

# Fluxo em redes

**Wladimir Araújo Tavares**<sup>1</sup>

<sup>1</sup>Universidade Federal do Ceará - Campus de Quixadá

16 de setembro de 2022

## 1 Fluxo em redes

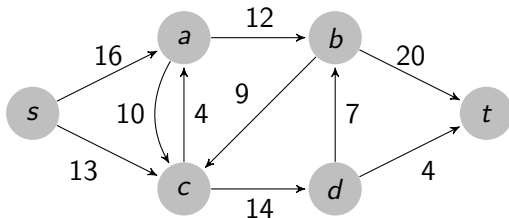
- Diversas aplicações em diferentes contextos:
  - ▶ Calcular a maior taxa de material que pode ser enviado desde da fonte até o sorvedouro.
  - ▶ Enviar a maior quantidade de caminhões considerando que as estradas possuem um limite de número de caminhões por unidade de tempo.
  - ▶ Minimizar o custo de destruir pontes a fim de desconectar duas cidades  $s$  e  $t$ .
  - ▶ Problema de emparelhamento de objetos (casamentos, residentes/hospitais, tarefas/máquinas)

# Fluxo em redes

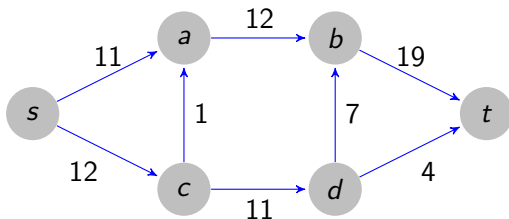
- Uma rede de fluxo  $G = (V, E)$  é um grafo dirigido em que cada arco e tem uma capacidade  $c(e) > 0$ .
- Possui dois vértices especiais, fonte  $s$  e sorvedouro  $t$ .
- Problema: maximizar a quantidade total de fluxo de  $s$  para  $t$  satisfazendo duas restrições:
  - ▶ o fluxo em cada aresta e não excede a capacidade da aresta, ou seja,  $f(u, v) \leq c(u, v)$  (Restrição de capacidade).
  - ▶ Para cada vértice  $v \neq s$  e  $v \neq t$ , o fluxo de entrada é igual ao fluxo de saída, ou seja,  $\sum_{v \in N^+(u)} f(u, v) = \sum_{v \in N^-(u)} f(u, v)$  (Conservação do fluxo).

# Fluxo em redes

- Capacidade



- Fluxo Máximo (23 unidades)

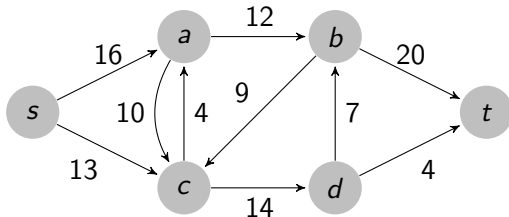


# Corte mínimo

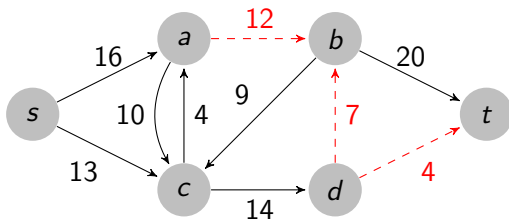
- Queremos remover algumas arestas do grafo tal que depois da remoção das arestas, não exista um caminho de  $s$  para  $t$ .
- O custo de remover uma aresta  $e$  será igual a sua capacidade  $c(e)$
- O problema do custo mínimo consiste em encontrar um corte com o custo total mínimo.
- Teorema 26.7 (CLRS, 2ª Edição): Fluxo máximo = Custo mínimo

# Exemplo

- Capacidade

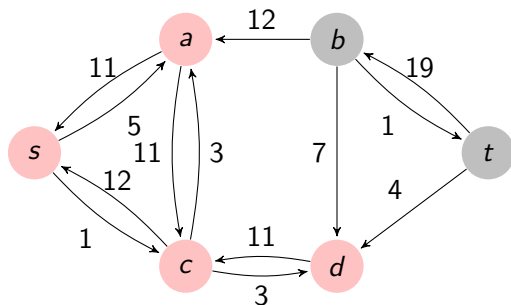
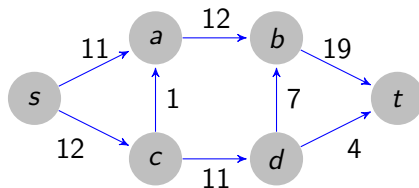
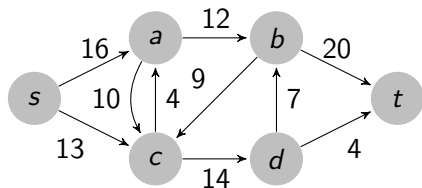


- Corte mínimo



# Corte Mínimo

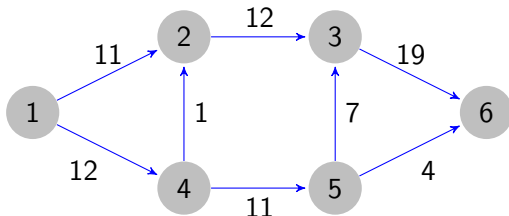
- Subtraia o fluxo máximo do grafo original.





# Decomposição em Fluxos

- Um fluxo válido pode ser decomposto em caminhos aumentantes:



- ▶  $s \rightarrow a \rightarrow b \rightarrow t : 11$
- ▶  $s \rightarrow c \rightarrow a \rightarrow b \rightarrow t : 1$
- ▶  $s \rightarrow c \rightarrow d \rightarrow b \rightarrow t : 7$
- ▶  $s \rightarrow c \rightarrow d \rightarrow t : 4$

# Algoritmo de Ford-Fulkerson

- Um algoritmo simples e prático para encontrar o fluxo máximo.
- Encontre caminhos em ampliação válidos no grafo residual  $G_f$  até que não exista mais nenhum e some o fluxo de todos eles.
- $f$  é um fluxo máximo de  $G$  então não existe nenhum caminho de ampliação em  $G_f$ .

# Algoritmo de Ford-Fulkerson

---

## Algorithm FordFulkerson( $G, s, t$ )

---

```
1: function FordFulkerson( $G, s, t$ )
2:    $f^* \leftarrow 0$ 
3:   for cada  $(u, v) \in E(G)$  do
4:      $f[u, v] \leftarrow 0$ 
5:      $f[v, u] \leftarrow 0$ 
6:   while Enquanto existir um caminho em ampliação  $p$  de  $s$  até  $t$  na rede residual  $G_f$  do
7:      $c_f(p) \leftarrow \min\{c[u, v] - f[u, v] : (u, v) \in p\}$ 
8:      $f^* \leftarrow f^* + c_f(p)$ 
9:     for cada  $(u, v) \in p$  do
10:       $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
11:       $f[v, u] \leftarrow -f[u, v]$ 
```

---

- As capacidades são valores inteiros.
- Encontrar o caminho em ampliação deve ter complexidade  $O(n + m)$
- A complexidade do algoritmo FordFulkerson é  $O((n + m)f^*)$
- A complexidade depende do valor do fluxo máximo.

# Representação do grafo

```
typedef struct Edge{
    int from,to, rev, f, cap;
    Edge(int from, int to, int rev, int f, int cap):
        from(from),to(to), rev(rev), f(f), cap(cap) {};
};

vector <Edge> g[MAXN];

void addEdge(int s, int t, int cap)
{
    g[s].push_back( Edge(s,t, g[t].size(), 0, cap) );
    g[t].push_back( Edge(t,s, g[s].size()-1, 0, 0) );
}
```

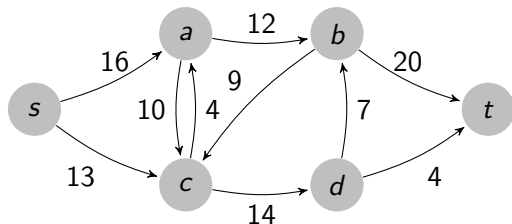
# Caminho em ampliação

```
int find_path(int s, int t, int f){
    int v, df;
    if( s == t ) return f;
    vis[s] = true;
    for(int i = 0; i < g[s].size(); i++){
        Edge & e = g[s][i];
        if( e.cap - e.f <= 0) continue;
        v = e.to;
        if( !vis[v] ){
            df = find_path(v, t, min(f, e.cap-e.f) );
            if( df > 0){
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
    }
    return 0;
}
```

# Ford-Fulkerson

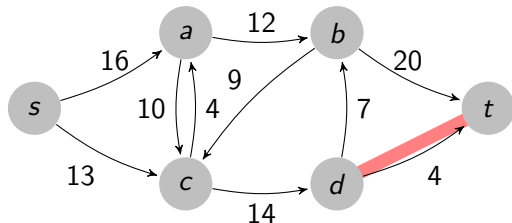
```
/*  
Algoritmo Ford-Fulkerson  
Complexidade  $O((|V|+|E|)*|f|) = O(V^2*|f|)$   
*/  
int ford_fulkerson(int _src , int _dest)  
{  
    int totflow , flow;  
    src = _src;  
    dest = _dest;  
    totflow = 0;  
    fill( vis , vis + nodes , false );  
    while( flow = find_path(src , dest , 0x7fffffff) )  
    {  
        totflow += flow;  
        fill( vis , vis + nodes , false );  
    }  
    printf("fluxo_maximo_%d\n" , totflow );  
    return totflow;  
}
```

# Execução do Ford-Fulkerson

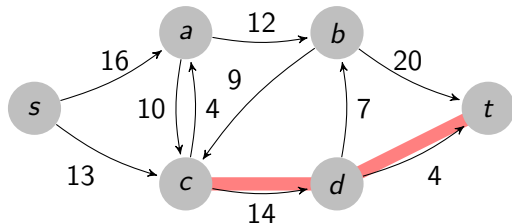




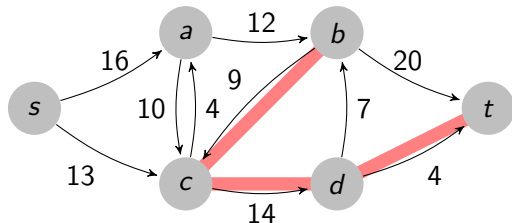
# Execução do Ford-Fulkerson



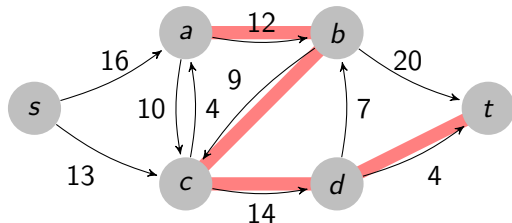
# Execução do Ford-Fulkerson



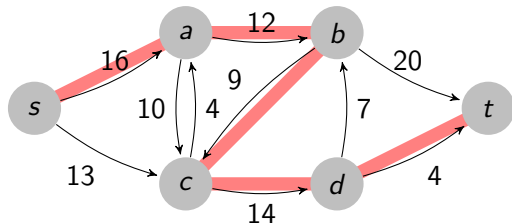
# Execução do Ford-Fulkerson



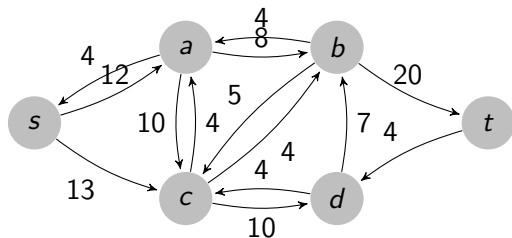
# Execução do Ford-Fulkerson



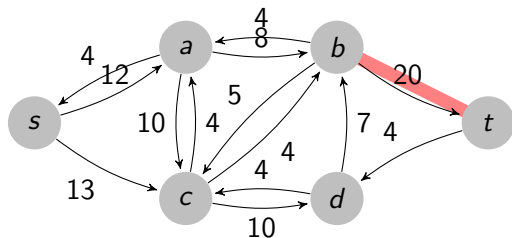
# Execução do Ford-Fulkerson



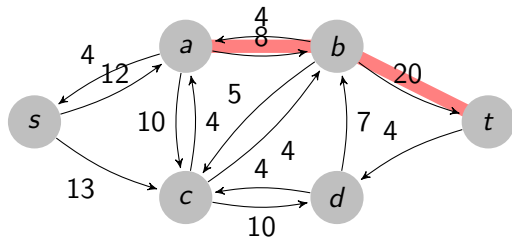
# Execução do Ford-Fulkerson



# Execução do Ford-Fulkerson

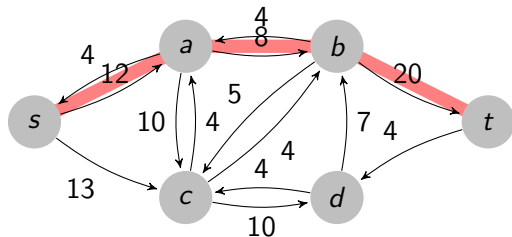


# Execução do Ford-Fulkerson

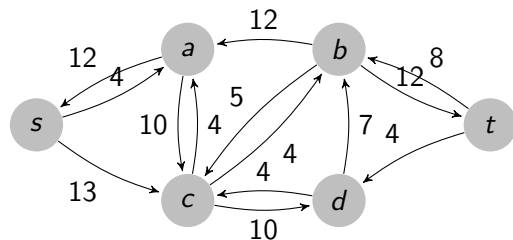




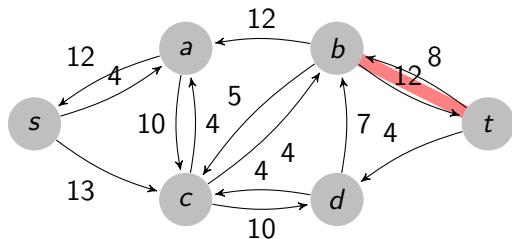
# Execução do Ford-Fulkerson



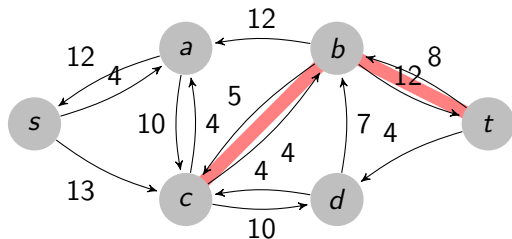
# Execução do Ford-Fulkerson



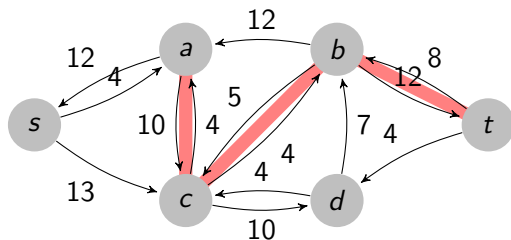
# Execução do Ford-Fulkerson



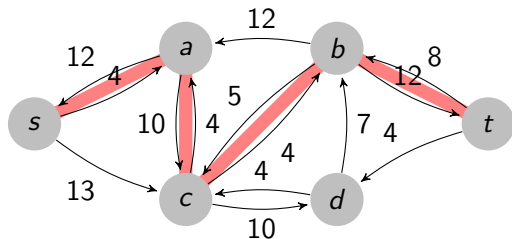
# Execução do Ford-Fulkerson



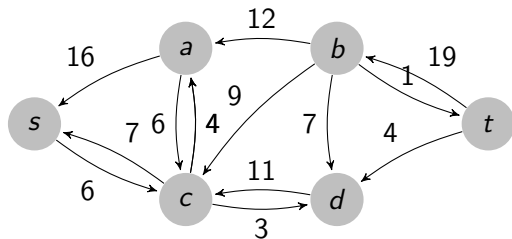
# Execução do Ford-Fulkerson



# Execução do Ford-Fulkerson



# Execução do Ford-Fulkerson



# Algoritmo Edmonds-Karp

- O algoritmo de Edmonds-Karp é o algoritmo de Ford-Fulkerson utilizando o caminho mínimo para o caminho em ampliação.
- Esta modificação melhorar a complexidade do algoritmo para tempo polinomial.
- A distância do caminho mais curto na rede residual aumenta monotonicamente em cada ampliação do fluxo.
- O algoritmo de Edmonds-Karp executa em  $O(VE^2)$



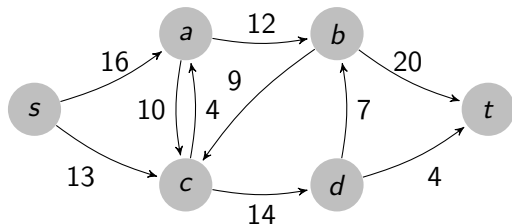
# EdmondsKarp

```
int bfs(int src, int dest, int *pred, int *edge){
    int q[nodes], qt;
    bool vis[MAXN];
    qt = 0; q[qt++] = src;
    fill( pred, pred + nodes, -1);
    fill( vis, vis + nodes, false);
    for(int qh = 0; qh < qt && pred[dest] == -1; qh++){
        int u = q[qh];
        vis[u] = true;
        for(int i = 0; i < (int)g[u].size(); i++){
            Edge & e = g[u][i];
            if( e.cap - e.f <= 0) continue;
            int v = e.to;
            if( !vis[v] ){
                pred[v] = u; edge[v] = i; q[qt++] = v;
            }
        }
    }
    if( pred[dest] == -1) return 0;
    else return 1;
}
```

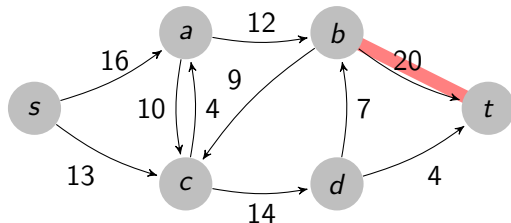
# EdmondsKarp

```
int edmonds_karp(int _src, int _dest)
{
    int flow, totflow;
    int pred[nodes];
    int edge[nodes];
    int src, dest;
    src = _src;
    dest = _dest;
    totflow = 0;
    while( bfs(src, dest, pred, edge) > 0 ){
        int flow = 0x7fffffff;
        for(int v = dest; pred[v] >= 0; v = pred[v]){
            Edge & e = g[ pred[v] ][ edge[v] ];
            flow = min( flow, e.cap - e.f );
        }
        for(int v = dest; pred[v] >= 0; v = pred[v]){
            Edge & e = g[ pred[v] ][ edge[v] ];
            e.f += flow; g[v][e.rev].f -= flow;
        }
        totflow += flow;
    }
    return totflow;
}
```

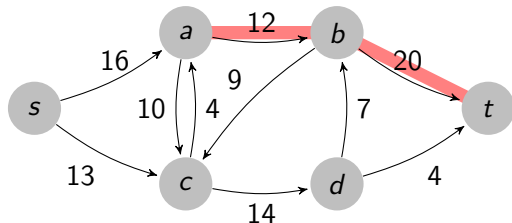
# Execução do Edmonds-Karp



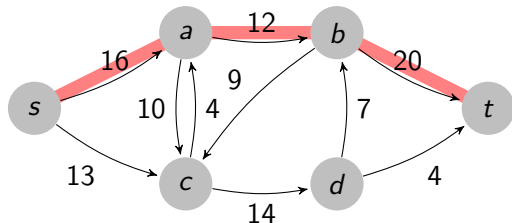
# Execução do Edmonds-Karp



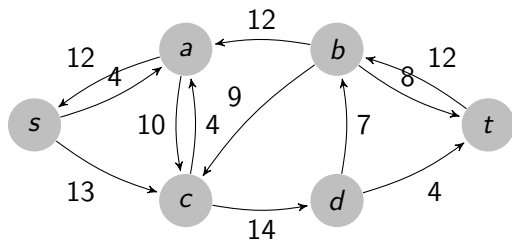
# Execução do Edmonds-Karp



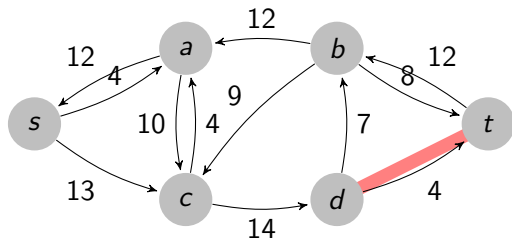
# Execução do Edmonds-Karp



# Execução do Edmonds-Karp

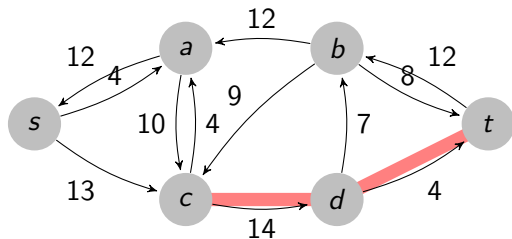


# Execução do Edmonds-Karp

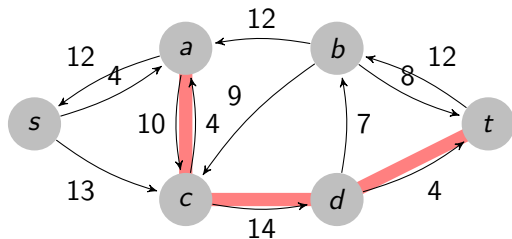




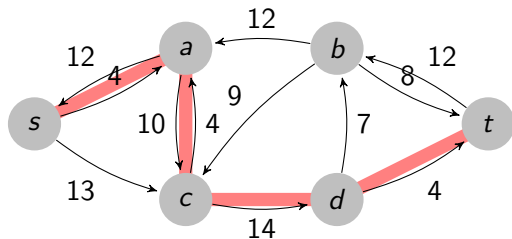
# Execução do Edmonds-Karp



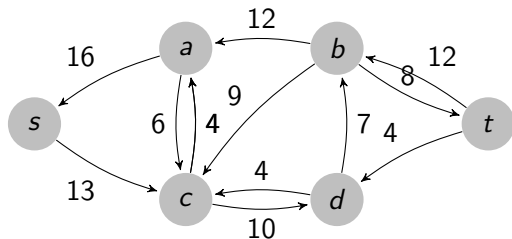
# Execução do Edmonds-Karp



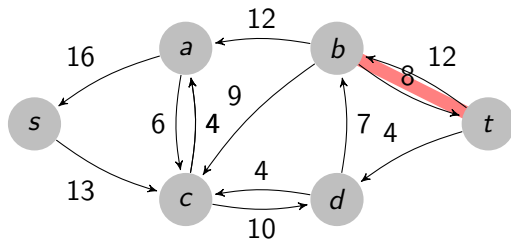
# Execução do Edmonds-Karp



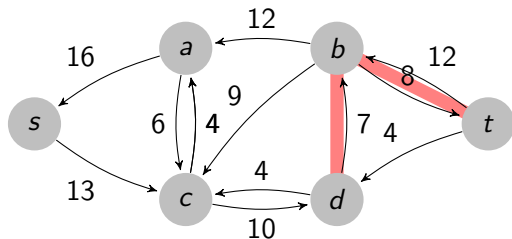
# Execução do Edmonds-Karp



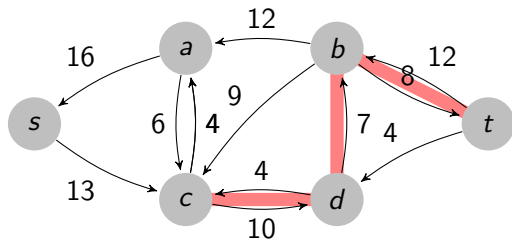
# Execução do Edmonds-Karp



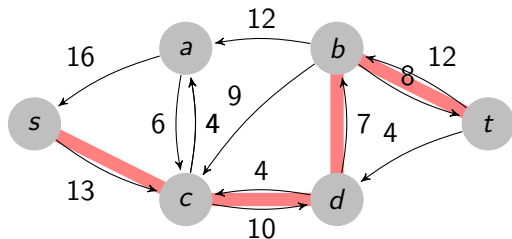
# Execução do Edmonds-Karp



# Execução do Edmonds-Karp

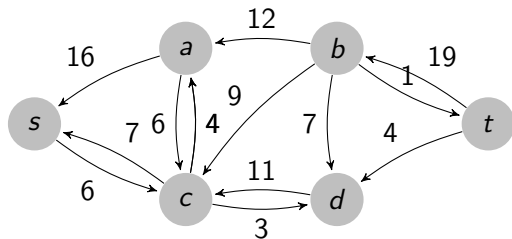


# Execução do Edmonds-Karp





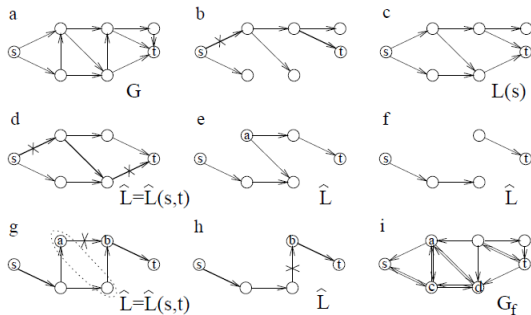
# Execução do Edmonds-Karp



# Algoritmo de Dinics

- A seguinte anedota mostra como as coisas eram feitas em URSS. Um americano e um russo que trabalhavam ambos no desenvolvimento de armas durante a Guerra Fria se encontram. O americano pergunta: "Quando vocês desenvolveram a bomba, como você faziam para executar a enorme quantidade de cálculos necessária com seus computadores ruins?". O russo respondeu: "Nós usamos algoritmos melhores".
- Encontrar os caminhos em ampliação no algoritmo FF é o gargalo do algoritmo.
- O algoritmo BFS é usado para construir grafo em camadas.
- O grafo em camadas é usado para encontrar os caminhos em ampliação.

# Algoritmo de Dinics



**Figura:** (d) A rede em camadas  $\hat{L}(s, t)$  de comprimento 3 e um caminho de  $s$  para  $t$ . (e) As arestas saturadas são removidas. (g) A nova rede em camadas de comprimento 4.

# Algoritmo de Dinics

```
bool bfs(int src, int dest)
{
    fill(dist, dist + nodes, -1);
    dist[src] = 0;
    int qh = 0, qt = 0;
    int q[MAXN];
    q[qt++] = src;
    for(int qh = 0; qh < qt; qh++)
    {
        int u = q[qh];
        for(int j = 0; j < g[u].size(); j++)
        {
            Edge & e = g[u][j];
            int v = e.to;
            if( dist[v] < 0 && e.f < e.cap )
            {
                dist[v] = dist[u] + 1;
                q[qt++] = v;
            }
        }
    }
    return dist[dest] >= 0;
}
```

# Algoritmo de Dinics

```
int dfs(int src, int dest, int f){
    if( src == dest ){
        return f;
    }
    for(int i = 0; i < g[src].size(); i++)
    {
        Edge & e = g[src][i];
        if( e.cap <= e.f ) continue;
        int v = e.to;
        if( dist[v] == dist[src] + 1){
            int df = dfs(v, dest, min(f, e.cap-e.f) );
            if(df > 0){
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
        }
    }
    return 0;
}
```

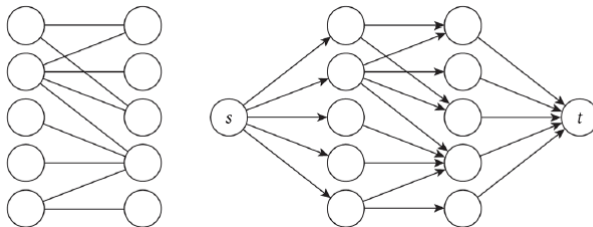
# Algoritmo de Dinics

```
int dinics(int src, int dest)
{
    int flow, totflow;
    totflow = 0;
    while( bfs(src, dest) )
    {
        while( flow = dfs(src, dest, INT_MAX) )
        {
            totflow += flow;
        }
    }
    printf("fluxo_maximo_%d\n", totflow);
    return totflow;
}
```

# Emparelhamento em grafos bipartidos

- Um grafo bipartido  $G(V, E)$  onde  $V = X \cup Y$  com  $X \cap Y = \emptyset$  e  $E \subseteq X \times Y$ .
- Grafos bipartidos modelam situações em os objetos são emparelhados com ou atribuídos para outros objetos. (Casamento, residentes/hospitais, tarefas/máquinas).
- Um emparelhamento em um grafo bipartido  $G$  é um conjunto  $M \subseteq E$  tal que todo vértice de  $G$  incide em no máximo um elemento de  $M$ .
- Um conjunto de arestas  $M$  é um emparelhamento perfeito se todo vértice de  $V$  incide em exatamente um aresta de  $M$ .

# Algoritmo para emparelhamento em grafos bipartidos



**Figura:** Um grafo bipartido e a correspondente rede de fluxos com todas as capacidades iguais a 1

- Converta o grafo  $G = (X \cup Y, E)$  em uma rede de fluxo  $G'$ : oriente as arestas de  $X$  para  $Y$ , adicione um vértice  $s$  e  $t$ , conecte  $s$  a cada vértice em  $X$ , conecte cada vértice de  $Y$  ao vértice  $t$  e sete todas as arestas com capacidade igual a 1.





# Problema de Circulação

- Dado um grafo direcionado  $G = (V, E)$  com a função de capacidade  $c : E \rightarrow \mathbb{Z}^+$  e uma função de demanda  $d : V \rightarrow \mathbb{Z}$ :
  - ▶  $d(v) > 0$ : nó destino e tem uma demanda de  $d(v)$  unidade de fluxo.
  - ▶  $d(v) < 0$ : nó fonte e tem uma oferta de  $-d(v)$  unidades de fluxo.
  - ▶  $d(v) = 0$ : nó recebe e transmite fluxo.
  - ▶  $S$  é o conjunto de vértices com demanda negativa.
  - ▶  $T$  é o conjunto de vértices com demanda positiva.
- Uma circulação é uma função  $f : E \rightarrow \mathbb{Z}^+$  que satisfaz
  - ▶ Para cada aresta  $e \in E$ ,  $0 \leq f(e) \leq c(e)$
  - ▶ Para cada vértice  $v$ ,  $\sum_{u \in N^-(v)} f[u, v] - \sum_{u \in N^+(v)} f[v, u] = d(v)$

# Redução do Problema de Circulação para o Problema do Fluxo Máximo

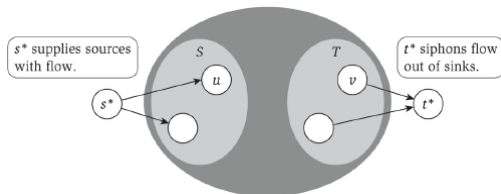
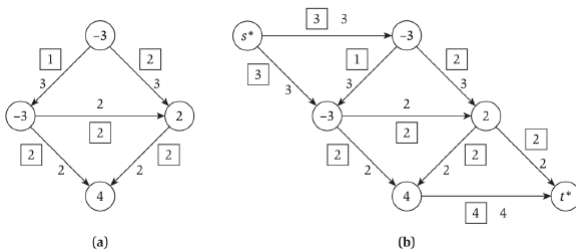


Figura: Redução do Problema de Circulação para o Problema do Fluxo Máximo

- Converta o grafo  $G$  em uma rede de fluxos  $G'$ :
  - 1 Adicione uma fonte  $s^*$  e um destino  $t^*$ .
  - 2 Conecte  $s^*$  a cada vértice  $v \in S$  usando uma aresta com capacidade  $-d(v)$ .
  - 3 Conecte cada vértice de  $v \in T$  ao vértice  $t^*$  usando uma aresta com capacidade  $d(v)$ .
  - 4 Verifique se o fluxo máximo é igual a  $D = \sum_{\{v:d(v)>0\}} d(v)$

# Circulação com demandas



**Figura:** (a) Uma instância do problema de circulação de demandas com a sua solução. Os números dentro do nó representam as demandas; números rotulando as arestas são as capacidades e os números dentro das caixas representam a função de circulação. (b) O resultado da redução desta instância em um problema equivalente de fluxo máximo.

# Problema de circulação com limites inferiores

- Nós queremos forçar que o fluxo em certas arestas.
- Dado um grafo  $G = (V, E)$  com a capacidade  $c(e)$  e um limite inferior  $0 \leq l(e) \leq c(e)$  para cada arestas e uma função de demanda  $d : V \rightarrow \mathbb{Z}$ .
- Uma circulação é uma função  $f : E \rightarrow \mathbb{Z}^+$  que satisfaz
  - ▶ Para cada aresta  $e \in E$ ,  $l(e) \leq f(e) \leq c(e)$
  - ▶ Para cada vértice  $v$ ,  $\sum_{u \in N^-(v)} f[u, v] - \sum_{u \in N^+(v)} f[v, u] = d(v)$
- Existe uma circulação viável para esse problema?

# Redução para o problema de circulação sem limites inferiores

- Seja  $f_0(e) = l(e) \forall e \in E$
- Seja  $L(v) = f_0^{in}(v) - f_0^{out}(v)$
- Encontre uma circulação  $f_1$  tal que  $f_1^{in}(v) - f_1^{out}(v) = d(v) - L(v)$
- A capacidade deixada em cada aresta será  $c(e) - l(e)$
- Defina um grafo  $G'$  com os mesmos vértices e arestas: a capacidade de uma aresta  $e$  é  $c(e) - l(e)$  e a demanda do vértice  $v$  é  $d(v) - L(v)$

# Projeto de Entrevista

- Empresa vende  $k$  produtos
- Empresa possui um banco de dados do histórico de compras de seus clientes.
- Empresa quer enviar uma pesquisa customizada para seus  $n$  clientes para entender suas preferências.
- A pesquisa deve satisfazer as seguintes restrições:
  - ▶ Cada cliente recebe perguntas sobre um subconjunto de produtos.
  - ▶ Um cliente recebe perguntas apenas sobre os produtos que ele comprou.
  - ▶ A pesquisa deve ser informativa mas não tão longa: Cada cliente  $i$  deve responder perguntas sobre um número de produtos entre  $c_i$  and  $c'_i$ .
  - ▶ Para cada produto devemos coletar informações suficientes: entre  $p_j$  e  $p'_j$  clientes devem ser responder perguntas sobre o produto  $j$ .
- É possível projetar uma pesquisa que satisfaça essas restrições?