Capítulo X

Fluxo Máximo entre Dois Vértices

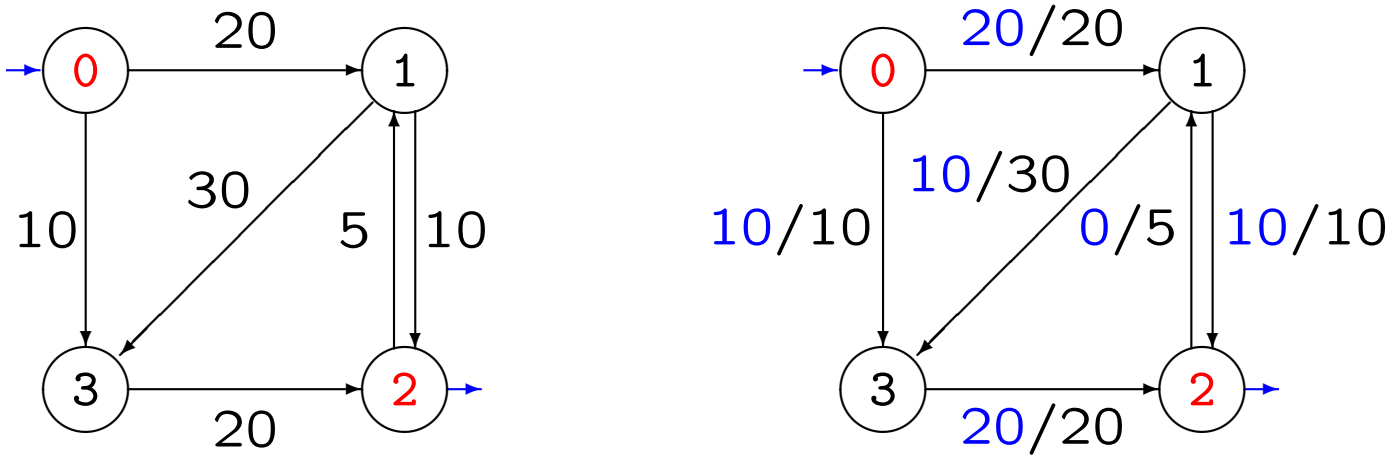
(num grafo orientado e pesado)

—

Algoritmo de Edmonds-Karp

Problema

Dado um grafo **orientado** e **pesado** (com pesos não negativos) e dois vértices f e d, como encontrar um **fluxo máximo de f para d**?

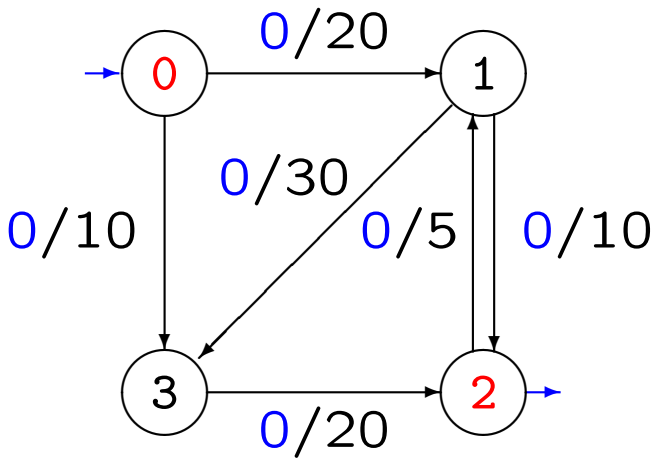


Valor de um Fluxo Máximo de 0 para 2: 30

Assume-se que qualquer vértice pertence a um caminho de f para d.

Portanto, o grafo é **simplesmente conexo** e $|A| \geq |V| - 1$.

Ideia Geral (de 0 para 2)



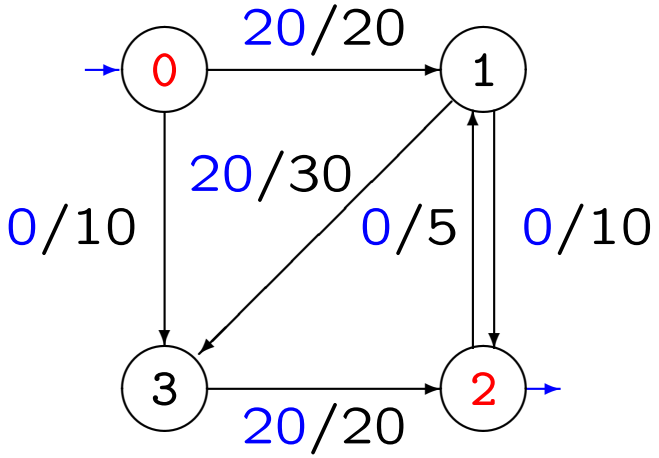
Descobrir um caminho para drenar:

0, 1, 3, 2.

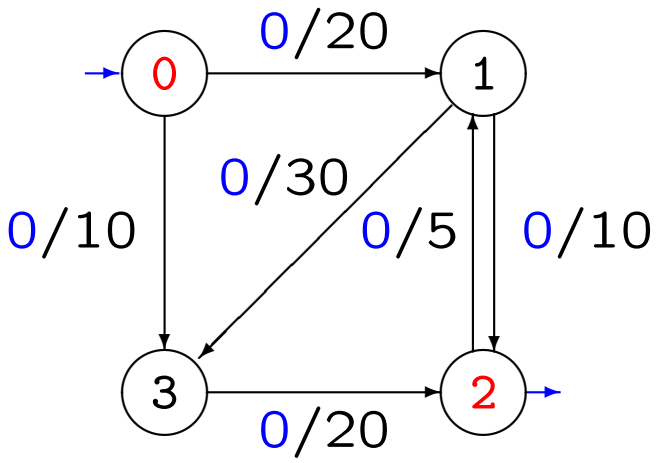
Incremento máximo permitido:

20.

Atualizar o fluxo.



Ideia Geral (de 0 para 2)



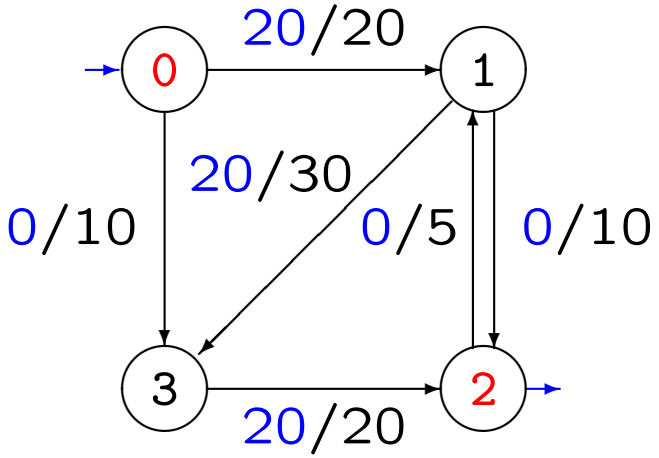
Descobrir um caminho para drenar:

0, 1, 3, 2.

Incremento máximo permitido:

20.

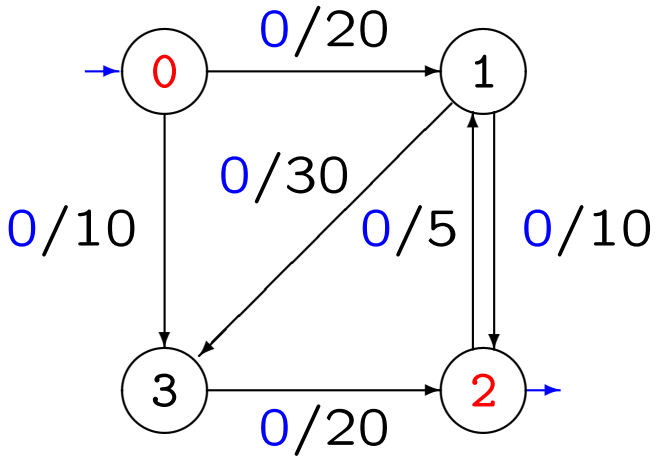
Atualizar o fluxo.



PROBLEMA

Já não há caminho para drenar e o valor de um fluxo máximo é > 20.

Ideia Geral (de 0 para 2)



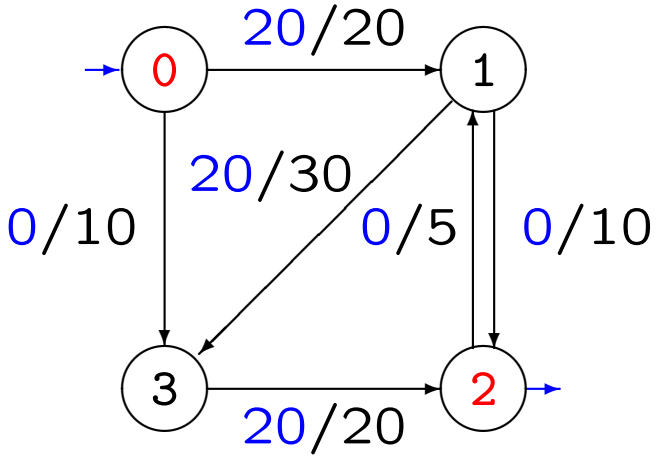
Descobrir um caminho para drenar:

0, 1, 3, 2.

Incremento máximo permitido:

20.

Atualizar o fluxo.



PROBLEMA

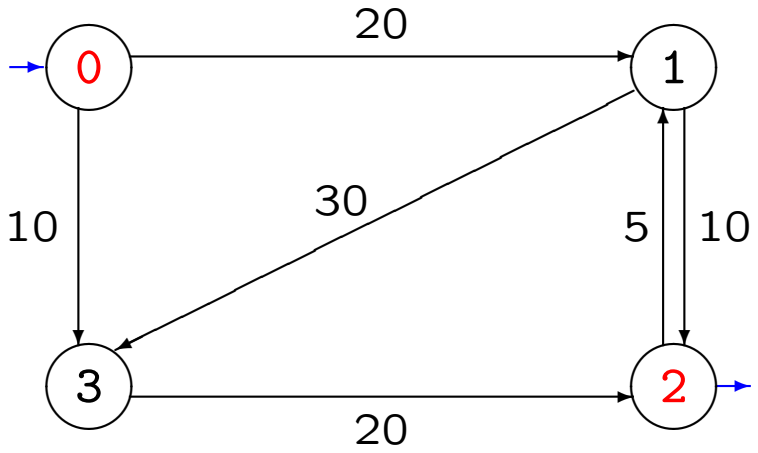
Já não há caminho para drenar e o valor de um fluxo máximo é > 20.

SOLUÇÃO

"Cancelar 10 unidades de fluxo do arco $(1, 3)$, desviando-as para $(1, 2)$"

Método de Ford-Fulkerson [1962]

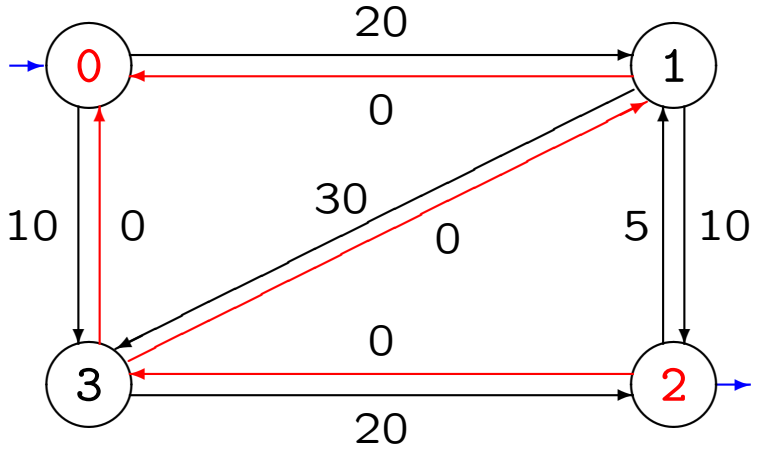
Grafo Original



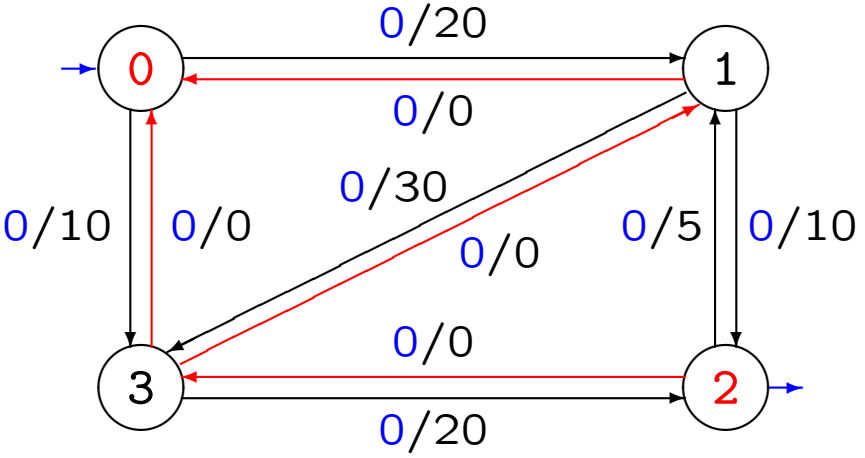
Trabalhar na Rede de Fluxos

(todos os arcos têm inverso)

Rede de Fluxos

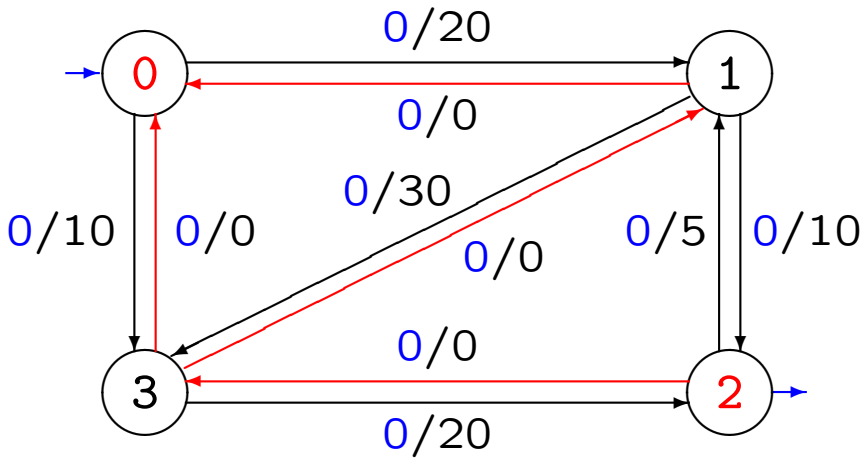


Fluxo Inicial

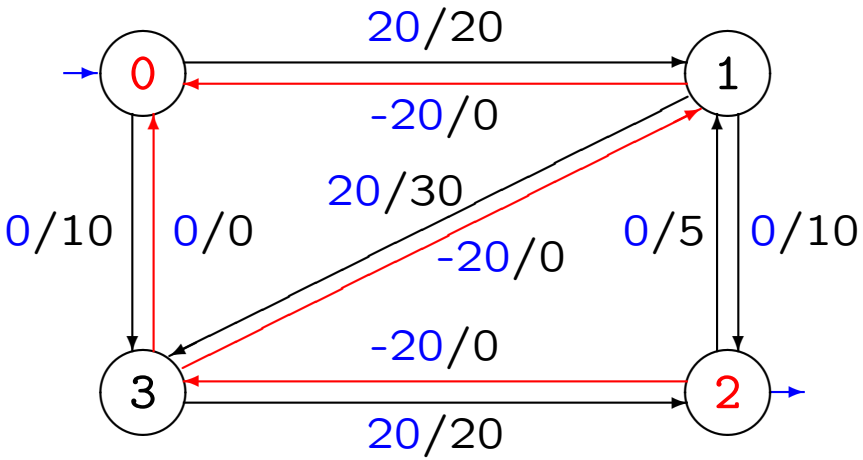


Método de Ford-Fulkerson (1)

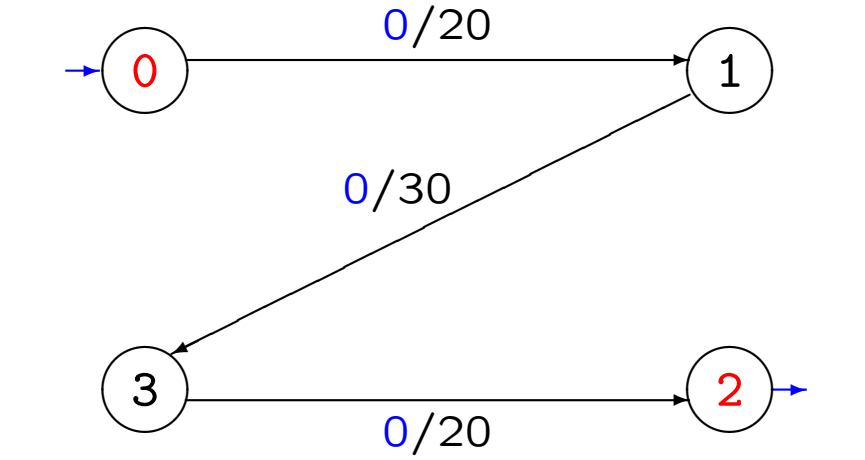
Estado Inicial

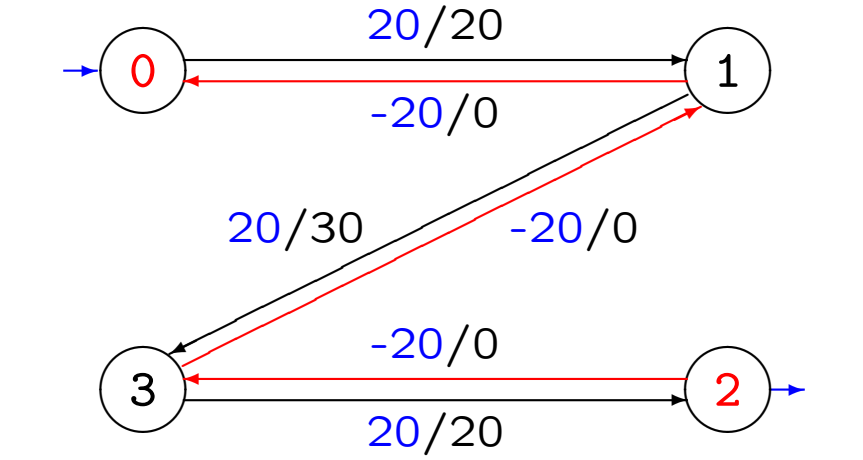


Estado Seguinte



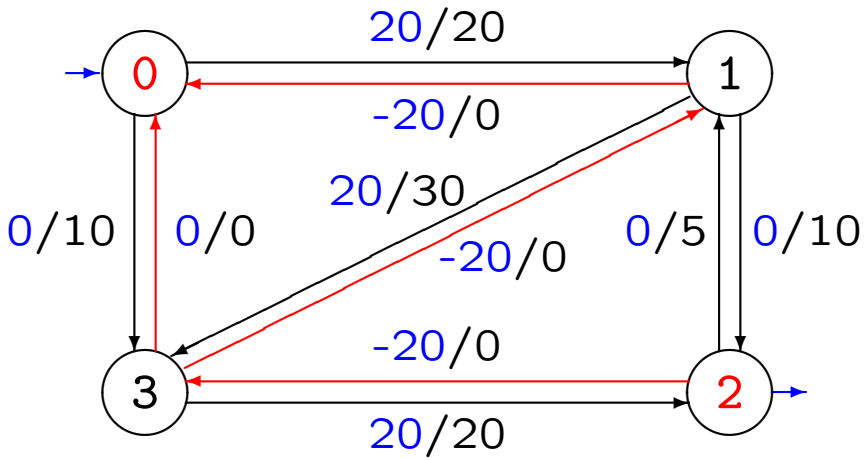
Caminho: 0, 1, 3, 2 **Incremento:** 20



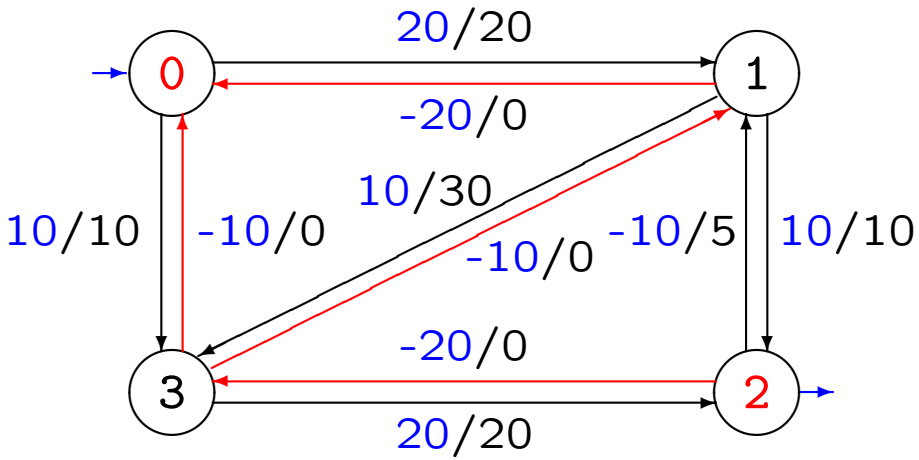


Método de Ford-Fulkerson (2)

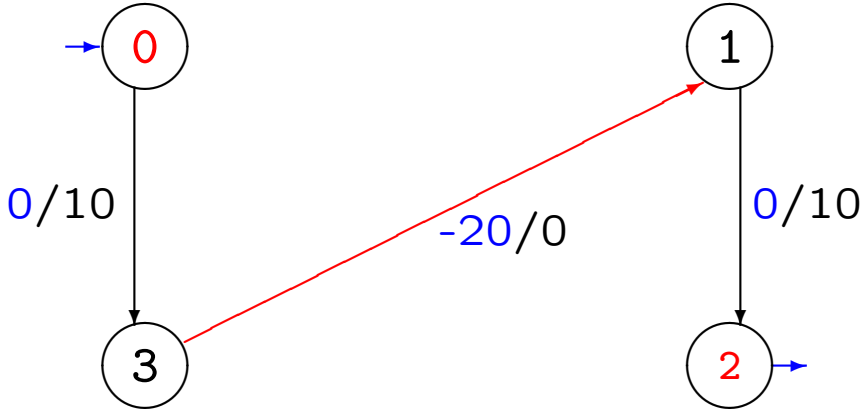
Estado Corrente

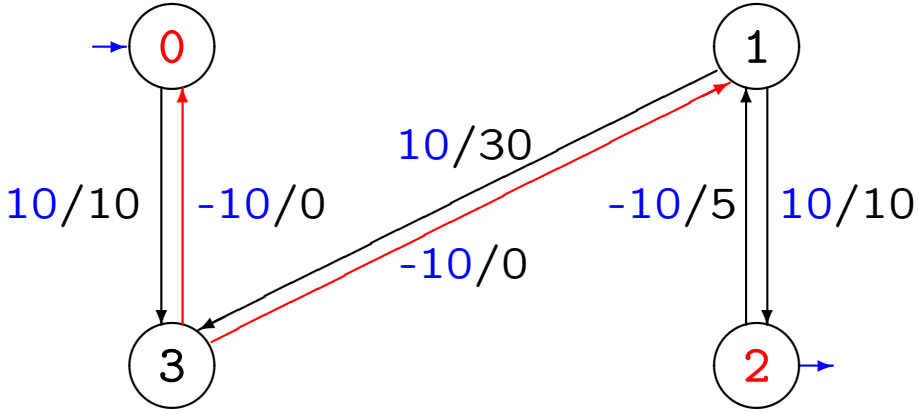


Estado Seguinte



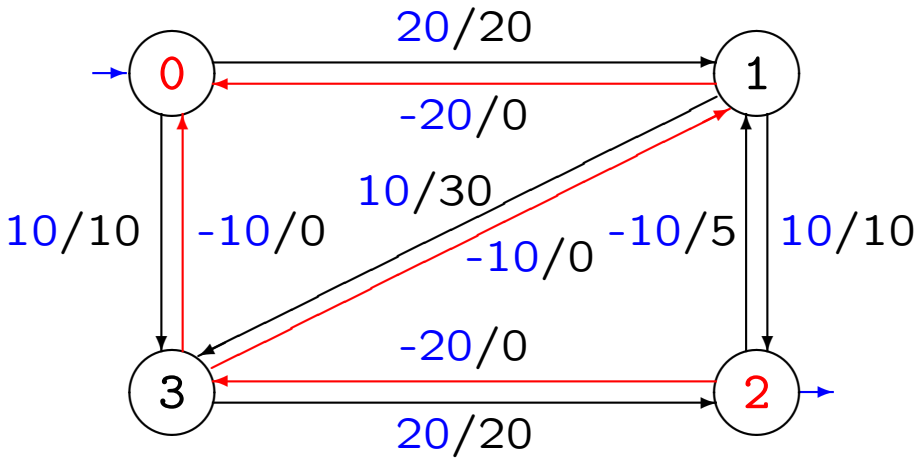
Caminho: 0, 3, 1, 2 **Incremento:** 10



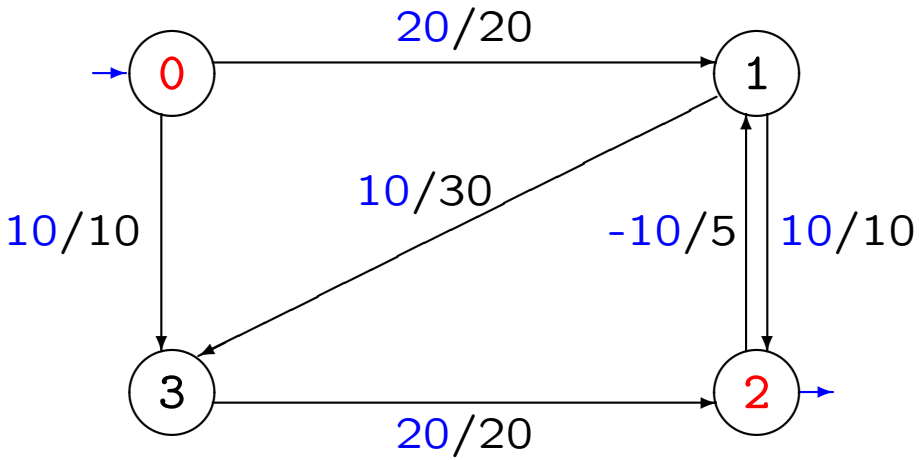


Método de Ford-Fulkerson (3)

Estado Corrente



Resultado

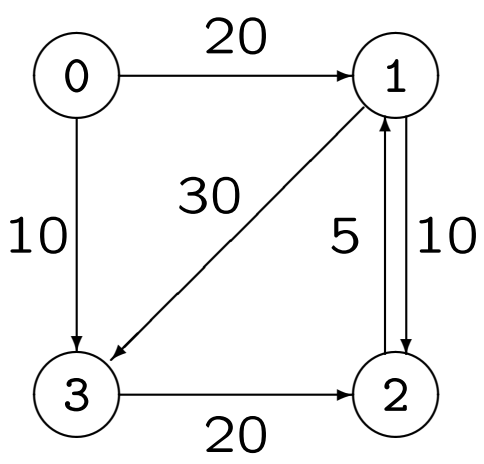


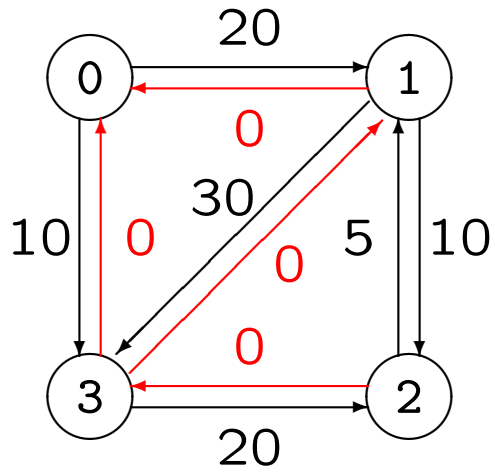
Como não há caminho de 0 para 2 na **rede residual** (i.e. quando se ignoram os arcos com um fluxo igual à capacidade), o fluxo é máximo (e o seu valor é 30).

Rede de Fluxos (*Flow Network*)

Seja $G = (V, A)$ um grafo orientado e pesado, cujos arcos têm um peso não negativo (que representa a capacidade do arco). A **rede de fluxos** de G é o grafo $R = (V, A')$, orientado e pesado, tal que:

- $A \subseteq A'$ (a rede tem todos os arcos de G);
- Se $(v, w) \in A$ e $(w, v) \notin A$, a rede tem o arco (w, v) com peso zero.





Fluxo (*Flow*)

Sejam:

- $R = (V, A')$ uma rede de fluxos;
- $f \in V$ o vértice fonte;
- $d \in V$ o vértice dreno; e
- $c(v, w)$ a capacidade do arco $(v, w) \in A'$.

Um **fluxo** em R é uma função $\phi : A' \longrightarrow \mathbb{R}$ que satisfaz as seguintes propriedades.

- (Capacidade) $\phi(v, w) \leq c(v, w)$, para qualquer $(v, w) \in A'$.
- (Simetria) $\phi(v, w) = -\phi(w, v)$, para qualquer $(v, w) \in A'$.
- (Conservação) $\sum_{\{w|(v,w)\in A'\}} \phi(v, w) = 0$, para qualquer $v \in V \setminus \{f, d\}$.

O **valor do fluxo** ϕ da fonte f para o dreno d é:

$$\sum_{\{v|(f,v)\in A'\}} \phi(f, v).$$

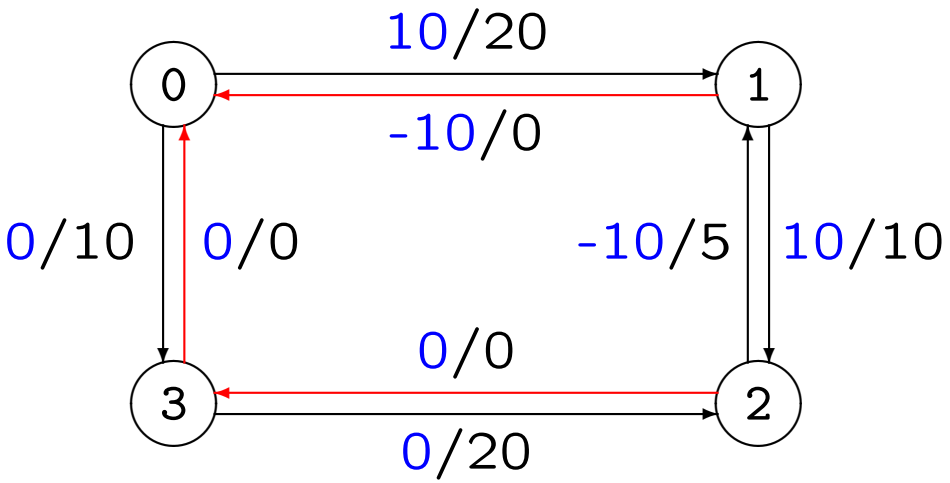
Rede Residual (*Residual Network*)

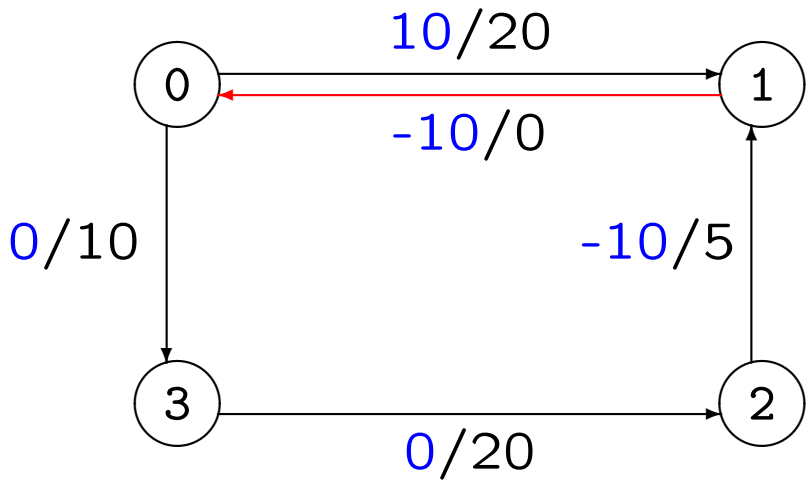
Sejam:

- $R = (V, A')$ uma rede de fluxos;
- $c(v, w)$ a capacidade do arco $(v, w) \in A'$; e
- ϕ um fluxo em R.

A **rede residual** de R induzida por ϕ é o grafo orientado $R_\phi = (V, A'')$, onde:

$$A'' = \{(v, w) \in A' \mid c(v, w) - \phi(v, w) > 0\}.$$





Observações

- O **método** de Ford-Fulkerson não especifica como se descobrem os caminhos da fonte f para o dreno d na rede residual.
- Se os pesos dos arcos forem números **irracionais**, o método pode não convergir.
- Se os pesos dos arcos forem números **racionais**, podem ser convertidos para números inteiros.
- Se os pesos dos arcos forem números **inteiros**, o número máximo de iterações não excede ...

  Justificação:

Observações

- O **método** de Ford-Fulkerson não especifica como se descobrem os caminhos da fonte f para o dreno d na rede residual.

- Se os pesos dos arcos forem números **irracionais**, o método pode não convergir.

- Se os pesos dos arcos forem números **racionais**, podem ser convertidos para números inteiros.

- Se os pesos dos arcos forem números **inteiros**, o número máximo de iterações não excede o valor dos fluxos máximos.

  Justificação: no início, o valor do fluxo é zero e, em cada iteração, o valor do fluxo é incrementado em, pelo menos, uma unidade.

Algoritmo de Edmonds-Karp [1972]

Aplica o método de Ford-Fulkerson,

descobrindo os caminhos da fonte f para o dreno d

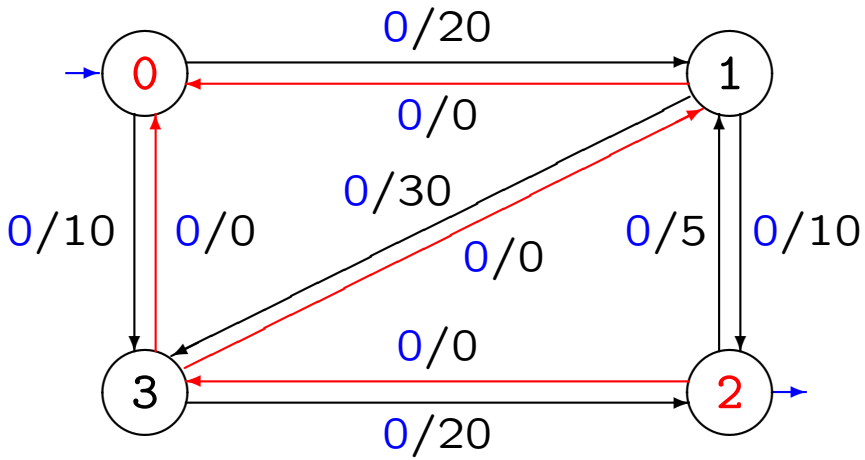
com uma pesquisa em largura na rede residual.

Esses caminhos têm comprimento mínimo,

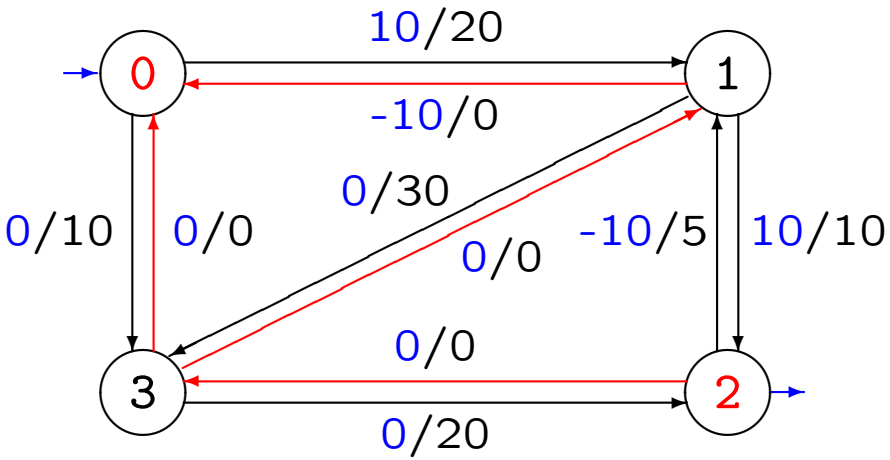
considerando que cada arco tem custo (distância) um.

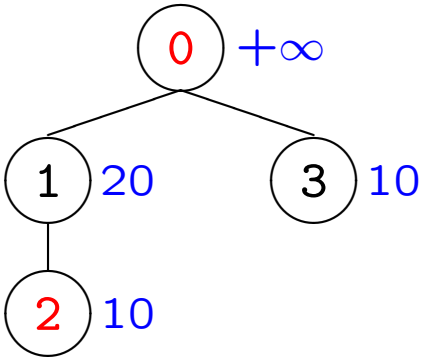
Algoritmo de Edmonds-Karp (1)

Estado Inicial



Estado Seguinte





via:

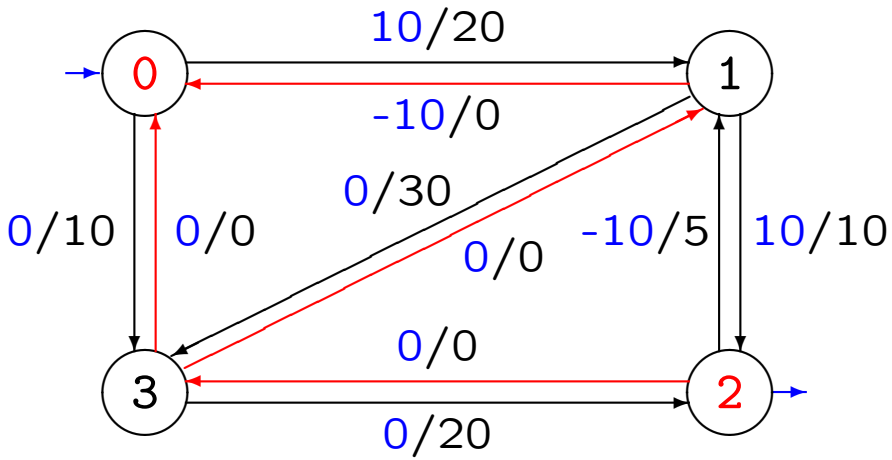
	0	1	0
0	1	2	3

incr:

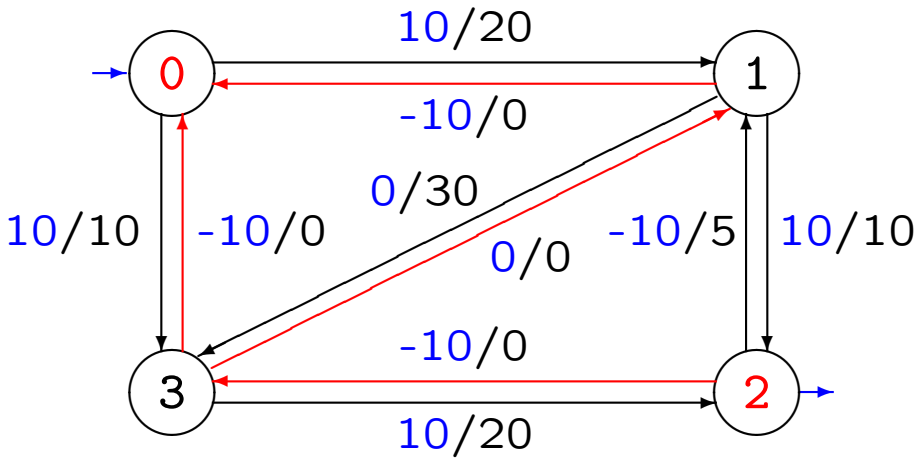
+∞	20	10	10
0	1	2	3

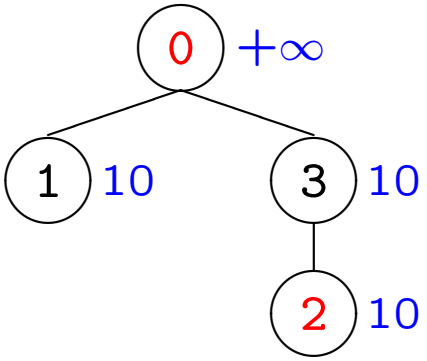
Algoritmo de Edmonds-Karp (2)

Estado Corrente



Estado Seguinte





via:

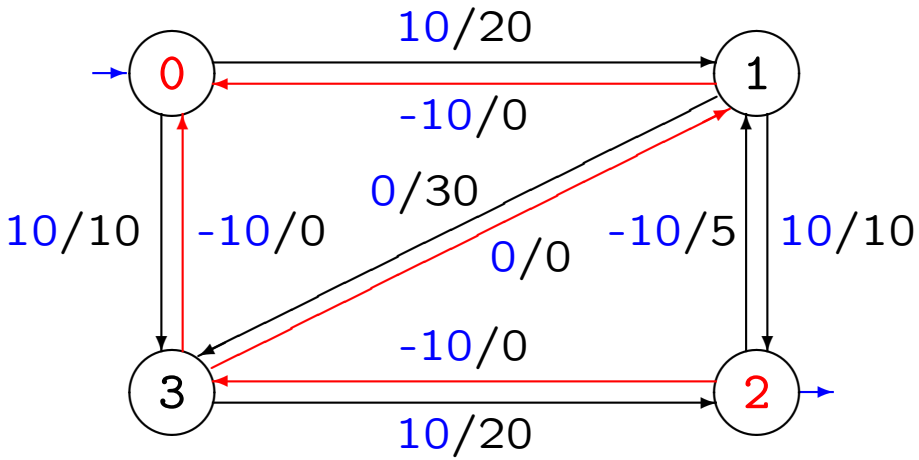
	0	1	2	3
	0	3	0	

incr:

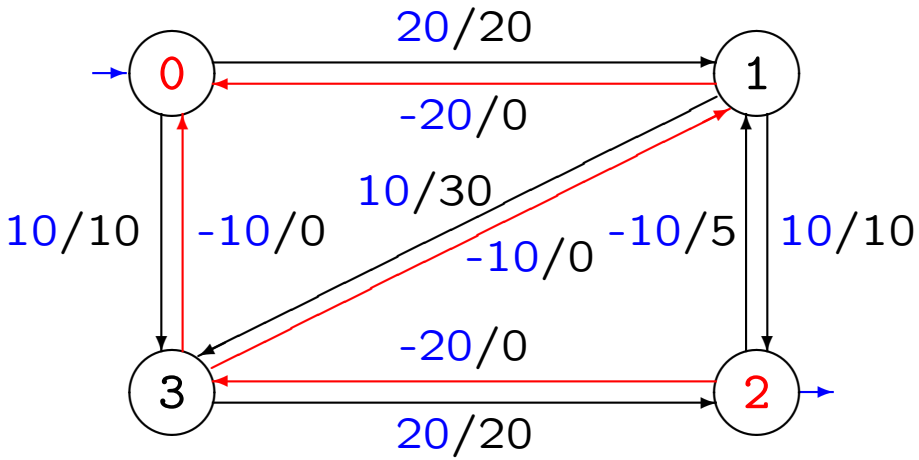
0	1	2	3
+∞	10	10	10

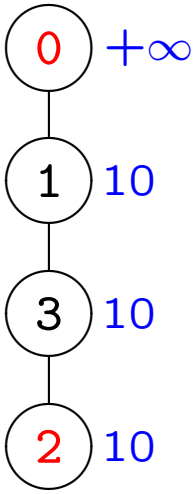
Algoritmo de Edmonds-Karp (3)

Estado Corrente



Estado Seguinte





via:

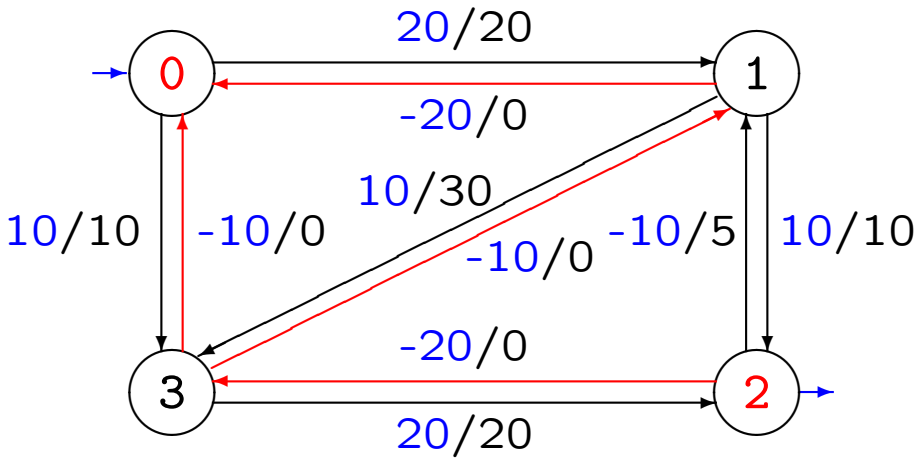
	0	3	1
0	1	2	3

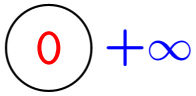
incr:

+∞	10	10	10
0	1	2	3

Algoritmo de Edmonds-Karp (4)

Estado Corrente





via:

incr:

Construir a Rede de Fluxos

```
Digraph<L> buildNetwork( Digraph<L> graph ) {
    // The network has every vertex and every edge in the graph.
    Digraph<L> network = graph.clone();

    // For every edge (v,w) in the graph (V,E),   if (w,v) ∉ E,
    // add edge (w,v) with capacity zero into the network.
    for every Edge<L> e in graph.edges() {
        Node node1 = e.firstNode();
        Node node2 = e.secondNode();
        if ( !graph.edgeExists(node2, node1) )
            network.addEdge(node2, node1, 0);
    }
    return network;
}
```

Fluxo Máximo

```
Pair<L, L[][]> edmondsKarp( Digraph<L> graph, Node source,
    Node sink ) {

    Digraph<L> network = buildNetwork(graph);
    int numNodes = network.numNodes();
    L[][] flow = new L[numNodes][numNodes];
    for every Edge<L> e in network.edges()
        flow[ e.firstNode() ][ e.secondNode() ] = 0;
    Node[] via = new Node[numNodes];
    L flowValue = 0;
    L increment;
```

```
    while ( ( increment = findPath(network, flow, source, sink, via) )
                != 0 ) {

        flowValue += increment;
        // Update flow.
        Node node = sink;
        while ( node != source ) {
            Node origin = via[node];
            flow[origin][node] += increment;
            flow[node][origin] -= increment;
            node = origin;
        }
    }

    return new PairClass<>(flowValue, flow);
}
```

```
L findPath( Digraph<L> network,  L[][] flow,  Node source,
    Node sink,  Node[] via ) {

    int numNodes = network.numNodes();

    Queue<Node> waiting = new QueueIn...<>(?);

    boolean[] found = new boolean[numNodes];
    for every Node v in network.nodes()
        found[v] = false;

    L[] pathIncr = new L[numNodes];

    waiting.enqueue(source);
    found[source] = true;
    via[source] = source;
    pathIncr[source] = +∞;
```

```
    do {
        Node origin = waiting.dequeue();
        for every Edge<L> e in network.outIncidentEdges(origin) {
            Node destin = e.secondNode();
            L residue = e.label() − flow[origin][destin];
            if ( !found[destin]  &&  residue > 0 ) {
                via[destin] = origin;
                pathIncr[destin] = Math.min(pathIncr[origin], residue);
                // destin == sink? Tratamento diferente (próximo slide)
            }
        }
    }
    while ( !waiting.isEmpty() );
    return 0;
}
```

```
do {
        Node origin = waiting.dequeue();
        for every Edge<L> e in network.outIncidentEdges(origin) {
            Node destin = e.secondNode();
            L residue = e.label() − flow[origin][destin];
            if ( !found[destin]   &&   residue > 0 ) {
                via[destin] = origin;
                pathIncr[destin] = Math.min(pathIncr[origin], residue);
                if ( destin == sink )
                    return pathIncr[destin];
                waiting.enqueue(destin);
                found[destin] = true;
            }
        }
    }
while ( !waiting.isEmpty() );
```

Complexidade do Algoritmo de Edmonds-Karp

Grafo $G = (V, A)$ em Vetor de Listas de Incidências

construir a rede de fluxos	$O(\|V\| \times \|A\|)$
inicializar o fluxo	$\Theta(\|A\|)$
k $\times$ descobrir um caminho	$O(k \times \|A\|)$
k $\times$ atualizar o fluxo	$O(k \times \|V\|)$
TOTAL	$O((\|V\| + k) \times \|A\|)$

Complexidade do Algoritmo de Edmonds-Karp

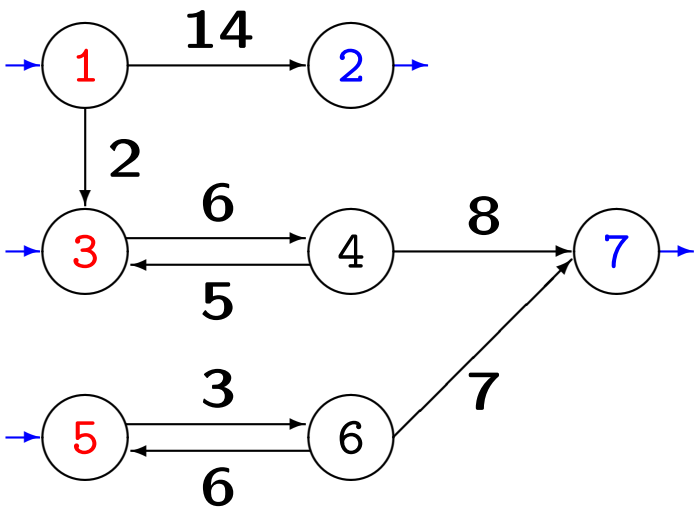
Grafo $G = (V, A)$ em Vetor de Listas de Incidências

construir a rede de fluxos	$O(\|V\| \times \|A\|)$
inicializar o fluxo	$\Theta(\|A\|)$
k $\times$ descobrir um caminho	$O(k \times \|A\|)$
k $\times$ atualizar o fluxo	$O(k \times \|V\|)$
TOTAL	$O((\|V\| + k) \times \|A\|)$

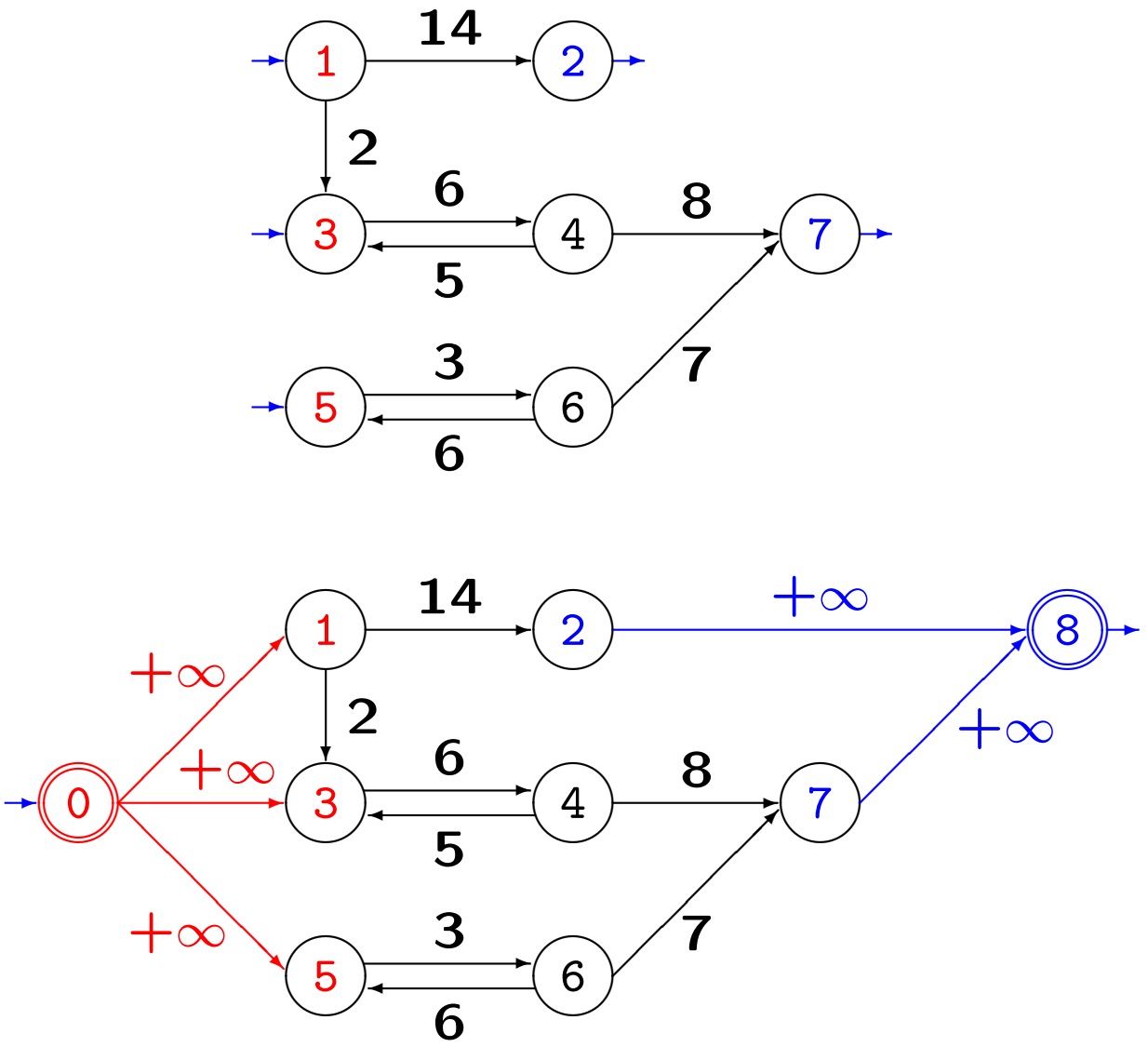
Resultado: O número de iterações (k) do algoritmo de Edmonds-Karp não excede $|V| \times |A|$ (nem excede o valor dos fluxos máximos).

Portanto, a complexidade do algoritmo é $O(|V| \times |A|^2)$.

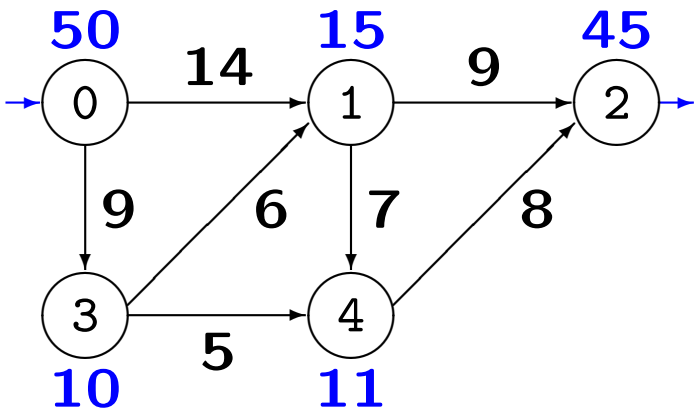
Generalização para Múltiplas Fontes ou Drenos



Generalização para Múltiplas Fontes ou Drenos



Generalização para Pesos nos Vértices



Generalização para Pesos nos Vértices