Trabalho final de Teoria dos Grafos

Ana Carolina Gontijo Graça

2019.1

1 O Problema

O problema relacionado ao algoritmo de Johnson é a busca de caminhos mínimos em grafos com arcos negativos. Tal problema consiste na busca de um caminho dentre os arcos do grafo (apenas no sentido do arco) de forma que o custo de percorrer este caminho seja mínimo em relação a todos os outros caminhos possíveis entre os vértices selecionados. Este problema está definido utilizando grafos direcionados com pesos nas arestas. Sem perda de generalidade pode-se analisar este problema caso o grafo seja não-direcionado, basta adicionar arcos em ambas as direções, enquanto que se o grafo for sem pesos nas arestas basta fazer com que todas as arestas do grafo tenham peso 1.

2 Problemas Relacionados

Um problema de grafos relacionado com o problema de caminhos mínimos é o problema do caixeiro viajante, que consiste no problema de buscar um caminho que passe por todos os vértices com custo mínimo. Este problema é *NP*-completo e um dos grandes problemas da teoria da computação. Diversos problemas da vida real podem ser resolvidos com o problema de caminhos mínimos em grafos, como a distancia mínima de estradas entre cidades em que cidades são vértices e estradas são arestas com comprimentos no lugar dos pesos. Outro problema é uma linha de produção em que cada etapa pode ser feita com métodos que tem custos em dinheiro associados em arestas a partir do método anterior. Esse seria um grafo n-partido com n etapas, com vértices sendo os métodos e arestas sendo custos. Tendo um vértice representando o produto e um vértice representando o material cru. O caminho mínimo do material cru ao produto é o custo mínimo de produção.

3 A Estrutura dos Dados

Para a implementação da estrutura dos grafos, a estrutura de dados implementada inclui uma classe de nós e uma classe de grafos que vai incluir todos os nós naquele grafo.

3.1 Nó

A classe de nó vai incluir um inteiro que representa o índice do nó no grafo em que ele está incluído e um dicionário que lista os arcos que saem do nó representado . A chave de um arco é o índice do nó destino deste arco no grafo e o valor é o peso do arco.

3.2 Grafo

A classe de um grafo vai incluir apenas uma lista dos nós que estão neste grafo e um inteiro representando a quantidade de nós que existem no grafo.

4 Formulação Matemática e Prova de Corretude

O algoritmo de Johnson é um algoritmo de busca de caminhos mínimos em grafos. Ele é construído sobre o algoritmo de **Bellman Ford**, que por sua vez é construído sobre o algoritmo de **Dijkstra**. Todos foram implementados e serão explicados neste trabalho.

4.1 Dijkstra

O **Dijkstra** calcula o custo mínimo para ir de um nó v até cada um dos outros nós w. O custo mínimo de todos os nós é infinito inicialmente, exceto pelo custo mínimo até o próprio v, que é 0. A cada nó acessível a partir dos nós que já foram acessados adiciona-se o custo mínimo até aquele nó u calculando o custo mínimo de todos os w que acessam este u até u mais o custo mínimo de v até w.

A formulação matemática do **Dijkstra** que comprova sua corretude se baseia no fato de que um caminho mínimo não pode incluir um caminho maior que si mesmo, em um grafo com pesos positivos. Sendo assim, partindo de um vértice w se obtém os n primeiros caminhos mínimos a partir de w. Qualquer caminho encontrado após este momento terá pelo menos um custo igual ao maior caminho já encontrado. Seja $c[m]$ o $m$ésimo custo mínimo encontrado a partir de w. Seja $v[m]$ o vértice atingido no $m$ésimo custo mínimo encontrado.

Temos que:

$$c[n+1] \geq c[n] \tag{1}$$

Bem como, sabemos que nenhum caminho além dos previamente encontrados precisa ser analisado para se chegar a $v[n]$, pois se o $c[n]$ foi o $n$ésimo caminho isso significa que qualquer outro caminho encontrado depois será maior ou igual, resultando portanto num caminho para $v[n]$ maior ou igual ao previamente encontrado (considerando a existência de arestas nulas).

Além disso, supondo um caminho de v_0 a v_n que se deseja analisar se é mínimo que contém um caminho não mínimo. Neste caminho analisado, o trajeto de v_0 a v_i não é mínimo. Podemos criar um novo caminho que utiliza o caminho mínimo de v_0 a v_i no lugar deste trajeto do antigo caminho e mantém o restante como estava. Este caminho será menor que o caminho analisado e, portanto, o caminho analisado não será mínimo.

Sendo assim, todos os caminhos mínimos encontrados são efetivamente mínimos, e essa é a formulação matemática que garante isso.

4.1.1 Prova de corretude:

A corretude do algoritmo de Dijkstra se baseia em algumas propriedades do algoritmo. Sejam VIS os vértices que já foram visitados, $C[v]$ o conjunto de caminhos possíveis até v, por fim seja s o vértice inicial.

1. $c\_min[v]$ é o custo mínimo de ir até v com vértices intermediários em VIS
2. $\forall\ v \in VIS, \forall\ c \in C[v] : c\_min[v] \leq c$
3. Para cada vértice a ser adicionado em uma iteração, seu vértice anterior tem que já ter sido visitado.

, Temos que . Por indução nas iterações:

- Caso base: Quando nenhum vértice foi visitado, as distâncias mínimas são 0 se $v = s$ e ∞ se $v \neq s$. Isso respeita as propriedades ditas anteriormente, pois o caminho de custo mínimo pra cada vértice só tem vértices visitados (como não tem vértices no caminho de s a s, é um caminho nulo de custo 0), por outro lado é impossível chegar em qualquer outro vértice a partir de s passando por vértices visitados, pois nem o s foi visitado ainda. Evidentemente, 0 é caminho mínimo de s a s com arestas positivas.
- Caso indutivo: No fim da iteração de número k as propriedades valem.
- Passo indutivo: No início da iteração $k+1$ as propriedades seguem valendo, considerando que o vértice v_{k+1} é inserido nesta iteração. Como v_{k+1} foi inserido, isso quer dizer que ele era o vértice que tinha o menor custo mínimo fora dos vértices visitados. O custo mínimo de chegar até ele é relativo a um caminho que contém apenas vértices visitados , pela primeira propriedade. Para mostrar que este é efetivamente o menor caminho (que um caminho passando por outros vértices não pode ser menor que esse) mostramos que para "cruzar a fronteira" de vértices visitados e não visitados, é necessário passar por uma aresta, supondo que vamos passar por w para chegar a v_{k+1}. O caminho até w vai precisar passar por essa fronteira, o custo mínimo para atravessar essa fronteira é $c\_min[w] \geq c\_min[v_{k+1}]$ (caso contrário, v_{k+1} não teria sido escolhido). Como o grafo só tem arestas positivas, o caminho de w até v tem custo ≥ 0. O custo de chegar a v passando por w passa a ser

$\geq c\_min[w] + 0 \geq c\_min[v_{k+1}]$. Sendo assim, incluindo outros caminhos para chegar a v_{k+1} o custo mínimo se mantém, comprovando a segunda propriedade. Todos os nós que terão seus valores de *c_min* modificados nessa iteração vão ser nós atingíveis por v_{k+1} e esses valores só são modificados se o novo caminho criado passando por v_{k+1} for menor que o caminho mínimo anterior. Como ele era o caminho mínimo anterior, um caminho novo menor que ele é obrigatoriamente menor que todos os outros previamente encontrados. A propriedade de que o caminho relacionado ao valor do caminho mínimo até cada nó só contém nós já visitados se mantém pois v_{k+1} acaba de ser visitado e o caminho até ele também só continha nós visitados.

4.2 Bellman Ford

O algoritmo de **Bellman Ford** se baseia no algoritmo de **Dijkstra**, como já dito anteriormente. Suas modificações permitem que o novo algoritmo encontre caminhos mínimos mesmo em grafos com arestas de pesos negativos. A complexidade do algoritmo de **Bellman Ford** é $O(|V| \times |E|)$ pois diferente do algoritmo de Dijkstra ele não pode assumir que um caminho mínimo vai ser composto de caminhos mínimos.

Diferente do caso do **Dijkstra**, em grafos com arcos com pesos não necessariamente positivos os caminhos mínimos podem incluir caminhos não mínimos. Para realizar o cálculo dos caminhos e verificar se existe um ciclo negativo basta iterar sobre a quantidade de nós, o caminho mínimo com mais arestas possível tem $n - 1$ arestas. Caso com mais de $n - 1$ arestas o caminho passe a ser menos custoso do que o com no máximo $n - 1$ arestas isso quer dizer que existe um ciclo negativo no caminho.

4.2.1 Prova de corretude:

Seja s o nó inicial

- Supondo que o grafo não contém ciclos negativos atingíveis por s. Seja v atingível por s e seja *c_min_caminho* seu caminho mínimo de s a v. O caminho mínimo *c_min_caminho* tem no máximo $|V| - 1$ arestas, pois se não existem ciclos negativos o caminho mínimo não pode ter ciclos (pois adicionar um ciclo não diminuiria o custo do caminho) e o caminho máximo (em número de arestas) entre dois nós em um grafo é igual a $|V| - 1$ (caso contrário, algum nó seria repetido pelo princípio da casa dos pombos). Como são realizados $|V| - 1$ relaxamentos (isto é, atualizações de custos) para cada nó, o caminho será comparado com todos os caminhos de comprimento até $|V| - 1$ que levam até ele e portanto o caminho mínimo será escolhido. Se v não é atingível por s, o caminho mínimo é igual a ∞ evidentemente.

- Supondo que o grafo contém ciclos positivos. Seja CIC o ciclo, então a soma dos pesos nesse ciclo é < 0. Se o algoritmo não retornar false, isso quer dizer que $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$. No somatório disso, $\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$. No ciclo, $v_0 = v_k$, então isso implicaria que $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, o que implica que o ciclo não é negativo, o que é absurdo. Sendo assim, em algum a comparação acima achará um erro e retornará false.

4.3 Johnson

Por fim o algoritmo de Johnson. Ele é um algoritmo com o foco de encontrar o menor caminho de todo vértice para todo vértice. Ele busca reescrever este problema em um grafo que tem arcos negativos por um problema equivalente porem com um grafo apenas com arcos positivos. Para isso, é utilizado o algoritmo de Bellman-Ford no grafo inicial com um vértice adicional ligado a todos os vértices com custo 0.

O algoritmo de Johnson é composto por 3 etapas. A primeira etapa consiste em encontrar o caminho mínimo de todo vértice até cada vértice. Para isso é utilizado **Bellman Ford** a partir de um vértice artificial que vai para todos os vértices com custo 0.Cada vértice terá associado a ele o valor do caminho mínimo que chega nele a partir de qualquer outro vértice. A segunda etapa é a redefinição dos pesos utilizando os pesos associados aos vértices. Dado que o valor do vértice v é h_v, o novo peso de uma aresta entre (u, v) passa a ser o custo antigo somado a h_u e subtraído por h_v. Isso obrigatoriamente é positivo pois por definição $h_v \leq h_u + c(u, v)$. Sendo assim $h_u - h_v + c(u, v) \geq 0$.

4.3.1 Prova de corretude

Seja $c(v_i, v_j)$ o custo original dos arcos e $b(v_i, v_j)$ o custo do grafo auxiliar. Demonstração:

$$b(p) = \sum_{i=1}^{k} b(v_{i-1}, v_i) = \sum_{i=1}^{k} c(v_{i-1}, v_i) + h_{v_0} - h_{v_k} = c(p) + h_{v_0} - h_{v_k} \tag{2}$$

Sendo assim, $b(p)$ é mínimo se, e somente se, $c(p)$ é mínimo. Pela corretude de **Bellman Ford**, $c(p)$ é mínimo. Pela corretude de **Dijkstra** o algoritmo encontrará caminhos mínimos.

5 Exemplos

5.1 Grafos para os exemplos

Para o exemplo de execução de **Dijkstra** o grafo utilizado será o seguinte:

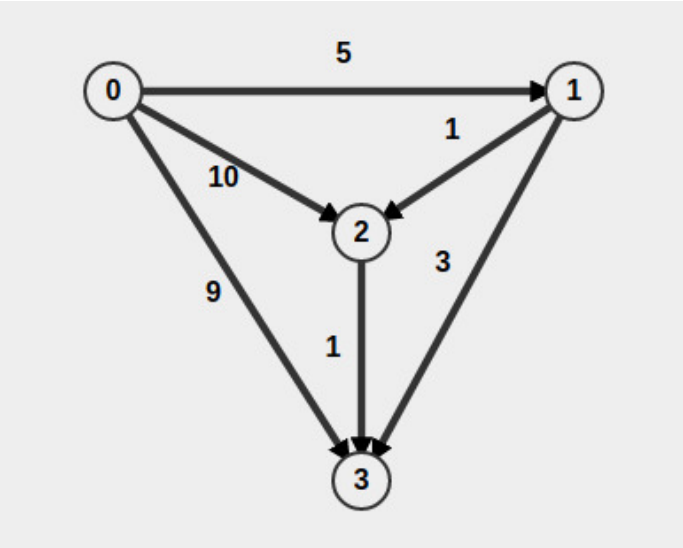

Figure 1: Exemplo 1 - Dijkstra

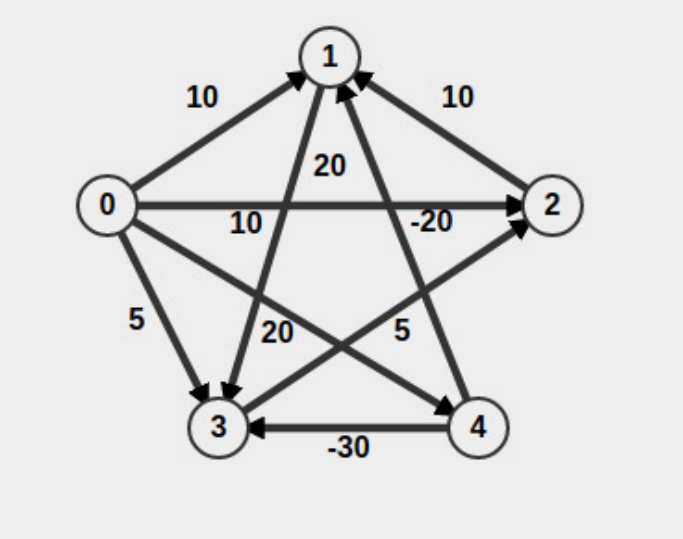

Figure 2: Exemplo 1 - Bellman Ford

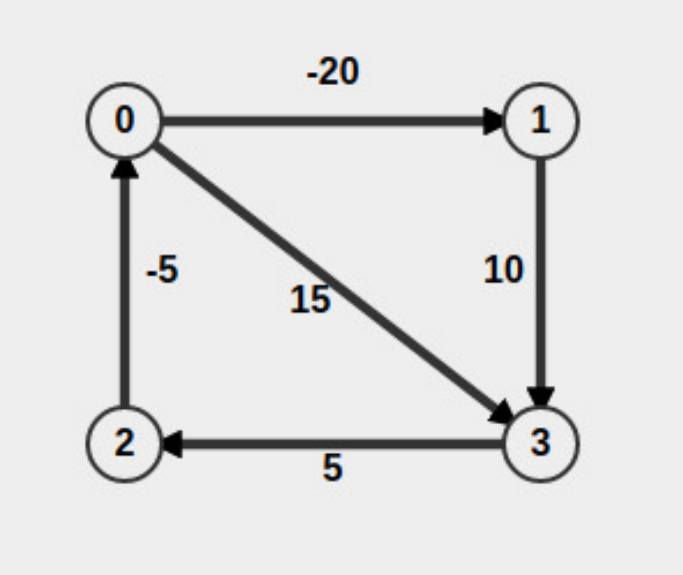

Figure 3: Exemplo 2 - Bellman Ford

- Dijkstra
  A execução do Dijkstra segue os passos a seguir ao rodar no grafo dado em 1:
  - Nó inicial: 0
  - Custos iniciais: [inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 0: 0
  - Insere o nó de maior valor: 0 — Novos custos mínimos: [0.0, inf, inf, inf]
  - Valores dos nós a serem inseridos: 1: 5.0, 2: 10.0, 3: 9.0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [0.0, 5.0, inf, inf]
  - Valores dos nós a serem inseridos: 2: 6.0, 3: 8.0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [0.0, 5.0, 6.0, inf]
  - Valores dos nós a serem inseridos: 3: 7.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [0.0, 5.0, 6.0, 7.0]

  - Caminhos mínimos:
  - Node number: 1 , Cost: 0.0
  - Node number: 2 , Cost: 5.0
  - Node number: 3 , Cost: 6.0
  - Node number: 4 , Cost: 7.0

- Bellman Ford
  A execução do Bellman Ford no grafo 2 é dada a seguir:

  - Nó inicial: 0
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Atualizar os custos mínimos 4 vezes.
  - 1 ª atualização - Custos mínimos: [0. inf inf inf inf]
  - 2 ª atualização - Custos mínimos: [0. 0. 10. -10. 20.]
  - 3 ª atualização - Custos mínimos: [0. 0. -5. -10. 20.]
  - 4 ª atualização - Custos mínimos: [0. 0. -5. -10. 20.]
  - Teste se existe loop negativo: *Não existe loop negativo*
  - Caminhos mínimos:
  - Node number: 1 , Cost: 0.0
  - Node number: 2 , Cost: 0.0
  - Node number: 3 , Cost: -5.0
  - Node number: 4 , Cost: -10.0
  - Node number: 5 , Cost: 20.0

A execução do Bellman Ford no grafo 3 é dada a seguir:

  - Nó inicial: 0
  - Custos iniciais: [inf, inf, inf, inf]
  - Atualizar os custos mínimos 3 vezes.
  - 1 ª atualização - Custos mínimos: [0. inf inf inf]
  - 2 ª atualização - Custos mínimos: [-10. -20. -5. -10.]
  - 3 ª atualização - Custos mínimos: [-20. -30. -15. -20.]
  - Teste se existe loop negativo: *Existe loop negativo incluindo o arco que vai de 0 para 1*

- Johnson
  A execução do Johnson no grafo 1 é dada a seguir:
  - Cria um novo nó completamente conectado aos outros.
  - Roda Bellman Ford neste novo grafo a partir do novo nó.
  - Nó inicial: 4
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Atualizar os custos mínimos 4 vezes.
  - 1 ª atualização - Custos mínimos: [inf inf inf inf 0.]
  - 2 ª atualização - Custos mínimos: [0. 0. 0. 0. 0.]
  - 3 ª atualização - Custos mínimos: [0. 0. 0. 0. 0.]
  - 4 ª atualização - Custos mínimos: [0. 0. 0. 0. 0.]
  - Teste se existe loop negativo: *Não existe loop negativo*
  - Custos mínimos no grafo auxiliar : [0. 0. 0. 0.]
  - Calculando os novos custos de arcos do nó 0 .
  - Nova aresta de 0 para 1 com peso: 5 + 0.0 - 0.0 = 5.0 .
  - Nova aresta de 0 para 2 com peso: 10 + 0.0 - 0.0 = 10.0 .
  - Nova aresta de 0 para 3 com peso: 9 + 0.0 - 0.0 = 9.0 .
  - Calculando os novos custos de arcos do nó 1 .
  - Nova aresta de 1 para 2 com peso: 1 + 0.0 - 0.0 = 1.0 .
  - Nova aresta de 1 para 3 com peso: 3 + 0.0 - 0.0 = 3.0 .
  - Calculando os novos custos de arcos do nó 2 .
  - Nova aresta de 2 para 3 com peso: 1 + 0.0 - 0.0 = 1.0 .
  - Roda Dijkstra para o nó 0 .
    * Nó inicial: 0
    * Custos iniciais: [inf, inf, inf, inf]
    * Insere o nó de maior valor: 0 — Novos custos mínimos: [0.0, inf, inf, inf]
    * Valores dos nós a serem inseridos: 1: 5.0, 2: 10.0, 3: 9.0
    * Insere o nó de maior valor: 1 — Novos custos mínimos: [0.0, 5.0, inf, inf]
    * Valores dos nós a serem inseridos: 2: 6.0, 3: 8.0
    * Insere o nó de maior valor: 2 — Novos custos mínimos: [0.0, 5.0, 6.0, inf]
    * Valores dos nós a serem inseridos: 3: 7.0
    * Insere o nó de maior valor: 3 — Novos custos mínimos: [0.0, 5.0, 6.0, 7.0]
  - Roda Dijkstra para o nó 1 .
    * Nó inicial: 1
    * Custos iniciais: [inf, inf, inf, inf]
    * Valores dos nós a serem inseridos: 1: 0
    * Insere o nó de maior valor: 1 — Novos custos mínimos: [inf, 0.0, inf, inf]
    * Valores dos nós a serem inseridos: 2: 1.0, 3: 3.0
    * Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, 0.0, 1.0, inf]
    * Valores dos nós a serem inseridos: 3: 2.0
    * Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, 0.0, 1.0, 2.0]

- Roda Dijkstra para o nó 2 .
  - Nó inicial: 2
  - Custos iniciais: [inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 2: 0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, inf, 0.0, inf]
  - Valores dos nós a serem inseridos: 3: 1.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, inf, 0.0, 1.0]
- Roda Dijkstra para o nó 3 .
  - Nó inicial: 3
  - Custos iniciais: [inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 3: 0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, inf, inf, 0.0]

[0. 5. 6. 7.]
[inf 0. 1. 2.]
[inf inf 0. 1.]
[inf inf inf 0.]

A execução do Johnson no grafo 2 é dada a seguir:

- Cria um novo nó completamente conectado aos outros.
- Roda Bellman Ford neste novo grafo.
- Nó inicial: 5 , Custos iniciais: [inf, inf, inf, inf, inf, inf]
- 1 ª atualização - Custos mínimos: [inf inf inf inf inf 0.]
- 2 ª atualização - Custos mínimos: [0. -20. 0. -30. 0. 0.]
- 3 ª atualização - Custos mínimos: [0. -20. -25. -30. 0. 0.]
- 4 ª atualização - Custos mínimos: [0. -20. -25. -30. 0. 0.]
- 5 ª atualização - Custos mínimos: [0. -20. -25. -30. 0. 0.]
- Teste se existe loop negativo: *Não existe loop negativo*
- Custos mínimos no grafo auxiliar : [0. 0. 0. 0.]
- Roda o Dijkstra para o nó 0 .
  - Nó inicial: 0
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 0: 0
  - Insere o nó de maior valor: 0 — Novos custos mínimos: [0.0, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 1: 30.0, 2: 45.0, 3: 35.0, 4: 20.0
  - Insere o nó de maior valor: 4 — Novos custos mínimos: [0.0, inf, inf, inf, 20.0]
  - Valores dos nós a serem inseridos: 1: 20.0, 2: 45.0, 3: 20.0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [0.0, 20.0, inf, inf, 20.0]
  - Valores dos nós a serem inseridos: 2: 45.0, 3: 20.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [0.0, 20.0, inf, 20.0, 20.0]
  - Valores dos nós a serem inseridos: 2: 20.0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [0.0, 20.0, 20.0, 20.0, 20.0]

- Roda Dijkstra para o nó 1 .
  - Nó inicial: 1
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 1: 0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [inf, 0.0, inf, inf, inf]
  - Valores dos nós a serem inseridos: 3: 20.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, 0.0, inf, 20.0, inf]
  - Valores dos nós a serem inseridos: 2: 20.0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, 0.0, 20.0, 20.0, inf]
- Roda Dijkstra para o nó 2 .
  - Nó inicial: 2
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 2: 0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, inf, 0.0, inf, inf]
  - Valores dos nós a serem inseridos: 1: 5.0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [inf, 5.0, 0.0, inf, inf]
  - Valores dos nós a serem inseridos: 3: 25.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, 5.0, 0.0, 25.0, inf]
- Roda Dijkstra para o nó 3 .
  - Nó inicial: 3
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 3: 0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, inf, inf, 0.0, inf]
  - Valores dos nós a serem inseridos: 2: 0.0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, inf, 0.0, 0.0, inf]
  - Valores dos nós a serem inseridos: 1: 5.0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [inf, 5.0, 0.0, 0.0, inf]
- Roda Dijkstra para o nó 4 .
  - Nó inicial: 4
  - Custos iniciais: [inf, inf, inf, inf, inf]
  - Valores dos nós a serem inseridos: 4: 0
  - Insere o nó de maior valor: 4 — Novos custos mínimos: [inf, inf, inf, inf, 0.0]
  - Valores dos nós a serem inseridos: 3: 0.0, 1: 0.0
  - Insere o nó de maior valor: 3 — Novos custos mínimos: [inf, inf, inf, 0.0, 0.0]
  - Valores dos nós a serem inseridos: 1: 0.0, 2: 0.0
  - Insere o nó de maior valor: 1 — Novos custos mínimos: [inf, 0.0, inf, 0.0, 0.0]
  - Valores dos nós a serem inseridos: 2: 0.0
  - Insere o nó de maior valor: 2 — Novos custos mínimos: [inf, 0.0, 0.0, 0.0, 0.0]

```
[ 0.  0. -5. -10.  20.]
[ inf 0.  15.  10.  inf]
[ inf 10.  0.  20.  inf]
[ inf 15.  5.  0.  inf]
[ inf -20. -25. -30.  0.]
```

A execução do Johnson no grafo 3 é dada a seguir:

- Cria um novo nó completamente conectado aos outros.
- Roda Bellman Ford neste novo grafo.
- Nó inicial: 4 , Custos iniciais: [inf, inf, inf, inf, inf]
- Atualizar os custos mínimos 4 vezes.
- 1 ª atualização - Custos mínimos: [inf inf inf inf 0.]
- 2 ª atualização - Custos mínimos: [-10. -20. -5. -10. 0.]
- 3 ª atualização - Custos mínimos: [-20. -30. -15. -20. 0.]
- 4 ª atualização - Custos mínimos: [-30. -40. -25. -30. 0.]
- Teste se existe loop negativo: *Existe loop negativo incluindo o arco que vai de 0 para 1*

6 Complexidade

6.1 Dijkstra

Sabemos que o algoritmo Dijkstra tem complexidade igual a O($|E| + |V| \times log(|V|)$) onde E é o conjunto das arestas do grafo e V é o conjunto de vértices do grafo. Esta complexidade é dada pela análise do pseudo-código a seguir.

Algorithm 1 Dijkstra

```
1:  procedure DIJKSTRA
2:      for v in V do
3:          mincost[v] ← ∞
4:      neighbors ← {s}
5:      while neighbors ≠ ∅ do
6:          n ← node that has minimum cost in neighbors
7:          neighbors ← neighbors−{n}
8:          cmin[n] ← n's cost in neighbors
9:          for u in n neighbors do
10:             if u not in visited then
11:                 neighbors ← neighbors+{u}
```

O **for** da linha 2 adiciona uma complexidade de $O(|V|)$ ao algoritmo. A linha 4 adiciona uma constante então é irrelevante em questão de complexidade assintótica. O **while** da linha 5 por si só vai ter uma complexidade de aproximadamente $O(|V|)$ (só é menor que isso caso nem todo vértice seja alcançável pelo vértice inicial), a linha 6 tem complexidade aproximadamente $O(log(|V|))$ pois os vizinhos que vão ser analisados são sempre os nós folha da árvore gerada pelos vértices atingíveis pelo nó inicial, sendo assim a complexidade é $O(|V| \times log(|V|)$ por enquanto. Por fim o **for** na linha 9 adiciona a complexidade de $O(|E|)$ pois no total o que este comando irá ter feito é passado por todas as arestas (a menos que o grafo não tenha todos os vértices atingíveis pelo vértice inicial, em que haverão arestas que não serão percorridas). Sendo assim a complexidade total do algoritmo é igual a $O(|E| + |V| \times log(|V|))$.

6.1.1 Análise temporal

Os resultados gráficos e as tabelas a seguir são relacionados aos mesmos dados, apenas apresentados de maneiras diferentes para melhor visualização. Os grafos usados para os testes são gerados aleatoriamente, com o cuidado de não gerar arcos duplicados. É possível perceber que para Dijkstra existe um nível de aleatoriedade nos resultados que está intimamente relacionado com a densidade dos grafos e sua aleatoriedade de geração. Tendo dois grafos com a mesma densidade, se em um todos os nós forem atingíveis a partir do nó inicial e em outro o nó inicial for um nó sozinho sem vizinhos, o Dijkstra irá ser muito mais rápido no segundo caso já que o $|V|$ da complexidade é relacionado aos nós atingíveis, enquanto que o primeiro caso será mais lento.

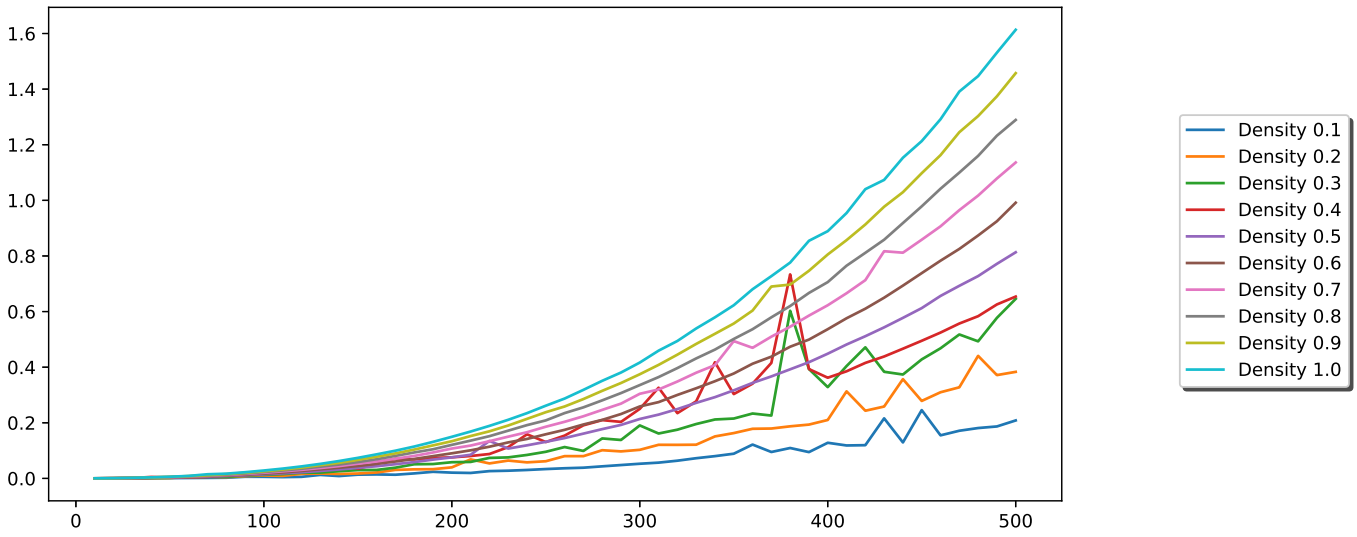

Figure 4: Resultados gráficos - Dijkstra

Table 1: Tabela de resultados - Dijkstra

	Densidades									
$\|V\|$	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
10	0.000108	0.000080	0.000126	0.000193	0.000199	0.000218	0.000209	0.000240	0.000245	0.000345
20	0.000035	0.000363	0.000407	0.000538	0.000546	0.000626	0.000736	0.000783	0.000842	0.000952
30	0.000693	0.000607	0.000838	0.001279	0.001156	0.001373	0.001682	0.001742	0.001858	0.002176
40	0.000842	0.001063	0.001370	0.005245	0.002040	0.002437	0.003131	0.003126	0.003429	0.003756
50	0.001048	0.001619	0.002114	0.003494	0.003204	0.003827	0.004332	0.005022	0.005519	0.005993
60	0.001534	0.004332	0.004876	0.003916	0.004987	0.005605	0.006438	0.007434	0.008400	0.008960
70	0.001863	0.005898	0.006358	0.005718	0.006580	0.007758	0.009054	0.010108	0.011408	0.014886
80	0.002422	0.004523	0.005620	0.010926	0.008984	0.010461	0.012126	0.013776	0.015518	0.017014
90	0.005917	0.008188	0.010097	0.012055	0.011614	0.013654	0.015949	0.017763	0.020112	0.022265
100	0.005739	0.010392	0.012892	0.018096	0.014827	0.017286	0.020281	0.022991	0.025601	0.028373
110	0.004691	0.009130	0.015453	0.020210	0.018105	0.021467	0.025437	0.028619	0.031888	0.035195
120	0.005692	0.016232	0.020516	0.030640	0.022164	0.026828	0.030838	0.034718	0.038822	0.043111
130	0.012599	0.017530	0.020887	0.029477	0.026763	0.032472	0.037338	0.042289	0.047684	0.052216
140	0.008477	0.016042	0.025943	0.055861	0.032151	0.038544	0.044296	0.049822	0.056022	0.062450
150	0.013472	0.017633	0.029822	0.034697	0.038066	0.044629	0.052762	0.058735	0.066657	0.073906
160	0.014302	0.021300	0.030483	0.052332	0.043761	0.052329	0.060917	0.068994	0.077945	0.086527
170	0.013237	0.030524	0.038668	0.070504	0.051089	0.060855	0.070801	0.081044	0.089894	0.099685
180	0.017998	0.032616	0.050871	0.068325	0.058751	0.069994	0.080820	0.093350	0.103749	0.114580
190	0.023796	0.033189	0.051708	0.073628	0.067672	0.080143	0.093285	0.104855	0.118742	0.131425
200	0.020772	0.039692	0.058258	0.075779	0.075733	0.090248	0.107372	0.120677	0.133725	0.149602
210	0.019593	0.068416	0.059262	0.080571	0.085046	0.100987	0.118121	0.135912	0.152350	0.168536
220	0.026233	0.053894	0.073711	0.087734	0.134845	0.113997	0.133829	0.151900	0.169351	0.188903
230	0.027739	0.064155	0.075604	0.112552	0.107333	0.129488	0.150124	0.171737	0.190225	0.210818
240	0.030339	0.057693	0.084395	0.158648	0.118972	0.141944	0.166102	0.191635	0.214121	0.234705
250	0.033650	0.061767	0.095732	0.130768	0.131541	0.158709	0.185871	0.208643	0.238092	0.261514
260	0.036477	0.080070	0.112781	0.153911	0.145075	0.174682	0.203579	0.234892	0.258636	0.287143
270	0.038519	0.080074	0.098774	0.190895	0.161085	0.193935	0.223454	0.255532	0.285377	0.318429
280	0.043390	0.101399	0.143783	0.209654	0.177220	0.210445	0.246439	0.280660	0.314902	0.350969
290	0.048064	0.097133	0.138221	0.203271	0.192396	0.231519	0.268924	0.306876	0.343082	0.380756
300	0.052742	0.102809	0.190588	0.250143	0.213651	0.258773	0.304333	0.335642	0.374581	0.416637
310	0.056741	0.120877	0.161461	0.326269	0.229433	0.274350	0.319822	0.364396	0.408270	0.459399
320	0.063709	0.120704	0.175196	0.234663	0.249257	0.300184	0.348434	0.396790	0.445062	0.494235
330	0.072617	0.121248	0.195537	0.277148	0.271739	0.324174	0.380089	0.431473	0.483636	0.539200
340	0.080087	0.151049	0.211929	0.417733	0.292275	0.349545	0.407377	0.463300	0.520098	0.579371
350	0.088815	0.163002	0.215299	0.303269	0.316738	0.377239	0.493848	0.501473	0.556951	0.622773
360	0.121682	0.178443	0.233385	0.340711	0.343496	0.412443	0.469629	0.536530	0.603658	0.680292
370	0.095010	0.179390	0.226017	0.415323	0.366822	0.437809	0.509793	0.579311	0.690061	0.727384
380	0.109326	0.187297	0.602865	0.733715	0.392348	0.473584	0.545526	0.619957	0.696965	0.776209
390	0.094624	0.193542	0.393363	0.393212	0.417942	0.499243	0.585491	0.666977	0.746685	0.854540
400	0.128150	0.210153	0.328280	0.362107	0.448535	0.536599	0.622739	0.705975	0.805347	0.889168
410	0.118477	0.313192	0.403975	0.385222	0.481531	0.575883	0.665852	0.764934	0.856831	0.954313
420	0.119490	0.243359	0.471504	0.415416	0.511394	0.610341	0.713017	0.811303	0.913200	1.040180
430	0.216023	0.258492	0.383481	0.438444	0.543219	0.649922	0.817031	0.858353	0.977056	1.073523
440	0.129375	0.356980	0.373630	0.466458	0.577357	0.693371	0.811759	0.918404	1.029359	1.153288
450	0.245635	0.278638	0.428013	0.494994	0.612448	0.738275	0.858465	0.978468	1.097704	1.213100
460	0.154914	0.309826	0.467924	0.524493	0.656227	0.782941	0.906531	1.041761	1.162553	1.291679
470	0.171397	0.327601	0.517692	0.556965	0.692601	0.825262	0.964715	1.099769	1.245046	1.391255
480	0.181127	0.440790	0.492952	0.583376	0.727613	0.873642	1.017083	1.159864	1.303092	1.446818
490	0.186859	0.371523	0.577365	0.625637	0.771695	0.924773	1.078667	1.232572	1.374079	1.531166
500	0.208475	0.383197	0.645760	0.653919	0.813237	0.991576	1.136199	1.289140	1.457433	1.613407

6.2 Bellman Ford

O algoritmo **Bellman Ford** tem complexidade igual a $O(|V| \times |E|)$ onde E é o conjunto das arestas do grafo e V é o conjunto de vértices do grafo. Esta complexidade, como na análise de complexidade do **Dijkstra** é dada pela análise do pseudo-código a seguir.

Algorithm 2 Bellman Ford

```
1:  procedure BELLMAN FORD
2:      for v in V do
3:          mincost[v] ← ∞
4:      mincost[s] ← 0
5:      i ← 0
6:      while i < |V| − 1 do
7:          for (begin, end, weight) in E do
8:              if cmin[end] > cmin[begin] +weight then
9:                  cmin[end] ← cmin[begin] +weight
10:         i ← i + 1
```

A primeira parte do algoritmo ao definir o custo inicial de cada vértice tem complexidade é $O(|V|)$. A segunda parte do algoritmo é uma repetição nos vértices em que cada repetição se analisa todas as arestas fazendo uma comparação, gerando uma complexidade de $O(|V| \times |E|)$.

6.2.1 Análise temporal

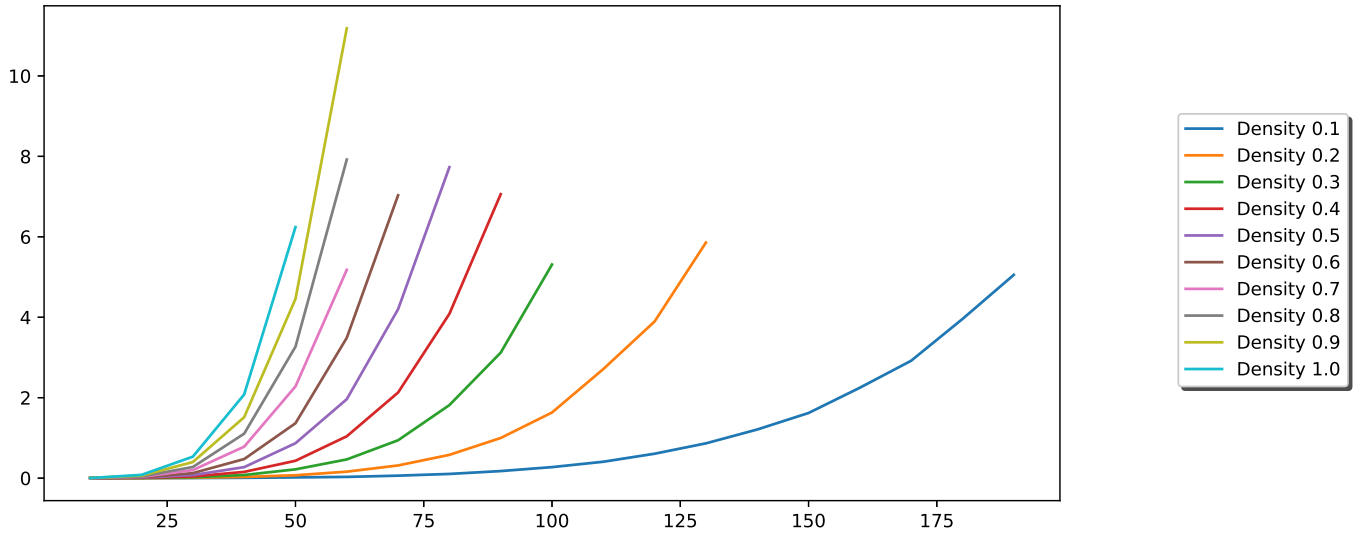

Figure 5: Resultados gráficos - Bellman Ford

Table 2: Tabela de resultados - Bellman Ford

	Densidades									
$\|V\|$	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
10	0.000032	0.000211	0.000390	0.000678	0.000744	0.001193	0.001426	0.002259	0.002785	0.002721
20	0.000038	0.001723	0.004367	0.007993	0.013966	0.020028	0.031883	0.042435	0.063972	0.081115
30	0.002746	0.009310	0.021312	0.042469	0.075867	0.124768	0.200179	0.277275	0.399771	0.534018
40	0.008116	0.030268	0.078171	0.155219	0.273695	0.474488	0.781884	1.105551	1.514605	2.080371
50	0.018372	0.071563	0.218512	0.431960	0.867906	1.364175	2.280097	3.267152	4.455144	6.240552
60	0.031044	0.160920	0.465061	1.040375	1.962813	3.488876	5.178582	7.920552	11.183659	-
70	0.061749	0.315465	0.941619	2.128651	4.199488	7.031801	-	-	-	-
80	0.104586	0.579022	1.816345	4.084790	7.730288	-	-	-	-	-
90	0.175703	0.997938	3.120205	7.059474	-	-	-	-	-	-
100	0.273982	1.631949	5.310783	-	-	-	-	-	-	-
110	0.407785	2.711858	-	-	-	-	-	-	-	-
120	0.607259	3.894926	-	-	-	-	-	-	-	-
130	0.865014	5.856808	-	-	-	-	-	-	-	-
140	1.209260	-	-	-	-	-	-	-	-	-
150	1.619682	-	-	-	-	-	-	-	-	-
160	2.247935	-	-	-	-	-	-	-	-	-
170	2.918961	-	-	-	-	-	-	-	-	-
180	3.956840	-	-	-	-	-	-	-	-	-
190	5.053336	-	-	-	-	-	-	-	-	-

6.3 Johnson

O algoritmo **Johnson** tem complexidade igual a $O((|V|^2 \times log(|V|)) + |V| \times |E|)$ onde E é o conjunto das arestas do grafo e V é o conjunto de vértices do grafo. Esta complexidade, como na análise de complexidade do **Dijkstra** é dada pela análise do pseudo-código a seguir.

Algorithm 3 Johnson

```
1:  procedure JOHNSON
2:      V ← V + {u}
3:      for v in V − {u} do
4:          E ← E + e(u, v, 0)
5:      c_min ← Run Bellman Ford from u
6:      if Bellman Ford returned a negative cycle then
7:          Return a negative cycle
8:      Create an auxiliar graph G' with the same vertices as G
9:      for (begin, end, weight) in E do
10:         E' ← (begin, end, e + c_min(v_i) − c_min(v_j))
11:     Create matrix (|V| × |V|) as b_min
12:     for v in V do
13:         Run Dijkstra for v
14:         Assign the Dijkstra result to the b_min vth line
```

A primeira parte do algoritmo tem complexidade $O(|V|)$. Uma rodada de Bellman Ford tem complexidade $O(|V| \times |E|)$. Uma rodada entre as arestas para atualizar os pesos, com peso $O(|E|)$. Ao rodar o Dijkstra para todos os vértice, a complexidade é $O(|V| \times (|V| log(|V|)))$. No total, a complexidade do algoritmo Johnson é $O((|V|^2 \times log(|V|)) + |V| \times |E|)$.

6.3.1 Análise temporal

Os resultados gráficos e as tabelas a seguir são relacionados aos mesmos dados, apenas apresentados de maneiras diferentes para melhor visualização. Os grafos usados para os testes são gerados aleatoriamente, com o cuidado de não gerar arcos duplicados.

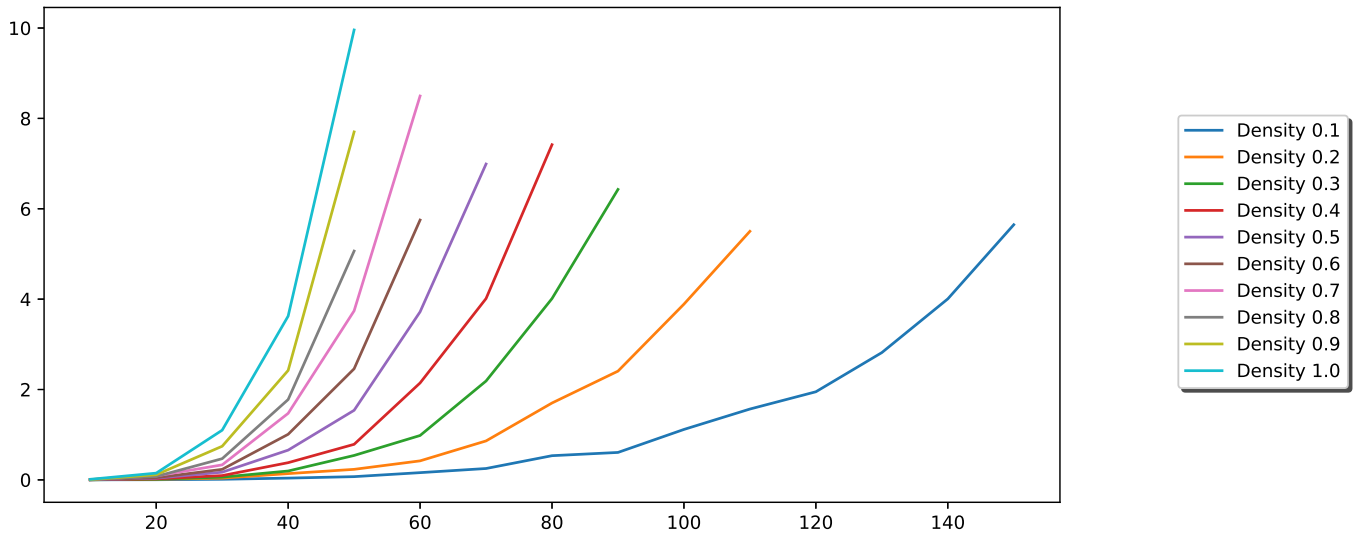

Figure 6: Resultados gráficos - Johnson

Table 3: Tabela de resultados - Johnson

	Densidades									
$\|V\|$	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
10	0.001790	0.004505	0.001591	0.002121	0.002626	0.003459	0.006348	0.005167	0.006383	0.012887
20	0.003353	0.013031	0.013497	0.022926	0.037208	0.049264	0.087088	0.075772	0.114250	0.149496
30	0.013247	0.038656	0.061083	0.094877	0.170244	0.232889	0.332165	0.469176	0.744202	1.099735
40	0.039279	0.138853	0.196255	0.380771	0.658486	1.008813	1.472355	1.778048	2.422430	3.624464
50	0.071520	0.233083	0.540626	0.787016	1.539662	2.458486	3.741148	5.063967	7.701031	9.957266
60	0.160112	0.420844	0.982190	2.144805	3.718867	5.750583	8.495827	-	-	-
70	0.251630	0.862130	2.187834	4.012559	6.990397	-	-	-	-	-
80	0.535363	1.701390	4.012637	7.417948	-	-	-	-	-	-
90	0.607247	2.406704	6.428017	-	-	-	-	-	-	-
100	1.116248	3.889186	-	-	-	-	-	-	-	-
110	1.567582	5.501277	-	-	-	-	-	-	-	-
120	1.950030	-	-	-	-	-	-	-	-	-
130	2.817203	-	-	-	-	-	-	-	-	-
140	4.005662	-	-	-	-	-	-	-	-	-
150	5.647139	-	-	-	-	-	-	-	-	-

[1]

[1]Os elementos que foram marcados como "-" são os elementos após testes que demoraram mais de 5 segundos para rodar.