Iterative Improvement

Capítulo 10 Levitin +… +
Material Prof. Gustavo Batista

Abordagem

Greedy strategy: constrói a solução para um problema de otimização peça-por-peça, sempre adicionando um peça ótima local a uma solução parcial

Iterative Improvement strategy: começa com uma solução possível (que satisfaz todas as soluções do problema) e passa a melhora-la através da aplicação repetida de algum passo simples

- The Maximum-Flow problem
- The Simplex Method
- Maximum Matching in Bipartide Graphs
- The Stable Marriage Problem

Iterative Improvement

Fluxo Máximo &
Fluxo de Custo Mínimo

Gustavo Batista & GraçaPimentel

Fluxo **Máximo**

- Podemos interpretar um grafo orientado como um fluxo em rede:
  - Existe uma origem que produz um material em uma taxa fixa (**source**)
  - Existe um depósito que consome esse material na mesma taxa (**sink**)
  - O peso *uij* em cada aresta direcionada (i,j) é um inteiro positivo chamado a capacidade da aresta (**capacity**)
    - Esse valor representa o limite superior da quantidade de material que pode ser enviada de i a j via essa aresta
  - Um dígrafo que satisfaz essas propriedades é chamado **flow network**
- O fluxo em qualquer ponto do grafo é a taxa na qual o material se move.

Existem diversas aplicações como líquidos em tubos, peças em linhas de montagem, correntes elétricas, etc..

Online-judge UVA Min-Cost Max-Flow Related Problems

http://online-judge.uva.es/board/viewtopic.php?f=22&t=11719&p=

Fluxo Máximo

- Cada aresta orientada pode ser entendida como um canal:
  - Sendo que cada canal possui uma capacidade estabelecida
- Vértices são junções de canais, além da origem e do depósito:
  - Não há acumulação de material nos vértices
    - Ou seja, a taxa de entrada deve ser igual à taxa de saída de material
  - Chamamos essa propriedade de *conservação de fluxo*

Fluxo Máximo

Em fluxo máximo, deseja-se calcular a maior taxa de fluxo de material:

- Da origem até o depósito
- Sem violar quaisquer restrições de capacidade

Definições

Um **fluxo em rede** é um grafo orientado $G = (V, A)$:

- Cada aresta $(u, v) \in A$ tem uma capacidade não negativa $c(u, v) \geq 0$
- Se $(u, v) \notin A$ então $c(u, v) = 0$
- Dois vértices são especiais: origem s e depósito t
- Por conveniência, assume-se que existe um caminho $s \leadsto v \leadsto t$

Definições

Um **fluxo** em G é uma função de valor real $f : V \times V \rightarrow \mathbb{R}$ que satisfaz:

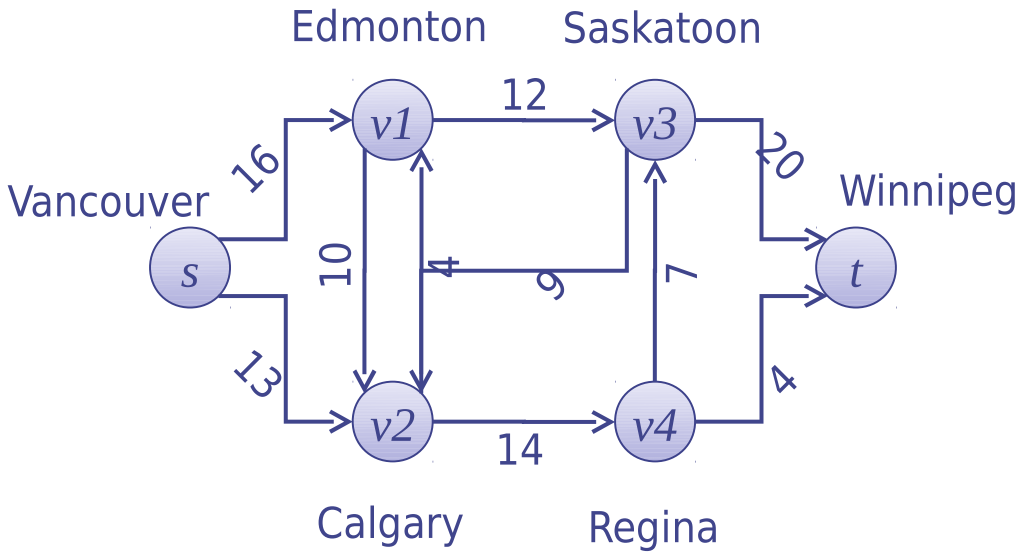
Restrição de Capacidade: para todo $u, v \in A$, $f(u, v) \leq c(u, v)$

Simetria oblíqua: para todo $u, v \in A$, $f(u, v) = -f(v, u)$

Conservação de fluxo: para todo $u \in V - \{s, t\}$,

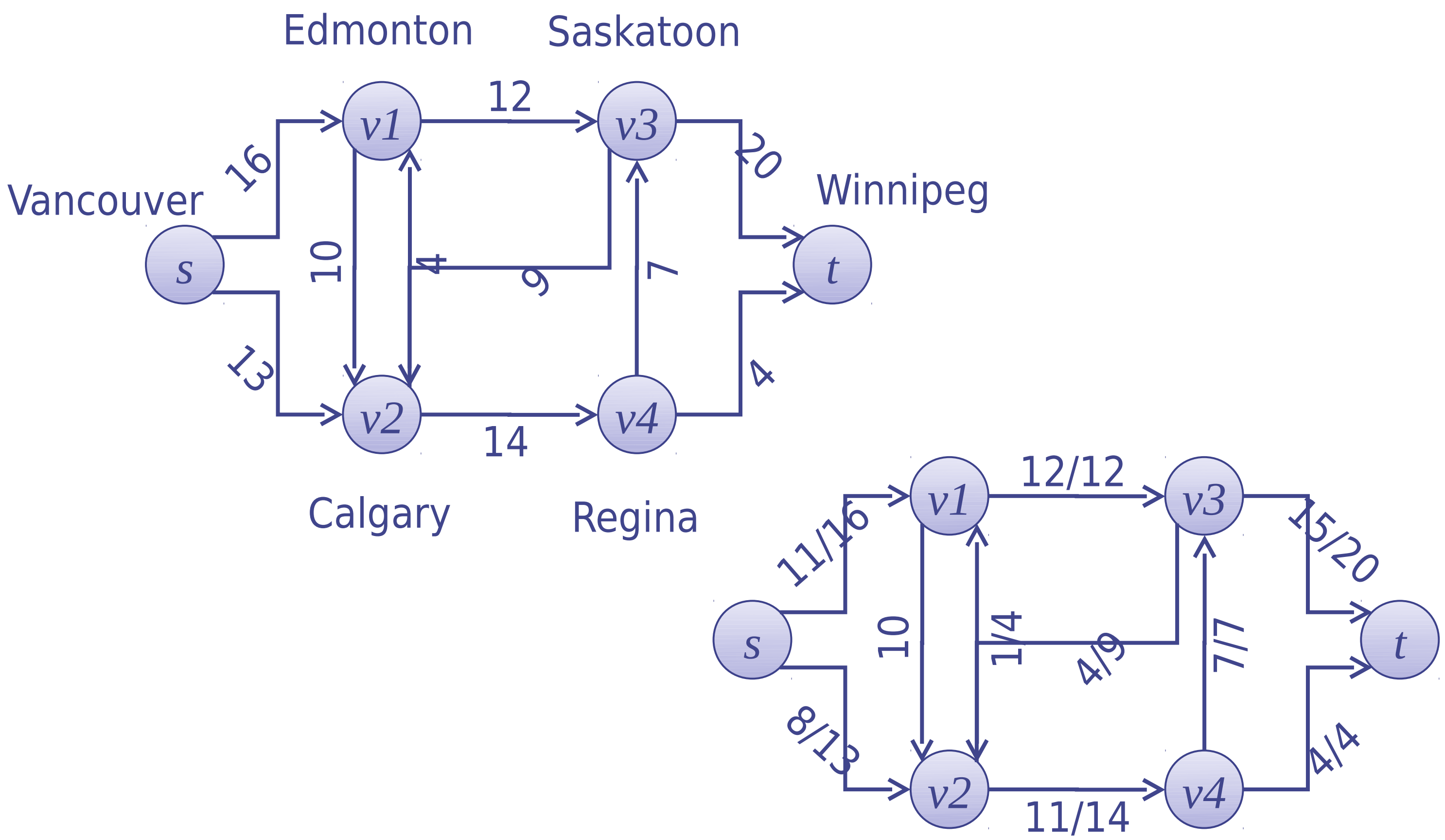
$$\sum_{v \in V} f(u, v) = 0$$

Exemplo



Edmonton
Saskatoon
12
v1
v3
16
20
Vancouver
Winnipeg
10
4
9
7
s
t
13
4
v2
v4
14
Calgary
Regina

Exemplo


Edmonton
Saskatoon
Vancouver
Winnipeg
Calgary
Regina
s
v1
v2
v3
v4
t
16
13
10
4
12
9
7
20
4
14
s
v1
v2
v3
v4
t
11/16
8/13
10
1/4
12/12
4/9
7/7
15/20
4/4
11/14

Definições

O **valor de um fluxo** em G é definido como:

$$|f| = \sum_{v \in V} f(s,v)$$

ou seja, o fluxo que sai da origem.

No problema de fluxo máximo, queremos encontrar um fluxo de valor máximo de *s* a *t* sobre *G*

Método de Ford-Fulkerson

Engloba diversas implementações com diferentes complexidades.
Utiliza três idéias importantes:

- Redes residuais
- Caminhos em ampliação, e
- Cortes

Redes residuais

A rede residual consiste em arestas que podem admitir mais fluxo:
A capacidade residual de dois vértices *u*, *v* é dada por:

$$cf\ (u, v) = c\ (u, v) - f\ (u, v)$$

Redes residuais

Por exemplo, se

$c\ (u, v) = 16$ e $f\ (u, v) = 11$, então

$cf\ (u, v) = 5$

Portanto pode aumentar a capacidade em 5 unidades antes de exceder a restrição.

Mas, se:

$c\ (u, v) = 16$ e $f\ (u, v) = -4$, então

$cf\ (u, v) = 20$

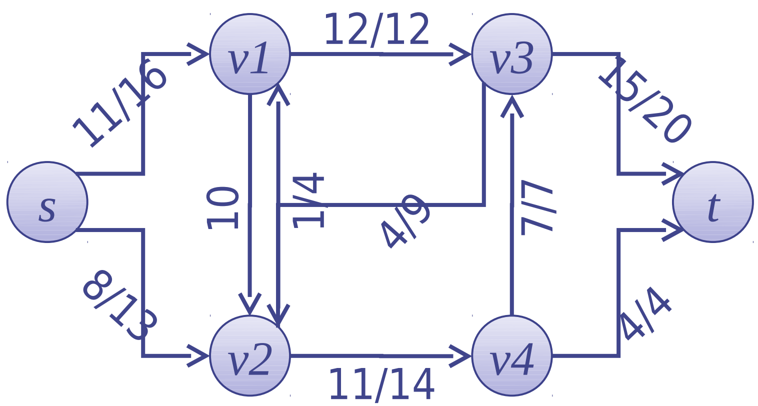
Pois, pode-se cancelar o fluxo contrário em 4 unidades e empurrar mais 16 unidades

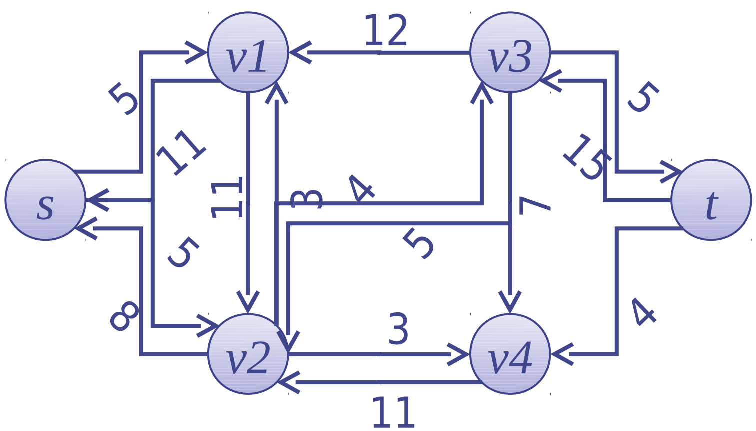
Redes residuais

Dados um fluxo em rede $G = (V, A)$ e um fluxo f, a rede residual de G induzida por f é $G_f = (V, A_f)$, onde:

$$A_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

Redes residuais





Redes residuais

As arestas em Af são as arestas em A ou suas inversas:

Se $f(u, v) < c(u, v)$ então $cf(u, v) = c(u, v) - f(u, v) > 0$

Se $f(v, u) < 0$, então $cf(v, u) = c(v, u) - f(v, u) > 0$

Em ambos os casos (u, v) e (v, u) $\in$ Af

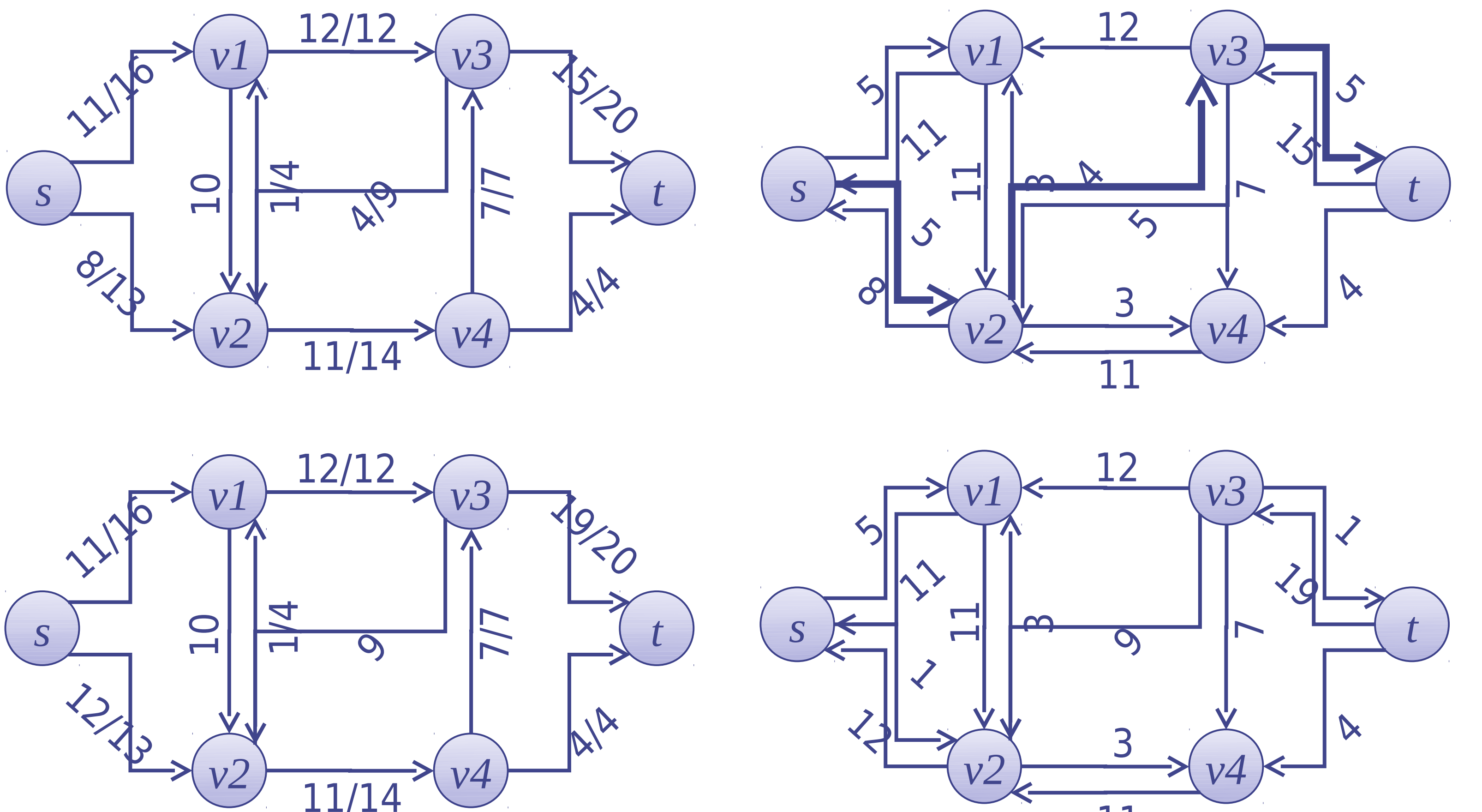
Caminhos em ampliação

Augmenting path

Dados um fluxo em rede $G = (V, A)$, e um fluxo f, um caminho em ampliação p é um caminho simples de s a t na rede residual Gf

Um caminho residual admite algum fluxo positivo adicional de s a t sem violar as capacidades das arestas

Caminhos em ampliação



Caminhos em ampliação

Capacidade residual é a quantidade máxima que se pode aumentar o fluxo em cada aresta de um caminho de ampliação *p*:

$$cf\ (p) = min\ \{cf\ (u, v): (u, v) \text{ está em } p\ \}$$

Ford-Fulkerson

O algoritmo básico de Ford-Fulkerson consiste em encontrar caminhos de ampliação, e *iterativamente* aumentar o fluxo em *G*

Ford-Fulkerson

```
Ford-Fulkerson(G, s, t)
   for cada aresta (u, v) ▪ A
      do f[u,v] = 0
         f[v,u] = 0
   while existe p de s até t na rede residual Gf
      do cf(p) = min{cf(u,v): (u,v) está em p}
         for cada aresta (u,v) em p
            do f[u,v] = f[u,v] + cf(p)
               f[v,u] = -f[u,v]
```

Edmonds-Karp

- Um ponto em aberto do algoritmo de Ford-Fulkerson é como encontrar os caminhos em ampliação *p*
- O algoritmo de *Edmonds-Karp* utiliza uma *busca em largura* para encontrar esses caminhos
  - Portanto *p* é o caminho mínimo entre *s* e *t* considerando o número de arestas
  - O algoritmo de Edmonds-Karp possui complexidade $O(VA2)$
- Existem outras variações mais eficientes...

Edmonds-Karp

```
struct grafo {
    // geral
    int adj[MAXVT][MAXVT];
    int nadj[MAXVT];
    int dest[MAXVT * MAXVT];
    int nvt;
    int nar;

    // especifico para caminhos minimos origem unica
    int ant[MAXVT];

    //especifico para fluxo em redes
    int cap[MAXVT * MAXVT];
    int ar_ant[MAXVT];
    int fluxo[MAXVT * MAXVT];
};
```

Edmonds-Karp

```c
void inicializa_grafo(grafo *g, int n = 0) {
    g->nvt = n;
    g->nar = 0;
    memset(g->nadj, 0, sizeof(g->nadj));
}

void insere_aresta(grafo *g, int ini, int dest, int cap) {
    g->cap[g->nar] = cap;
    g->dest[g->nar] = dest;
    g->adj[ini][g->nadj[ini]++] = g->nar++;

    g->cap[g->nar] = 0;
    g->dest[g->nar] = ini;
    g->adj[dest][g->nadj[dest]++] = g->nar++;
}
```

Edmonds-Karp

```c
int edmonds_karp(grafo *g, int ini, int fim) {
    int f, fmax;

    memset(g->fluxo, 0, sizeof(g->fluxo));
    while (busca_largura_residual(g, ini, fim)) {
        f = capacidade_residual(g, ini, fim);
        aumenta_fluxo(g, f, ini, fim);
        fmax += f;
    }
    return fmax;
}
```

```
int busca_largura_residual(grafo *g, int ini, int fim) {
    int i, a, v, w, capres;
    int marc[MAXVT];
    queue<int> q;

    memset(marc, 0, sizeof(marc));
    q.push(ini);
    marc[ini] = 1;
    while (!q.empty()) {
        v = q.front(); q.pop();
        if (v == fim) return 1;
        for (i = 0; i < g->nadj[v]; i++) {
            a = g->adj[v][i]; w = g->dest[a];
            capres = g->cap[a] - g->fluxo[a];
            if (!marc[w] && (capres > 0)) {
                marc[w] = 1;
                q.push(w);
                g->ant[w] = v;
                g->ar_ant[w] = a;
            }
        }
    }
    return 0;
}
```

Edmonds-Karp

```
int capacidade_residual(grafo *g, int ini, int fim) {
    int v, a, capres;

    v = fim;
    a = g->ar_ant[fim];
    capres = g->cap[a] - g->fluxo[a];
    while (g->ant[v] != ini) {
        v = g->ant[v];
        a = g->ar_ant[v];
        capres = capres < (g->cap[a] - g->fluxo[a]) ?
                         capres : g->cap[a] - g->fluxo[a];
    }
    return capres;
}
```

Edmonds-Karp

```
void aumenta_fluxo(grafo *g, int f, int ini, int fim) {
    int v, a;

    v = fim;
    while (v != ini) {
        a = g->ar_ant[v];
        g->fluxo[a] += f;
        g->fluxo[inv(a)] -= f;
        v = g->ant[v];
    }
}
```

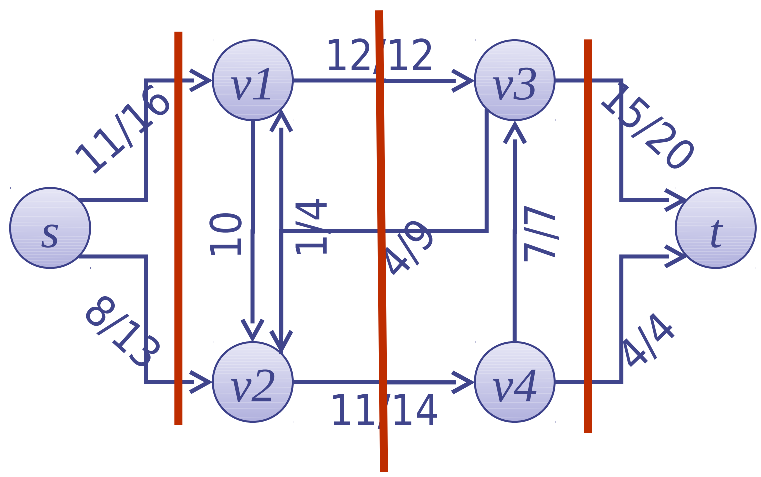
```
int inv(int a) { return a ^ 0x1; }
```

Cortes

- Um *corte* em um grafo é a *partição* dos vértices em dois conjuntos disjuntos
- Uma aresta de cruzamento conecta dois vértices em *partições* diferentes
- Um *corte-st* é um corte que coloca o vértice *s* em uma aresta e o vértice *t* em outra

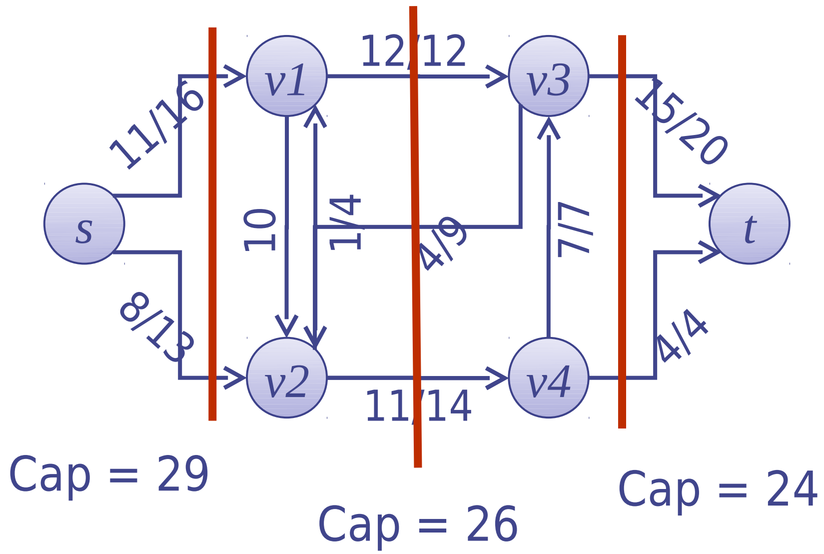
Cortes

Um corte-*st* divide um fluxo em rede em dois componentes conexos, interrompendo o fluxo de *s* para *t*



Cortes

A *capacidade* de um corte-*st* é a soma das capacidades das arestas de cruzamento *st* (forward)

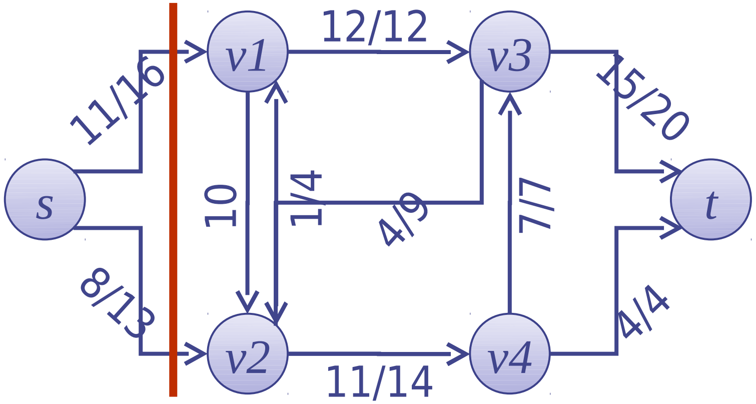


Cortes

O *fluxo* através de um corte-*st* é a diferença entre a soma do fluxo das arestas de cruzamento *st* menos o fluxo das arestas de cruzamento *ts* (*forward – backward*)

Cortes

Para qualquer fluxo de rede, o fluxo através de qualquer corte-*st* é igual ao valor do fluxo*.

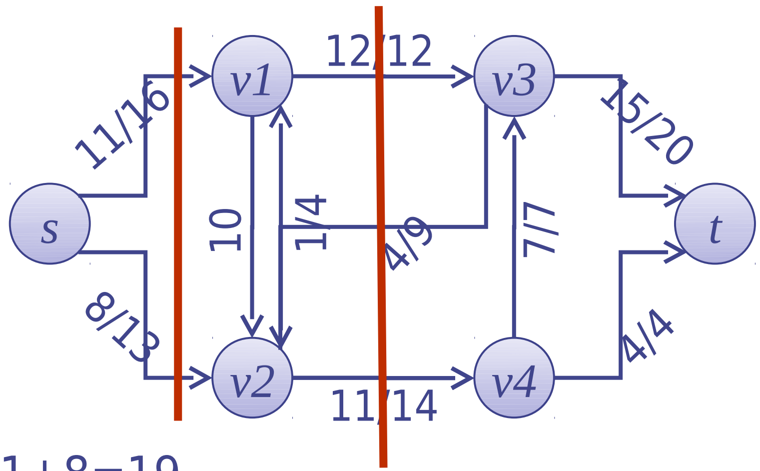


11+8=19

*Observe que este fluxo em rede não é máximo

Cortes

Para qualquer fluxo de rede, o fluxo através de qualquer corte-*st* é igual ao valor do fluxo*.



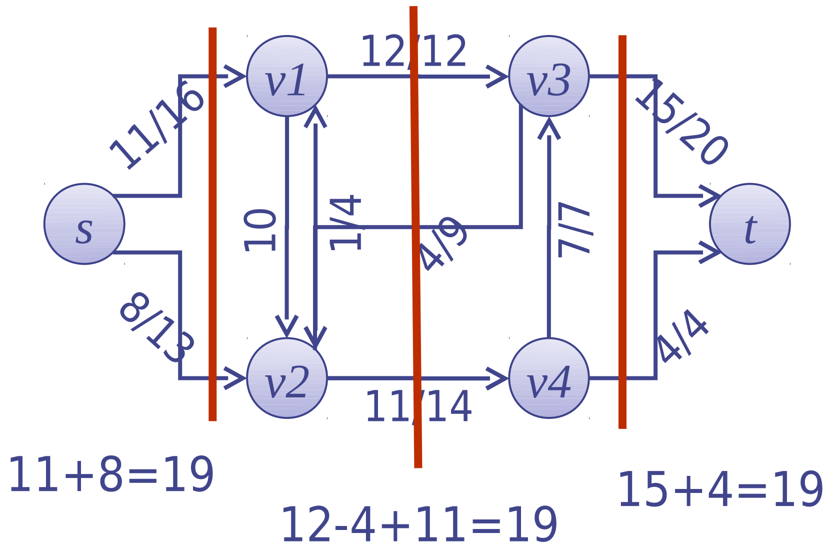
11+8=19

12-4+11=19

*Observe que este fluxo em rede não é máximo

Cortes

Para qualquer fluxo de rede, o fluxo através de qualquer corte-*st* é igual ao valor do fluxo*.



*Observe que este fluxo em rede não é máximo

Corte mínimo

No problema de corte mínimo, deve-se encontrar um corte-*st* no qual a capacidade do corte é menor do que a capacidade de qualquer outro corte-*st*.

Na verdade, resolver esse problema é igual a resolver o problema de fluxo máximo

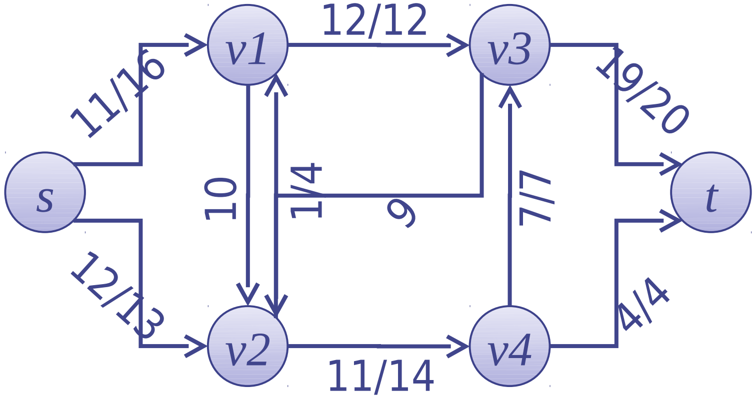
Corte mínimo

O fluxo máximo em um fluxo em rede é igual a capacidade mínima de um corte-*st*

Porque? É o corte mínimo que limita o fluxo na rede, e conseqüentemente define o fluxo máximo.

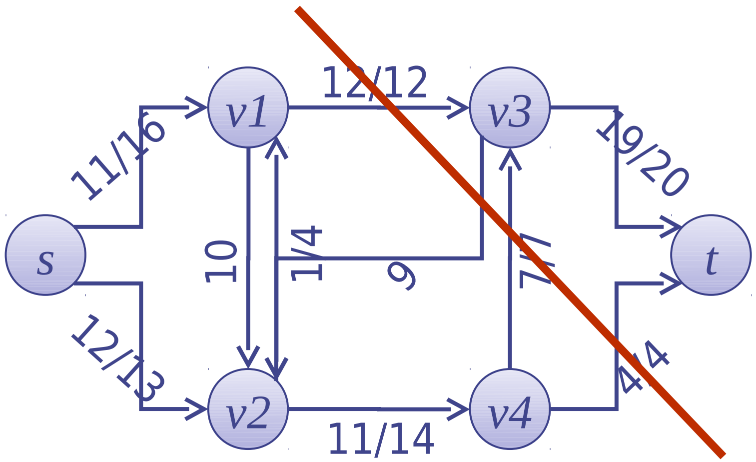
Corte mínimo

Observe a seguinte rede com fluxo máximo igual a 23. Existe um conjunto de arestas *forward* cheias que limitam o fluxo.



Corte mínimo

Observe a seguinte rede com fluxo máximo igual a 23. Existe um conjunto de arestas *forward* cheias que limitam o fluxo.



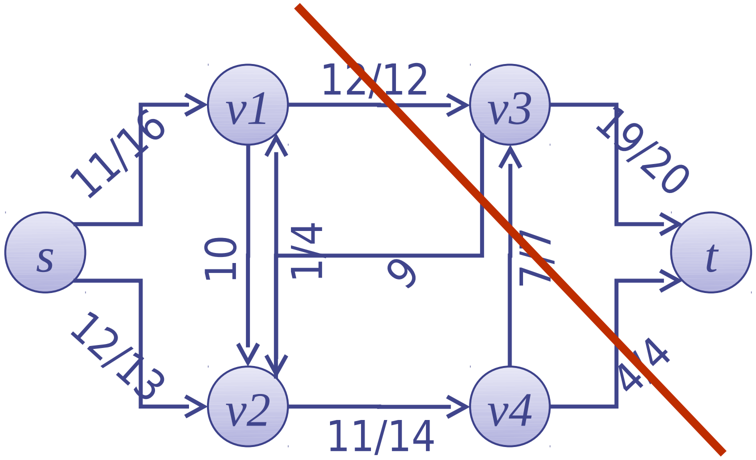
Corte mínimo

Como achar as arestas (ou vértices) que fazem parte de um corte mínimo?

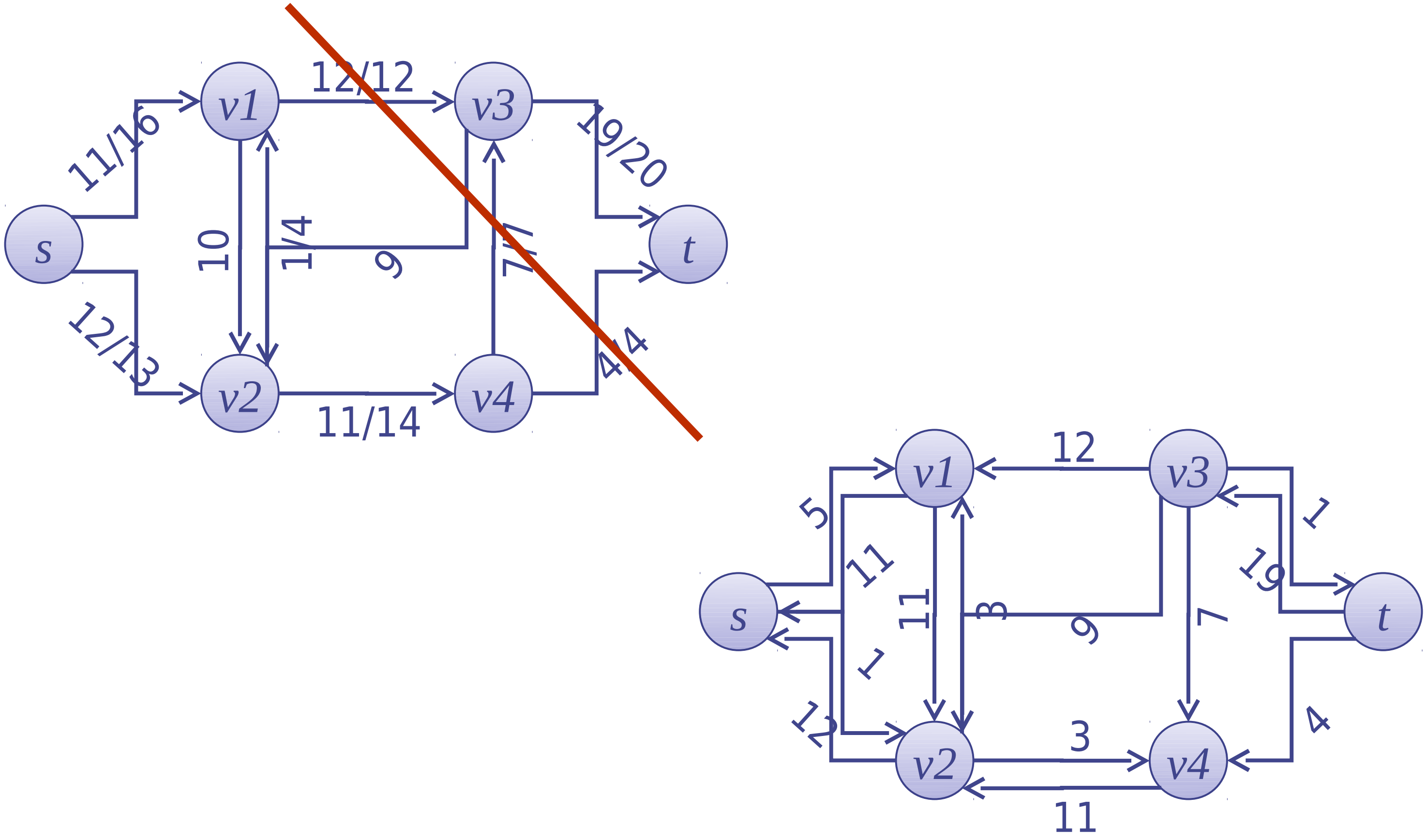
Segundo Sedgewick:

"The Ford-Fulkerson algorithms gives precisely such a flow and cut: When the algorithm terminates, identify the first full foward or empty backward edge on every path from *s* to *t* in the graph. Let *Cs* be the set of all vertices that can be reached from *s* with an undirected path that does not contain a full forward or empty backward edge and let *Ct* be the remaining vertices. Then, *t* must be in *Ct*, so (*Cs*, *Ct*) is na *st*-cut, whose cut consists entirely of full forward or empty backward edges."

Corte mínimo



Corte mínimo



Corte Mínimo

- Dessa forma, podemos utilizar uma busca em largura sobre a rede residual a partir de *s*
- Os vértices *u*-*v* que pertencem ao corte mínimo são aqueles que *u* foi marcado na busca em largura e *v* não foi marcado
- Veja que não é preciso fazer uma chamada adicional a busca em largura. Pode-se utilizar o resultado da última busca realizada pelo algoritmo de Edmonds-Karp

Fluxos de **custo mínimo**

- Com frequência o problema de fluxo máximo possui mais de uma solução
- Nessas situações pode-se estar interessado em encontrar, entre as soluções, uma que possua uma característica especial:
  - Um fluxo máximo com a menor quantidade de arestas, ou
  - Um fluxo máximo de menor caminho

Fluxos de custo mínimo

Esses problemas são considerados mais difíceis que o problema de fluxo máximo e caem em uma categoria genérica de *problemas de fluxo de custo mínimo*

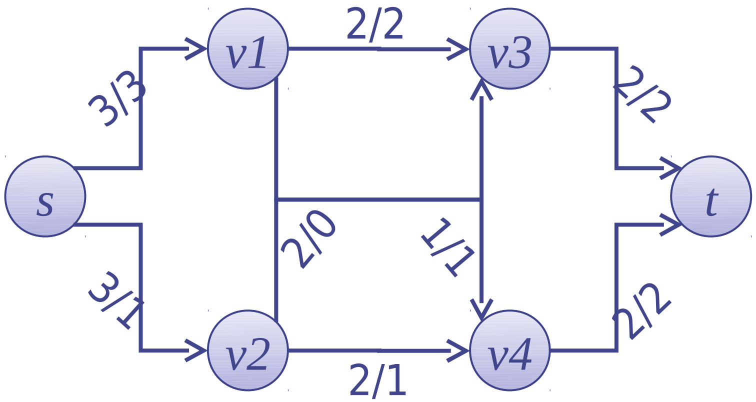
Nesses problemas especificamos que cada aresta possui também um custo associado. O objetivo é encontrar entre todas as soluções de fluxo máximo, aquela que possui o menor custo

Fluxos de custo mínimo

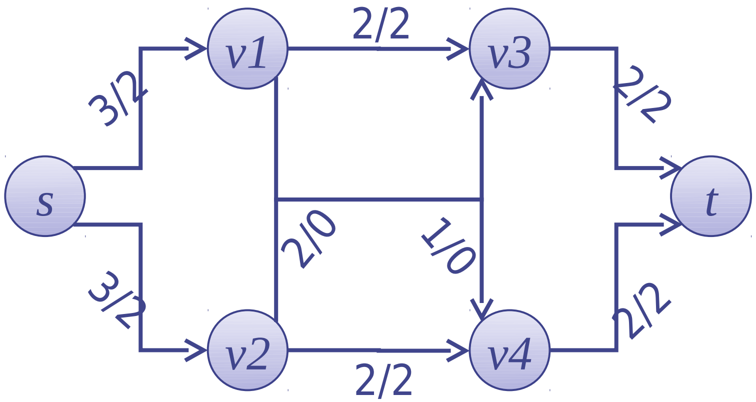
O *custo do fluxo* de uma aresta em uma rede de fluxo é o **produto** do fluxo da aresta pelo seu custo

O *custo de um fluxo* é a soma dos custos dos fluxos das arestas que pertencem ao fluxo

Fluxos de custo mínimo

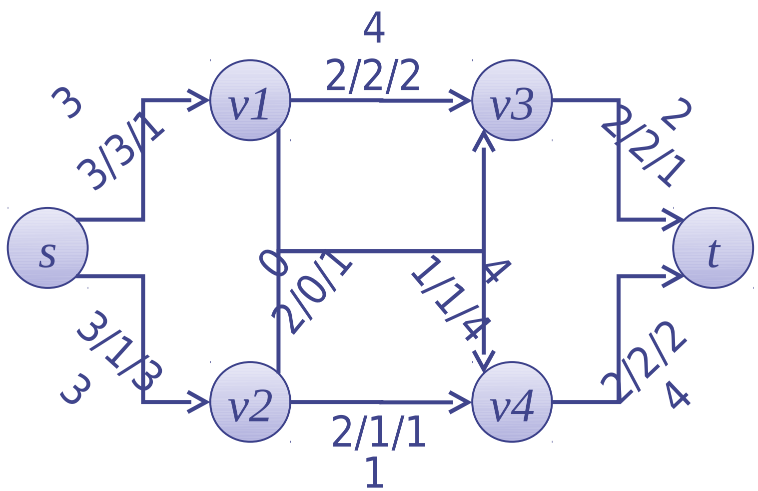


Fluxo = 4

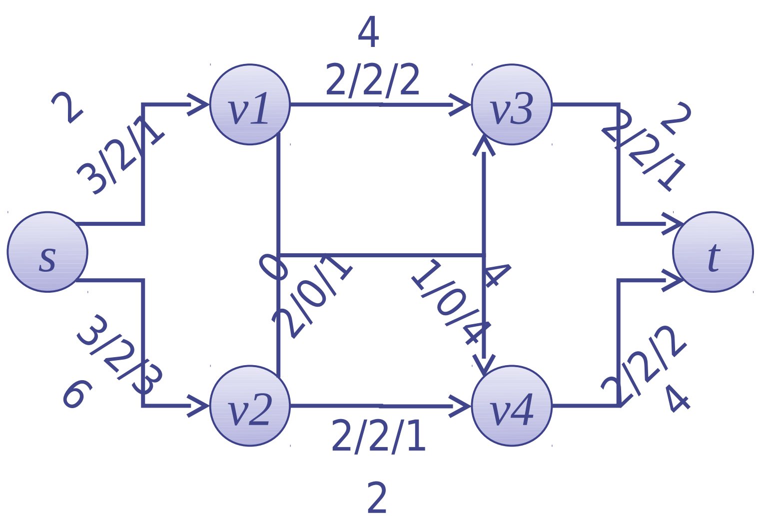


Fluxo = 4

Fluxos de custo mínimo



Fluxo = 4
Custo = 21

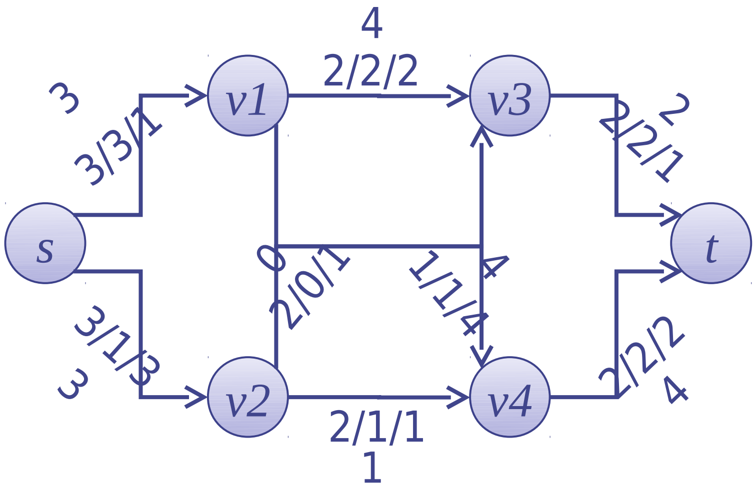


Fluxo = 4
Custo = 20

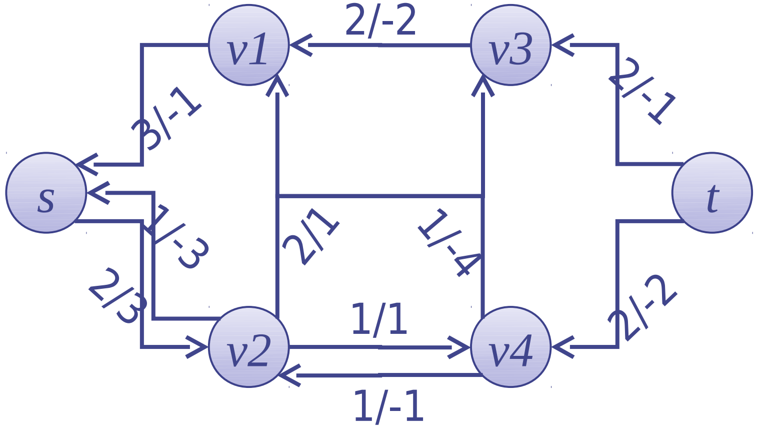
Fluxos de custo mínimo

- Para custo mínimo, precisamos redefinir a rede residual:
  - Ele tem as mesmas características da rede para problemas de fluxo máximo, entretanto adicionamos uma informação: se o fluxo é positivo, então a aresta de retorno tem custo $-x$. Se o fluxo é menor do que a capacidade, então a aresta de ida tem custo x. Nos demais casos a aresta (ida ou retorno) não é representada.

Fluxos de custo mínimo



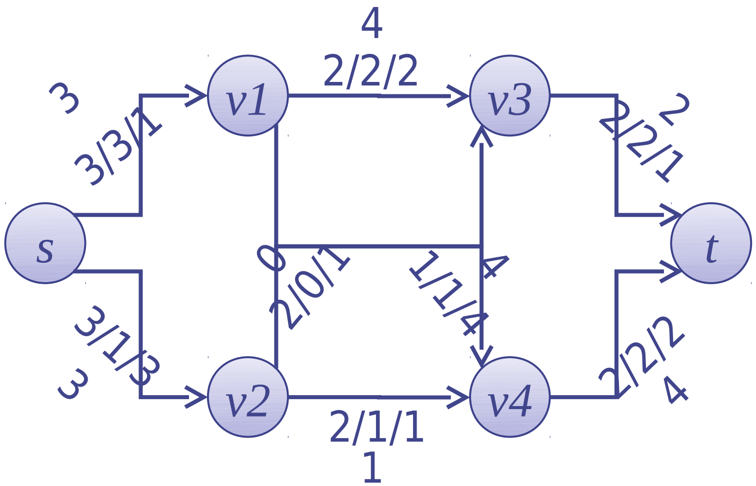
Fluxo = 4
Custo = 21



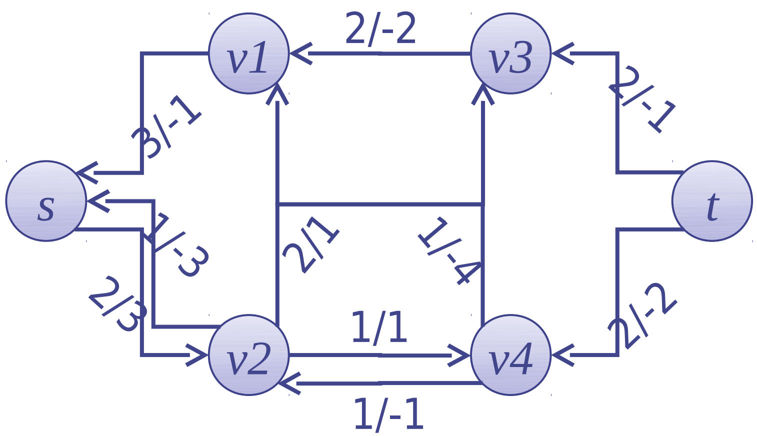
Fluxo de custo mínimo

Um fluxo máximo é um **fluxo máximo de custo mínimo** se e somente se a sua rede residual não contém ciclos negativos direcionados

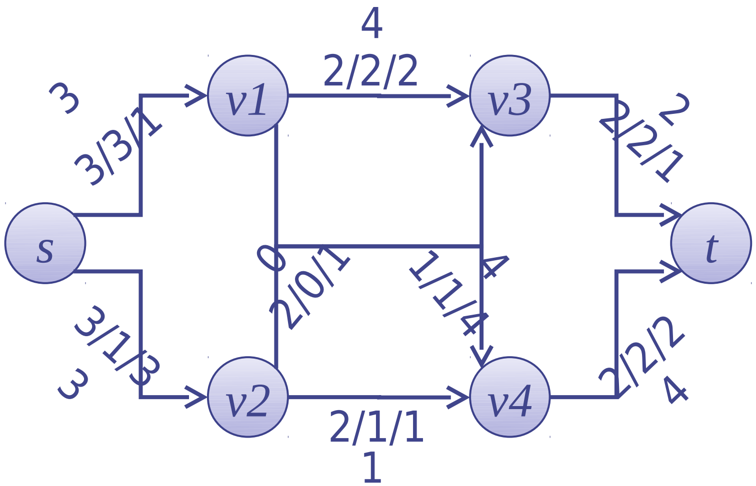
Fluxos de custo mínimo



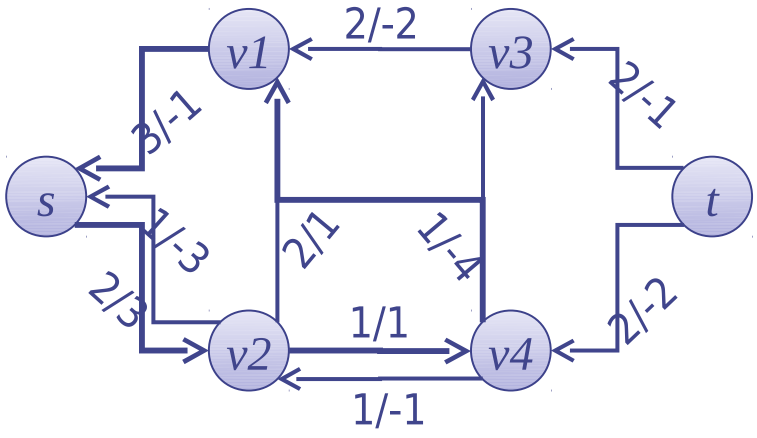
Fluxo = 4
Custo = 21



Fluxos de custo mínimo



Fluxo = 4
Custo = 21

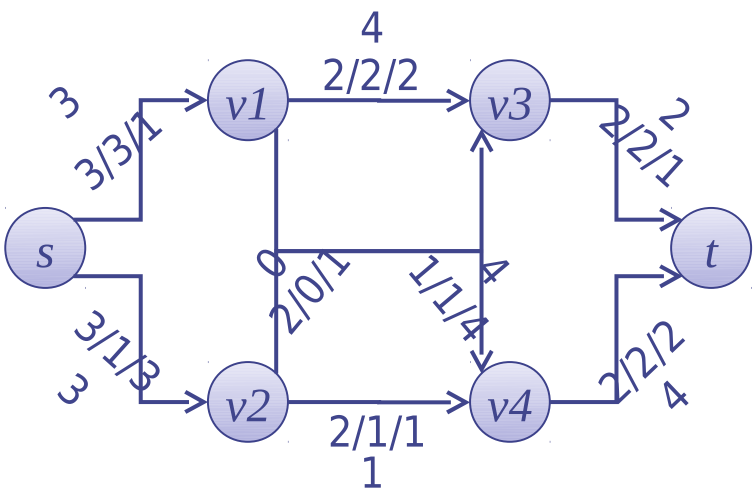


Fluxo de custo mínimo

Seja *x* a capacidade mínima das arestas no ciclo, podemos adicionar *x* às arestas de ida e –x às arestas de volta:

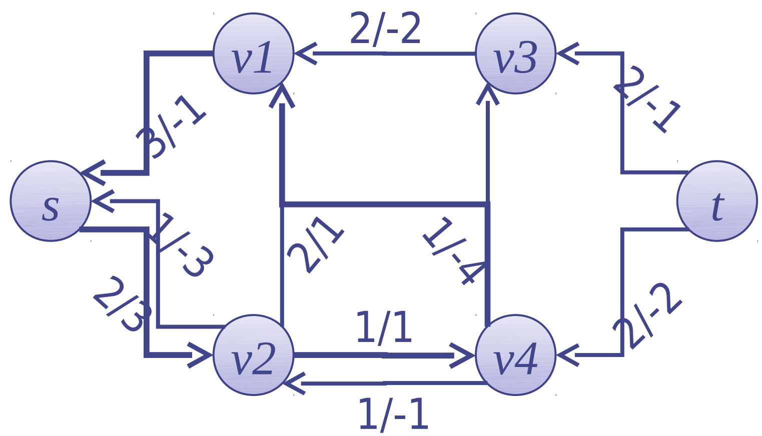
- Isso não altera a diferença de fluxo de entrada/saída de cada vértice
- Mas modifica o custo da rede em *x* vezes o custo do ciclo

Fluxos de custo mínimo

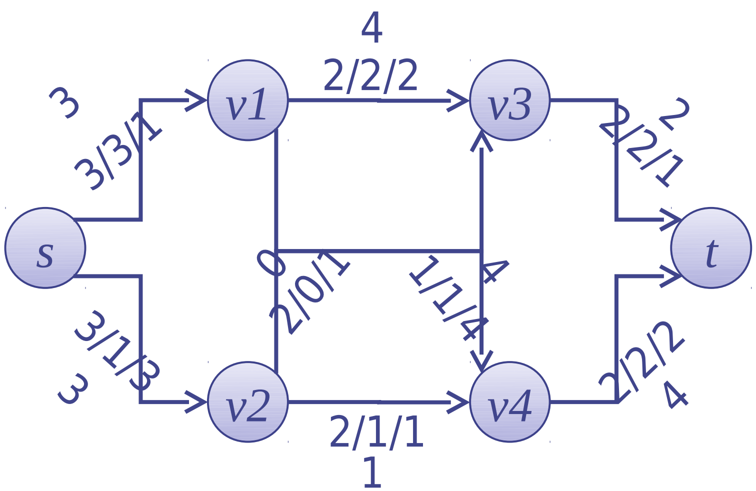


Fluxo = 4
Custo = 21

Custo do ciclo = -1
Capacidade mínima = 1

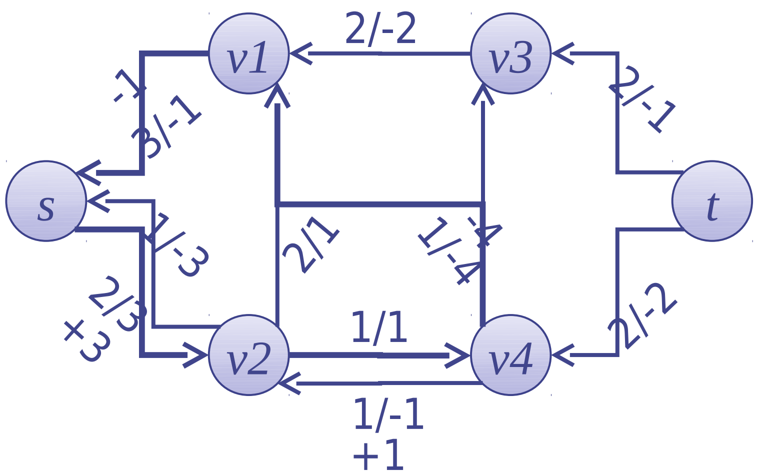


Fluxos de custo mínimo

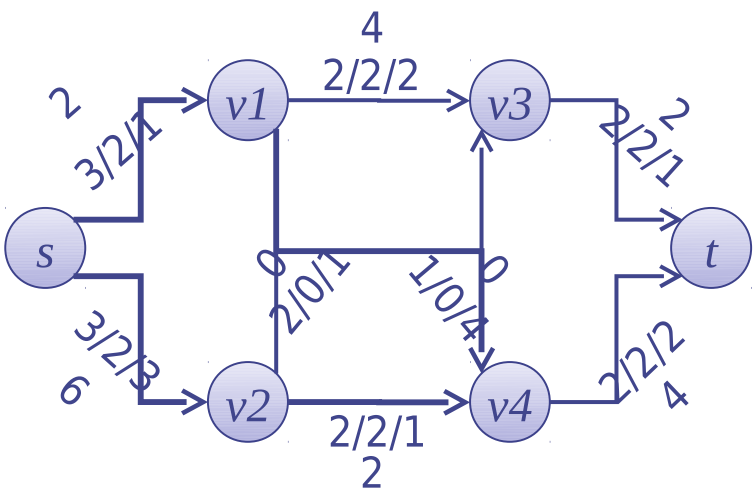


Fluxo = 4
Custo = 21

Custo do ciclo = -1
Capacidade mínima = 1

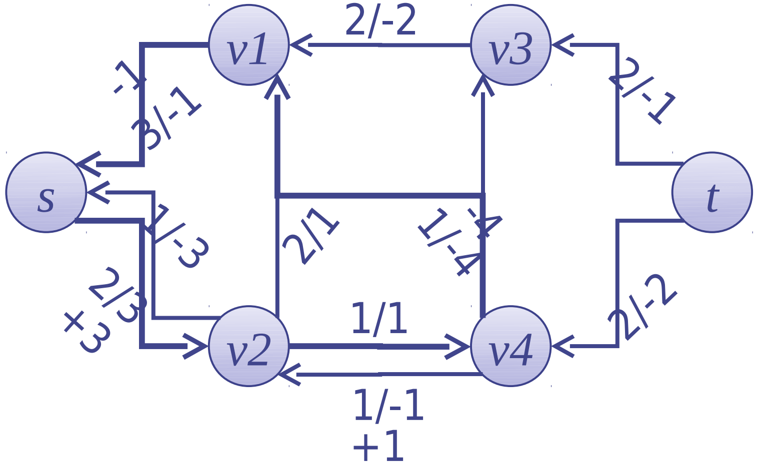


Fluxos de custo mínimo



Fluxo = 4
Custo = 20

Custo do ciclo = -1
Capacidade mínima = 1



Fluxos de custo mínimo

O algoritmo para encontrar um fluxo máximo de custo mínimo é:

1. Encontrar um fluxo máximo
2. Procurar por um ciclo negativo, caso não exista terminar
3. Aumentar o fluxo no ciclo negativo e voltar ao passo 2

```
void maxflow_mincost(grafo *g, int ini, int fim) {
    int a, x, w, capres;

    edmonds_karp(g, ini, fim);
    bellman_ford_residual(g, ini);
    while ((x = bellman_ford_cycle_test_mincost()) != -1) {
        a = g->ar_ant[x];
        capres = g->cap[a] - g->fluxo[a];
        for(w = g->dest[a]; w != x; w = dest[a]) {
            a = ar_ant[w];
            aux = g->cap[a] - g->fluxo[a];
            if (aux < capres) capres = aux;
        }
        a = g->ar_ant[x];
        g->fluxo[a] += capres; g->fluxo[inv(a)] -= capres;
        for(w = g->dest[a]; w != x; w = dest[a]) {
            a = ar_ant[w];
            g->fluxo[a] += capres; g->fluxo[inv(a)] -= capres;
        }
        bellman_ford_residual(g, ini);
    }
}
```