UFC – UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

Curso de Engenharia da Computação

Disciplina: Algoritmos em Grafos

Pseudocódigos

Equipe:

Gerônimo Pereira Aguiar - 385145

Pedro Renoir Silveira Sampaio - 389113

Samuel Hericles Souza Silveira - 389118

Sobral - CE 2019.2

1 - Bellman-Ford

1.1 - Pseudocódigo

```
Bellman_Ford (G, weights, initial)
  | for every vertex ∈ V
  |   |  λ [vértice] ← ∞
  |   |  π [vértice] ← null
  | λ [initial] ← 0
  | for i from 1 to | V | -1
  |   |  for every edge = (u, v) ∈ A
  |   |   |  if λ [v] > λ [u] + weights(u, v)
  |   |   |   | λ [v] ← λ [u] + weights(u, v)
  |   |   |   | π [v] ← u
```

1.2 - Funcionamento

Como o algoritmo Dijkstra, o algoritmo Bellman-Ford usa a técnica de relaxamento, ou seja, realiza sucessivas aproximações das distâncias até finalmente chegar na solução. A principal diferença entre Dijkstra e Bellman-Ford é que no algoritmo de Dijkstra é utilizada uma fila de prioridades para selecionar os vértices a serem relaxados, enquanto o algoritmo de Bellman-Ford simplesmente relaxa todas as arestas. G é um grafo no formato G = (V,A), π[v] indica o predecessor de v, λ[v] é o custo de sair da origem e chegar até o vértice v e inicial é o vértice inicial. Após o término do algoritmo, para cada v pertencente aos vértices de G, π[v] → y representa uma aresta selecionada para a árvore geradora mínima (se y ≠ nulo) e, pensando em árvore, λ[v] representa a distância da raiz até o vértice v.

1.3 - Complexidade

O algoritmo de Bellman-Ford executa em tempo O(V x E) onde V é o número de vértices e E o número de arestas.

2 - Busca em profundidade

2.1 - Pseudocódigo

```
Busca-em-Profundidade (n, Adj, r)
Para  u ← 1  até  n  faça
 | cor[u] ← branco
cor[r] ← cinza
P ← Cria-Pilha (r)
Enquanto  P  não estiver vazia faça
 | u ← Cópia-Topo-da-Pilha (P)
 | v ← Próximo (Adj[u])
 | Se  v ≠ nil
 | |  Então  se cor[v] = branco
 | | |  Então  cor[v] ← cinza
 | | | |  Coloca-na-Pilha (v, P)
 | | | | |  Senão  cor[u] ← preto
 | | | | | | Tira-da-Pilha (P)
 | Retorna cor[1..n]
```

2.2 - Funcionamento

Formalmente, um algoritmo de busca em profundidade realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda cada vez mais, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (backtrack) e começa no próximo nó. Numa implementação não-recursiva, todos os nós expandidos recentemente são adicionados a uma pilha, para realizar a exploração. Quando ocorrem buscas em grafos muito grandes, que não podem ser armazenadas completamente na memória, a busca em profundidade não termina, em casos onde o comprimento de um caminho numa árvore de busca é infinito. O simples artifício de “lembrar quais nós já foram visitados” não funciona, porque pode não haver memória suficiente. Isso pode ser resolvido estabelecendo-se um limite de aumento na profundidade da árvore.

2.3 - Complexidade

Complexidade: O(m+n)

3 - Floyd-Warshall

3.1 - Pseudocódigo

```
Floyd-Warshall(Inteiro[1..n,1..n] grafo)
|  # Inicialização
|  VAR Inteiro[1..n,1..n] dist := grafo
|  VAR Inteiro[1..n,1..n] pred
|  Para i DE 1 A n
|  |  Para j DE 1 A n
|  |  |  Se dist[i,j] < Infinito ENTÃO
|  |  |  |  pred[i,j] := i
|  # Laço principal do algoritmo
|  Para k DE 1 A n
|  |  Para i DE 1 A n
|  |  |  Para j DE 1 A n
|  |  |  |  Se dist[i,j] > dist[i,k] + dist[k,j] ENTÃO
|  |  |  |  |  dist[i,j] = dist[i,k] + dist[k,j]
|  |  |  |  |  pred[i,j] = pred[k,j]
| Retorne dist
```

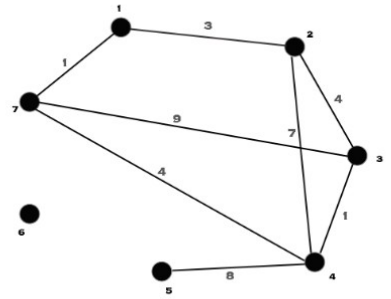
3.2 - Funcionamento

O algoritmo de Floyd-Warshall recebe como entrada uma matriz de adjacência que representa um grafo (V,E) orientado e valorado. O valor de um caminho entre dois vértices é a soma dos valores de todas as arestas ao longo desse caminho. As arestas E do grafo podem ter valores negativos, mas o grafo não pode conter nenhum ciclo de valor negativo. O algoritmo calcula, para cada par de vértices, o menor de todos os caminhos entre os vértices. Por exemplo, o caminho de menor custo.Assumindo que os vértices de um grafo orientado G são V= 1,2,3... n, considere um subconjunto 1,2,3, ... ,k. Para qualquer par de vértices(i,j) em V, considere todos os caminhos de i a j cujos vértices intermédios pertencem ao subconjunto 1,2,3, ... ,k, e p como o mais curto de todos eles. O algoritmo explora um relacionamento entre o caminho *p* é os caminhos mais curtos de i a j com todos os vértices intermediários em 1,2,3, ... ,k-1. O relacionamento depende de k ser ou não um vértice intermédio do caminho p.

3.3 - Complexidade

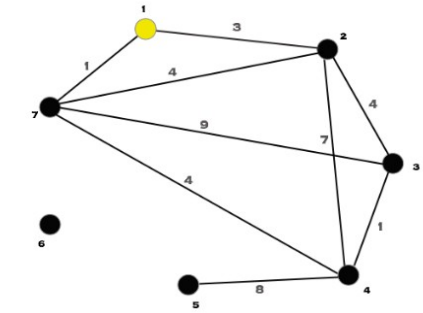
Complexidade de O(V^3).

3.4 - Exemplo de execução



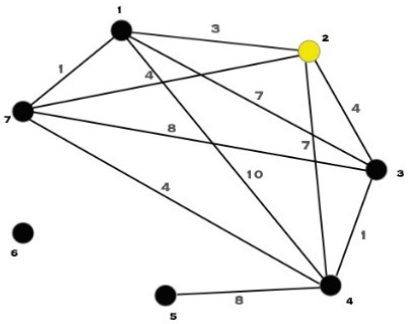
	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	∞	∞	∞	∞	1
V_2	3	0	4	7	∞	∞	∞
V_3	∞	4	0	1	∞	∞	9
V_4	∞	7	1	0	8	∞	4
V_5	∞	∞	∞	8	0	∞	∞
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	∞	9	4	∞	∞	0

Figura 1 - Estrutura do grafo inicial



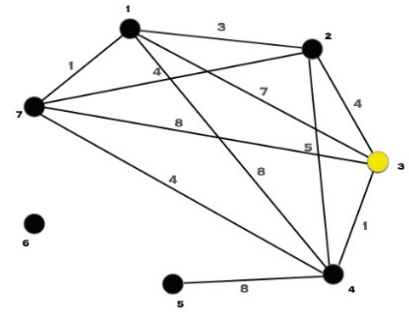
	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	∞	∞	∞	∞	1
V_2	3	0	4	7	∞	∞	4
V_3	∞	4	0	1	∞	∞	9
V_4	∞	7	1	0	8	∞	4
V_5	∞	∞	∞	8	0	∞	∞
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	4	9	4	∞	∞	0

Figura 2 - Começamos executando no vértice 1 é a única distância que alteramos é entre os vértices 2 e 7.



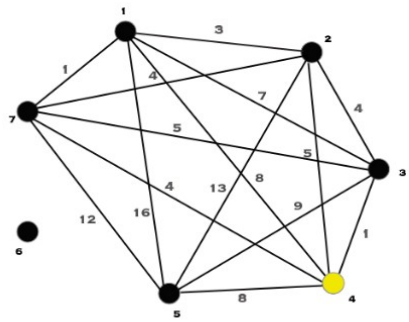
	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	7	10	∞	∞	1
V_2	3	0	4	7	∞	∞	4
V_3	7	4	0	1	∞	∞	8
V_4	10	7	1	0	8	∞	4
V_5	∞	∞	∞	8	0	∞	∞
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	4	8	4	∞	∞	0

Figura 3 - Agora vamos para o vértice 2 e alteramos as distâncias entre os pares (1,3), (1,4) e (3,7).



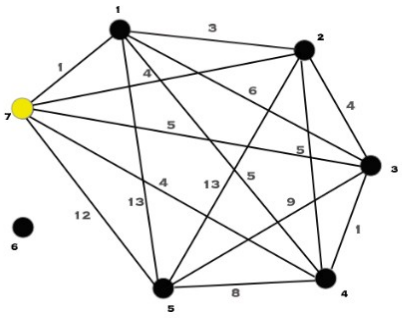
	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	7	8	∞	∞	1
V_2	3	0	4	5	∞	∞	4
V_3	7	4	0	1	∞	∞	8
V_4	8	5	1	0	8	∞	4
V_5	∞	∞	∞	8	0	∞	∞
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	4	8	4	∞	∞	0

Figura 4 - Vamos para o vértice 3, onde atualizamos as distâncias entre os pares (1,4) e (2,4).



	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	7	8	16	∞	1
V_2	3	0	4	5	13	∞	4
V_3	7	4	0	1	9	∞	5
V_4	8	5	1	0	8	∞	4
V_5	16	13	9	8	0	∞	12
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	4	5	4	12	∞	0

Figura 5 - Para o vértice 4, onde atualizamos as distâncias dos pares (1,5), (2,5), (3,5), (3,7) e (5,7).



	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	0	3	6	5	13	∞	1
V_2	3	0	4	5	13	∞	4
V_3	6	4	0	1	9	∞	5
V_4	5	5	1	0	8	∞	4
V_5	13	13	9	8	0	∞	12
V_6	∞	∞	∞	∞	∞	0	∞
V_7	1	4	5	4	12	∞	0

Figura 6 - Quando executamos o processo nos vértices 5 e 6, nada é alterado. Quando executamos no vértice 7, alteramos as distâncias dos pares (1,3), (1,4), (1,5).

4 - Shortest-Path

4.1 - Pseudocódigo

```
Shortest-Path-Main(W)
L = W
Para i = 2 até n
| L = Shortest-Path(L,W)
Retorne L

Shortest-Path (L(m)□, W)
Cria L(m+1) = inf
Para todo i ∈ V
| Para todo j ∈ V
|   | c = 0
|   | Para todo k ∈V
|   |   | c += L(m)[i,k] + W[k,i]
|   | L(m+1)[i,j] = c
| Retorne L(m+1)
```

4.2 - Funcionamento

Defina todas as distâncias dos vértices = infinito, exceto o vértice da fonte, defina a distância da fonte = 0. Empurre o vértice de origem em uma fila de prioridade mínima no formato (distância, vértice), pois a comparação na fila de prioridade mínima será de acordo com as distâncias dos vértices. Define o vértice com a distância mínima da fila de prioridade (primeiro o vértice estourado = origem). Atualize as distâncias dos vértices conectados ao vértice estourado no caso de "distância atual do vértice + peso da borda < distância do próximo vértice" e empurre o vértice com a nova distância da fila de prioridade. Se o vértice popped já tiver sido visitado antes, continue sem usá-lo. Aplique o mesmo algoritmo novamente até que a fila de prioridade esteja vazia.

4.3 - Complexidade

Complexidade: $O(n^2)$

4.4 - Exemplo de execução

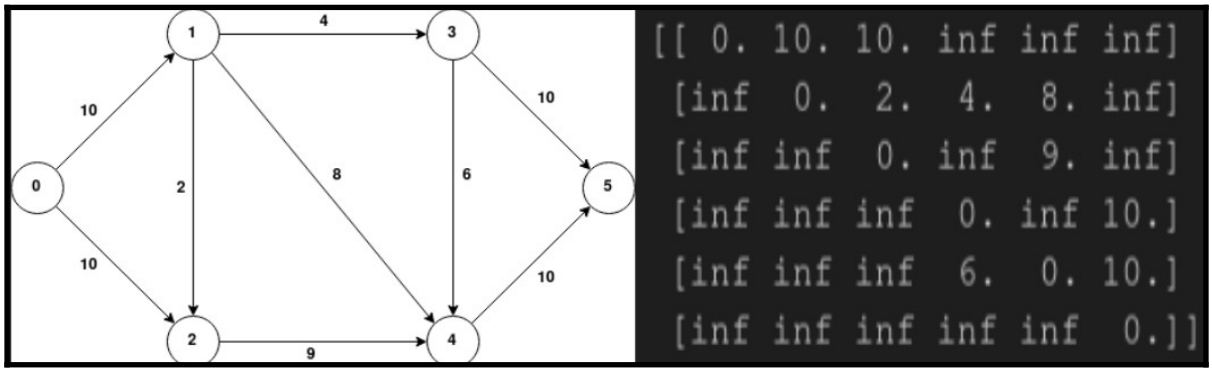


Figura 7 - Entrada da matriz de pesos do grafo direcionado w. Neste momento é definido também $L^0 = w$.

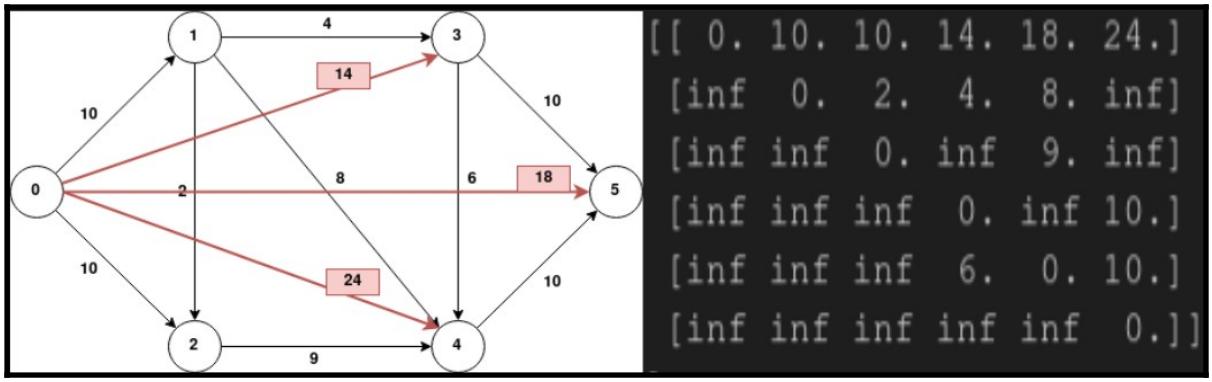


Figura 8 - Primeira iteração, entrada na função Shortest-Path para calcular $L^1 = c = L^0[0,k] + w[k,j]$.

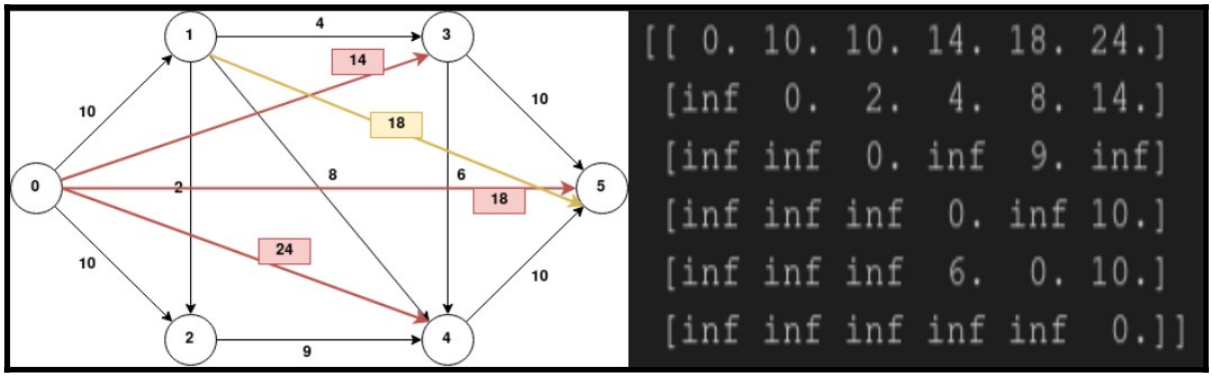


Figura 9 - Segunda iteração, entrada na função Shortest-Path para calcular $L^2 = c = L^1[1,k] + w[k,j]$.

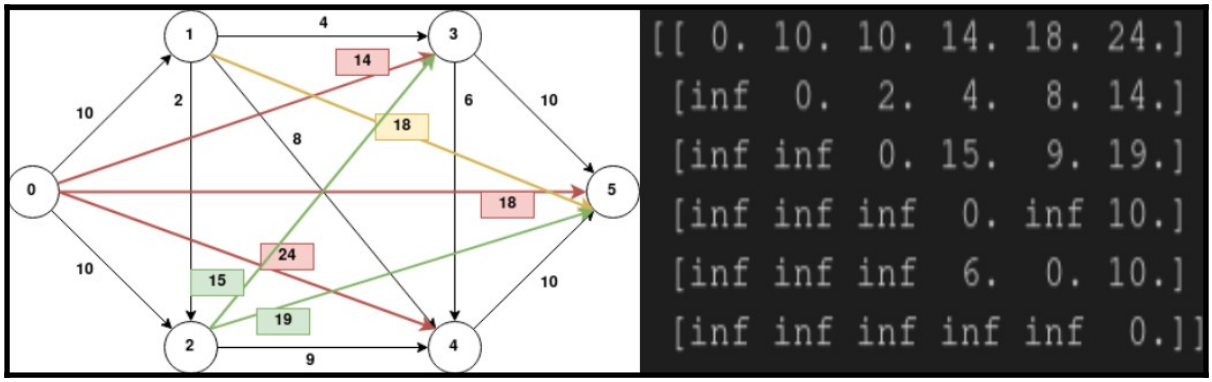


Figura 10 - Última iteração, entrada na função Shortest-Path para calcular $L^3 = c = L^2[2,k] + w[k,j]$. Aqui o algoritmo pára, pois há caminho mínimo de todos os vértices para todos os outros.

5 - Extend-Shortest-Path

5.1 - Pseudocódigo

```
MenorRec(G,w)
|  Inicializa L
|  Para cada i,j ϵV
|  |   Lij = calc(G,w,i,j,n)
| Retorna L

Calc(G,w,i,j,m)
| Se i = j
| |  Retorna 0
| Se m = 1
| |  Retorna w(i,j)
| c = inf
| Para cada k ϵV
| | Se c > Calc(G,w,i,k,m-1) + w(k,j)
| |  | c = Calc(G,w,i,k,m-1) + w(k,j)
| Retorne c
```

5.2 - Funcionamento

Algoritmo que calcula o menor caminho de todos os pares de vértices, calcula os pesos de caminhos curtos estendendo os caminhos mais curtos aresta por aresta.

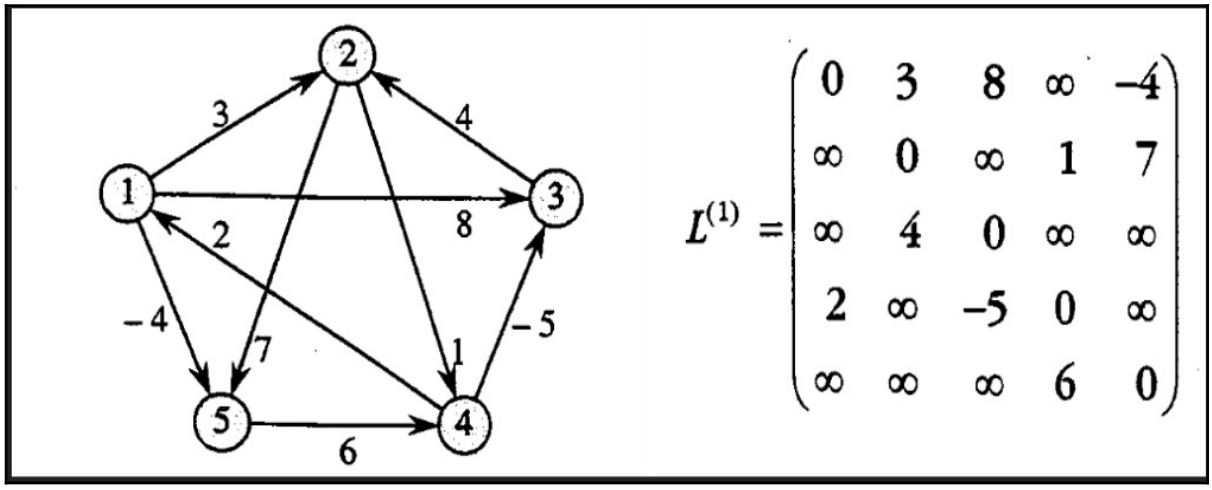
Tomando como nossa entrada W = (w_{ij}), calculamos agora uma séries de matrizes L^1, L^2, …, L^n-1, onde, para m = 1, 2, ..., n - 1 temos L^(m). A matriz final L^(n-1) contém os pesos reais de caminhos mais curtos. Observe que, consideramos-se $l_{ij}^1 = w_{ij}$ para todos os vértices i,j ϵ V, temos $L^1 = W$.

O núcleo do algoritmo é o procedimento a seguir que, dadas às matriz L^(m-1) e W, retorna a matriz L^(m). Isto é, ele estende os caminhos mais curtos calculados até agora por mais uma aresta.

5.3 - Complexidade

Execução: $O(n^3)$

5.4 - Exemplo de execução



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Figura 11 - Primeira iteração, grafo problema com suas arestas e a matriz L da primeira iteração antes entrar no algoritmo.

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

Figura 12 - Segunda iteração, matriz $L^2 = W^2$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figura 13 - Terceira iteração, matriz $L^3 = W^3$

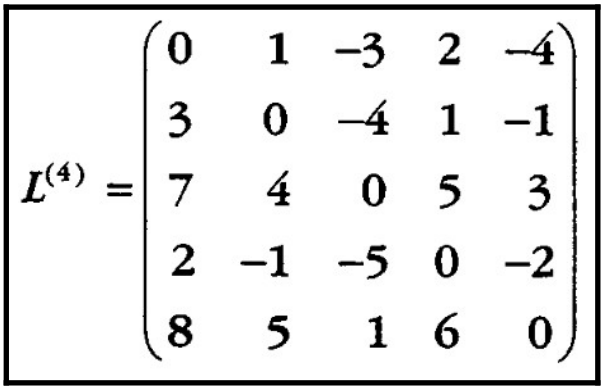


Figura 14 - Quarta e última iteração, matriz $L^4 = W^4$. Não é possível esboçar o grafo devido ser uma a restrições de dimensionalidade, ou seja, alguma aresta vai passar por cima da outra.

6 - Ford-Fulkerson

6.1 - Pseudocódigo

Ford-Fulkerson(G, s, t)
For cada aresta (u,v) <- E[G]
| **do** f[u,v] <- 0
| | f[v,u] <- 0
While existir um caminho p de s até t na rede residual G_f
| **do** $c_f(p)$ <- min{$c_f(u,v)$: (u,v) está em p}
| | **for** cada aresta (u,v) em p
| | | **do** f[u,v] <- f[u,v] + $c_f(p)$
| | | | f[v,u] <- f[u,v]

6.2 - Funcionamento

Em cada iteração do método de Ford-Fulkerson, encontramos algum caminhos aumentante *p* e aumentamos o fluxo *f* em cada aresta de *p* pela capacidade residual $c_f(p)$. A implementação do método a seguir calcula o fluxo máximo em um grafo *G = (V,E)* atualizando o fluxo *f[u,v]* entre cada par *u,v* de vértices que estão conectados por uma aresta. Se *u* e *v* não estão conectados por uma aresta em um ou outro sentido, supomos implicitamente que *f[u,v] = 0*. As capacidades *c(u,v)* são consideradas dadas juntamente com o grafo, e *c(u,v) = 0* se $(u,v) \notin E$. A capacidade residual $c_f(u,v)$ é calculada de acordo com a fórmula

$$(c_f(u,v) = c(u,v) - f(u,v).$$

A expressão $c_f(p)$ no código é na realidade apenas uma variável temporária que armazena a capacidade residual do caminho *p*. O algoritmo de Ford-Fulkerson, também conhecido como algoritmo dos pseudo caminhos aumentadores, resolve o problema do fluxo máximo. Cada iteração começa com um fluxo f que respeita às capacidades dos arcos. A primeira iteração começa com o fluxo nulo. O processo iterativo consiste no seguinte: enquanto existe pseudo caminho aumentador, encontra um pseudo caminho aumentador P, calcule a capacidade residual δ de P, envie δ unidades de fluxo ao longo de P e atualiza f .

6.3 - Complexidade

A complexidade do algoritmo é O(mf), em que m representa o número de arestas presentes no grafo G e f o fluxo máximo encontrado.

6.4 - Exemplo de execução

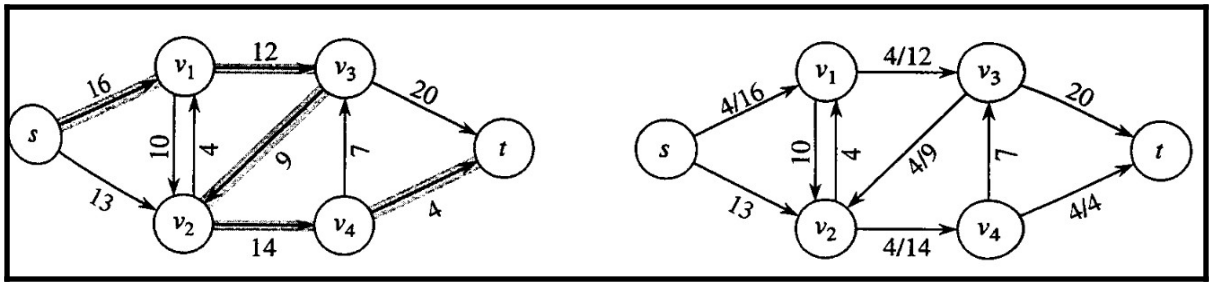


Figura 15 - Primeira iteração, a rede de entrada,o algoritmo encontra um caminho com maior peso (s -> v_1 -> v_3 -> v_2 -> v_4 -> t), veja na imagem a direita. Após isso, adiciona o 4 (menor peso) entre as arestas desse caminho na matriz fluxo e insere 4 na variável do fluxo máximo.

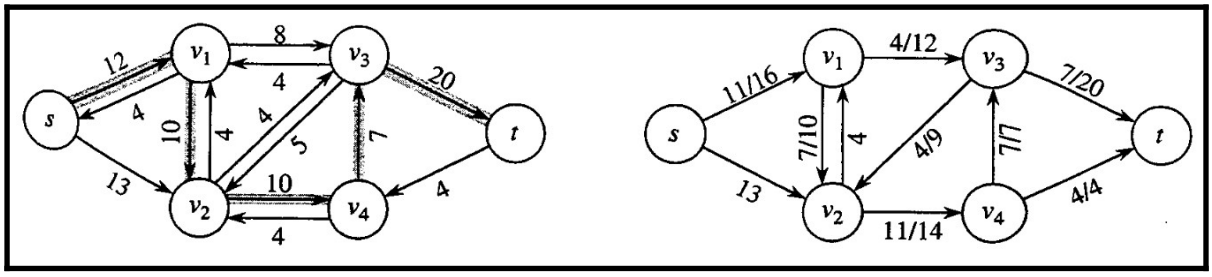


Figura 16 - Segunda iteração, o algoritmo encontra outro caminho com maior peso (s -> v_1 -> v_2 -> v_4 -> v_3 -> t), veja na imagem a direita. Após isso, adiciona o 7 (menor peso) entre as arestas desse caminho na matriz fluxo e insere 7 na variável do fluxo máximo.

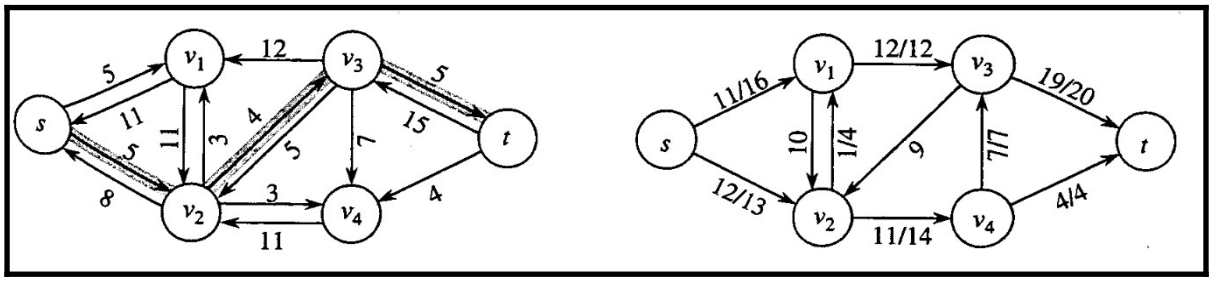


Figura 17 - Terceira iteração, o algoritmo encontra outro caminho com maior peso (s -> v_2 -> v_3 -> t), veja na imagem a direita. Após isso, adiciona o 5 (menor peso) entre as arestas desse caminho na matriz fluxo e insere 5 na variável do fluxo máximo.

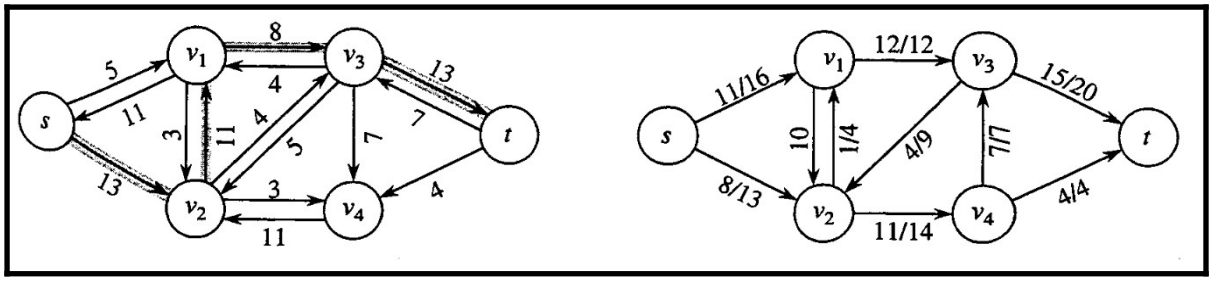


Figura 18 - Última iteração, o algoritmo encontra outro caminho com maior peso (s -> v_2 -> v_1 -> v_3 -> t), veja na imagem a direita. Após isso, adiciona o 8 (menor peso) entre as arestas desse caminho na matriz fluxo e insere 8 na variável do fluxo máximo.

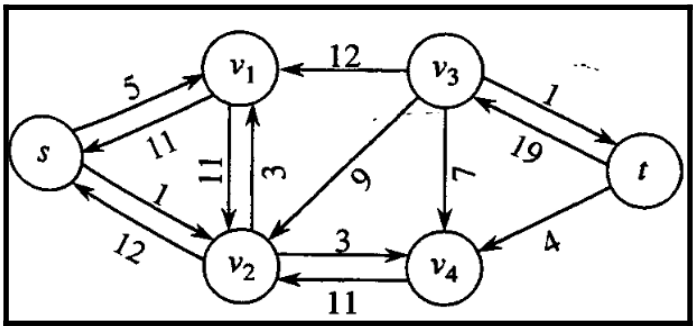


Figura 19 - A rede residual no último teste do loop while.

7 - Push–relabel

7.1 - Pseudocódigo

```
Push(f, h, v, w)
Enquanto e(v) > 0, h(w) < h(v) e (v, w) ∈ Gf
 Se (v, w) é uma aresta direta de Gf então
 | e ← (v, w)
 | ε ← min(e(v), u(e) – f(e))
 | f(e) ← f(e) + ε
 Se (v, w) é uma aresta inversa de Gf então
 | e ← (w, v)
 | ε ← min(e(v), f(e))
 | f(e) ← f(e) – ε
 Retorna (f, h)

Relabel(f, h, v)
Enquanto e(v) > 0 e h(w) ≥ h(v) para toda aresta (v, w) ∈ Gf
 | h(v) ← h(v) + 1
 Retorna(f, h)

Generic-Push-Relabel(G, s, t)
 Para cada v em V faça h(v) ← 0
 | h(s) ← |V |
 Para cada e ← (v, w) em Gf faça
 | Se v = s então f(e) ← u(e)
 | Senão f(e) ← 0
 Enquanto existe vértice v != t com e(v) > 0 faça
 | Seja v um vértice com excesso positivo
 | Se existe aresta (v, w) com h(w) < h(v) então push(f, h, v, w)
 | Senão relabel(f, h, v)
 Retorna f
```

7.2 - Funcionamento

Para calcular o fluxo descrito em 2, o algoritmo utiliza duas funções principais: push e relabel. Tais funções mantém os seguintes dados:

Fluxo de u para v f(u,v). A capacidade disponível é dada por c(u,v) - f(u,v). h(u). Nós chamamos a função push de u para v apenas se h(u) > h(v). Para todos u, h(u) é um inteiro positivo. e(u). Soma do fluxo de e para u.

Após cada passo do algoritmo, teremos um pré-fluxo que deve satisfazer às

seguintes condições:

$f(u,v) < c(u,v)$. O fluxo entre u e v não excede a capacidade.

$f(u,v) = -f(v,u)$. Mantemos o fluxo da rede.

$\Sigma f(v,u) = express(u) >= 0$ para todos os nós $u \neq s$. Somente a fonte pode produzir fluxo.

7.3 - Complexidade

O algoritmo push-relabel é um dos algoritmos de fluxo máximo mais eficientes. O algoritmo genérico tem uma complexidade de tempo $O(V^2 E)$.

7.4 - Exemplo de execução

A figura 20 é o exemplo de grafo para demonstrar o funcionamento do algoritmo.

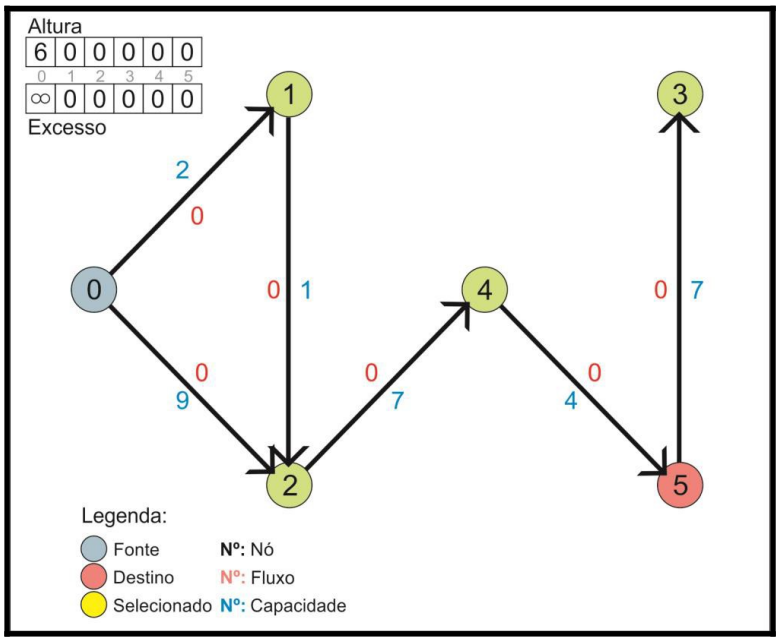


Figura 20 - Grafo em estado inicial.

Inicialmente, o algoritmo iguala a altura do vértice inicial com o número de vértices presente no grafo e inicia todos os outros vértices com altura zero.

A partir deste ponto, chamamos a função push do nó inicial para todos seus vizinhos enviando todo o fluxo permitido pela capacidade de suas arestas. A figura 21 demonstra a situação do fluxo após o push inicial, onde podemos observar a mudança de f(0,1) e f(0,2).

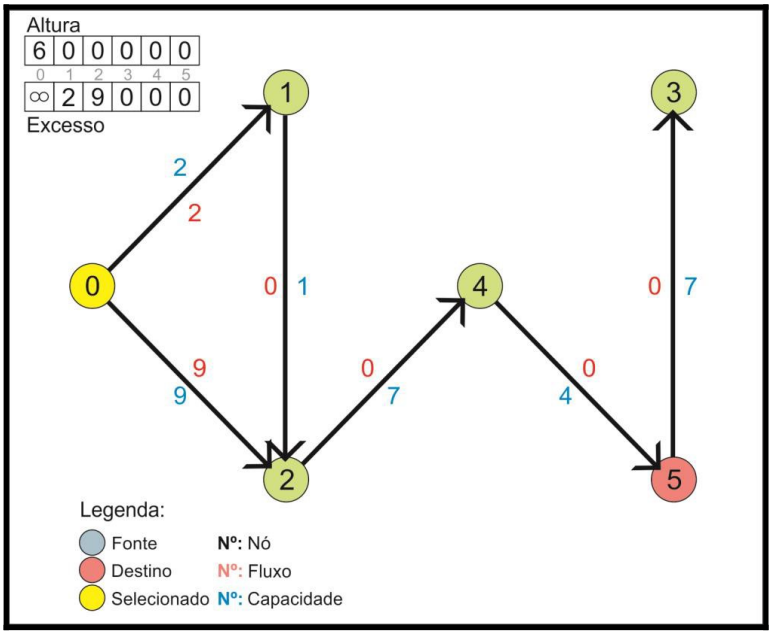


Figura 21 - Grafo após push da fonte para seus vizinhos.

Após o push inicial partindo do nó fonte, observamos além da mudança no fluxo, também o vetor que controla o excesso presente no fluxo atual tem seus valores no nó 1 e 2 atualizados. Como todos os nós fora o fonte estão com sua altura zero e podemos apenas enviar fluxo de um nó mais alto para um nó mais baixo, será necessário chamar a função relabel para aumentar minimamente a altura tornando possível o envio de fluxo do nó corrente.

Após o ajuste na altura, iremos enviar todo o fluxo possível do nó corrente para seus vizinhos através de função push. Caso ainda haja excesso de fluxo no nó corrente, enviaremos fluxo de volta para a fonte. As figuras a seguir demonstram a execução completa do algoritmo para o grafo exemplo.

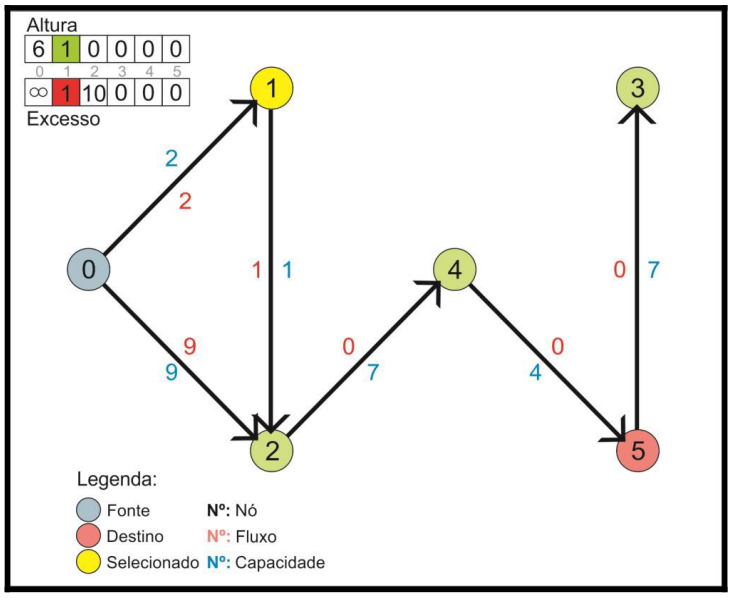


Figura 22 - Push a partir do nó um.

Podemos observar na figura 22 que só é possível realizar o push(1,2) pois a altura do nó 1 foi aumentada na função relabel. Mesmo após o push de 1 para seus vizinhos ainda há excesso no nó selecionado, então será necessário devolver fluxo para a fonte. A figura 23 demonstra que foi necessário tornar a altura do nó 1 maior que a altura do nó fonte, utilizando a função relabel, para podermos devolver fluxo. Isso se deve ao princípio citado anteriormente nessa sessão, onde para poder enviar fluxo de um nó a para um nó b, height(a) > height(b). Retornando o fluxo para a fonte, o excesso do nó 1 é zerado e podemos prosseguir para o próximo nó.

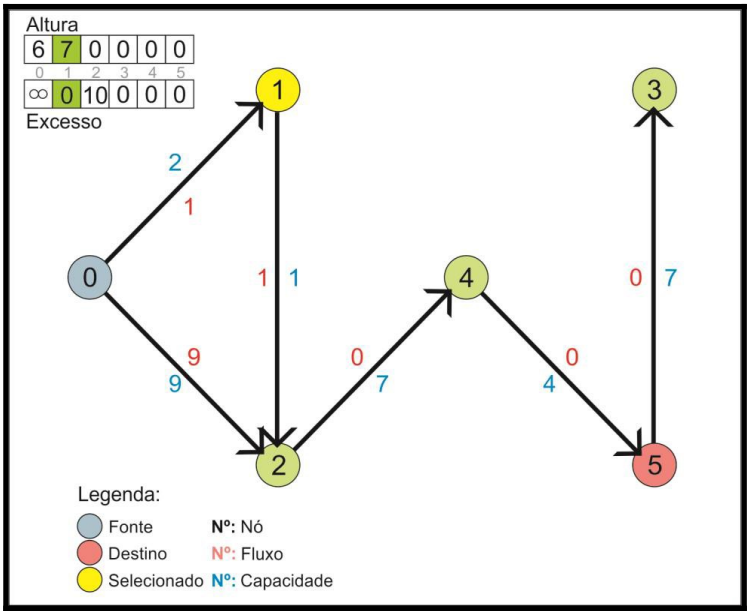


Figura 23 - Retorno de fluxo para fonte.

Como conseguimos zerar o excesso do nó 1, o algoritmo pula para o próximo nó e repete os passos descritos anteriormente.

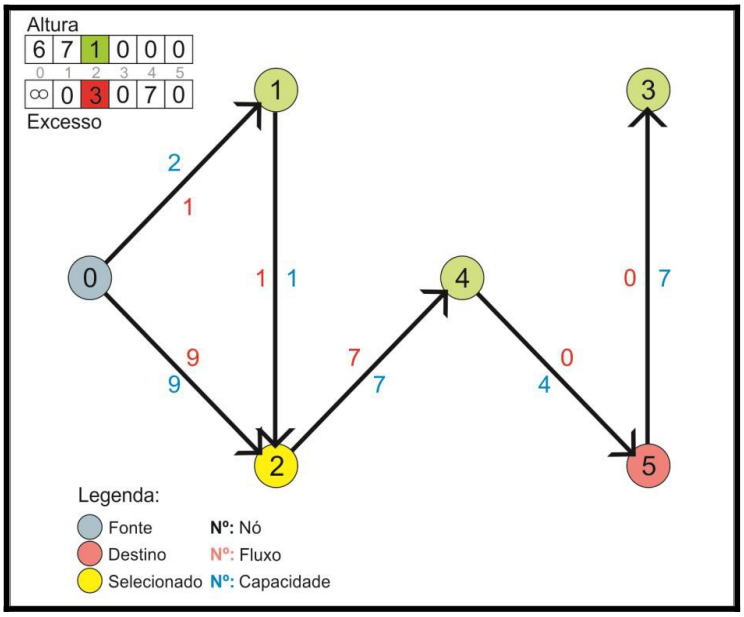


Figura 24 - Relabel(2) e push(2,4)

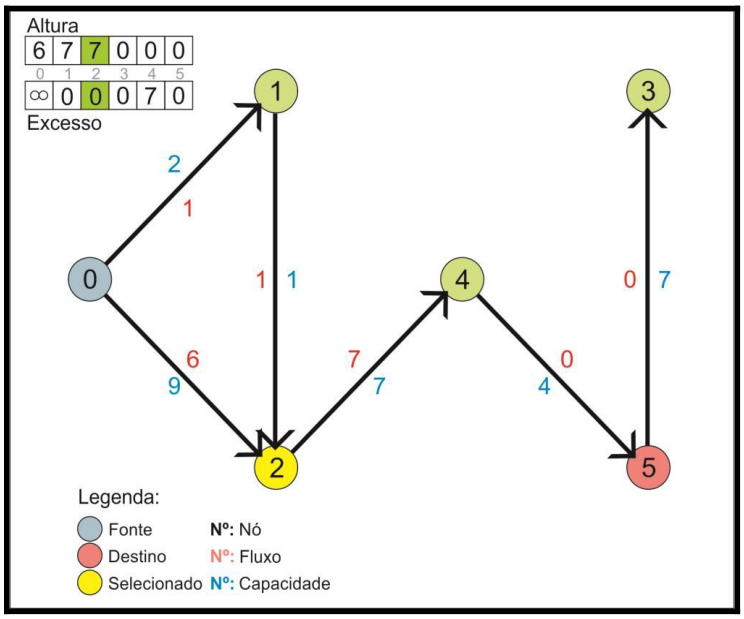


Figura 25 - Retorno de fluxo do nó 2 para fonte.

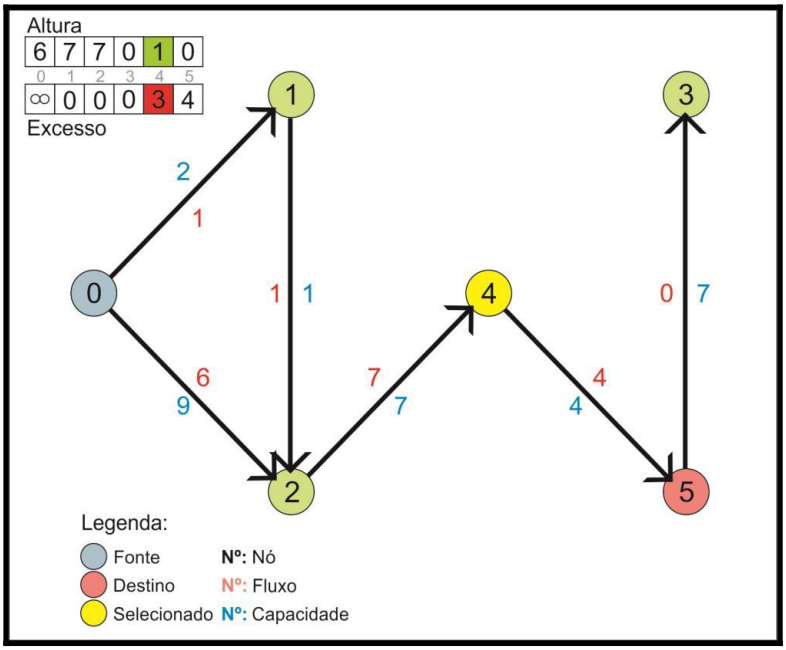


Figura 27 - Relabel(4) e push(4,5).

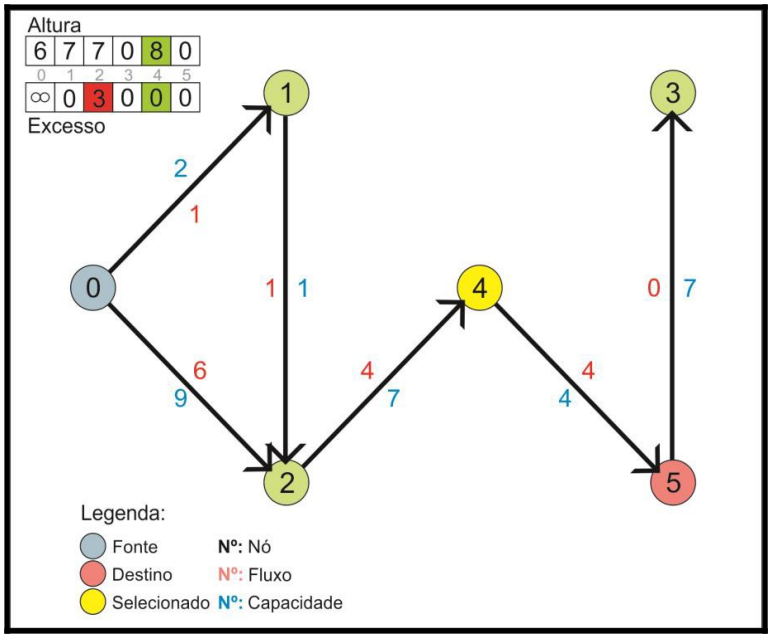


Figura 28 - Retorno de fluxo do nó 4 para o nó 2.

Na figura 28, que o retorno de fluxo não é feito diretamente para a fonte e sim para o nó 2, o que aumentará novamente o excesso do nó 2, fazendo necessário também o retorno do excesso dele.

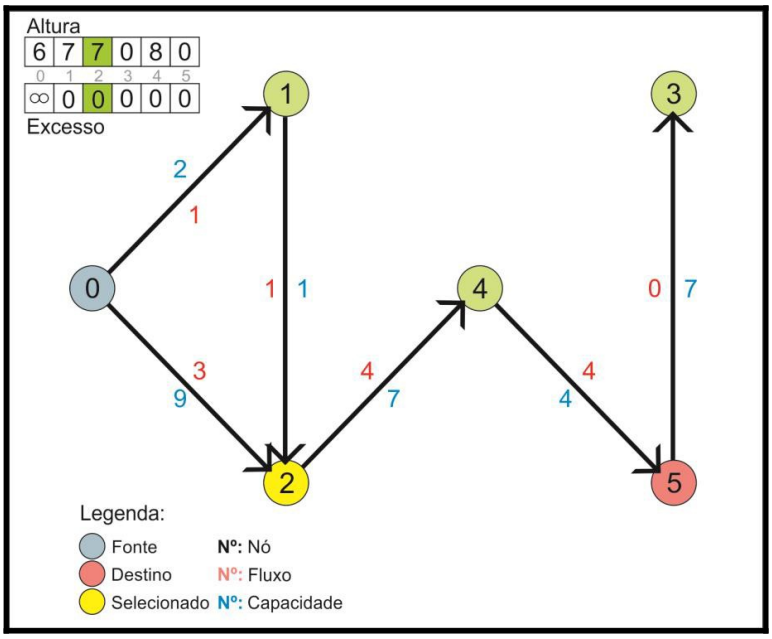


Figura 29 - Retorno de fluxo do nó 2 para fonte.

Na figura 29 podemos observar que não há nenhum nó com excesso de fluxo, então o algoritmo retorna a soma dos fluxos partindo da fonte, ou seja, o fluxo máximo que pode ser enviado do nó fonte para o nó destino.