ← S ∪ {u};
8:          for cada vértice v ∈ Adj[u] do
9:              RELAXA(u, v, w);
10:         end for
11:     end while
12: end procedure
```

Este algoritmo não possui detector de ciclo negativos. Aliás, como já mencionado, este algoritmo não suporta arestas negativas.

2.5.1. Análise do Algoritmo

Primeiramente ele realiza uma inicialização (linha 2) das arestas sendo sua complexidade $\mathcal{O}(E)$.

Em seguida, ele realiza uma remoção de cada nó que ele opera (linha 4 - 6) e logo depois realiza o relaxamento para cada item na franja (linha 8 e 9). Isso pode ser considerado $\mathcal{O}(V \log V)$.

Assim, o algoritmo, no pior caso tem complexidade de tempo $\mathcal{O}(E + V \log V)$.

2.6. Algoritmo Floyd-Warshall

Desenvolvido por Bernard Roy, Stephen Warshall e Robert Floyd, o algoritmo utiliza uma formulação de programação dinâmica para resolver o problema de caminhos mais curtos de todos os pares de grafos [Cormen 2002].

Algorithm 3 Floyd-Warshall

```
1:  procedure FLOYD-WARSHALL(W)
2:      n ← linhas[W];
3:      D ← W;
4:      for k ← 1, n do
5:          for i ← 1, n do
6:              for j ← 1, n do
7:                  d_ij^k ← min(d_ij^{k-1}, d_ik^{k-1} + d_kj^{k-1});
8:              end for
9:          end for
10:     end for
11:     return D;
12: end procedure
```

Além da matriz de pesos, também é necessário uma matriz de mesma dimensão que armazenará os predecessores de cada vértice que no pseudocódigo foi nomeada de D. Inicialmente, ela é preenchida com os valores das arestas obtidas dos dados de entrada e serão modificadas a cada relaxamento realizado ao longo da execução do algoritmo [Cormen 2002].

Este algoritmo assume que sua entrada não inclui nenhum ciclo negativo.

2.7. Análise do Algoritmo

Cada laço `for` deste algoritmo pode ser entendido como um somatório de um mesmo valor constante supondo que a operação da linha 7 é uma operação elementar. Assim, temos que sua complexidade é dada por

$$\sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n}1$$

Já que cada somatório é a repetição de o valor 1 n vezes, então este pode ser convertido para

$$n\sum_{k=1}^{n}\sum_{i=1}^{n}1$$

Portanto

$$n^3$$

Então, sua complexidade assintótica ‘pode ser descrita como $\mathcal{O}(n^3)$.

3. Experimentos

Para a realização dos experimentos, utilizou-se de um *script* em *Shell* para a execução dos testes de forma controlada e autônoma. Para cada iteração, foi analisado o tempo de execução e assim comparado com os demais algoritmos.

Para tal teste, utilizou-se de 3 (três) instâncias, entretanto, para que a execução de grafos com peso negativo sejam testados também, foi selecionado uma das instâncias e suas arestas foram alteradas para que se comporte como um grafo com arestas negativas, totalizando 4 instâncias. São elas: *rome99.gr*, *rg300_4730.gr*, *rg300_768_floyd.gr*, *rg300_768_floyd-n.gr* sendo a última modificada.

Isso foi necessário pois as instâncias com arestas de peso negativo encontradas na literatura eram grandes e colocaria o projeto em risco já que executar 20 (vinte) iterações de um problema poderia levar tempo demais para logo analisar, gerar gráficos e tabelas e a conclusão deste.

Para cada instância foi executada 20 (vinte) vezes no mesmo ambiente de testes para que os dados tenham uma média de tempo confiante.

Todos os resultados de cada iteração são exibido em anexo neste relatório, junto com cada algoritmo implementado.

3.1. Formato de Saída

Como meio de tornar o algoritmo útil, decidiu-se que ele teria como saída de arquivo a impressão de todos os caminhos por ele analisados (que no caso é todos para todos). O formato deste está da seguinte forma:

```
[origem,destino](distancia) caminho_origem_ate_destino
```

Um exemplo e exibido a seguir:

```
[1,2](8) 1 4 2
[1,3](9) 1 4 2 3
[1,4](5) 1 4
```

3.2. Análise de Tempo do Problema de Caminho Mínimo de Todos para Todos

Para melhor visualização dos resultados, foram desenvolvidos tabelas e gráficos.

Abaixo é exibido a tabela com os valores médios de cada algoritmo sobre cada instância. As tabelas com os valores de cada iteração é exibido em anexo no relatório.

Deve-se notar que a instância *rg300_768_floyd-n.gr* é a instância que foi alterada pelo grupo para que se comporte como um grafo com arestas de peso negativo e por isso, foi adicionado um *-n* no seu nome para diferenciar das outras.

Como já dito anteriormente, o Algoritmo de Dijkstra não suporta grafos com arestas negativas e por isso, seus dados sobre esta instância não se aplicam.

Tabela 2. Tabela com os valores de tempo médio de cada algoritmo nas quatro instâncias com o objetivo de obter caminhos mínimos de todos para todos.

Instância	Bellman-Ford (s)	Dijkstra (s)	Ford-Warshall (s)
rome99.gr	420.0757	93.59347	149.3292
rg300_4730.gr	1.788467	0.11464	0.118966
rg300_768_floyd.gr	0.294253	0.094997	0.116002
rg300_768_floyd-n.gr	0.289948	Não se aplica.	0.098479

A Figura 5 exibe um gráfico comparando os resultados de cada algoritmo sobre a instância *rome99.gr*.

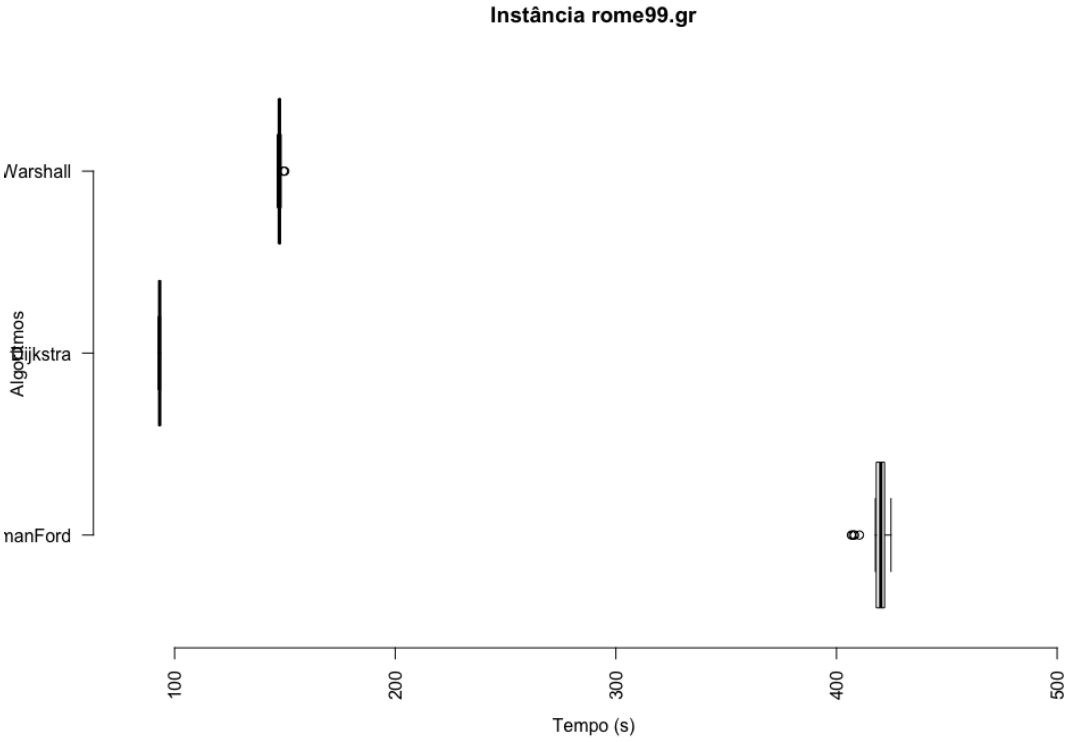

Figura 5. Tempo de execução de cada algoritmo sobre a instância *rome99.gr*. Fonte: Autor.

A Figura 6 exibe um gráfico comparando os resultados de cada algoritmo sobre a instância *rg300_4730.gr*.

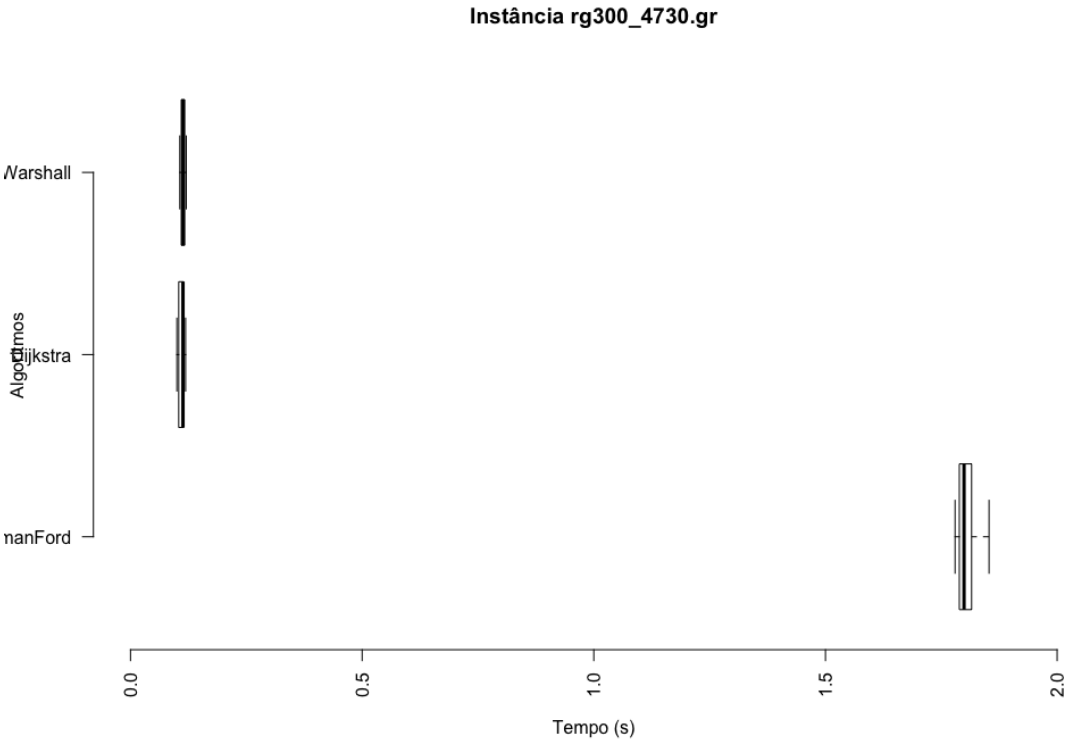

Figura 6. Tempo de execução de cada algoritmo sobre a instância *rg300_4730.gr*. Fonte: Autor.

A Figura 7 exibe um gráfico comparando os resultados de cada algoritmo sobre a instância *rg300_768_floyd.gr*.

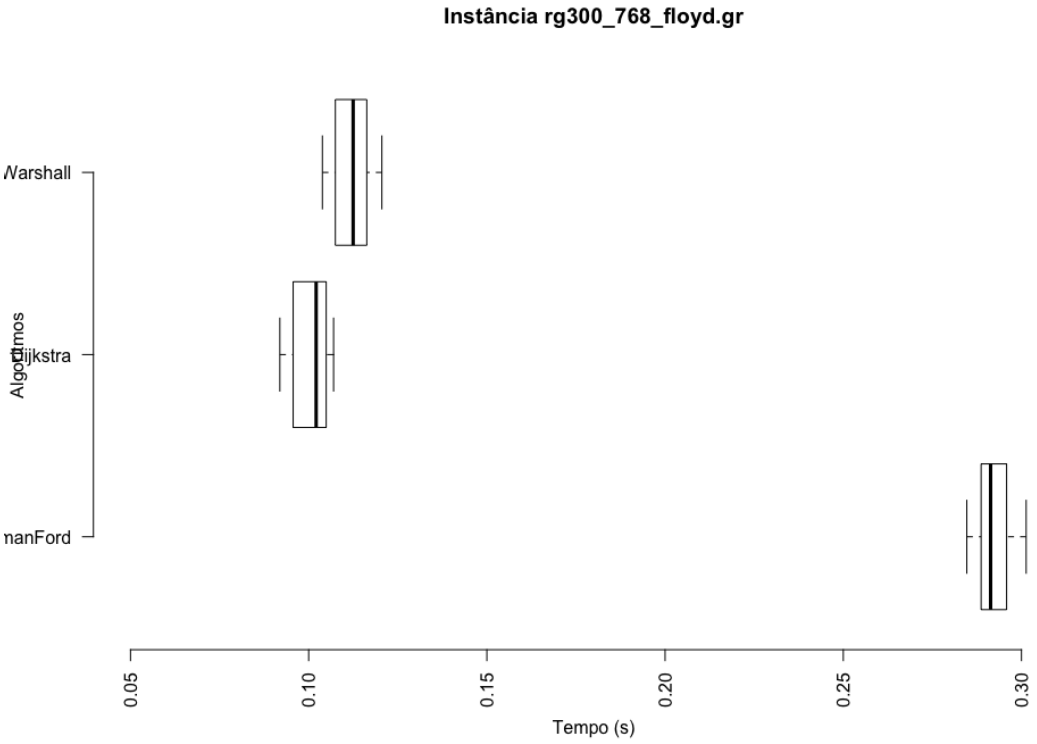

Figura 7. Tempo de execução de cada algoritmo sobre a instância *rg300_768_floyd.gr*. Fonte: Autor.

A Figura 8 exibe um gráfico comparando os resultados de cada algoritmo sobre a instância *rg300_768_floyd-n.gr*. Lembrando que o Algoritmo de Dijkstra não opera sobre grafos que tenham arestas com peso negativo.

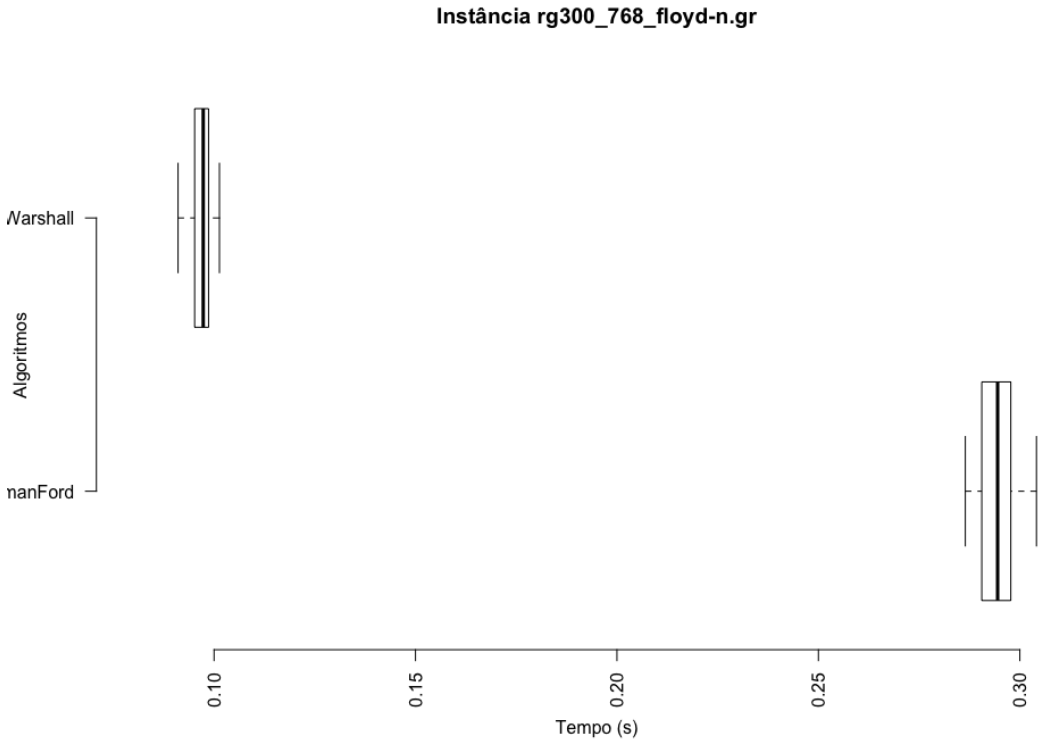

Figura 8. Tempo de execução de cada algoritmo sobre a instância com arestas negativas *rg300_768_floyd-n.gr*. Fonte: Autor.

3.3. Conclusão

Primeiramente, algo interessante a se notar é que o problema de caminhos mínimos também está relacionado a programação linear. É possível reduzir um caso especial de programação linear ao fato de encontrar caminhos mais curtos a partir de uma única origem. Tal problema pode ser resolvido com algoritmo de Bellman-Ford, sendo assim resolvendo também o problema de programação linear [Cormen 2002].

Os resultados foram expressos por meio de tabelas e gráficos (Boxplot) pelo fato de ambos serem itens fundamentais para a compreensão do leitor do relatório. Realizou-se 20 iterações de teste devido ao tempo computacional elevado pelo tamanho das instâncias utilizadas assim como a complexidade dos algoritmos.

Em termos de implementação, todos os códigos possuem grande facilidade de implementação. Principalmente o Algoritmo Floyd Warshall que se baseia numa simples estratégia que utiliza três laços `for` aninhados e realiza uma única verificação e atribuição dentro deles. Entretanto, em análise assintótica sua facilidade se torna um problema.

Na análise assintótica houve uma grande disputa entre o Bellman-Ford e Dijkstra. O Algoritmo de Floyd Warshall ficou fora dessa disputa por ser de complexidade de tempo $\mathcal{O}(n^3)$ e o Bellman tem complexidade $\mathcal{O}(n^2)$ e o Dijkstra $\mathcal{O}(E + V \log V)$ no pior caso. O algoritmo Dijkstra teve sucesso sua complexidade de tempo ser menor que todos os outros. A estrutura utilizada nele foi projetada pelos integrantes do grupo. Também houve vários problemas com *Warnings* que o compilador do Bellman-Ford relatou que, como já mencionado, o grupo não soube explicar a origem.

4. Algoritmos

4.1. Shell Script para Execução Autônoma

```
#!/bin/bash

# Questiona quantas iteracoes
echo "Quantas iteracoes?"
read quantidade_iteracoes;
echo

# Remove dados de iteracoes passadas
eval "rm saida*"
eval "rm tempo*"

# Compila os arquivos novamente
eval "gcc bellman/bellman.c -o bellman/bellman"
eval "gcc floyd/floyd.c -o floyd/floyd"
eval "gcc dijkstra/dijkstra.c -o dijkstra/dijkstra"

# Vetor de instancias
instancias=( rome99.gr rg300_4730.gr rg300_768_floyd.gr rg300_768_floyd-n.gr )

# Algoritmos a serem testados
algoritmos=( bellman dijkstra floyd )

# Execucoes
for algoritmo in "${algoritmos[@]}"
do
   echo $algoritmo

  for instancia in "${instancias[@]}"
  do
     echo $instancia

    for (( i = 0; i < "$quantidade_iteracoes"; i++ )); do
      echo "$i"

      cmd="./$algoritmo/$algoritmo $instancia"
      date
      echo $cmd
      $cmd
    done
    echo

  done
  echo

done
```

4.2. Bellman Ford em C

```c
/*
 * Trabalho de Projeto e Análise de Algoritmo
 * Mestrado em Ciência da Computação - Turma 16.1
 *
 * Alunos (nome, matricula, e-mail):
 *    Conrado
 *    Danilo Santos Souza                16.1.10149 - danilo.gdc@gmail.com
 *    Rodolfo Labiapari Mansur Guimarães 16.1.10163 - rodolfolabiapari@gmail.com
 *    Thiago Schons                      16.1.10186 - thiagoschons2@gmail.com
 *
 * Este arquivo executa o Algoritmo de Bellman Ford.
 *
 *
 * Para executar o arquivo utilize o comando:
 *    "./nomeDoPrograma benchmark"
 *
 *
 * Saída está descrita da seguinte forma: [origem, destino](distancia) caminho
 * Abaixo é exibido um exemplo
 * [1,2](8) 1 4 2
 * [1,3](9) 1 4 2 3
 * [1,4](5) 1 4
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/*
 * Estrutura que armazena os valores de uma aresta
 */
typedef struct Edges {
   int origem;
   int destino;
   float peso;
} Edge;


/*
 * Utilizou-se de uma pilha para imprimir a ordem de saída na forma
 * origem -> destino. O algoritmo naturalmente imprime de forma inversa, e por
 * isso necessitou de uma pilha.
 */
typedef struct Pilhas {
    int noh;
    struct Pilhas * prox;
} Pilha;


/*
 * Procedimento de empilhar um novo item na pilha
 */
void pushPilha (Pilha ** s, int noh) {
   Pilha * noh_pilha = calloc(1, sizeof(Pilha));

   noh_pilha->noh = noh;
```

```
57      noh_pilha->prox = *s;
58
59      *s = noh_pilha;
60  }
61
62  /*
63   * Procedimento de retirar um item da pilha
64   */
65  int popPilha (Pilha ** s) {
66      int retorno;
67      Pilha * proximo;
68
69      if (s != NULL) {
70          retorno = (*s)->noh;
71          proximo = (*s)->prox;
72
73          free(*s);
74
75          *s = proximo;
76
77          return retorno;
78
79      } else {
80          return -1;
81      }
82  }
83
84  /*
85   * Procedimento que realiza a retirada dos dados da pilha imprimindo cada um
86   * deles.
87   */
88  void imprimePilha(Pilha * s, FILE * f) {
89
90      if (s == NULL) {
91          return;
92      } else {
93
94          while (s != NULL) {
95              fprintf(f, " %d", s->noh);
96              popPilha(&s);
97          }
98      }
99      fprintf(f, "\n");
100 }
101
102
103 /*
104  * Procedimento que retira todos os itens da pilha
105  */
106 void esvaziaPilha(Pilha * s) {
107
108     Pilha * atual, * proximo;
109
110     if (s == NULL) {
111         return;
112     } else {
113         atual = s;
114         proximo = s->prox;
115
```

```
116         while (proximo != NULL) {
117            free (atual);
118
119            atual = proximo;
120            proximo = atual->prox;
121         }
122
123         free (atual);
124      }
125   }
126
127   /*
128    * Procedimento responsável por ler o arquivo e recolher as informações do
129    * grafo nele contido.
130    */
131   void le_arquivo(char * diretorio, Edge ** lista_arestas, int * vertices, int * arestas,
        ↪  int * maior_peso) {
132
133      // Define o ponteiro pro arquivo em modo de leitura
134      FILE * bench = fopen(diretorio, "r");
135
136      int i = 0, origem_tmp = 0, destino_tmp = 0, peso_tmp = 0;
137      char criou_lista = 0;
138
139      // Contador de quantas arestas foram lidas
140      int count_arestas = 0;
141
142
143      // Lê do arquivo o comando da linha
144      char comando = fgetc(bench);
145
146      // Enquando não for final de arquivo
147      while (comando != EOF) {
148
149         // Verifica qual comando é o comando
150         switch (comando) {
151            // Comentários serão ignorados
152            case 'c':
153                  while(fgetc(bench) != '\n') ;
154
155               break;
156
157            // Informações iniciais do grafo como número de vértices e
158               // arestas
159            case 'p':
160               if (!(fgetc(bench) == ' ')) {
161                  printf("Erro na inicializacao!\n");
162                  exit(2);
163               }
164               if (!(fgetc(bench) == 's')) {
165                  printf("Erro na inicializacao!\n");
166                  exit(2);
167               }
168               if (!(fgetc(bench) == 'p')) {
169                  printf("Erro na inicializacao!\n");
170                  exit(2);
171               }
172
173               // Le o número de vertices e arestas
```

```
174             fscanf(bench, "%d %d", vertices, arestas);
175
176             // Flag indicando que a lista foi criada
177             criou_lista = 1;
178
179             // Variável indicando o maior peso encontrado no momento
180             * maior_peso = -1;
181
182             // Aloca a quantidade exata de arestas lida previamente pelo
183                // arquivo
184             *lista_arestas = calloc(*arestas, sizeof(Edge));
185
186             break;
187
188          case 'a':
189
190             // Verifica a flag de criaçao da lista
191             if (criou_lista == 0) {
192                printf("Lista não criada!\n");
193                exit(-1);
194             }
195
196             // Lê a aresta
197             fscanf(bench, "%d %d %d", &origem_tmp, &destino_tmp, &peso_tmp);
198
199             // Atribou as informações no vetor, na primeira posição vazia
200             (*lista_arestas)[count_arestas].origem  = origem_tmp;
201             (*lista_arestas)[count_arestas].destino = destino_tmp;
202             (*lista_arestas)[count_arestas].peso    = peso_tmp;
203
204             // Verifica se encontrou algum peso maior que o já encontrado
205             if (peso_tmp > * maior_peso)
206                * maior_peso = peso_tmp;
207
208             // Le a aresta e seu valor
209             count_arestas++;
210
211             // Quebra a linha
212             fgetc(bench);
213
214             break;
215
216          default:
217             break;
218       }
219
220       // Le o proximo comando
221       comando = fgetc(bench);
222    } //while
223
224    // Verifica se a contagem de leitura de arestas foi realmente exato
225    if ((criou_lista == 0) || (count_arestas != *arestas)) {
226       printf("Numero de arestas está incorreto em relação ao arquivo.\n");
227       exit(-1);
228    }
229
230    // Para a representação do infinito, utilizou-se o maior peso encontrado ao
231    // quadrado
232    *maior_peso = *maior_peso * *maior_peso;
```

```
233
234     // Fecha o arquivo aberto
235     fclose(bench);
236  }
237
238  /*
239   * Procedimento simples para impressão de das distâncias encontradas
240   */
241  void imprimeDistancias(int * distancia, int vertices) {
242     int i;
243     printf("\n\n\n");
244     for (i = 0; i < vertices; i++)  {
245        printf("%4d ", i);
246     }
247
248     printf("\n");
249
250     for (i = 0; i < vertices; i++)  {
251        printf("%4d ", distancia[i]);
252     }
253
254  }
255
256  /*
257   * Procedimento que imprime os predecessores de cada vértice
258   */
259  void imprimePredecessores(int * predecessor, int vertices) {
260     int i;
261     printf("\n\n");
262     for (i = 0; i < vertices; i++)  {
263        printf("%4d ", i);
264     }
265
266     printf("\n");
267
268     for (i = 0; i < vertices; i++)  {
269        printf("%4d ", predecessor[i]);
270     }
271  }
272
273
274  /*
275   * Algoritmo De Belmman-Ford.
276   * Baseado no pseudocódigo do livro do Cormen.
277   */
278  void bellmanFord(Edge * lista_arestas, int ** distancia, int ** predecessor, int
       ↪  vertices, int arestas, int origem, int maior_peso) {
279     int i, j, peso;
280
281     // Aloca um vetor de distâncias e de predecessores temporários
282     int * distancia_temp   = calloc(vertices, sizeof(int));
283     int * predecessor_temp = calloc(vertices, sizeof(int));
284
285     // Inicializa os vetores temporários com
286     for (i = 0; i < vertices; i++) {
287        // Distância como 'infinito'
288        distancia_temp[i] = maior_peso;
289        // predecessor como inexistente no momento
290        predecessor_temp[i] = -1;
```

```
291     }
292
293     // Informa que a distância da origem pra ela mesma é 0
294     distancia_temp[origem - 1] = 0;
295
296     // Executa o algoritmo
297     for (i = 0; i < vertices; i++) {
298        for (j = 0; j < arestas; j++) {
299           peso = lista_arestas[j].peso;
300
301           // Verifica se existe uma distância menor por outro caminho
302           if (distancia_temp[lista_arestas[j].origem - 1] + peso <
              ↪  distancia_temp[lista_arestas[j].destino - 1] ){
303              // Se sim, realiza as atualizações
304              distancia_temp[lista_arestas[j].destino - 1] =
                 ↪  distancia_temp[lista_arestas[j].origem - 1] + peso;
305
306              predecessor_temp[lista_arestas[j].destino - 1] = lista_arestas[j].origem -
                 ↪  1;
307           }
308        }
309     }
310
311     // Realiza uma verificação final de ciclos negativos
312     for (i = 0; i < arestas; i++) {
313        peso = lista_arestas[i].peso;
314
315        if (distancia_temp[lista_arestas[i].origem - 1] + peso <
           ↪  distancia_temp[lista_arestas[i].destino - 1]) {
316           printf("[%d,%d] Graph contains a negative-weight cycle\n",
              ↪  lista_arestas[i].origem, lista_arestas[i].destino);
317           exit(-1);
318        }
319     }
320
321     // Retorna os vetores criados
322     *distancia = distancia_temp;
323     *predecessor = predecessor_temp;
324  }
325
326
327
328  /*
329   * Procedimento final que imprime o caminho para melhor visualização do usuário
330   * bem como o valor total da distância.
331   */
332  void imprimeTodosCaminhos(FILE * file, int * distancia, int * predecessor, int vertices,
       ↪  int origem) {
333
334     int anterior, i;
335
336     Pilha * stack;
337
338     for(i = 0; i < vertices; i++) {
339        if (i != origem -1) {
340           if (predecessor[i] != -1) {
341
342              fprintf(file, "[%d,%d](%d)", origem, i + 1, distancia[i]);
343
```

```
344                 stack = NULL;
345
346                 pushPilha(&stack, i + 1);
347
348                 anterior = predecessor[i];
349                 while(anterior != origem - 1) {
350                    pushPilha(&stack, anterior + 1);
351
352                    anterior = predecessor[anterior];
353                 }
354
355                 fprintf(file, " %d", origem);
356
357                 imprimePilha(stack, file);
358              }
359           }
360        }
361     }
362
363     /*
364      * Desalocas as memórias alocadas pelo algoritmo
365      */
366     void desaloca(Edge ** lista_arestas, int ** distancia, int ** predecessor) {
367
368        free(*lista_arestas);
369        free(*distancia);
370        free(*predecessor);
371     }
372
373
374     int main(int argc, char** argv) {
375
376        if(argc == 2) {
377           // Variáveis de cálculo de tempo
378           clock_t tempo_inicio, tempo_final;
379           double intervalo_real = 0;
380
381           Edge * lista_arestas = NULL;
382           int i, vertices = 0, arestas = 0, * distancia = 0, * predecessor = 0, maior_peso =
                ↪  0;
383           // Arquivo de saída de dados dos caminhos
384           FILE * file = fopen("saida_bellman.txt", "w+");
385           // Arquivo de tempos de execução
386           FILE * tempos = fopen("tempos_bellman.txt", "a");
387
388           le_arquivo(argv[1], &lista_arestas, &vertices, &arestas, &maior_peso);
389
390           for (i = 0; i < vertices; i++) {
391
392              tempo_inicio = clock();
393                 bellmanFord(lista_arestas, &distancia, &predecessor, vertices, arestas, i +
                      ↪  1, maior_peso);
394              tempo_final = clock();
395
396              intervalo_real += (double)(tempo_final - tempo_inicio) / CLOCKS_PER_SEC;
397
398              //imprimeTodosCaminhos(file, distancia, predecessor, vertices, i + 1);
399           }
400
```

```
401         fprintf(tempos, "%f\n", intervalo_real);
402 
403         fclose(file);
404         fclose(tempos);
405 
406         desaloca(&lista_arestas, &distancia, &predecessor);
407 
408     }
409     else {
410         printf("Argumentos Inválidos!\n");
411         exit(-1);
412     }
413 
414     return (EXIT_SUCCESS);
415 }
```

4.3. Algoritmo de Dijkstra em C

```c
/*
 * Trabalho de Projeto e Análise de Algoritmo
 * Mestrado em Ciência da Computação - Turma 16.1
 *
 * Alunos (nome, matricula, e-mail):
 *    Conrado
 *    Danilo Santos Souza                16.1.10149 - danilo.gdc@gmail.com
 *    Rodolfo Labiapari Mansur Guimarães 16.1.10163 - rodolfolabiapari@gmail.com
 *    Thiago Schons                      16.1.10186 - thiagoschons2@gmail.com
 *
 * Este arquivo executa o Algoritmo de Dijkstra.
 * Deve-se ter atenção à estrutura de dados utilizada já que os autores
 * decidiram utilizar um estrutura modificada para facilitar a execução deste.
 *
 *
 * Para executar o arquivo utilize o comando:
 *    "./nomeDoPrograma benchmark"
 *
 *
 * Saída está descrita da seguinte forma: [origem, destino](distancia) caminho
 * Abaixo é exibido um exemplo
 * [1,2](8) 1 4 2
 * [1,3](9) 1 4 2 3
 * [1,4](5) 1 4
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*
 * Estrutura que guarda informações do nó adjacente, possuindo
 * um ponteiro para o próximo nó adjacente
 * Esta estrutura depente da estrutura NohsIndividuais.
 */
typedef struct Nohs {
      int noh_id;         // Identificação do Noh
      int peso;           // Peso que esta adjacência tem
      struct Nohs * prox; // Ponteiro para a próxima adjacência
} Noh;

/*
 * Estrutura principal. Ela será um vetor que armazenará informações individuáis
 * de cada vertice além de um ponteiro para todos os seus adjacentes.
 * Assim, quando necessitar de uma informação de determinado noh, poderá ser
 * acessado em O(1). Além de forncer todos os seus adjacentes indicados pelo
 * ponteiro prox.
 */
typedef struct NohsIndividuais {
      char visitado;      // Flag
      int peso_atual;   // Distância deste nó até o nó de origem
      struct Nohs * prox; // Ponteiro para a próxima adjacência
} NohIndividual;


/*
```

```
57   * Utilizou-se de uma pilha para imprimir a ordem de saída na forma
58   * origem -> destino. O algoritmo naturalmente imprime de forma inversa, e por
59   * isso necessitou de uma pilha.
60   */
61  typedef struct Pilhas {
62      int noh;
63      struct Pilhas * prox;
64  } Pilha;
65
66
67  /*
68   * Procedimento de empilhar um novo item na pilha
69   */
70  void pushPilha (Pilha ** s, int noh) {
71     Pilha * noh_pilha = calloc(1, sizeof(Pilha));
72
73     noh_pilha->noh = noh;
74     noh_pilha->prox = *s;
75
76     *s = noh_pilha;
77  }
78
79  /*
80   * Procedimento de retirar um item da pilha
81   */
82  int popPilha (Pilha ** s) {
83     int retorno;
84     Pilha * proximo;
85
86     if (s != NULL) {
87        retorno = (*s)->noh;
88        proximo = (*s)->prox;
89
90        free(*s);
91
92        *s = proximo;
93
94        return retorno;
95
96     } else {
97        return -1;
98     }
99  }
100
101
102 /*
103  * Procedimento que realiza a retirada dos dados da pilha imprimindo cada um
104  * deles.
105  */
106 void imprimePilha(Pilha * s, FILE * f) {
107
108    if (s == NULL) {
109       return;
110    } else {
111
112       while (s != NULL) {
113          fprintf(f, " %d", s->noh);
114          popPilha(&s);
115       }
```

```
116     }
117     fprintf(f, "\n");
118  }
119
120
121  /*
122   * Procedimento que retira todos os itens da pilha
123   */
124  void esvaziaPilha(Pilha * s) {
125
126     Pilha * atual, * proximo;
127
128     if (s == NULL) {
129        return;
130     } else {
131        atual = s;
132        proximo = s->prox;
133
134        while (proximo != NULL) {
135           free (atual);
136
137           atual = proximo;
138           proximo = atual->prox;
139        }
140
141        free (atual);
142     }
143  }
144
145
146  /*
147   * Função que realiza a criação de um novo Nó
148   * preenchendo seus dados de acordo com os paramentros
149   */
150  Noh * criaNovoNoh(int destino, int peso) {
151
152     // Aloca uma nova estrutura Noh
153     Noh *noh = calloc(1, sizeof (Noh));
154
155     // Atribui as novas informações
156     noh->noh_id     = destino;
157     noh->peso       = peso;
158     noh->prox    = NULL;
159
160     // retorna seu endereço
161     return noh;
162  }
163
164  /*
165   * Procedimento que recebe dados para a criação de um novo nó adjacente
166   * já adicionando-o na sua respectiva lista colocando-o na primeira posição
167   * evitando a necessidade de percorrer a lista
168   */
169  void criaNovaAdjacencia(NohIndividual * lista, int origem, int destino, int peso) {
170
171     // Recebe o primeiro item adjacente
172     Noh * primeiro = lista[origem - 1].prox;
173
174     // Cria um novo nó
```

```
175     Noh * novo = criaNovoNoh(destino, peso);
176 
177     // Atribui este novo nó no início da lista
178     novo->prox = primeiro;
179     lista[origem - 1].prox = novo;
180 }
181 
182 /*
183  * Procedimento que realiza a criação da base da lista de adjacência que é
184  * feita pela struct NohIndividual. Ela guarda informações de cada nó
185  * individual como 'se foi visitado' e sua distância da origem no momento.
186  * Também tem um ponteiro para a estrutura Noh que representa os Nohs adjacentes
187  * deste nó, informando o peso da aresta e seu identificador.
188  */
189 NohIndividual * criaListaAdjacencia(int vertices) {
190 
191     // Aloca os nohs individuais que armazenaram as listas de adjacencia
192     NohIndividual * lista_adjacencia = calloc(vertices, sizeof(NohIndividual));
193     int i;
194 
195     // Define os valores iniciais de cada um desses nós
196     for (i = 0; i < vertices; i++) {
197         lista_adjacencia[i].peso_atual = 0;
198         lista_adjacencia[i].visitado = 0;
199         lista_adjacencia[i].prox = NULL;
200     }
201 
202     // Retorna a lista de todos os vértices vazia
203     return lista_adjacencia;
204 }
205 
206 /*
207  * Procedimento que realiza a impressão da lista de ajdacência.
208  */
209 void imprimeAdjacencia(NohIndividual * lista, int vertices, int arestas) {
210 
211     Noh * atual = NULL;
212     int i;
213 
214     printf("\nVértices: %d, Arestas: %d:\n", vertices, arestas);
215 
216     for (i = 0; i < vertices; i++) {
217         printf("%d ->", i + 1);
218         atual =  lista[i].prox;
219 
220         while (atual != NULL) {
221             printf(" %d", atual->noh_id);
222 
223             atual = atual->prox;
224         }
225         printf("\n");
226     }
227 }
228 
229 /*
230  * Procedimento responsável por ler o arquivo e recolher as informações do
231  * grafo nele contido.
232  */
233 NohIndividual * le_arquivo(char * diretorio, int * vertices, int * arestas) {
```

```
234
235     // Define o ponteiro pro arquivo em modo de leitura
236     FILE * bench = fopen(diretorio, "r");
237
238     int i, origem_tmp, destino_tmp, peso_tmp;
239     NohIndividual * lista_adjacencia = NULL;
240
241     // Contador de quantas arestas foram lidas
242     int count_arestas = 0;
243
244
245     // Lê do arquivo o comando da linha
246     char comando = fgetc(bench);
247
248     // Enquando não for final de arquivo
249     while (comando != EOF) {
250
251        // Verifica qual comando é o comando
252        switch (comando) {
253           // Comentários serão ignorados
254           case 'c':
255              // Le a linha inteira de comentario
256              while(fgetc(bench) != '\n') ;
257
258              break;
259
260           // Informações iniciais do grafo como número de vértices e
261              // arestas
262           case 'p':
263              if (!(fgetc(bench) == ' ')) {
264                 printf("Erro na inicializacao!\n");
265                 exit(-2);
266              }
267              if (!(fgetc(bench) == 's')) {
268                 printf("Erro na inicializacao!\n");
269                 exit(-2);
270              }
271              if (!(fgetc(bench) == 'p')) {
272                 printf("Erro na inicializacao!\n");
273                 exit(-2);
274              }
275
276              // Le o número de vertices e arestas
277              fscanf(bench, "%d %d", vertices, arestas);
278
279              // Cria a lista de adjacencia pra alimentá-la
280              lista_adjacencia = criaListaAdjacencia(*vertices);
281              break;
282
283           case 'a':
284              // Verifica se ja tenha lido a quantidade de arestas antes de ler
285                 // cada uma.
286              if (*vertices == 0 || * arestas == 00) {
287                 printf("Nenhuma aresta ou vértice lido\n");
288                 exit(-1);
289              }
290
291              // Le a aresta e seu valor
292              count_arestas++;
```

```
293
294             fscanf(bench, "%d %d %d", &origem_tmp, &destino_tmp, &peso_tmp);
295
296             // Adiciona a aresta à adjacencia
297             criaNovaAdjacencia(lista_adjacencia, origem_tmp, destino_tmp, peso_tmp);
298
299             // Quebra a linha
300             fgetc(bench);
301
302             break;
303
304          default:
305             break;
306
307       }
308
309       // Le o proximo comando
310       comando = fgetc(bench);
311    } //while
312
313
314    // Verifica se a contagem de leitura de arestas foi realmente exato
315    if ((*arestas < 1 || *vertices < 1) ||
316          (lista_adjacencia == NULL) ||
317          (count_arestas != *arestas)) {
318       printf("Leitura de arquivo obteve problemas.\n");
319       exit(-1);
320    }
321
322    // Fecha o arquivo aberto
323    fclose(bench);
324
325    // Retorna a lista
326    return lista_adjacencia;
327 }
328
329
330
331
332
333
334 /*
335  * Procedimento de inicialização do algoritmod e Dijkstra. Ele realiza a
336  * inicialização dos valores de distância (peso_atual) de cada nó e também
337  * sinalizando que eles ainda não foram visitados.
338  * Define que todos os vértices possuem o vertice de origem como o vértice
339  * anterior.
340  * Também inicializa os valores do vertice fonte que deverá ter propriedades
341  * diferente dos demais.
342  */
343 void inicializaDijkstra(int fonte, int vertices, int * vertice_anterior, NohIndividual *
      ↪  lista_adjacencia) {
344    int i;
345    // Inicializa todos dados individuais dos vertices
346    for (i = 0; i < vertices; i++) {
347
348       lista_adjacencia[i].peso_atual = -1;
349       lista_adjacencia[i].visitado = 0;
350
```

```
351         vertice_anterior[i] = fonte;
352     }
353 
354     // Redefine as informações do vertice fonte
355     lista_adjacencia[fonte - 1].peso_atual = 0;
356     vertice_anterior[fonte - 1] = -1;
357 }
358 
359 /*
360  * Procedimento que lista determinado vertice, passado por parâmetro, como
361  * vertice visitado, retirando da lista de disponíves.
362  * Procedimento utilizado para definir que o nó origem serja o primeiro a ser
363  * descartado de uso.
364  */
365 int extraiVertice(NohIndividual * lista_adjacencia, int id) {
366 
367     // Altera o valor de visitado para true.
368     lista_adjacencia[id - 1].visitado = 1;
369 
370     // Retorna o id do noh utilizado
371     return id;
372 }
373 
374 /*
375  * Procedimento que realiza o relaxamento do algoritmo de Dijkstra.
376  * Realiza a verificação dos nos adjacentes alterando as distancias dos seus
377  * respectivos a procura de novos caminhos.
378  */
379 void relaxamento(NohIndividual * lista, int vertice_anterior[], int origem_id) {
380 
381     // Recebe o primeiro noh adjacente
382     Noh * atual  = lista[origem_id - 1].prox;
383 
384     float peso_atual_temp;
385 
386     // Enquanto tiver adjacente para analisar
387     while (atual != NULL) {
388 
389         // Recebe a distância do nó atual
390         peso_atual_temp = lista[atual->noh_id - 1].peso_atual;
391 
392         // Verifica se no novo calculo, existe uma distância menor
393         if (peso_atual_temp < 0 ||
394                 (lista[origem_id - 1].peso_atual + atual->peso
395                     <
396                 peso_atual_temp)) {
397 
398             // Recebe o novo relaxamento
399             lista[atual->noh_id - 1].peso_atual = lista[origem_id - 1].peso_atual +
                  ↪  atual->peso;
400 
401             // define que o vertice antecessor a esse é o de origem aqui analisado
402             vertice_anterior[atual->noh_id - 1] = origem_id;
403         }
404 
405         atual = atual->prox;
406     }
407 }
408 
```

```c
/*
 * Função que realiza a procura de uma aresta ainda não utilizada e que
 * tenha o menor custo possível de distância.
 */
int extraiVerticeMenosCustoso(NohIndividual * listaAdjacencia, int vertices, int
  ↪  arestas) {

   NohIndividual * lista = listaAdjacencia;

   int i = 0;

   // salta todos os nós que não podem ser utilizados como:
      // nós já visitados ou arestas inexistentes
   while ((lista[i].visitado == 1 || lista[i].peso_atual < 1) && i < arestas)
      i++;

   // Verifica se excedeu a quantidade de arestas
   if (i != arestas -1) {

      // Se não tiver excedido, define o primeiro disponível como o menor para
         // futuras comparações
      int menor = i;

      // Comparando com o restante dos vértices
      for (++i; i < vertices; i++) {

         // Verifica se existe algum outro vertice disponível com aresta
            // menor que o atual
         if (lista[i].visitado == 0 &&
               lista[i].peso_atual > 0 &&
            lista[i].peso_atual < lista[menor].peso_atual) {

            // Indica qual é o menor para o retorno da função
            menor = i;
         }
      }

      // Define o vertisse como visitado
      lista[menor].visitado = 1;

      // Lembrando que o ID é indexado de 1
      menor++;

      // Retorna o id do vertice
      return menor;
   }
   else {
      // Tratamento de erro
      printf("Não foi encontrado uma nova aresta para operar.\nPrograma Finalizado.\n");
      exit(2);
   }
}

/*
 * Procedimento final que imprime o caminho para melhor visualização do usuário
 * bem como o valor total da distância.
 * IMPRIME O CAMINHO DE FORMA INVERSA: destino <- origem
 */
```

```c
void imprimeCaminho(int vertices, int vertice_anterior[vertices], NohIndividual
  ↪  lista_adjacencia[vertices],  int origem, int destino) {

   // Diz que o destino é o primeiro nó a ser percorrido anterior
   int anterior = destino;

   // Indica que será descrito o caminho para o usuário
   printf("\nCaminho Inverso:\n\t%d", destino);

   // Imprime o caminho de forma inversa
   while (vertice_anterior[anterior - 1] != origem) {
      printf(" <- %d", vertice_anterior[anterior - 1]);

      anterior = vertice_anterior[anterior - 1];
   }

   // Imprime o último item do caminho (que é a origem)
   printf(" <- %d.", vertice_anterior[anterior - 1]);

   // Imprime também a distância do caminho
   printf("\nDistância pecorrida: %d\n.", lista_adjacencia[destino - 1].peso_atual);
}


/*
 * Procedimento que realiza a impressão dos dados em arquivo para a análise.
 * Para a impressão, utiliza-se a pilha para que o caminho inverso seja impresso
 * de forma natural (origem -> destino)
 */
void imprimeTodosCaminhosArquivo(FILE * file, int vertices, int
  ↪  vertice_anterior[vertices], NohIndividual lista_adjacencia[vertices], int origem)
  ↪  {

   int anterior, i;

   // Pilha para armazenamento do caminho inverso
   Pilha * stack;

   // Para cada vertice diferente da origem
   for(i = 0; i < vertices; i++) {
      if (i != origem -1) {

         // Verifica se o vértice é inválido
         if (vertice_anterior[i] != -1) {

            // Imprime as informações básicas do arquivo como início, fim e
               // custo
            fprintf(file, "[%d,%d](%d)", origem, i + 1, lista_adjacencia[i].peso_atual);

            // Inicializa a pilha para que não exista lixo
            stack = NULL;

            // Coloca o primeiro item na pilha
            pushPilha(&stack, i + 1);

            // Informa qual é o próximo item a ser colocado na pilha
            anterior = vertice_anterior[i];

            // Enquanto não for a origem, adiciona os intermediários na pilha
```

```
522                 while(anterior != origem) {
523                     pushPilha(&stack, anterior);
524 
525                     anterior = vertice_anterior[anterior - 1];
526                 }
527 
528                 // Imprime a origem no arquivo
529                 fprintf(file, " %d", origem);
530 
531                 // Imprime os itens restantes no arquivo
532                 imprimePilha(stack, file);
533             }
534         }
535     }
536 }
537 
538 
539 /*
540  * Algoritmo de Dijkstra
541  * Baseado no pseudocódigo do livro do Cormen.
542  */
543 void dijkstra(int fonte, NohIndividual * lista_adjacencia, int vertices, int arestas,
      ↪   int vertice_anterior[vertices]) {
544 
545     // Quantidade de vertices adicionados no vertor resultante.
546         // Serve como medida para indicar término do cálculo
547     int quantidade_vetor_resultantes = 0;
548 
549     // Inicializa as variáveis do algorimo
550     inicializaDijkstra(fonte, vertices, vertice_anterior, lista_adjacencia);
551 
552     // Extrai o vertice mais leve
553     int vertice_mais_proximo = extraiVertice(lista_adjacencia, fonte);
554 
555     // Enquanto tiver vertice pra analizar
556     while (++quantidade_vetor_resultantes < vertices) {
557 
558         // Realiza o relaxamento da fronteira
559         relaxamento(lista_adjacencia, vertice_anterior, vertice_mais_proximo);
560 
561         // Extrai vertice mais leve
562         vertice_mais_proximo = extraiVerticeMenosCustoso(lista_adjacencia, vertices,
          ↪   arestas);
563     }
564 }
565 
566 /*
567  * Procedimento responsável por desalocar todos os dados alocados para a
568  * execução do algoritmo de Dijkstra.
569  */
570 void desaloca(NohIndividual ** lista_adjacencia, int vertices) {
571     // Cria ponteiros para a exclusão e indicação do próximo
572     Noh * deletar, * atual;
573     int i;
574 
575     // Para cada vertice, exclui seus adjacentes
576     for (i = 0; i < vertices; i++) {
577 
578         if ((*lista_adjacencia)[i].prox != NULL) {
```

```
579             deletar = (*lista_adjacencia)[i].prox;
580             atual = deletar->prox;
581 
582             while (atual != NULL) {
583                 free(deletar);
584 
585                 deletar = atual;
586                 atual = atual->prox;
587             }
588 
589             free(deletar);
590         }
591     }
592 
593     // Exclui o vetor base de vertices NohIndividual
594     free(*lista_adjacencia);
595 }
596 
597 
598 int main(int argc, char** argv) {
599 
600     if(argc == 2) {
601         // Variáveis de cálculo de tempo
602         clock_t tempo_inicio, tempo_final;
603         double intervalo_real = 0;
604 
605         int i, vertices = 0, arestas = 0, origem = 1, destino = 3;
606         // Arquivo de saída de dados dos caminhos
607         FILE * file = fopen("saida_dijkstra.txt", "w+");
608         // Arquivo de tempos de execução
609         FILE * tempos = fopen("tempos_dijkstra.txt", "a");
610 
611         NohIndividual * lista_adjacencia = le_arquivo(argv[1], &vertices, &arestas);
612 
613         if (vertices < 1)
614             exit(-1);
615 
616         // Vetor com os valores de vértices predecessores
617         int vertice_anterior[vertices];
618 
619         // Executa o algoritmo de Dijkstra de todos para todos
620         for (i = 0; i < vertices; i++) {
621 
622             // Calcula o tempo de execução
623             tempo_inicio = clock();
624                 dijkstra(i + 1, lista_adjacencia, vertices, arestas, vertice_anterior);
625             tempo_final = clock();
626 
627             // Soma o tempo calculado de cada origem
628             intervalo_real += (double)(tempo_final - tempo_inicio) / CLOCKS_PER_SEC;
629 
630             //imprimeTodosCaminhosArquivo(file, vertices, vertice_anterior,
                ↪  lista_adjacencia, i + 1);
631         }
632 
633         // Persiste o tempo total de execução
634         fprintf(tempos, "%f\n", intervalo_real);
635 
636         // Desaloca todos os itens utilizados no algoritmo
```

```
637         desaloca(&lista_adjacencia, vertices);
638 
639         // Fecha de forma correta os arquivos abertos
640         fclose(file);
641         fclose(tempos);
642     }
643     // caso contário, cancela a execução
644     else {
645         printf("Argumentos Inválidos!\n");
646         exit(-1);
647     }
648 
649     return (EXIT_SUCCESS);
650 }
```

4.4. Floyd Warshall em C

```c
/*
 * Trabalho de Projeto e Análise de Algoritmo
 * Mestrado em Ciência da Computação - Turma 16.1
 *
 * Alunos (nome, matricula, e-mail):
 *    Conrado
 *    Danilo Santos Souza                16.1.10149 - danilo.gdc@gmail.com
 *    Rodolfo Labiapari Mansur Guimarães 16.1.10163 - rodolfolabiapari@gmail.com
 *    Thiago Schons                      16.1.10186 - thiagoschons2@gmail.com
 *
 * Este arquivo executa o Algoritmo de Floyd Warshall.
 *
 *
 * Para executar o arquivo utilize o comando:
 *    "./nomeDoPrograma benchmark"
 *
 *
 * Saída está descrita da seguinte forma: [origem, destino](distancia) caminho
 * Abaixo é exibido um exemplo
 * [1,2](8) 1 4 2
 * [1,3](9) 1 4 2 3
 * [1,4](5) 1 4
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


/*
 * Procedimento que realiza a alocação das matrizes que o algoritmo necessita
 * para executar.
 */
void criaMatrizes(int *** distancia, int *** proximo, int vertices) {
   // Instancia o primeiro nível das matrizes
   *distancia = (int **) calloc(vertices, sizeof(int*));
   *proximo   = (int **) calloc(vertices, sizeof(int*));
   int i, j;

   // Instancia os outros níveis de cada matriz
   for (i = 0; i < vertices; i++){
      (*distancia)[i] = (int *) calloc(vertices, sizeof(int));
      (*proximo)[i]   = (int *) calloc(vertices, sizeof(int));
   }

   // Inicializa cada um de suas células
   for (i = 0; i < vertices; i++){
      for (j = 0; j < vertices; j++){
         (*distancia)[i][j] = -1;
         (*proximo)[i][j] = -1;
      }

      (*distancia)[i][i] = 0;
   }
}

/*
```

```
57   * Procedimento responsável por desalocar todas as matrizes utilizadas pelo
58   * programa
59   */
60  void destroiMatrizes(int *** distancia, int *** proximo, int vertices) {
61     int i;
62     for (i = 0; i < vertices; i++) {
63        free((*distancia)[i]);
64        free((*proximo)[i]);
65     }
66
67     free(*distancia);
68     free(*proximo);
69  }
70
71  /*
72   * Procedimento simples para a impressão dos dados de distancia
73   */
74  void imprimeDistancias(int ** d, int vertices) {
75     int i, j;
76     printf("\n");
77     printf("      0:   1:   2:   3:   4:\n");
78     for (i = 0; i < vertices; i++){
79        for (j = 0; j < vertices; j++){
80           if (j == 0) {
81              printf("\033[1m\033[37m");
82              printf("%3.d:  ", i);
83              printf("\033[0m");
84           }
85           printf("%2d   ", d[i][j]);
86        }
87        printf("\n");
88     }
89  }
90
91
92  /*
93   * Procedimento simples para a impressão dos dados de proximos
94   */
95  void imprimeProximos(int ** p, int vertices) {
96     int i, j;
97     printf("\n");
98
99     printf("      0:   1:   2:   3:   4:\n");
100    for (i = 0; i < vertices; i++){
101       for (j = 0; j < vertices; j++){
102          if (j == 0) {
103             printf("\033[1m\033[37m");
104             printf("%3.d: ", i);
105             printf("\033[0m");
106          }
107          printf("%3d ", p[i][j]);
108       }
109       printf("\n");
110    }
111 }
112
113 /*
114  * Procedimento responsável por ler o arquivo e recolher as informações do
115  * grafo nele contido.
```

```
116   */
117  void le_arquivo(char * diretorio, int *** matriz_distancia, int *** matriz_proximo, int
       ↪   * vertices) {
118
119     int ** m_distancia = * matriz_distancia;
120     int ** m_proximo   = * matriz_proximo;
121     int maior_distancia = -1;
122     char criou_matriz = 0;
123
124     // Define o ponteiro pro arquivo em modo de leitura
125     FILE * bench = fopen(diretorio, "r");
126
127     int i, j, origem_tmp, destino_tmp, peso_tmp, arestas_temp = 0, count_arestas = 0;
128
129     // Lê do arquivo o comando da linha
130     char comando = fgetc(bench);
131
132     // Enquando não for final de arquivo
133     while (comando != EOF) {
134
135        // Verifica qual comando é o comando
136        switch (comando) {
137           // Comentários serão ignorados
138           case 'c':
139                 while(fgetc(bench) != '\n') ;
140
141              break;
142
143           // Informações iniciais do grafo como número de vértices e
144              // arestas
145           case 'p':
146              if (!(fgetc(bench) == ' ')) {
147                 printf("Erro na inicializacao!\n");
148                 exit(2);
149              }
150              if (!(fgetc(bench) == 's')) {
151                 printf("Erro na inicializacao!\n");
152                 exit(2);
153              }
154              if (!(fgetc(bench) == 'p')) {
155                 printf("Erro na inicializacao!\n");
156                 exit(2);
157              }
158
159              // Le o número de vertices e arestas
160              fscanf(bench, "%d %d", vertices, &arestas_temp);
161
162              // Cria a lista de adjacencia pra alimentá-la
163              criaMatrizes(&m_distancia, &m_proximo, *vertices);
164
165              // Flag indicando criação de matriz
166              criou_matriz = 1;
167
168              break;
169
170           case 'a':
171              // Verifica se a matriz já foi criada
172              if (criou_matriz == 0) {
173                 printf("Matriz não criada!\n");
```

```
174                exit(-1);
175             }
176
177             count_arestas++;
178
179             // Lê a aresta do arquivo
180             fscanf(bench, "%d %d %d", &origem_tmp, &destino_tmp, &peso_tmp);
181
182             // adiciona as informações na estrutura
183             m_distancia[origem_tmp - 1][destino_tmp - 1] = peso_tmp;
184
185             m_proximo[origem_tmp - 1][destino_tmp - 1] = destino_tmp;
186
187             if (peso_tmp > maior_distancia)
188                maior_distancia = peso_tmp;
189
190             // Quebra a linha
191             fgetc(bench);
192             break;
193
194          default:
195
196             break;
197
198       }
199
200       // Le o proximo comando
201       comando = fgetc(bench);
202    } //while
203
204    // Verifica se a contagem de leitura de arestas foi realmente exato
205    if ((criou_matriz == 0) || (count_arestas != arestas_temp)) {
206       printf("Numero de arestas está incorreto em relação ao arquivo.\n");
207       exit(-1);
208    }
209
210    // Para a representação do infinito, utilizou-se o maior peso encontrado ao
211    // quadrado
212    maior_distancia = maior_distancia * maior_distancia;
213
214    // Inicializa a matriz distância com o valor infinito
215    for (i = 0; i < *vertices; i++) {
216       for (j = 0; j < *vertices; j++) {
217          if (m_distancia[i][j] == -1)
218             m_distancia[i][j] = maior_distancia;
219       }
220    }
221
222    *matriz_distancia = m_distancia;
223    *matriz_proximo = m_proximo;
224
225    // Fecha o arquivo aberto
226    fclose(bench);
227 }
228
229
230 /*
231  * Procedimento final que imprime o caminho para melhor visualização do usuário
232  * bem como o valor total da distância.
```

```c
 */
void imprimeCaminho(FILE * file, int ** proximo, int ** distancia, int origem, int
  ↪  destino) {
   int caminho;

   if (proximo[origem - 1][destino - 1] == -1) {
      printf("\nErros nos resultados [%d,%d]!\n", origem, destino);
      exit(-1);
   }
   else {
      fprintf(file, "[%d,%d](%d)", origem, destino, distancia[origem - 1][destino - 1]);

      caminho = origem;
      fprintf(file, " %d", caminho);

      while (caminho != destino) {
         caminho = proximo[caminho - 1][destino - 1];
         fprintf(file, " %d", caminho);
      }

      fprintf(file, "\n");
   }
}


/*
 * Algoritmo de Floyd Warshall.
 * Baseado no livro do Cormen
 */
void floydWarshall(int ** m_distancia, int ** m_proximo, int vertices) {
   int i, j, k;

   for (k = 0; k < vertices; k++) {
      for (i = 0; i < vertices; i++) {
         for (j = 0; j < vertices; j++) {
            if (i != k && j != k) {

               if (m_distancia[i][k] + m_distancia[k][j] < m_distancia[i][j]) {

                  m_distancia[i][j] = m_distancia[i][k] + m_distancia[k][j];
                  m_proximo[i][j]  = m_proximo[i][k];
               }
            }
         }
      }
   }
}

int main(int argc, char** argv) {

   if(argc == 2) {
      // Variáveis de cálculo de tempo
      clock_t tempo_inicio, tempo_final;
      double intervalo_real;

      int i, j, ** m_proximo = NULL, vertices, ** m_distancia = NULL;
      // Arquivo de saída de dados dos caminhos
      FILE * tempos = fopen("tempos_floyd.txt", "a");
      // Arquivo de tempos de execução
```

```
291         FILE * file_resultados = fopen("saida_floyd.txt", "w+");
292 
293         le_arquivo(argv[1], &m_distancia, &m_proximo, &vertices);
294 
295         tempo_inicio = clock();
296            floydWarshall(m_distancia, m_proximo, vertices);
297         tempo_final = clock();
298 
299         intervalo_real = (double)(tempo_final - tempo_inicio) / CLOCKS_PER_SEC;
300 
301         fprintf(tempos, "%f\n", intervalo_real);
302 
303         /*for (i = 0; i < vertices; i++) {
304            for (j = 0; j < vertices; j++) {
305               imprimeCaminho(file_resultados, m_proximo, m_distancia, i + 1, j + 1);
306            }
307         }*/
308 
309         fclose(file_resultados);
310         fclose(tempos);
311 
312         destroiMatrizes( &m_distancia, &m_proximo, vertices);
313 
314      }
315      else {
316         printf("Argumentos Inválidos!\n");
317         exit(-1);
318      }
319 
320      return (EXIT_SUCCESS);
321   }
```

5. Tempo de Cada Iteração

Instância	Iteração	Bellman-Ford (s)	Dijkstra (s)	Floyd-Warshall (s)
rome99.gr	1	420.075737	93.593469	149.329210
rome99.gr	2	406.778267	93.155683	149.901242
rome99.gr	3	407.557925	92.526349	148.299100
rome99.gr	4	408.164779	92.922567	147.298434
rome99.gr	5	410.388223	93.273871	147.728493
rome99.gr	6	417.608766	93.188093	147.472081
rome99.gr	7	420.067948	92.877990	147.649888
rome99.gr	8	423.579021	93.481498	147.947385
rome99.gr	9	418.428915	93.143297	147.331580
rome99.gr	10	419.904089	92.777797	147.839273
rome99.gr	11	421.807243	93.477219	147.339988
rome99.gr	12	422.480544	93.329407	146.594208
rome99.gr	13	423.582300	93.035231	147.465285
rome99.gr	14	421.011317	93.371306	146.509349
rome99.gr	15	420.321190	93.387431	147.291747
rome99.gr	16	419.562213	93.327032	148.152184
rome99.gr	17	421.270161	92.939897	146.992861
rome99.gr	18	421.778354	93.259190	147.915625
rome99.gr	19	424.687490	93.319218	146.724432
rome99.gr	20	419.337160	93.211982	147.195511
rg300_4730.gr	1	1.788467	0.114640	0.118966
rg300_4730.gr	2	1.799135	0.114424	0.108971
rg300_4730.gr	3	1.794091	0.116044	0.112652
rg300_4730.gr	4	1.787104	0.113575	0.116166
rg300_4730.gr	5	1.792932	0.102091	0.114990
rg300_4730.gr	6	1.831822	0.104733	0.118885
rg300_4730.gr	7	1.813434	0.112657	0.110113
rg300_4730.gr	8	1.830302	0.119035	0.111056
rg300_4730.gr	9	1.808650	0.113649	0.112631
rg300_4730.gr	10	1.805976	0.113598	0.115899
rg300_4730.gr	11	1.799024	0.100087	0.117284
rg300_4730.gr	12	1.817401	0.117614	0.112459
rg300_4730.gr	13	1.825335	0.100887	0.114111
rg300_4730.gr	14	1.853041	0.116985	0.106702
rg300_4730.gr	15	1.804325	0.103362	0.106481
rg300_4730.gr	16	1.786946	0.114903	0.120008
rg300_4730.gr	17	1.779710	0.106953	0.120401
rg300_4730.gr	18	1.793920	0.113056	0.109078
rg300_4730.gr	19	1.789469	0.105754	0.107920
rg300_4730.gr	20	1.782268	0.103506	0.116683
rg300_768_floyd.gr	1	0.294253	0.094997	0.116002
rg300_768_floyd.gr	2	0.299323	0.096287	0.120532
rg300_768_floyd.gr	3	0.296152	0.105085	0.117771
rg300_768_floyd.gr	4	0.287375	0.093086	0.105078

rg300_768_floyd.gr	5	0.288541	0.091911	0.117342
rg300_768_floyd.gr	6	0.284657	0.104280	0.113558
rg300_768_floyd.gr	7	0.289068	0.103326	0.106788
rg300_768_floyd.gr	8	0.292276	0.098178	0.114823
rg300_768_floyd.gr	9	0.287061	0.104746	0.106932
rg300_768_floyd.gr	10	0.291642	0.104740	0.107975
rg300_768_floyd.gr	11	0.293828	0.105183	0.103842
rg300_768_floyd.gr	12	0.295847	0.101016	0.112194
rg300_768_floyd.gr	13	0.286082	0.105628	0.112173
rg300_768_floyd.gr	14	0.295810	0.100730	0.109862
rg300_768_floyd.gr	15	0.297365	0.093277	0.116141
rg300_768_floyd.gr	16	0.291032	0.103123	0.106996
rg300_768_floyd.gr	17	0.289630	0.101063	0.112675
rg300_768_floyd.gr	18	0.288871	0.106986	0.117593
rg300_768_floyd.gr	19	0.288800	0.093148	0.116462
rg300_768_floyd.gr	20	0.301314	0.105861	0.112277
rg300_768_floyd-n.gr	1	0.289948	—	0.098479
rg300_768_floyd-n.gr	2	0.291154	—	0.097173
rg300_768_floyd-n.gr	3	0.298522	—	0.097863
rg300_768_floyd-n.gr	4	0.286472	—	0.097455
rg300_768_floyd-n.gr	5	0.289122	—	0.098396
rg300_768_floyd-n.gr	6	0.289937	—	0.091062
rg300_768_floyd-n.gr	7	0.294530	—	0.094978
rg300_768_floyd-n.gr	8	0.296994	—	0.096971
rg300_768_floyd-n.gr	9	0.292182	—	0.092122
rg300_768_floyd-n.gr	10	0.300920	—	0.093879
rg300_768_floyd-n.gr	11	0.299106	—	0.099244
rg300_768_floyd-n.gr	12	0.295046	—	0.101335
rg300_768_floyd-n.gr	13	0.295537	—	0.096205
rg300_768_floyd-n.gr	14	0.289527	—	0.101148
rg300_768_floyd-n.gr	15	0.293377	—	0.099427
rg300_768_floyd-n.gr	16	0.294441	—	0.096725
rg300_768_floyd-n.gr	17	0.304122	—	0.095385
rg300_768_floyd-n.gr	18	0.299738	—	0.098785
rg300_768_floyd-n.gr	19	0.293361	—	0.098140
rg300_768_floyd-n.gr	20	0.296438	—	0.093913

Referências

Cormen, T. H. (2002). *Algoritmos: teoria e prática*. Elsevier.

Goldbarg, M. (2012). *Grafos: Conceitos, algoritmos e aplicações*. Elsevier Brasil.

Hernandes, F., Berton, L., and Castanho, M. J. d. P. (2009). O problema de caminho mínimo com incertezas e restrições de tempo. *Pesquisa Operacional*, 29(2):471–488.

Netto, P. O. B. (2003). *Grafos: teoria, modelos, algoritmos*. Edgard Blücher.