

Marcio Lima Inácio

---

# Introdução à Otimização Combinatória Aplicada

### Aula 1 Otimização Combinatória

Definição: Problema de Otimização

- Entrada (instância)
- Conjunto de soluções viáveis
  - Soluções válidas
  - Respeitam as restrições do problema

Quero achar a melhor solução

- Função objetivo
  - Associa um valor real a cada solução viável

Melhor: maximização ou minimização – De acordo com a função objetivo

Otimização Combinatória

- Variáveis discretas
- O conjunto de soluções viáveis é finito

Problema do escalonamento

Dadas  $n$  tarefas, cada uma com uma duração, aloca-las em  $m$  máquinas minimizando a maior soma de tempos (*makespan*).

Instâncias

$$m = 2$$



$t_1$



$t_2$

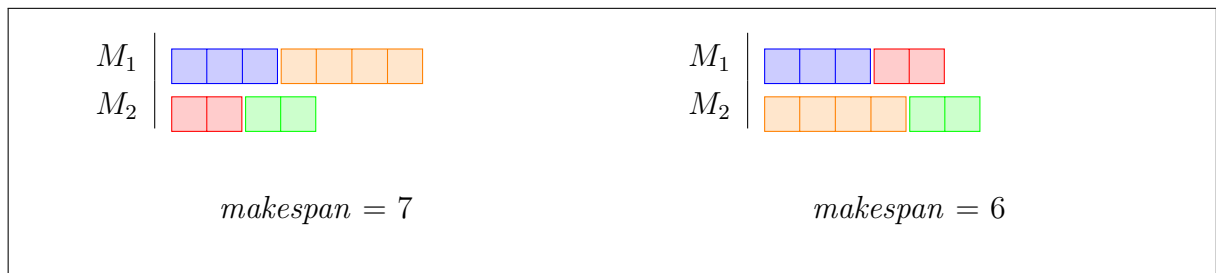


$t_3$



$t_4$

Exemplos de soluções



## Aula 2 Definições básicas

### 2.1 Grafos

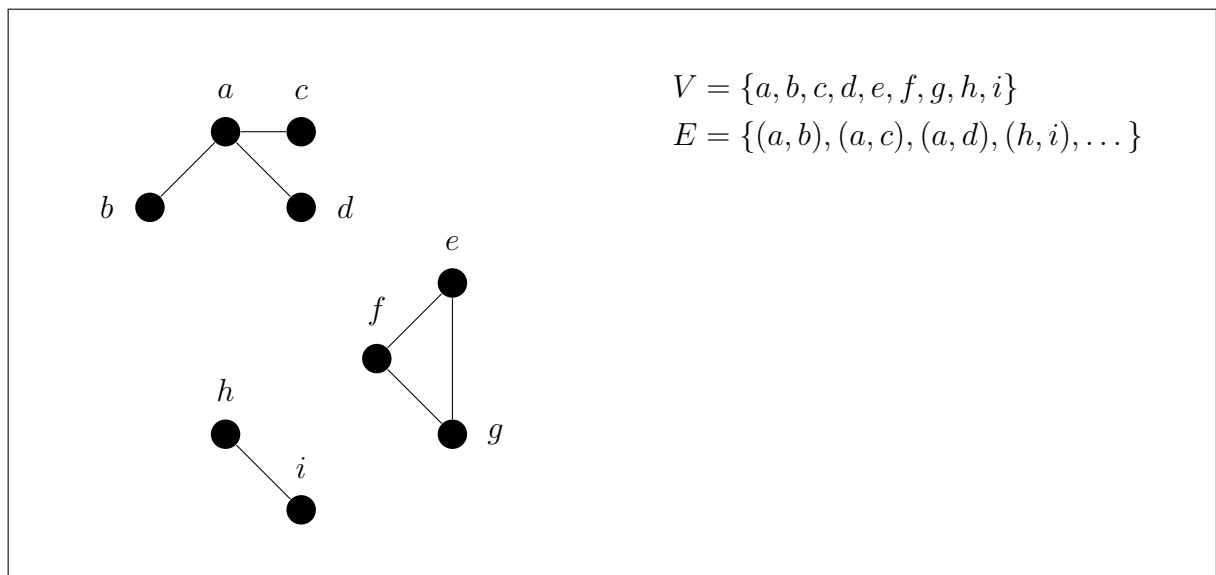
Estrutura matemática que representa relacionamentos par-a-par (arestas) entre objetos (vértices).

Formalização:  $G = (V, E)$

$V$  = conjunto de vértices

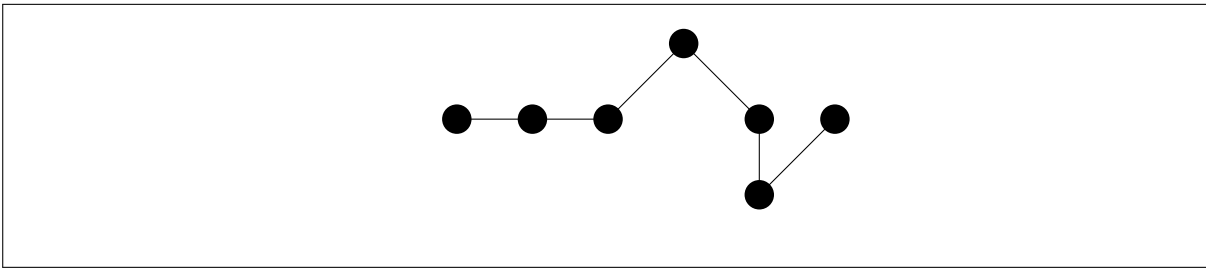
$E$  = conjunto de arestas

$E$  é um conjunto de pares.



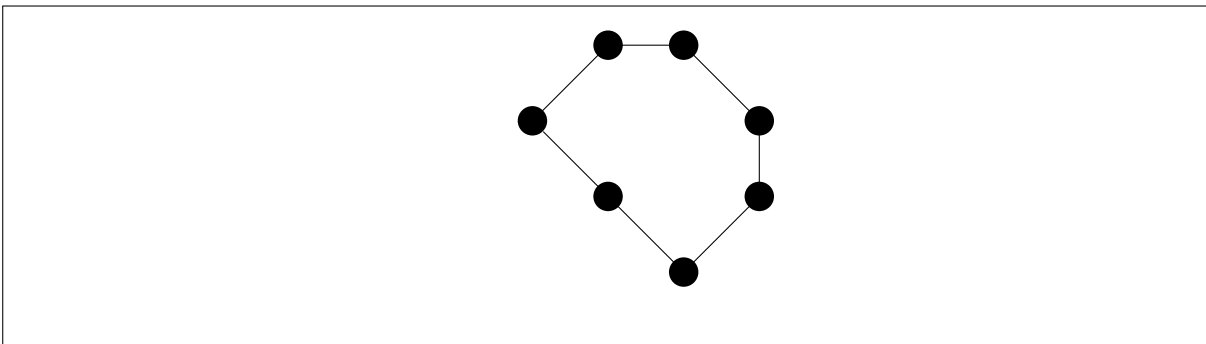
Tipos comuns de grafos

- Caminho
  - Conexo
  - Não possui bifurcação



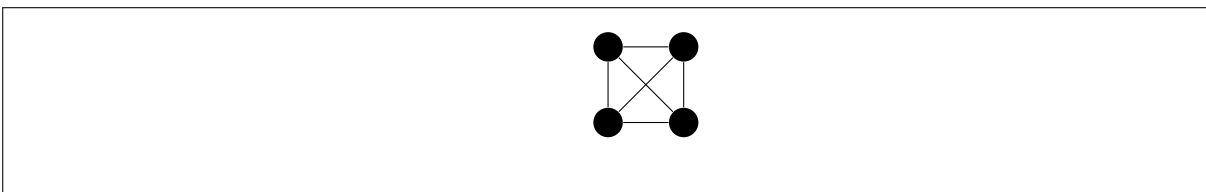
- Circuito

- Fecha o ciclo



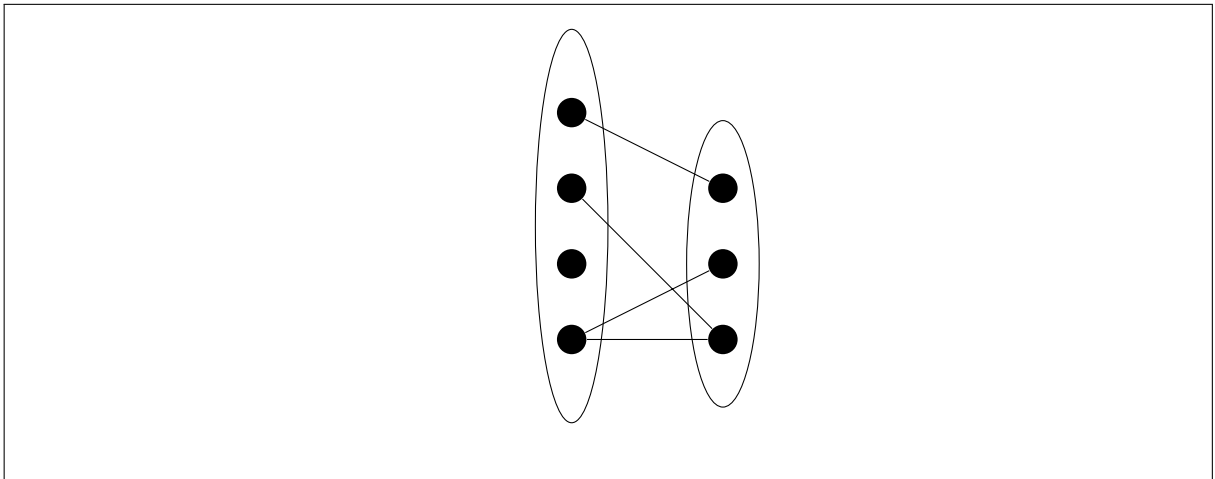
- Completo

- Todos os nós conectados entre si
- Representado por  $K_n$ , sendo  $n$  o número de vértices



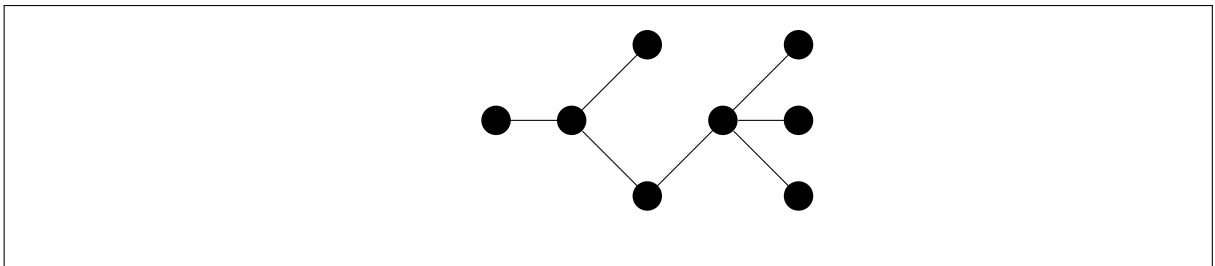
- Bipartidos

- Dois subconjuntos de vértices sem arestas entre split
- Tipos de entidades distintas



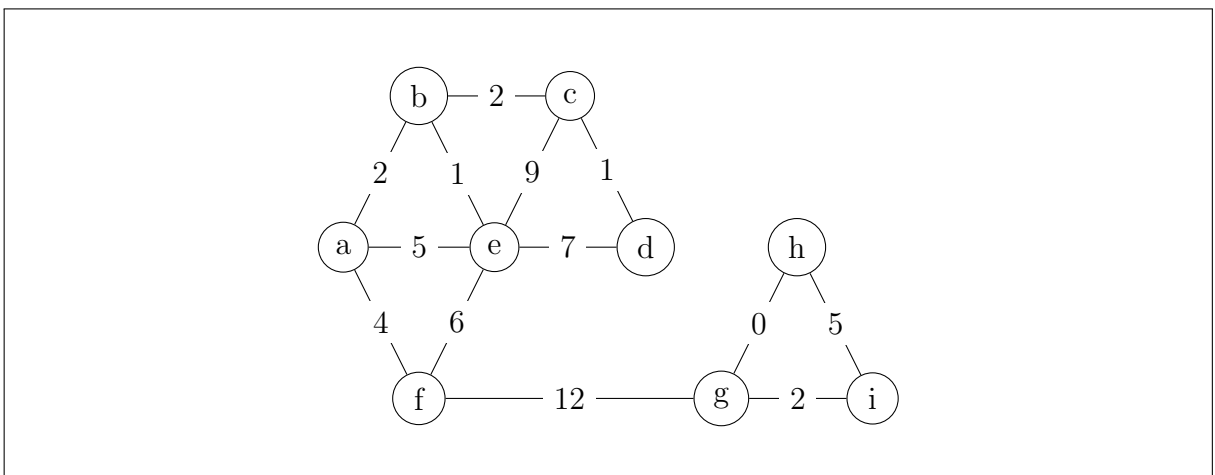
- Árvores

- Grafos conexos e acíclicos
- Caminhos são um tipo específico de árvore
- Para qualquer par de nós  $a$  e  $b$ , existe um único caminho conectando  $a$  a  $b$



Os grafos podem ser orientados (ou dirigidos) ou não-orientados. No caso dos grafos orientados, as arestas (também chamadas de arcos) têm uma direção.

Os grafos também podem ter pesos (ou custos) associados às arestas.



Os pesos são formalizados como uma função separada:  $f : E \mapsto \mathbb{R}$ .

## 2.2 Análise de algoritmos

Mostra que

- Um algoritmo está correto
- Estimar o tempo de execução

É uma análise matemática, então não é preciso implementar o algoritmo.

Notação assintótica: termos de menor ordem e constantes são de desconsiderados.

$$20n^2 + 500n \rightarrow O(n^2)$$

## Aula 3 Exemplos de problemas

### 3.1 Árvore de Steiner

- **Entrada:**  $G = (V, E)$  com  $V = R \cup S$ , sendo  $R$  terminais e  $S$  vértices de Steiner, e função  $w$  de peso na arestas.
- **Soluções viáveis:** árvores que conectam todos os vértices em  $R$ .
- **Função objetivo:** soma dos pesos das arestas na árvore.
- **Objetivo:** encontrar uma árvore de peso mínimo.

### 3.2 Bin Packing/Empacotamento

- **Entrada:** conjunto  $L = \{1, \dots, n\}$  de itens retangulares, item  $i$  com largura  $w_i$  e altura  $h_i$ , largura  $W$  e altura  $H$  do recipiente retangular.
- **Soluções viáveis:** partição  $L_1, L_2, \dots, L_q$  de  $L$  tal que os itens em  $L_k$  cabem num recipiente  $W \times H$ .
- **Função objetivo:** número  $q$  de recipientes (*bins*) utilizados.
- **Objetivo:** encontrar solução de custo mínimo.

### 3.3 Caixeiro Viajante (TSP - *Traveling Salesman Problem*)

- **Entrada:**  $G = (V, E)$  e função  $w$  de peso nas arestas.
- **Soluções viáveis:** circuitos hamiltonianos (passam por todos os vértices sem repetição) de  $G$ .
- **Função objetivo:** soma dos pesos das arestas do circuito.
- **Objetivo:** encontrar o circuito de menor custo

### 3.4 Problema da Mochila

- **Entrada:** conjunto de  $n$  itens, cada item  $i$  tem peso  $w_i$  e valor  $v_i$  e tamanho  $W$  da mochila.
- **Soluções viáveis:** conjuntos de itens  $S \subseteq \{1, \dots, n\}$  com  $\sum_{i \in S} w_i \leq W$ .
- **Função objetivo:** soma dos valores dos itens em  $S$ .
- **Objetivo:** encontrar uma solução de valor máximo.

## Aula 4 Dificuldades

Primeira abordagem de resolução de problemas: busca por força bruta.

O espaço de busca é finito, então dá para enumerar todas as soluções guardando a melhor encontrada. Com tempo suficiente, resolve o problema.

Mas não explora as estruturas combinatórias do problema.  $\rightarrow$  Muito esforço.

Com um espaço muito grande, fica inviável usar.

No TSP qualquer sequência dos  $n$  vértices é candidata a solução.

Algoritmo:

1. Gere as  $n!$  sequências de vértices
2. Cada sequência é um circuito hamiltoniano
3. Calcule seu custo e compare com o melhor já encontrado

**São  $(n - 1)!$  sequências no espaço de busca.**

No problema da mochila, qualquer subconjunto dos  $n$  elementos é candidato a solução.

Algoritmo:

1. Gere os  $2^n$  possíveis subconjuntos
2. Para cada um, teste se os itens cabem na mochila  $\rightarrow$  Viabilidade
3. Se couberem, calcule o custo da solução e compare com o melhor já encontrado

**Crescimento exponencial**

## 4.1 Complexidade

- Algoritmo eficiente: complexidade de tempo no pior caso é polinomial no tamanho  $n$  da entrada  $\rightarrow O(n^k)$  com  $k$  constante.
- Problemas de decisão: Problemas com resposta sim ou não.

Classes de problemas:

- **Classe P:** problemas de decisão que possuem algoritmos eficientes.
- **Classe NP:** problemas de decisão cuja solução ("resposta sim") pode ser verificada em tempo polinomial.
- **Classe NP-Completo:** problemas  $Q$  tais que  $Q \in NP$  e todo problema em NP é redutível a  $Q$ .

### Redução

Um problema  $A$  é redutível a  $B$  se podemos utilizar um algoritmo que resolve  $B$  para resolver  $A$ , ou seja,  $B$  é pelo menos tão difícil quanto  $A$ .

Não há um certificado de dificuldade absoluto: "esse problema não é possível de resolver". Existe a dificuldade relativa: "esse problema é tão complexo quanto aquele".

- **Classe NP-Difícil:** problemas  $Q$  tais que todo problema em  $NP$  é redutível a  $Q$ . Esses problemas não precisam ser necessariamente  $NP$  ( $\neq NP$ -Completo).

## 4.2 Abordagens

Se  $P \neq NP$ , não é possível ter algoritmos para problemas  $NP$ -Difíceis que:

- Encontrem soluções ótimas
- Em tempo polinomial
- Para qualquer entrada

Tem que abrir mão de alguma característica

- Métodos exatos
  - Não funcionam em tempo hábil (exponencial ou superpolinomial)
  - Ao contrário da força bruta, explora as estruturas combinatórias para eliminar pedaços do espaço de busca para facilitar
- Heurísticas



- Não encontra necessariamente soluções ótimas
- Procura soluções “boas o bastante”
- Avaliação empírica com o uso de *benchmarks* de instâncias
- Algoritmos de aproximação
  - Não encontra necessariamente soluções ótimas (subconjunto das heurísticas)
  - Garantia de que o algoritmo é polinomial
  - Garantia de que o resultado vai estar dentro de uma margem de erro da solução ótima
- Parametrização
  - Não funciona para todas as instâncias
  - Um parâmetro é fixado
  - Garantia de encontrar a solução ótima para as instâncias com o parâmetro fixo

## Aula 5 Métodos exatos

- Procuram a solução ótima
- Considera as estruturas combinatórias do problema → Diferença da força bruta
- Não garante tempo polinomial no pior caso (Problemas NP-Difíceis)

Exemplos de métodos exatos

- Algoritmos gulosos
- Programação dinâmica
- ***Branch and bound***
- **Programação linear / Programação linear inteira**
- Programação por restrições ×

### 5.1 Algoritmos gulosos

Realizam decisões melhores em curto prazo, esperando que isso leve ao resultado ótimo.

Existem casos que um algoritmo guloso resolve o problema em tempo polinomial para todas as instâncias. Exemplos:

- Caminho mínimo → Dijkstra

- MST  $\rightarrow$  Prim e Kruskal
- Compressão de Dados  $\rightarrow$  Huffman
- Mochila Fracionária

Para problemas NP-Difíceis, dá para usar algoritmos gulosos como heurísticas.

### 5.1.1 Mochila Fracionária

Os itens que são colocados na mochila podem ser divisíveis. Pode pegar frações de itens.

Objetivo: pegar os itens com o maior custo-benefício. Razão  $\frac{\text{valor}}{\text{peso}}$ . Garantir que ninguém melhor está fora da mochila.

$W = 50$			Ordem decrescente: 1, 2, 3
			Fração item 1: 100%
$v_1 = 60$	$w_1 = 10$	$\frac{v_1}{w_1} = 6$	Espaço livre: $50 - 10 = 40$
			Fração item 2: 100%
$v_2 = 100$	$w_2 = 20$	$\frac{v_2}{w_2} = 5$	Espaço livre: $40 - 20 = 20$
			Fração item 3: $\frac{20}{30} = 66\%$
$v_3 = 120$	$w_3 = 30$	$\frac{v_3}{w_3} = 4$	Espaço livre: $20 - 20 = 0$

**Função MochilaFrac**( $n$  : itens,  $w$  : peso,  $v$  : valor,  $W$  : capacidade)

Ordene e renomeie os itens para que  $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$ ;  
 Seja  $q$  um inteiro tal que  $X = \sum_{i=1}^q w_i \leq W$  e  $\sum_{i=1}^{q+1} w_i > W$   
 Cabe ainda uma fração  $\frac{W-X}{w_{q+1}}$  do item  $q+1$   
**retorna**  $\frac{v_1}{w_1} + \frac{v_2}{w_2} + \dots + \frac{v_q}{w_q} + \frac{W-X}{w_{q+1}} \frac{v_{q+1}}{w_{q+1}}$

**fim**

## 5.2 Branch and Bound

Busca exaustiva inteligente: enumeração (*branch*) e poda (*bound*).

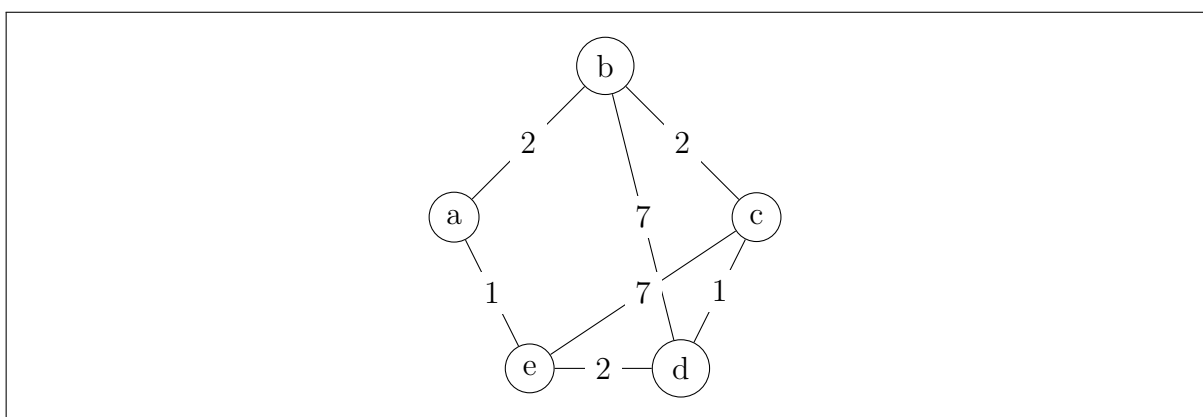
Cria uma árvore de enumeração das soluções e, elimina ramos pouco promissores (não percorre esses ramos).

- Como fazer a enumeração?
- Como percorrer a árvore?
- Como podar?

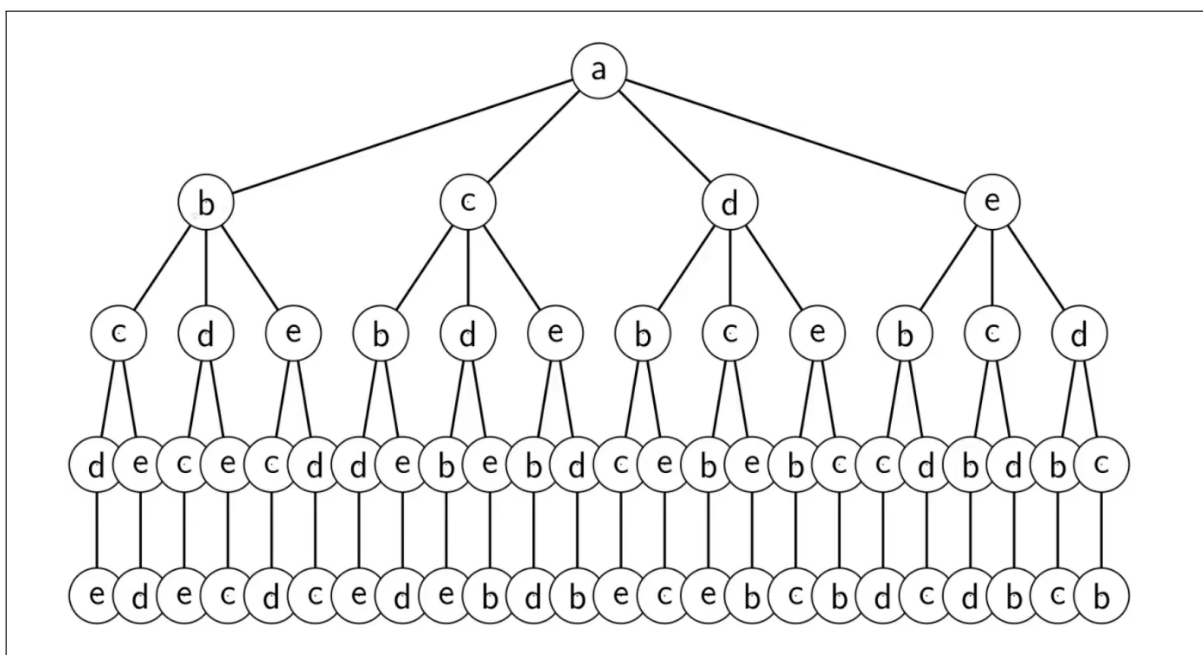
### 5.2.1 Caixeiro Viajante

Enumerar todas as sequências de vértices sendo que

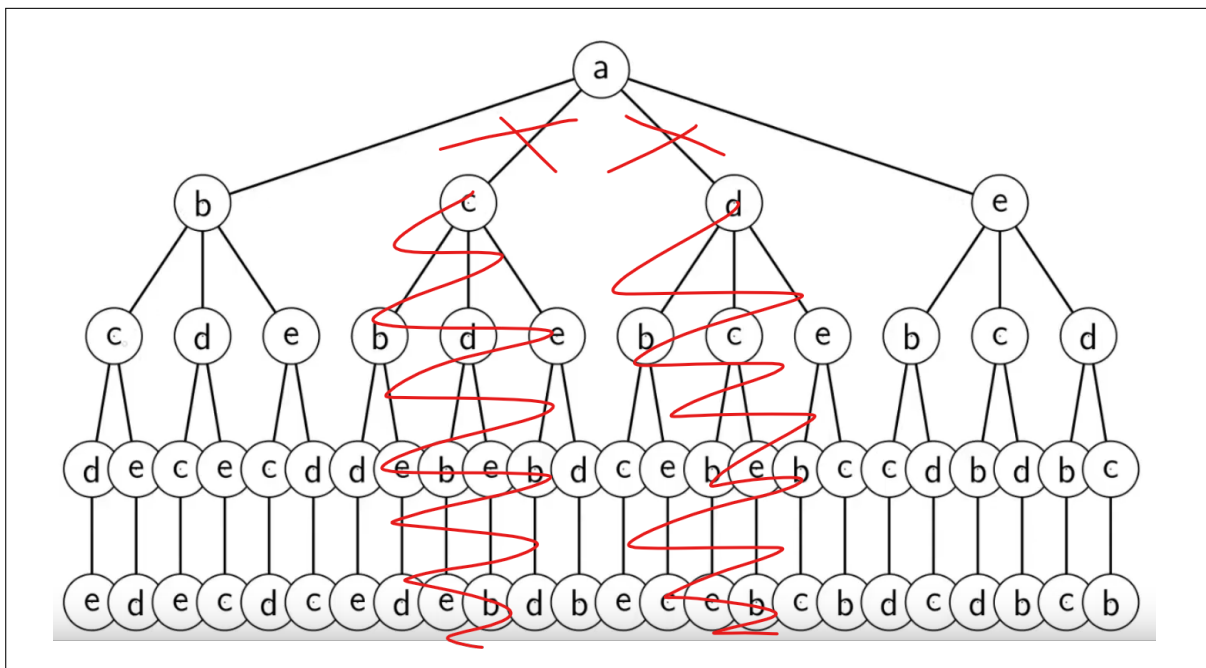
- Cada nó da árvore representa um nó do grafo original
- Cada ramificação na árvore representa uma aresta percorrida



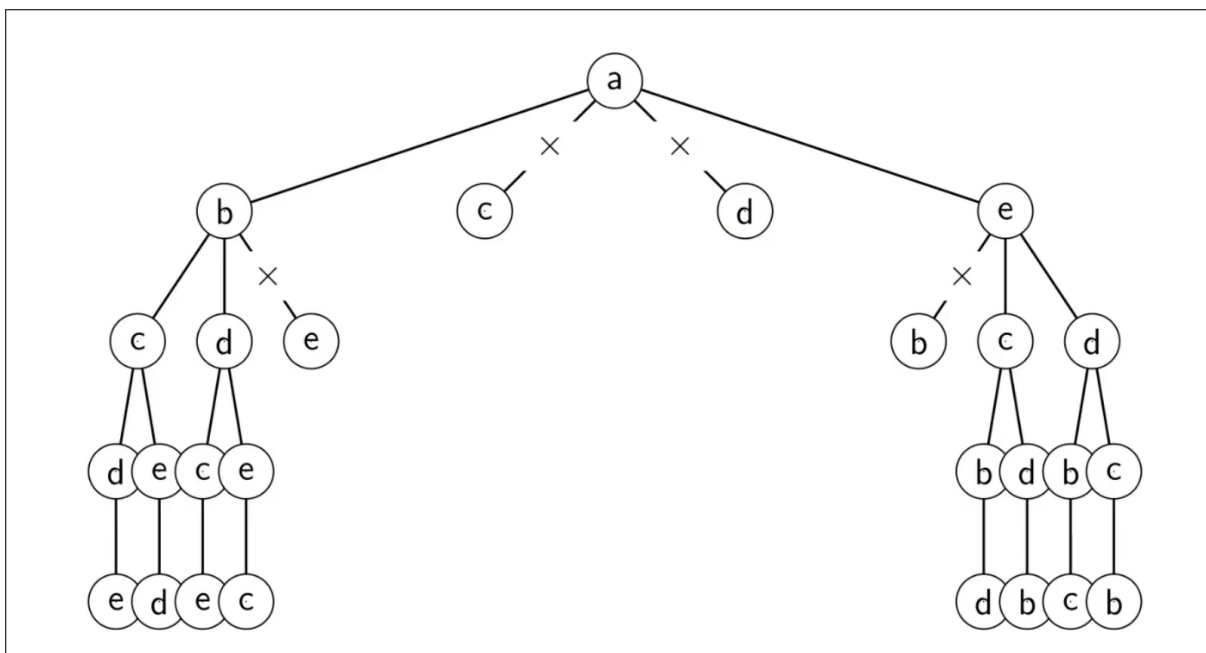
Árvore de enumeração:



Pode podar aqueles ramos que indicam um caminho impossível (sem aresta)



Mais podas podem ser feitas

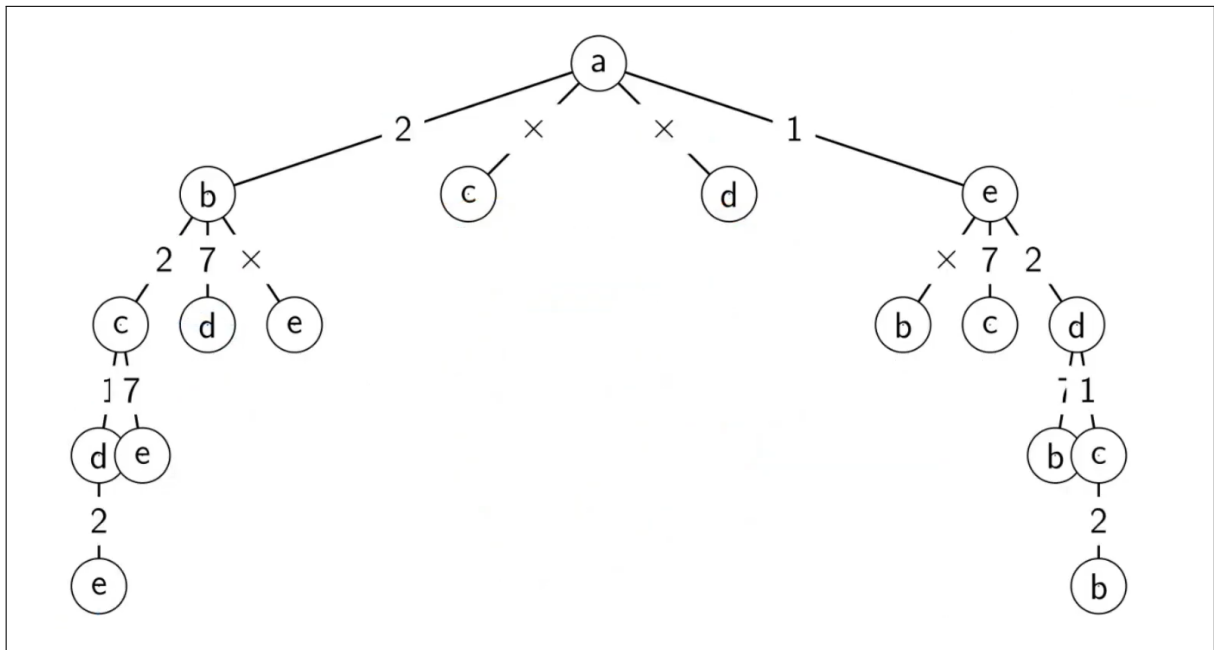


Levando os custos das arestas em consideração:

O caminho  $a \xrightarrow{2} b \xrightarrow{2} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{1} a$  tem custo total 8.

O caminho  $a \xrightarrow{2} b \xrightarrow{7} d$  tem custo 9 por si só. Então tudo abaixo vai ser pior que o caminho anterior, não é promissor usar.

Com todos esses caminhos removidos, a árvore final que precisa ser percorrida fica:



### 5.3 Programação linear (inteira)

Modelo matemático de otimização.

- Conjunto variáveis relacionamentos
- Conjunto de restrições  $\rightarrow$  inequações lineares
- Função objetivo  $\rightarrow$  expressão linear

Sempre trabalhar com constantes multiplicando variáveis e soma desses valores.

Uma solução é viável se todas as restrições são atendidas.

A forma padrão para um problema de minimização com  $n$  variáveis e  $m$  restrições é:

$$\begin{aligned} & \text{minimizar } \sum_{j=1}^n c_j x_j \\ & \text{sujeito a } \sum_{j=1}^n a_{ij} x_j \geq b_i \forall i \in \{1, \dots, m\} \\ & x_j \geq 0 \forall j \in \{1, \dots, n\} \end{aligned}$$

$a_{ij}$ ,  $b_i$  e  $c_j$  são constantes.

$x_j$  são variáveis.

Em um programa linear inteiro, as variáveis são inteiras.

### 5.3.1 Problema da Mochila

Sejam  $x_i$  variáveis binárias que indicam se o item  $i$  foi escolhido.

$$\begin{aligned} & \text{maximizar } \sum_{i=1}^n v_i x_i \\ & \text{sujeito a } \sum_{i=1}^n w_i x_i \leq W \\ & x_i \in \{0, 1\} \forall i \in \{1, \dots, n\} \end{aligned}$$

Programas lineares podem ser resolvidos em tempo polinomial  $\rightarrow$  Mochila fracionária.

Programas lineares inteiros são, normalmente, NP-Difíceis.

Resolvedores de PLI usam o *branch and bound* junto com PL.

Roda PL dentro de cada nó para encontrar limitantes inferiores para aquele ramo inteiro. Depois faz a poda.

## Aula 6 Heurísticas

Algoritmos que encontram uma solução viável.

**Não** garante que seja solução ótima.

Tendem a ser mais rápidos.

Duas categorias:

- Construtivas: Constroem uma solução viável
- Busca local: Partem de uma solução inicial e tentam melhorar através de modificações

### 6.1 Heurística construtiva para a Mochila

**Função** MochilaGuloso( $n$  : itens,  $w$  : peso,  $v$  : valor,  $W$  : capacidade)

ordene e renomeie os itens para que  $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$ ;

seja  $q$  um inteiro tal que  $\sum_{i=1}^q w_i \leq W$  e  $\sum_{i=1}^{q+1} w_i > W$ ;

**retorna**  $v_1 + v_2 + \dots + v_q$

**fim**

Exemplo que não funciona:

$$W = B$$

$$v_1 = 2 \quad w_1 = 1 \quad \frac{v_1}{w_1} = 2$$

$$v_2 = B \quad w_2 = B \quad \frac{v_2}{w_2} = 1$$

Ordem decrescente: 1, 2

Fração item 1: 100%

Espaço livre:  $B - 1$

Fração item 2: 0%

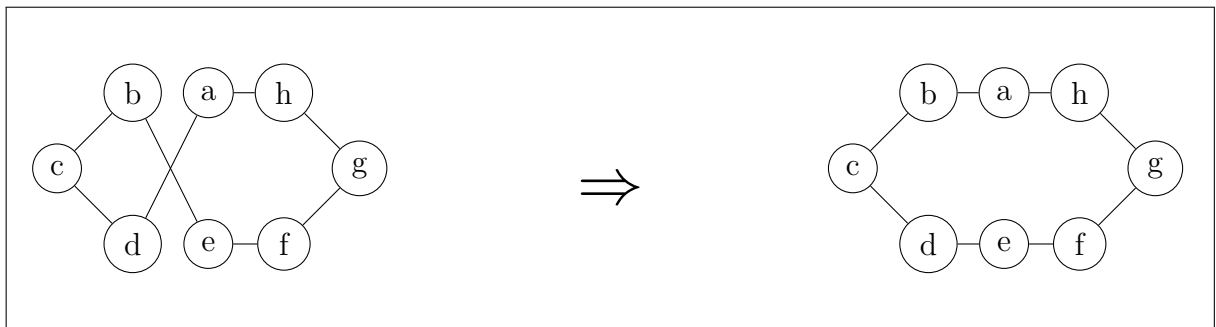
A solução final é fica com valor 1 e espaço livre  $B - 1$ . Mas a solução ótima é, claramente, pegar o item 2. Então a solução gulosa não é ótima, mas é uma solução viável.

### 6.1.1 Heurística de busca local para o TSP

Vizinhança de 2-OPT (troca de 2 arestas) de um circuito hamiltoniano  $C$ :

- Conjunto de circuitos hamiltonianos que são obtidos removendo 2 arestas de  $C$  e inserindo outras 2 arestas

Exemplo de troca 2-OPT:



**Função** TSP-2OPT( $G = (V, E), w$ )

  encontra um circuito hamiltoniano inicial  $C$ ;

**enquanto** houver um circuito  $C'$  na vizinhança 2-OPT de  $C$  tal que

$w(C') < w(C)$  **faça**

      |  $C \leftarrow C'$

**fim**

**retorna**  $\underline{C}$

**fim**

### Aula 1 Abordagens

$P \neq NP \rightarrow$  **Não** é possível encontrar um algoritmo que encontre soluções ótimas em tempo polinomial para qualquer entrada.

Existem abordagens que abrem mão de alguma dessas características.

#### 1.1 Heurísticas

Abre mão de que a solução seja ótima. Encontra uma solução viável. Podem ser:

- Construtivas – Constrói uma solução viável
- De Busca Local – Modifica uma solução pré-existente

#### 1.2 Algoritmos gulosos

As heurísticas construtivas são normalmente estratégias gulosas.

Algoritmos gulosos tomam decisões locais de curto prazo que leve a soluções. Normalmente produz soluções ótimas para problemas específicos, mas não é garantido para problemas NP-Difíceis.

É necessário criatividade para propor métodos gulosos para um problema.

### Aula 2 Problema do Escalonamento

- **Entrada:** conjunto de tarefas  $\{1, \dots, n\}$ , cada tarefa  $i$  tem tempo de processamento  $t_i$  e  $m$  máquinas idênticas.
- **Soluções viáveis:** partição das tarefas em  $m$  conjuntos  $M_1, M_2, \dots, M_m$ .
- **Função objetivo:**  $\max_{j=1\dots m} \sum_{i \in M_j} t_i$  (makespan).
- **Objetivo:** encontrar solução de custo mínimo.

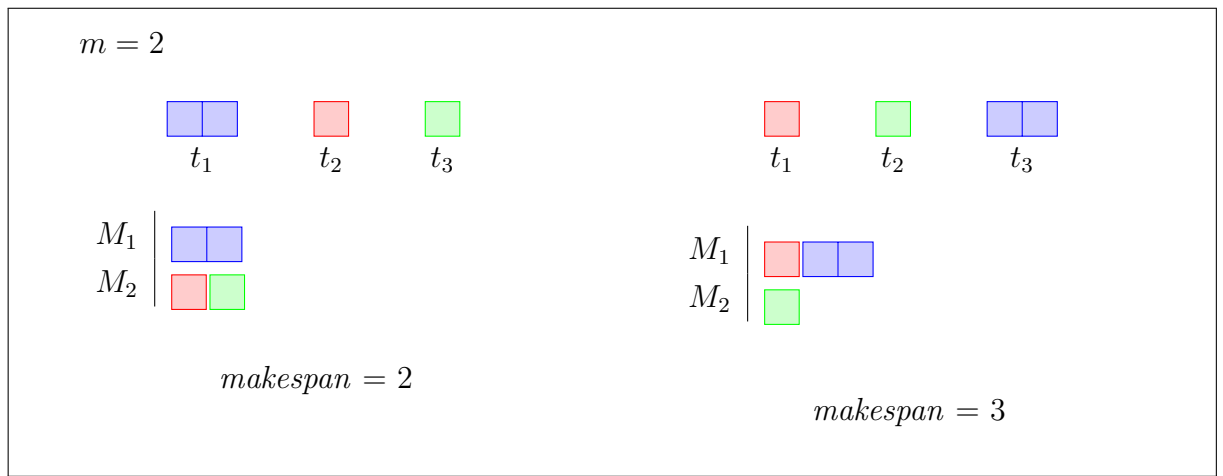


```

Função EscalonaGuloso( $n, t, m$ )
  para  $j \leftarrow 1$  até  $m$  faça  $M_j = \emptyset$ ;
  para  $i \leftarrow 1$  até  $n$  faça
    seja  $j$  uma máquina em que  $\sum_{i \in M_j} t_i$  é mínimo;
     $M_j \leftarrow M_j \cup \{i\}$ ;
  fim
  retorna  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$ ;
fim

```

## 2.1 Algoritmo guloso para o escalonamento



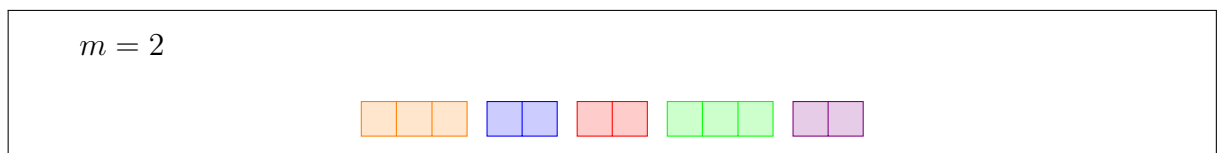
A ordem nos trabalhos mudou e o algoritmo não encontrou a solução ótima. A ideia é lidar primeiro com os trabalhos mais longos.

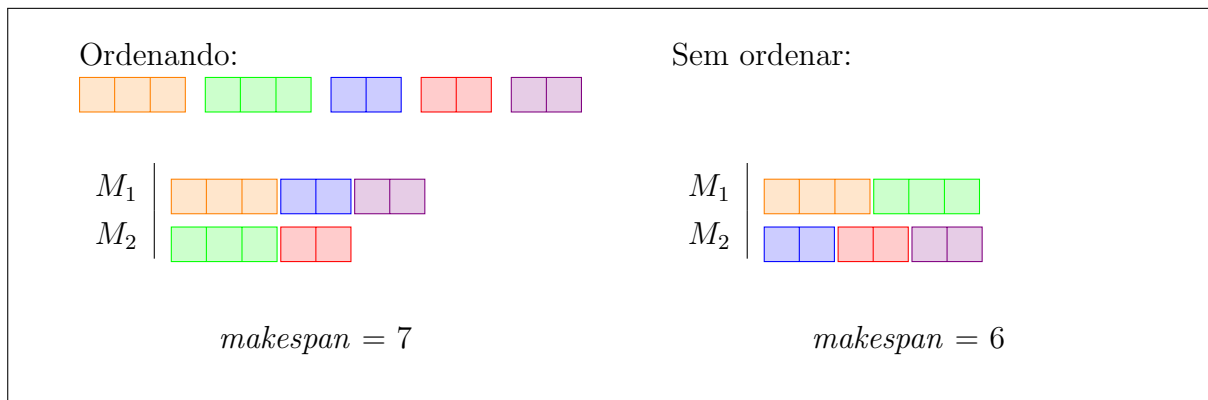
```

Função EscalonaGuloso2( $n, t, m$ )
  para  $j \leftarrow 1$  até  $m$  faça  $M_j = \emptyset$ ;
  Ordene e renomeie os itens de modo que  $t_1 \geq t_2 \geq \dots \geq t_n$ ;
  para  $i \leftarrow 1$  até  $n$  faça
    seja  $j$  uma máquina em que  $\sum_{i \in M_j} t_i$  é mínimo;
     $M_j \leftarrow M_j \cup \{i\}$ ;
  fim
  retorna  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$ ;
fim

```

Neste caso, ordenar ainda não gera a solução ótima. Não ordenar ainda gera um makespan menor:





Como estamos lidando com heurísticas, quase sempre é possível indicar um caso que o algoritmo mais simples funciona melhor que o mais sofisticado. Mas ordenar ainda é melhor em geral.

Lidando com problemas NP-Difíceis. O ideal é executar diferentes métodos e selecionar o melhor resultado. Variedade é importante, principalmente se os diversos métodos já são velozes.

### Aula 3 Problema da Mochila


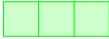

- **Entrada:** conjunto de  $n$  itens, cada item  $i$  tem peso  $w_i$  e valor  $v_i$  e tamanho  $W$  da mochila.
- **Soluções viáveis:** conjuntos de itens  $S \subseteq \{1, \dots, n\}$  com  $\sum_{i \in S} w_i \leq W$ .
- **Função objetivo:** soma dos valores dos itens em  $S$ .
- **Objetivo:** encontrar uma solução de valor máximo.

#### Função MochilaGuloso( $n, w, v, W$ )

```

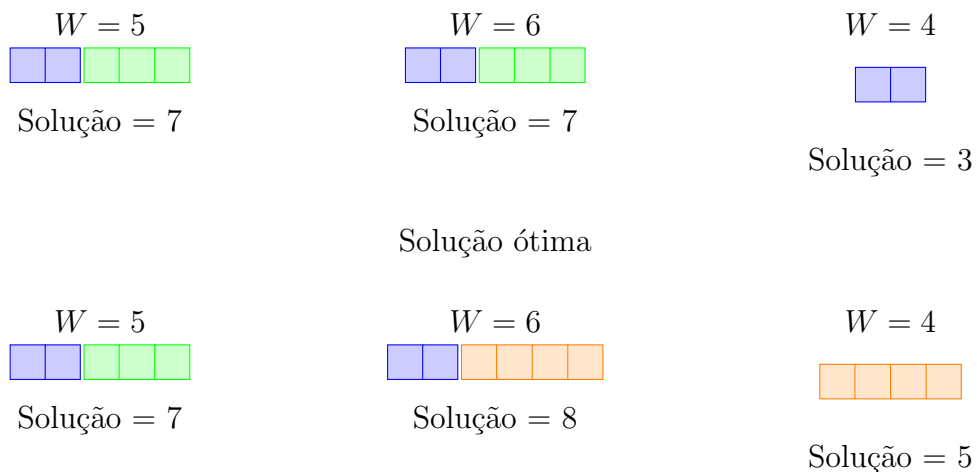
   $S \leftarrow \emptyset$  ;
   $R \leftarrow W$  ;
  Ordene e renomeie os itens para que  $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$ 
  para  $i \leftarrow 1$  até  $n$  faça
    se  $w_i \leq R$  então
       $S \leftarrow S \cup \{i\}$  ;
       $R \leftarrow R - w_i$ 
    fim
  fim
  retorna S
fim
```

/\* Solução \*/  
 /\* Espaço restante \*/  
 /\* Se couber na mochila \*/  
 /\* Adicione  $i$  à mochila \*/

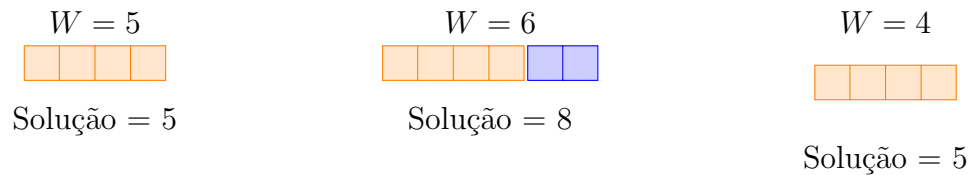
		
$w_1 = 2$	$w_2 = 3$	$w_3 = 4$
$v_1 = 3$	$v_2 = 4$	$v_3 = 5$
$\frac{v_1}{w_1} = \frac{3}{2}$	$\frac{v_2}{w_2} = \frac{4}{3}$	$\frac{v_3}{w_3} = \frac{5}{4}$

$\frac{3}{2} > \frac{4}{3} > \frac{5}{4}$

Algoritmo guloso baseado na razão  $\frac{\text{valor}}{\text{peso}}$



Algoritmo guloso baseado no valor



**Função** MochilaGuloso2( $n, w, v, W$ )

```

   $S \leftarrow \emptyset$  ;
   $R \leftarrow W$  ;
  Ordene e renomeie os itens para que  $v_1 \geq v_2 \geq \dots \geq v_n$  para  $i \leftarrow 1$  até  $n$  faça
    se  $w_i \leq R$  então
       $S \leftarrow S \cup \{i\}$  ;
       $R \leftarrow R - w_i$ 
    fim
  fim
  retorna S
fim

```

/\* Solução \*/  
/\* Espaço restante \*/  
/\* Se couber na mochila \*/  
/\* Adicione  $i$  à mochila \*/

O algoritmo guloso baseado em razão se deu melhor em uma instância e o baseado em valor foi melhor em duas instâncias.  $\implies$  O melhor é combinar os algoritmos.

## Aula 4 Problema da Coloração

- **Entrada:** um grafo  $G = (V, E)$ .
- **Soluções viáveis:** partição  $V_1, V_2, \dots, V_q$  de  $V$  tal que toda aresta  $(u, v) \in E$  tem extremos em partes distintas.
- **Função objetivo:** número  $q$  de partes (cores) utilizadas.
- **Objetivo:** encontrar solução de custo mínimo.

**Função** ColoracaoGuloso( $G = (V, E)$ )

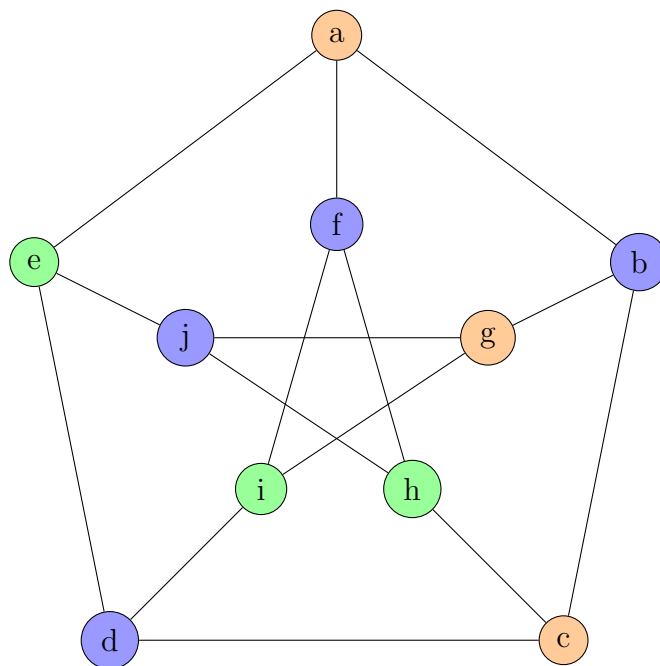
```

para  $v \in V$  faça
    /* Cores representadas por índices em ordem          */
    seja  $i$  a menor cor não presente nos vizinhos de  $v$ ;
    atribua cor  $i$  a  $v$ ;
fim
seja  $i_{\max}$  a maior cor utilizada;
retorna  $i_{\max}$ 
fim

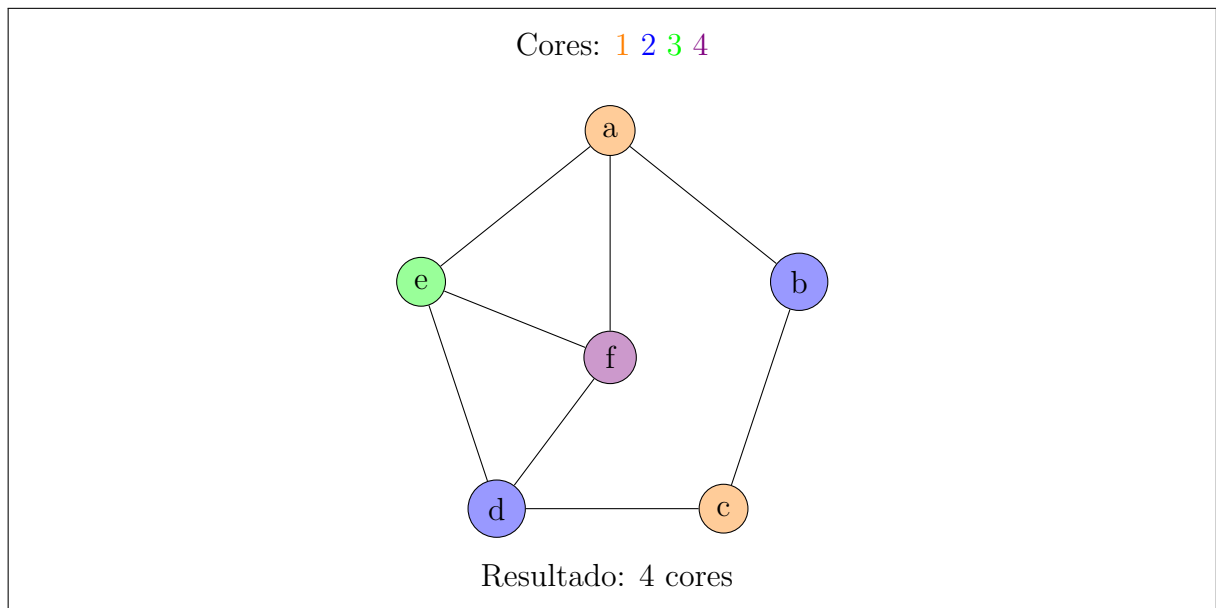
```

Grafo de Petersen

Cores: 1 2 3 4

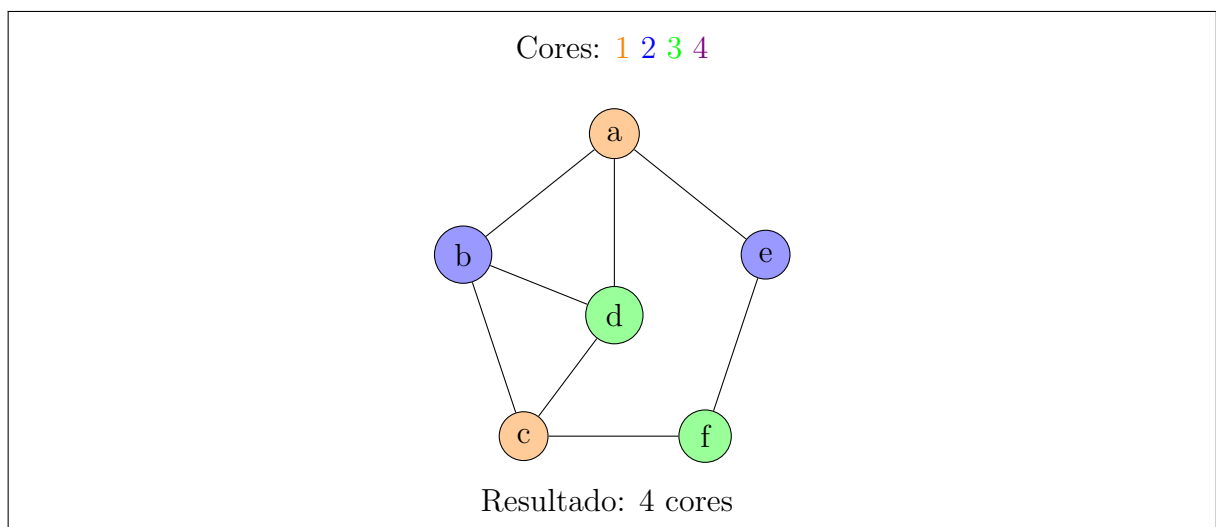


Resultado: 3 cores



O problema aqui foi um vértice de maior grau (f). Uma ideia é começar a trabalhar com os vértices de maior grau.

Reordenando, temos:



**Função** ColoracaoGuloso2( $G = (V, E)$ )

ordene e renomeie os vértices em ordem decrescente de grau;

**para**  $v \leftarrow 1$  até  $|V|$  **faça**

    seja  $i$  a menor cor não presente nos vizinhos de  $v$ ;

    atribua a cor  $i$  a  $v$ ;

**fim**

seja  $i_{\max}$  a maior cor utilizada;

**retorna**  $i_{\max}$

**fim**

## Aula 5 Corte em grafos

Um corte é uma bipartição dos vértices de um grafo.

O tamanho do corte é o número de arestas que “atravessam” o corte.



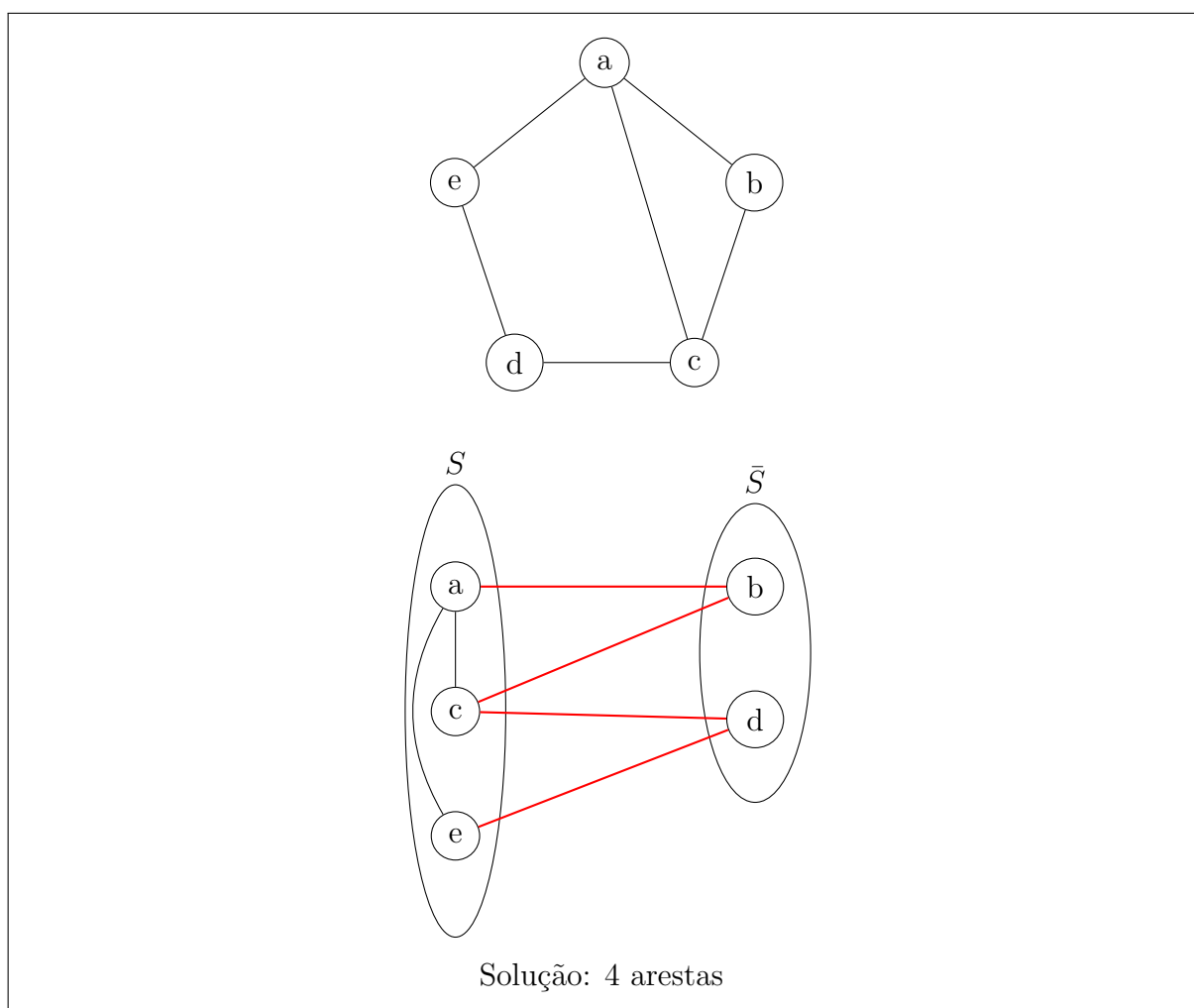
### 5.1 Problema do corte máximo

- **Entrada:** um grafo  $G = (V, E)$ .
- **Soluções viáveis:** um corte de  $G$ , i.e., um conjunto  $S$  tal que  $\emptyset \neq S \subset V$ .
- **Função objetivo:** número de arestas que sai de  $S$ :  $|\delta(S)|$ .
- **Objetivo:** encontrar solução de custo máximo.

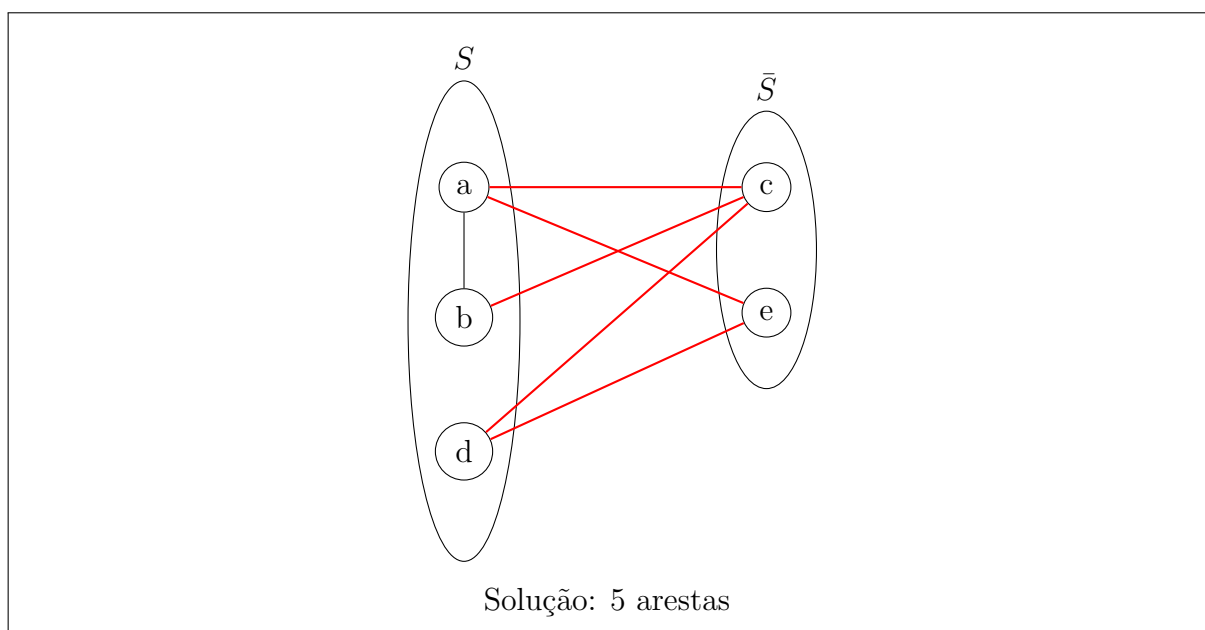
```

Função CorteMaximoGuloso( $G = (V, E)$ )
   $S \leftarrow \emptyset$ ;
   $\bar{S} \leftarrow \emptyset$ ;
  para  $v \in V$  faça
    /* Coloca o vértice no conjunto que tem menos vizinhos dele
       (aumentando o número de arestas que atravessam o corte) */
    se  $|\{(v, u) \in \delta(v) : u \in S\}| \leq |\{(v, u) \in \delta(v) : u \in \bar{S}\}|$  então
      |  $S \leftarrow S \cup \{v\}$ ;
    senão
      |  $\bar{S} \leftarrow \bar{S} \cup \{v\}$ 
    fim
  fim
fim

```



Se atribuir mais cedo os vértices de maior grau ( $a, c$ ), parece que é melhor. Já que esses vértices “influenciam mais” outros.



```

Função CorteMaximoGuloso2( $G = (V, E)$ )
   $S \leftarrow \emptyset$ ;
   $\bar{S} \leftarrow \emptyset$ ;
   $alcancados \leftarrow \{s\}$ ; /* Inicializa com algum vértice inicial */
  para  $v \in alcancados$  com maior grau faça
    se  $|\{(v, u) \in \delta(v) : u \in S\}| \leq |\{(v, u) \in \delta(v) : u \in \bar{S}\}|$  então
       $S \leftarrow S \cup v$ ;
       $alcancados \leftarrow alcancados \cup \delta(v)$ ;
    senão
       $\bar{S} \leftarrow \bar{S} \cup \{v\}$ ;
       $alcancados \leftarrow alcancados \cup \delta(v)$ ;
    fim
  fim
fim

```

## Aula 6 Problema do Caixeiro Viajante

- **Entrada:**  $G = (V, E)$  e função  $w$  de peso nas arestas.
- **Soluções viáveis:** circuitos hamiltonianos (visitam todos os vértices sem repetição) de  $G$ .
- **Função objetivo:** soma dos pesos das arestas do circuito.
- **Objetivo:** encontrar um circuito de menor custo.

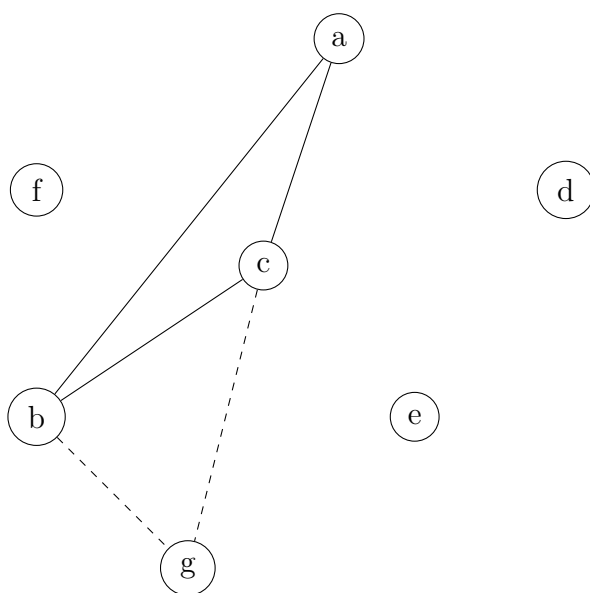
Escolher em cada iteração o vértice com menor custo.

```

Função TSP-Guloso1( $G = (V, E), w$ )
  Seja  $C$  um circuito trivial;
  enquanto  $C$  não é hamiltoniano faça
    escolha uma aresta  $(u, v) \in C$  e um vértice  $x \notin C$  que minimizem
       $w(u, x) + w(x, v) - w(u, v)$ ;
     $C \leftarrow C - uv + ux + xv$ ;
  fim
  retorna  $C$ 
fim

```

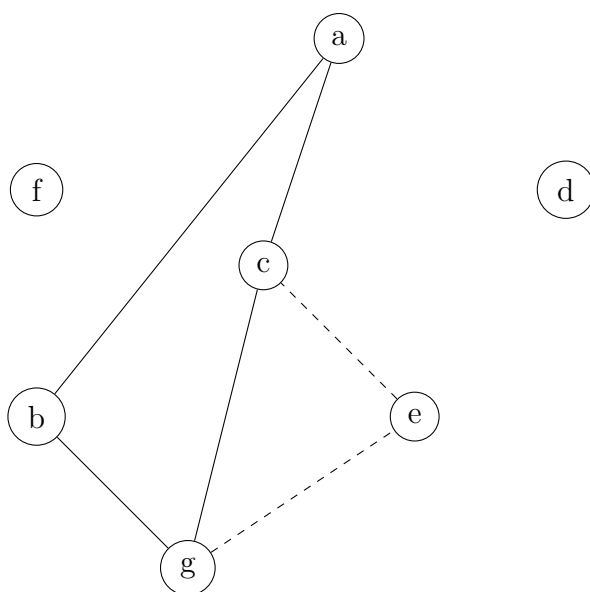




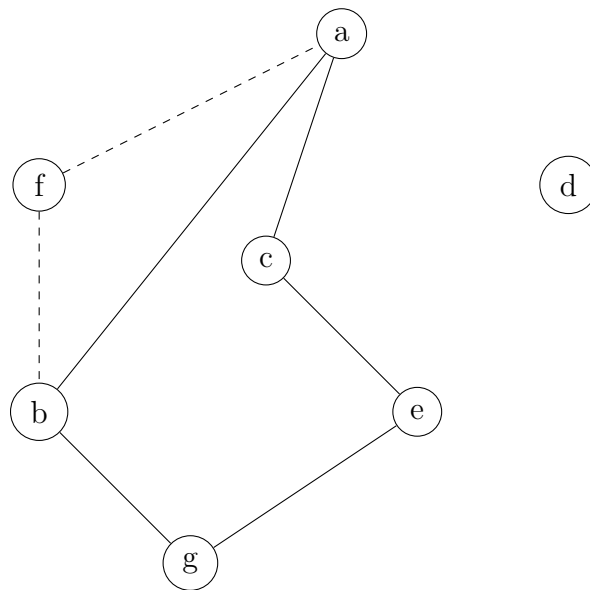
Circuito trivial:  $\bar{a}bc$ .

Vértice que minimiza:  $g$ , quando inserido entre  $b$  e  $c$ .

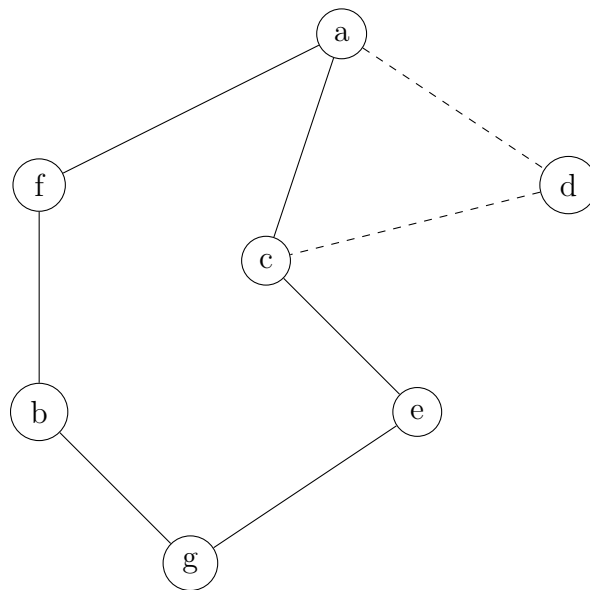
Então, remove a aresta  $\bar{bc}$  e adiciona  $\bar{b}gc$ .



Em seguida, insere o vértice  $f$ .



E o vértice  $d$  por último.



## 6.1 Vértices mais próximos e mais distantes

Seja  $w(v, C)$  a menor distância de  $v$  a um vértice de  $C$ .

```

Função TSP-Guloso2( $G = (V, E), w$ )
  Seja  $C$  um circuito trivial;
  enquanto  $C$  não é hamiltoniano faça
    escolha um vértice  $c \notin C$  que minimiza  $w(x, C)$ ;
    insira  $x$  em  $C$  ao lado do vértice mais próximo dele;
  fim
retorna  $C$ 
fim

```

É mais fácil calcular a distância de um nó para um conjunto de nós do que calcular o critério de peso de arestas do algoritmo TSP-Guloso1.

Para o algoritmo que insere o nó mais distante:

```

Função TSP-Guloso3( $G = (V, E), w$ )
  Seja  $C$  um circuito trivial, de preferência formado pelos dois vértices mais
  distantes;
  enquanto  $C$  não é hamiltoniano faça
    escolha um vértice  $c \notin C$  que maximiza  $w(x, C)$ ;
    insira  $x$  em  $C$  ao lado do vértice mais próximo dele;
  fim
retorna  $C$ 
fim

```

Esse algoritmo não necessariamente se preocupa em adicionar as arestas de menor peso. Mas é uma heurística e gera uma solução viável.

## 6.2 Vizinho mais próximo

Não mantém um circuito. Constroi um caminho. Ao final cria um circuito ligando as extremidades.

```

Função TSP-Guloso4( $G = (V, E), w$ )
  Seja  $v$  um vértice qualquer;
   $C \leftarrow (v)$ ;
  enquanto  $C$  não contém todos os vértices faça
    Seja  $C = (v_1, \dots, v_i)$ ;
    Escolha um vértice  $x \notin C$  que minimiza  $w(v_i, x)$ ;
    Insira  $x$  no final de  $C$ ;
  fim
retorna  $C$ 
fim

```

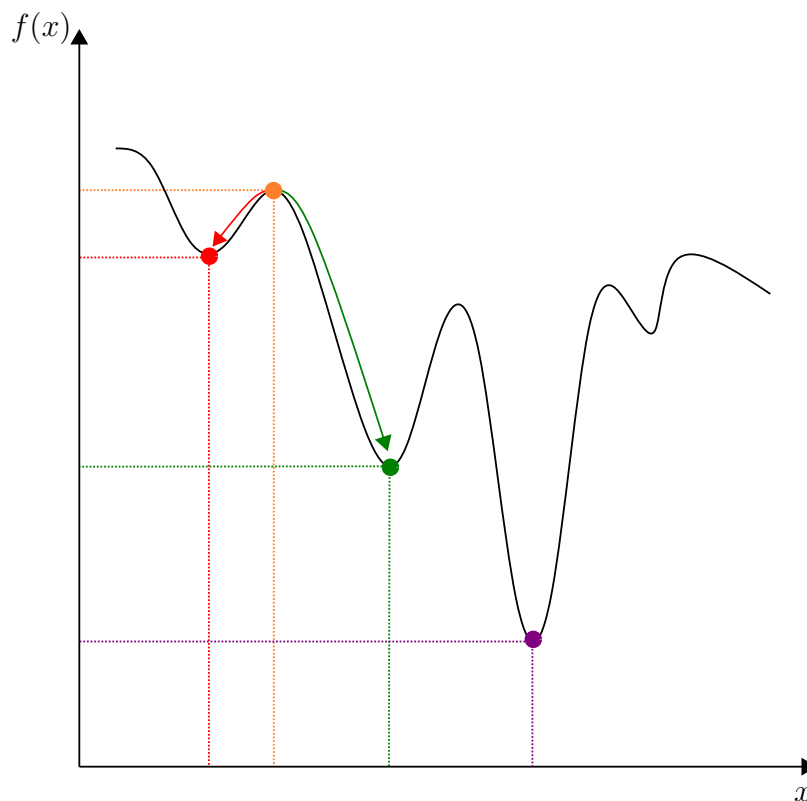
O algoritmo **TSP-Guloso4** é mais eficiente porque não precisa calcular a distância de cada vértice ao conjunto de vértices (circuito). Calcula apenas a distância com relação a um único vértice  $v_i$ .

## SEMANA 3

## HERÍSTICAS DE BUSCA LOCAL

Heurísticas geram soluções viáveis, mas sem garantir a qualidade.

- Construtivas
- De busca local

**Aula 1 Introdução****1.1 Ótimo local vs. ótimo global**

O algoritmo inicia no ponto laranja e vai, incrementalmente, encontrando soluções melhores na vizinhança.

Seguindo o caminho vermelho, ele encontra um mínimo local, já que nenhum vizinho tem um valor de função melhor que ele (menor, já que é um problema de minimização).

Mas seguir o caminho verde, pode ser melhor do que seguir o caminho vermelho.

De qualquer forma, o ótimo global é o roxo, mas a busca local não encontra ele, porque olha apenas a vizinhança.

**Busca local garante apenas que estamos em um ótimo local, não necessariamente global.**

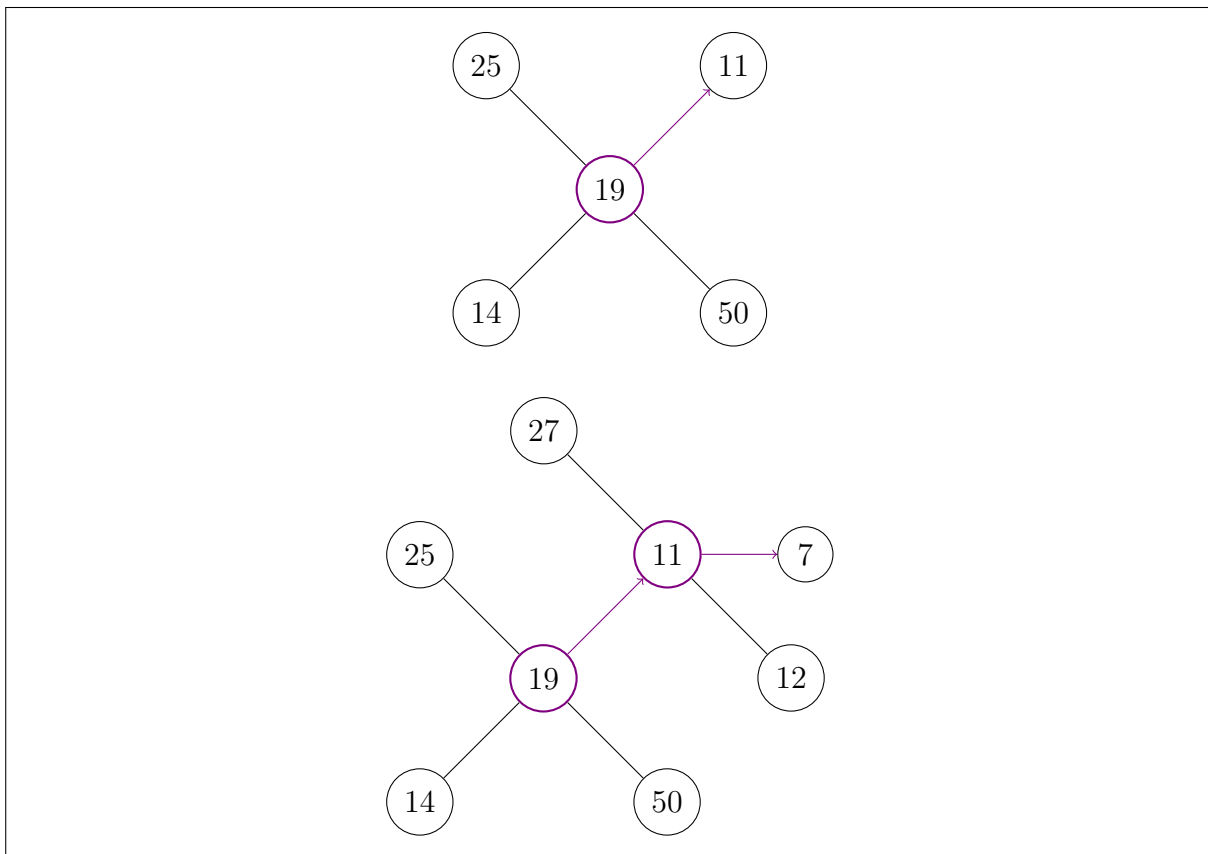
Também não é possível garantir que o ótimo local é pelo menos próximo do ótimo global. É um ótimo local, mas pode ser ainda muito longe do ótimo global.

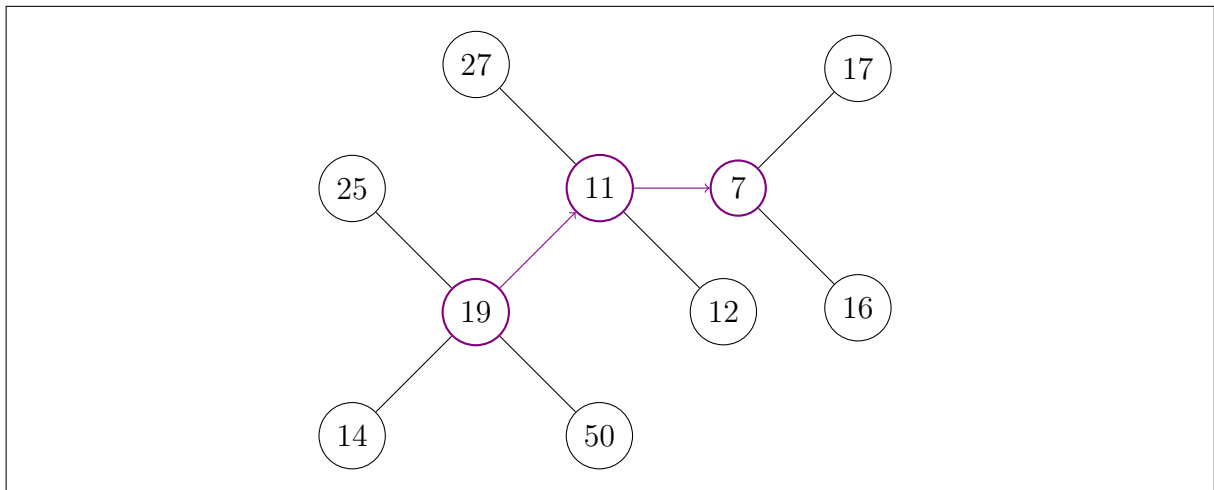
Positivo: A solução final é sempre melhor que o ponto de partida.

## 1.2 Conceito de vizinhança

Vizinhança de uma solução é o conjunto de soluções obtidas com uma pequena mudança.

Partindo de uma solução, encontra-se as vizinhas (segundo algum critério). Depois de identificadas as soluções vizinhas, é possível encontrar se existe alguma melhor. Seguindo uma estratégia gulosa, modifica-se a solução inicial partindo para a melhor vizinha.





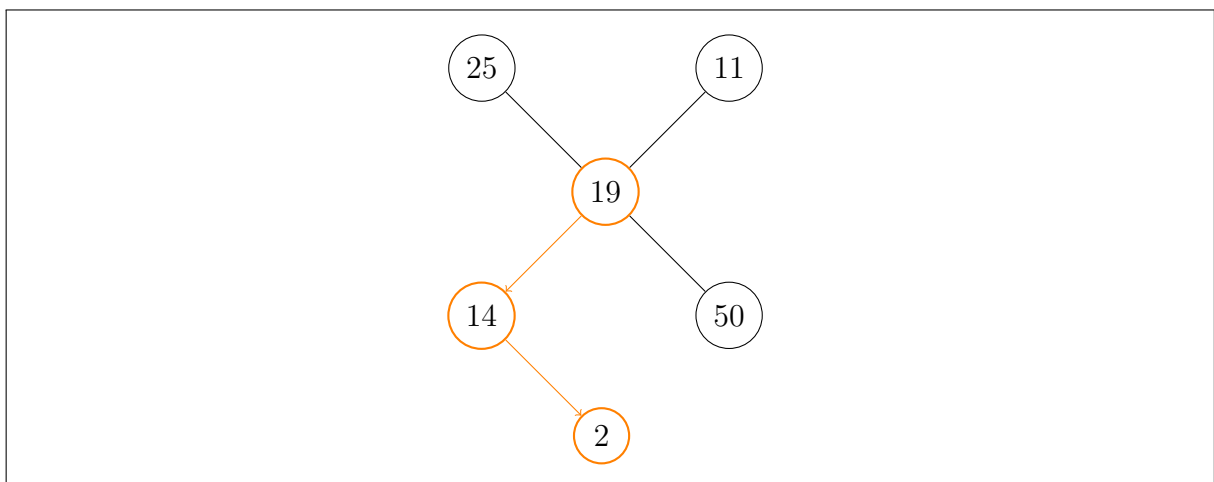
Ao final, o nó de valor 7 é o mínimo local, então é a solução final.

A definição de vizinhança influencia no resultado.

É também importante tomar alguma decisões na hora de percorrer a vizinhança:

- First fit - Ao percorrer os vizinhos, quando se encontra um que seja melhor que a solução atual, parte para ele.
- Best fit - Verifica todos os vizinhos para escolher o melhor: guloso.
- Random - Busca aleatória. Mais variedade para encontrar mínimos locais diferentes.

Por exemplo, pode ser que depois do nó de valor 14, exista uma solução melhor ainda.



First fit normalmente dá passos menores de melhoria, com mais passos. Já o best fit, dá menos passos maiores.

First fit pode ser mais eficiente, mas não necessariamente.

```

 $S \leftarrow$  Solução inicial;
enquanto Vizinhaça da solução atual  $S$  tiver uma solução  $S'$  melhor faça
  |  $S \leftarrow S'$ ;
fim
retorna  $S$ 

```

A solução inicial pode vir de uma heurística construtiva ou de soluções aleatórias.  
 Questões: como definir a vizinhaça? Como percorrer a vizinhaça?

## Aula 2 Problema do escalonamento

- **Entrada:** conjunto de tarefas  $\{1, \dots, n\}$ , cada tarefa  $i$  tem tempo de processamento  $t_i$  e  $m$  máquinas idênticas.
- **Soluções viáveis:** partição das tarefas em  $m$  conjuntos  $\{M_1, M_2, \dots, M_m\}$ .
- **Função objetivo:**  $\max_{j=1 \dots m} \sum_{i \in M_j} t_i$  (makespan).
- **Objetivo:** encontrar solução de custo mínimo.

Vizinhaça de um escalonamento  $\mathcal{M}$ : escalonamentos que diferem de  $\mathcal{M}$  pela posição de um item.

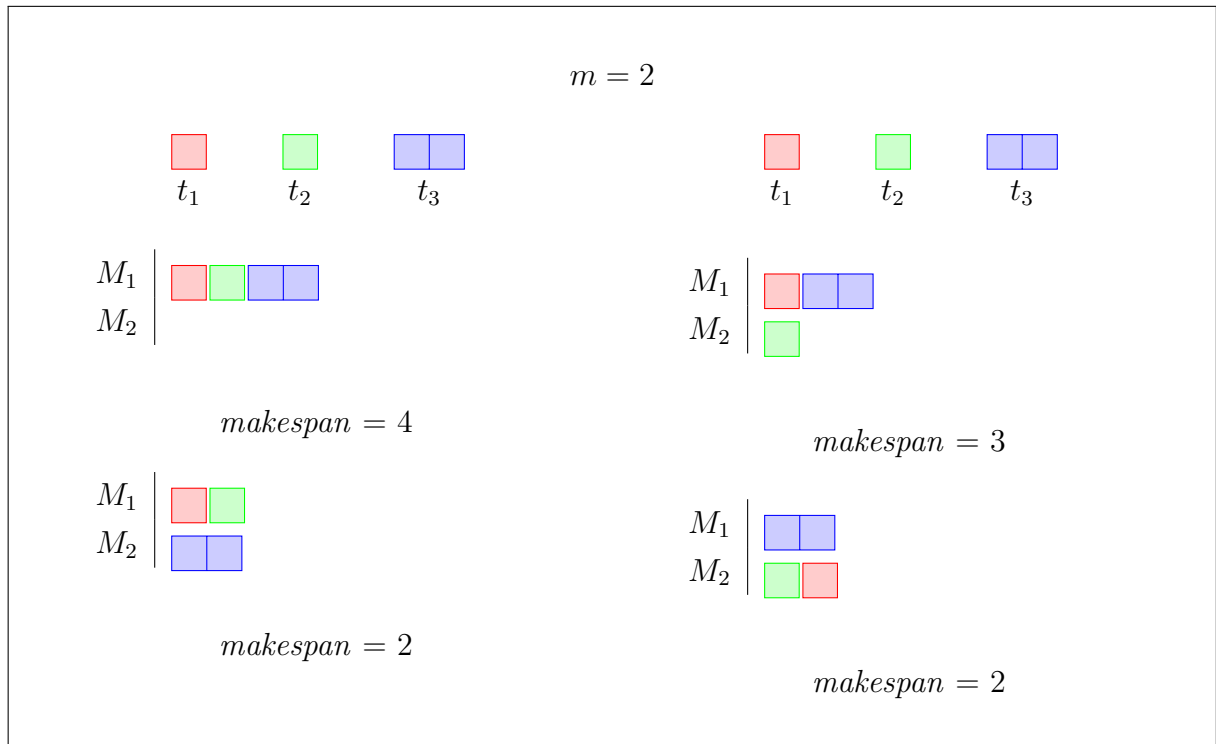
Seja  $l(j) = \sum_{i \in M_j} t_i$  a carga da máquina  $j$ .

```

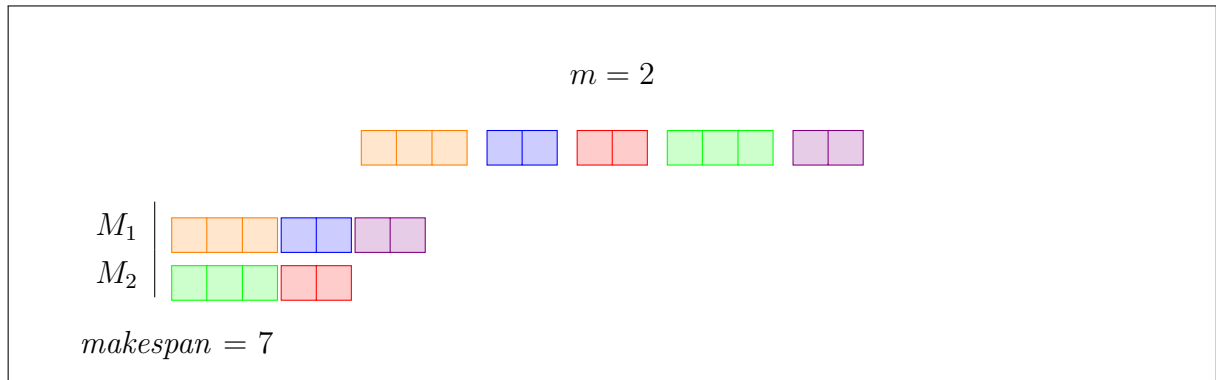
Função EscalonaBuscaLocal(n, t, m)
  |  $\mathcal{M} \leftarrow$  um escalonamento inicial;
  | /*  $l(j')$  é o makespan atual */
  | enquanto houver um item  $i'$  na máquina mais carregada  $j'$  e uma máquina  $j$ 
  |   | tal que  $l(j) + t_{i'} < l(j')$  faça
  |     |  $M_{j'} \leftarrow M_{j'} \setminus \{i'\};$ 
  |     |  $M_j \leftarrow M_j \cup \{i'\};$ 
  |   fim
  | retorna  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$ 
fim

```





No seguinte exemplo, não há nenhum movimento possível que melhore o resultado.



Trocando o conceito de vizinhança para lidar com dois itens de diferença, nós conseguimos encontrar uma solução melhor.



Normalmente as diferentes vizinhanças são combinadas. Iniciando com as vizinhanças mais simples e, quando não dá pra melhorar nada com elas, usa as vizinhanças mais complexas.

**Função EscalonaBuscaLocal2( $n, t, m$ )**

```

 $\mathcal{M} \leftarrow$  um escalonamento inicial;
enquanto houver  $i'$  na máquina mais carregada  $j'$  e  $i$  numa máquina  $j$  tal que
 $l(j) - t_i + t_{i'} < l(j')$  e  $l(j') - t_{i'} + t_i < l(j)$  faça
     $M_{j'} \leftarrow M_{j'} \setminus \{i'\} \cup \{i\};$ 
     $M_j \leftarrow \{i\} \cup \{i'\};$ 
fim
retorna  $\max_{j=1,\dots,m} \sum_{i \in M_j} t_i$ 
fim

```

**Aula 3 Problema do Corte Máximo**

- **Entrada:** um grafo  $G = (V, E)$ .
- **Soluções viáveis:** um corte de  $G$ , i.e., um conjunto  $S$  tal que  $\emptyset \neq S \subset V$ .
- **Função objetivo:** número de arestas que sai de  $S$ , i.e.,  $|\delta(S)|$ .
- **Objetivo:** encontrar solução de custo máximo.

Vizinhança de um corte  $S$ : cortes que tem apenas um vértice a mais ou a menos que  $S$ .

$c(v)$  = número de arestas incidentes a  $v$  que atravessam o corte.

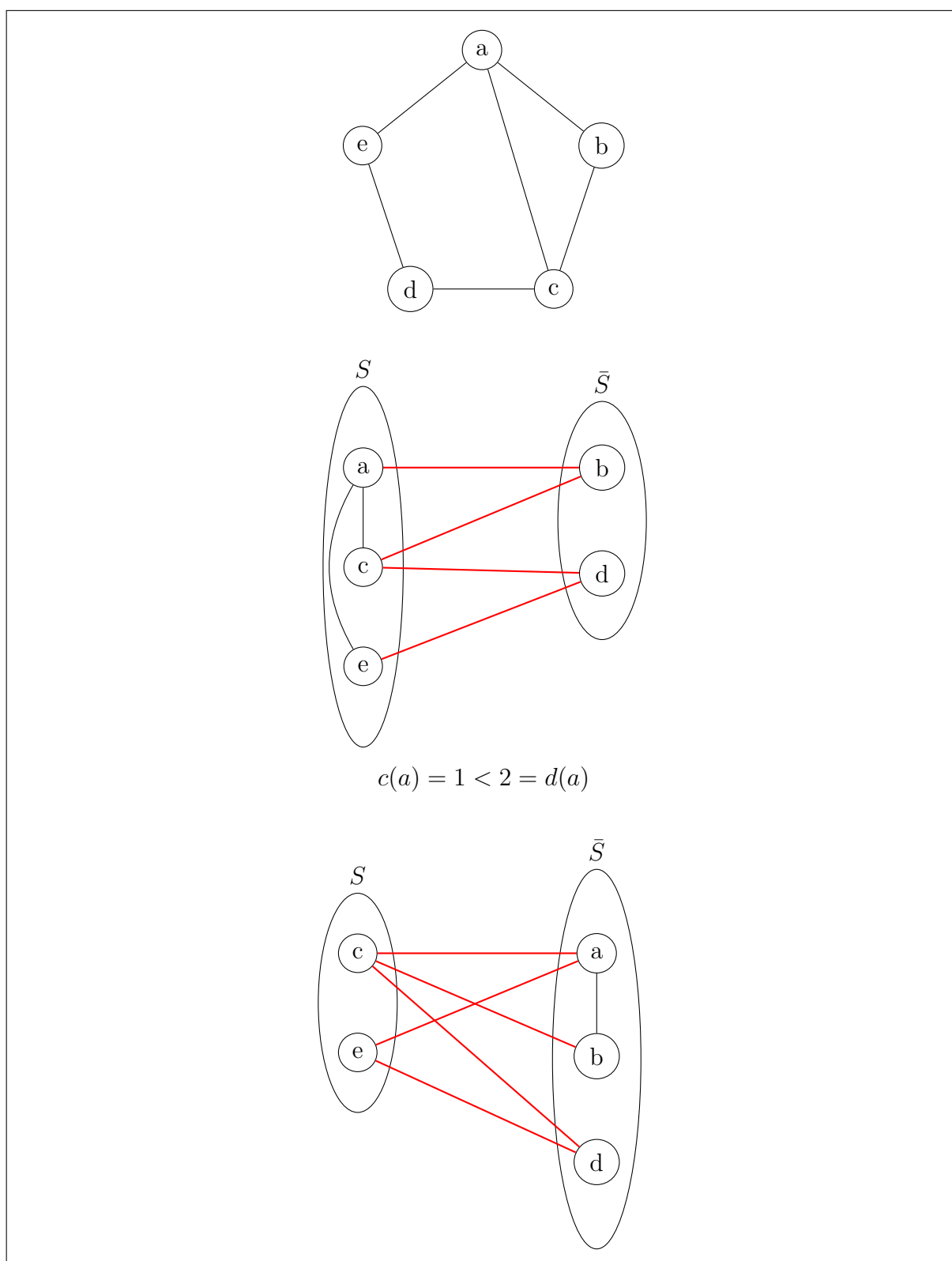
$d(v)$  = número de arestas incidentes a  $v$  que não atravessam o corte.

**Função CorteMaximoBuscaLocal( $G = (V, E)$ )**

```

 $S \leftarrow$  um corte inicial;
 $\bar{S} \leftarrow V \setminus S;$ 
enquanto houver vértice  $v$  tal que  $c(v) < d(v)$  faça
    se  $v \in S$  então
         $S \leftarrow S \setminus \{v\};$ 
         $\bar{S} \leftarrow \bar{S} \cup \{v\};$ 
    senão
         $\bar{S} \leftarrow \bar{S} \setminus \{v\};$ 
         $S \leftarrow S \cup \{v\};$ 
    fim
fim
retorna  $S$ 
fim

```



Nesse caso, a heurística de busca local melhorou o resultado obtido pelo algoritmo guloso.

## Aula 4 Problema da Localização de Instalações

Localização de Instalações sem Capacidades  $\rightarrow$  UFL

- **Entrada:** conjunto de clientes  $D$ , conjunto de instalações  $F$ , distância  $d : D \times F \mapsto \mathbb{R}^+$  e custo de abertura  $f : F \mapsto \mathbb{R}^+$ .
- **soluções viáveis:** um subconjunto  $F'$ , tal que  $\emptyset \neq F' \subseteq F$ .
- **Função objetivo:** custo de abertura mais custo de conexão, i.e.,  $\sum_{i \in F'} f(i) + \sum_{j \in D} d(j, a(j))$ , com  $a(j)$  instalação mais próxima de  $j$  em  $F'$ .
- **Objetivo:** encontrar solução de custo mínimo.

Vizinhança de um conjunto de instalações  $F'$ : conjuntos que tem apenas uma instalação a mais ou a menos que  $F'$ .

$D(i)$  = conjunto de clientes mais próximos de  $i$  que de qualquer instalação em  $F'$ .  $\rightarrow i$  não está em  $F'$ , mas atrairia esses clientes se estivesse.

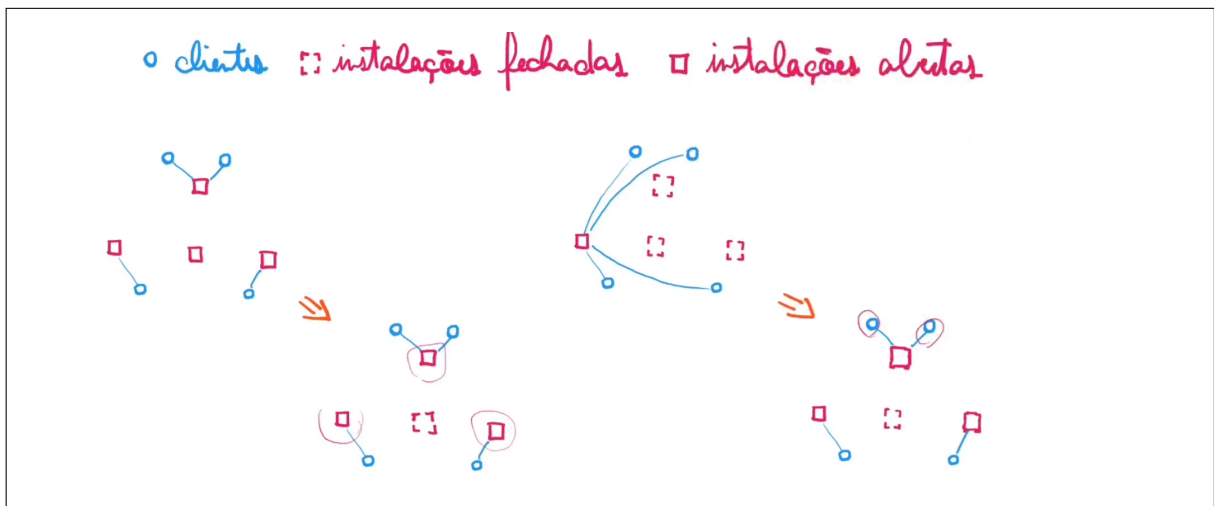
$\Delta(i) = \sum_{j \in D_j} d(j, a(j)) - \sum_{j \in D_i} d(j, i)$ .  $\rightarrow$  Ganho de conexão.

**Função UFL-BuscaLocal**( $D, F, d, f$ )

```

 $F' \leftarrow$  um conjunto inicial de instalações;
enquanto houver uma instalação  $i'$  tal que  $i' \in F'$  com  $f(i') > \Delta(i')$  ou
 $i' \in F \setminus F'$  com  $f(i') < \Delta(i')$  faça
    se  $i' \in F'$  então
        /*  $i'$  não se paga */
         $F' \leftarrow F' \setminus \{i'\}$ ;
    senão
        /*  $i'$  deveria estar aberto */
         $F' \leftarrow F' \cup \{i'\}$ ;
    fim
fim
retorna  $F'$ 
fim

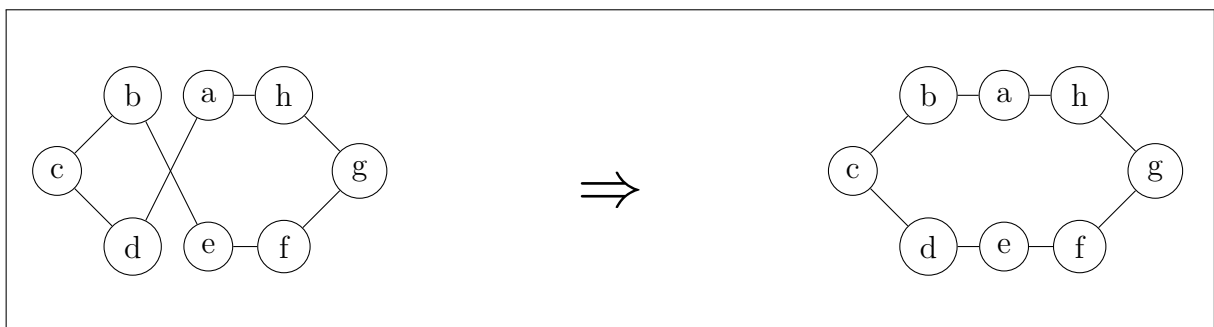
```



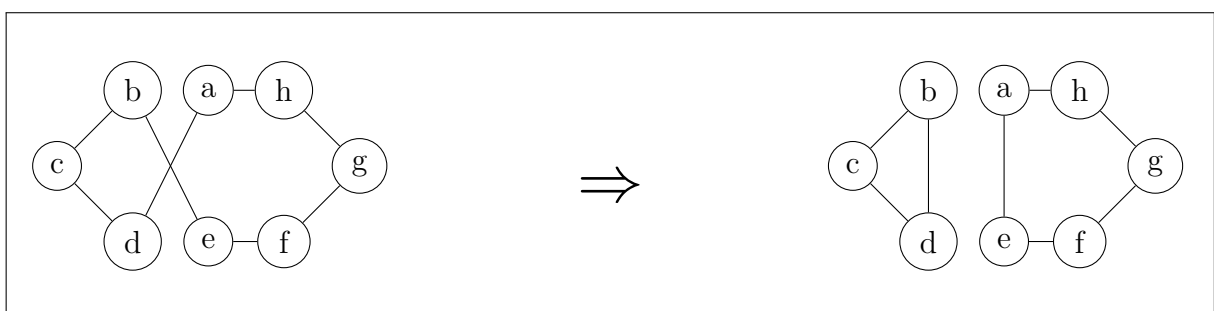
## Aula 5 Problema do caixeiro viajante

- **Entrada:**  $G = (V, E)$  e função  $w$  de peso nas arestas.
- **Soluções viáveis:** circuitos hamiltonianos de  $G$ .
- **Função objetivo:** soma dos pesos das arestas do circuito.
- **Objetivo:** encontrar um circuito de menor custo.

Vizinhança 2-OPT de um circuito hamiltoniano  $C$ : conjunto dos circuitos hamiltonianos que são obtidos trocando 2 arestas de  $C$ .



É importante garantir que a troca permita que a solução seja ainda viável. Por exemplo, a troca a seguir não pode ser feita.



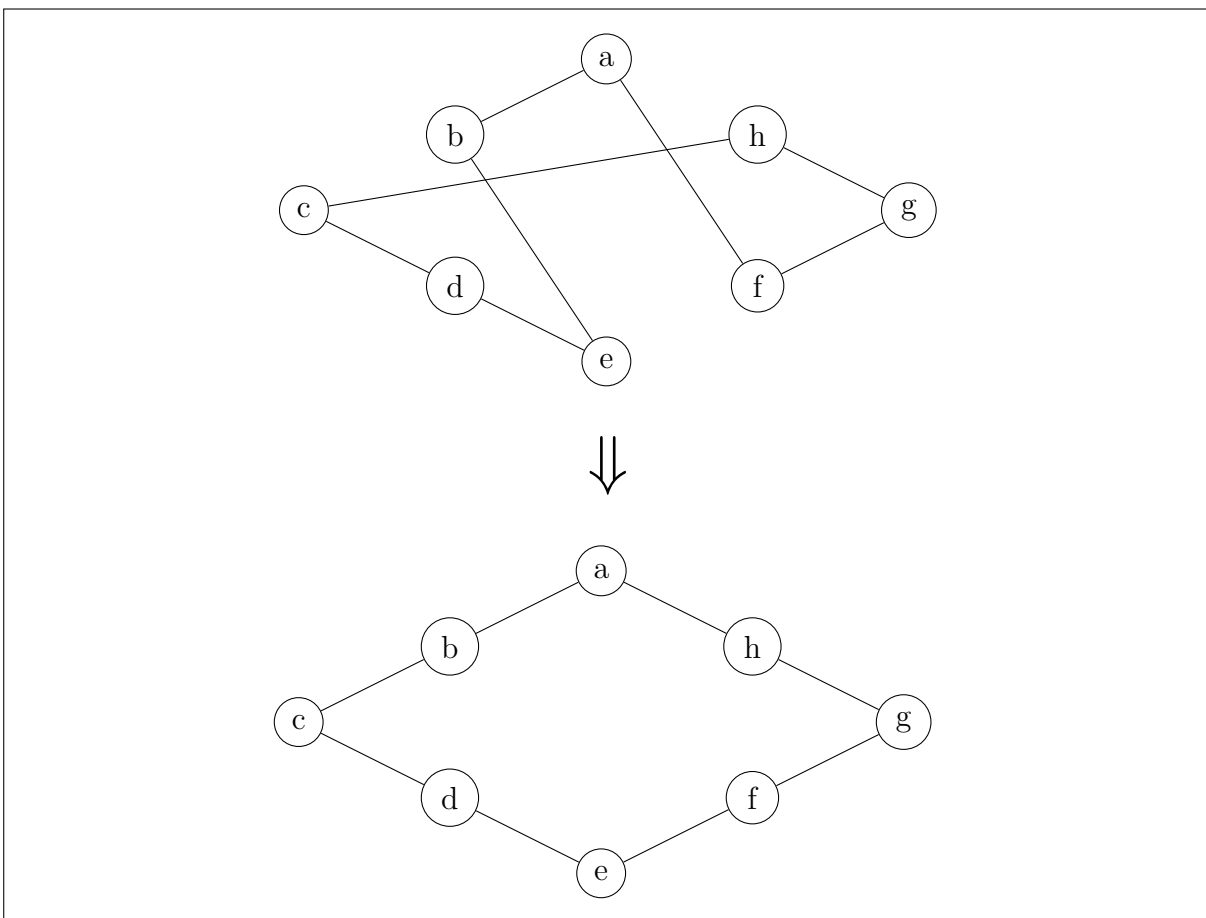
**Função**  $\text{TSP-2-OPT}(G = (V, E), w)$

```

   $C \leftarrow$  um circuito hamiltoniano inicial;
  enquanto houver um par de arestas  $(u, v)$  e  $(x, z)$  em  $C$  tal que
     $w(u, x) + w(x, z) < w(u, v) + w(x, z)$  faça
    |  $C \leftarrow C \setminus \{(u, v), (x, z)\} \cup \{(u, x), (v, z)\}$ ;
  fim
  retorna  $C$ 
fim

```

É possível utilizar uma vizinhança 3-OPT.



Na realidade é possível usar uma vizinhança  $k$ -OPT mais geral. Uma **categoria de vizinhanças**.

## Aula 6 Metaheurísticas de busca

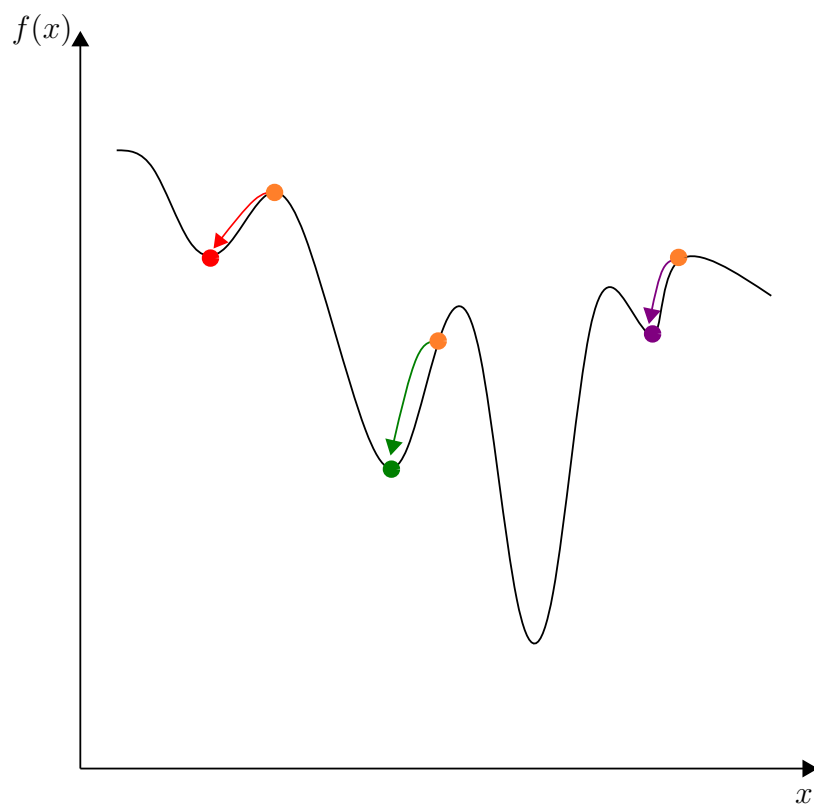
Extensões da busca local. Como fugir do mínimo local?

- Repeated local search
- Simulated annealing

- Tabu search
- Variable neighbourhood search
- GRASP

## 6.1 Repeated local search

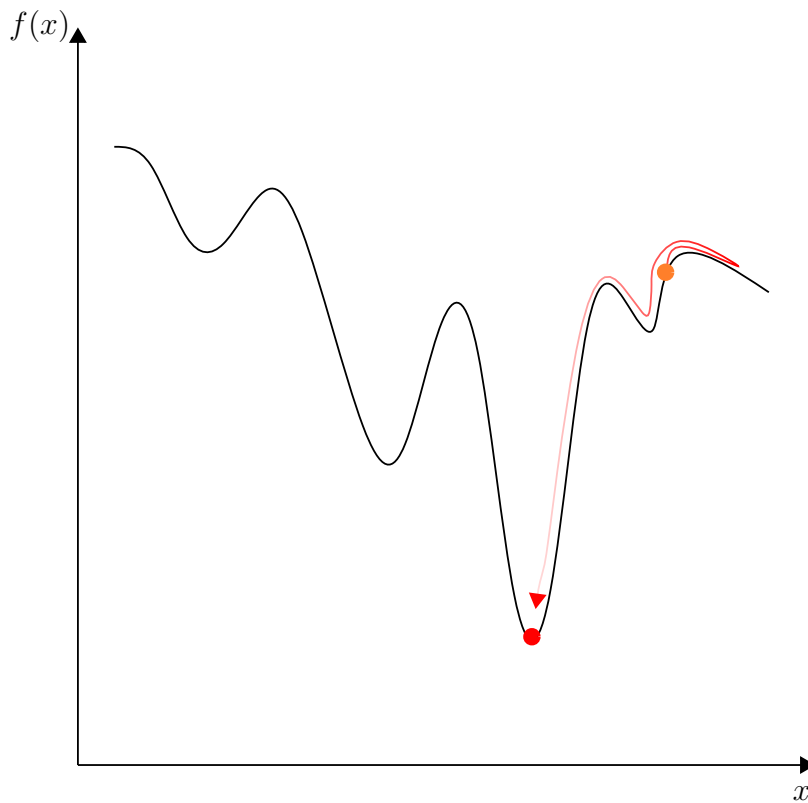
Realizar diversas buscas locais com instâncias e buscas aleatórias. Joga diferentes pontos iniciais de busca, para que alguma delas encontre uma solução razoável.



## 6.2 Simulated annealing (Cozimento simulado)

Ideia de temperatura. Enquanto a temperatura do sistema está alta, a busca tem grande chance de fazer escolhas que piorem a solução.

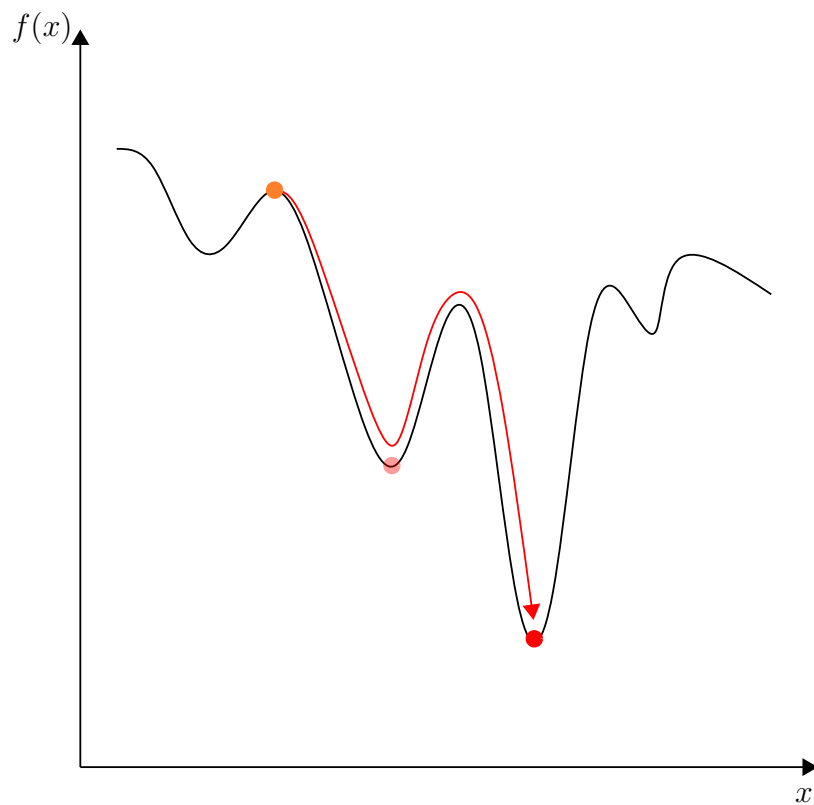
O algoritmo busca o espaço ao redor da solução inicial de forma mais aleatória (podendo ir para soluções piores), mas conforme o tempo passa (temperatura abaixa), converge para um mínimo local.



## 6.3 Tabu search

No início faz uma busca local até um mínimo local. Chegando no mínimo, ele passa a aceitar vizinhos que piorem para escapar do mínimo local. Espera encontrar em algum ponto um mínimo local melhor que o anterior.





Existe uma questão que, enquanto “sobe o morro”, sempre há pelo menos uma solução melhor (a que você veio antes). Para isso, tem que ter uma lista tabu indicando que soluções já percorreu e proibir esses movimentos.

#### 6.4 Variable neighbourhood search

Usa uma vizinhança mais simples. Quando chega em um mínimo local, usa uma vizinhança mais complexa e depois volta para a vizinhança mais simples. Pode usar diversos níveis de complexidade de vizinhanças.

#### 6.5 GRASP - Greedy Randomized Adaptive Search Procedure

Combina estratégia gulosa com aleatoriedade e busca local. Aleatoriza os algoritmos gulosos para que cada vez que executa, ele percorra um caminho diferente trazendo diversidade de soluções. Em cada passo, tem também um processo de busca local para melhorar o resultado obtido pelo algoritmo guloso.

# PROGRAMAÇÃO LINEAR INTEIRA

Está dentro dos métodos exatos. Abre mão do tempo polinomial para o pior caso, mas obtém sempre a solução ótima. Exemplos:

- Programação dinâmica
- *Branch and bound*
- Programação linear (inteira)
- Programação por restrições

## Aula 1 Definição

Modelo matemático que descreve problemas de otimização combinatória.

- Um conjunto de variáveis inteiras
- Um conjunto de restrições (inequações lineares)
- Uma função objetivo (expressão linear)

Solução viável: atribuição para as variáveis. Desde que satisfazam as restrições.

Existem também problemas mistos: com variáveis inteiras e contínuas.

### 1.1 Forma genérica de um PLI

$n$  variáveis  
 $m$  restrições  
minimizar  $\sum_{j=1}^n c_j x_j$   
sujeito a  $\sum_{j=1}^n a_{ij} x_j \geq b_i, \forall i \in \{1, \dots, m\}$   
 $x_j \in \mathbb{Z}^+, \forall j \in \{1, \dots, n\}$

$a_{ij}, b_i$  e  $c_j$  são constantes e  $x_j$  são as variáveis.

Função objetivo:  $\sum_{j=1}^n c_j x_j$ .

Se multiplicar a função objetivo por  $-1$ , o problema de minimização se torna de maximização. Igualmente, para as restrições, pode multiplicar por  $-1$  para obter uma inequação “menor que” ( $\leq$ ). Para garantir a igualdade, usamos duas inequações:

$$\sum_{j=1}^n a_{ij}x_j = b_i \iff \begin{cases} \sum_{j=1}^n a_{ij}x_j \geq b_i \\ \sum_{j=1}^n a_{ij}x_j \leq b_i \end{cases}$$

É comum usar variáveis binárias.

## 1.2 Programa linear

Pode-se fazer uma “relaxação da integridade” para obter um Programa Linear (PL), em que as variáveis deixam de ser inteiras e podem ser reais.

Um programa linear inteiro sempre pode ser “relaxado” para um PL. Com isso, podemos garantir que o PL é um **limitante** do PLI original.

Toda solução do PLI é uma solução do PL.

PL podem ser resolvidos em tempo polinomial. (PLI não garante isso)

Resolvedores de PLI normalmente combinam *branch and bound* com PL. Para encontrar melhores limitantes inferiores e realizar mais podas na árvore de busca.

## Aula 2 Problema da Mochila Binária

- Variáveis:  $x_i$   $i = 1, \dots, n$  binária, indicando se o item  $i$  está na mochila.
- Função objetivo:  $\max \sum_{i=1}^n v_i x_i$ . Maximizar o valor da mochila
- Restrição:  $\sum_{i=1}^n w_i x_i \leq W$ . O peso máximo da mochila deve ser respeitado.

Qual o problema obtido ao se relaxar a restrição de integralidade dos  $x_i$ ?  $\rightarrow$  Mochila Fracionária

## Aula 3 Problema do Escalonamento

- Variáveis:  $x_{ij}$   $i = 1, \dots, n$   $j = 1, \dots, m$  binária indicando se a tarefa  $i$  está na máquina  $j$ .
- $L_{\max}$ . Variável que representa o makespan.
- Função objetivo:  $\min L_{\max}$ . Minimizar o *makespan*.
- Restrições:

$$- \sum_{j=1}^m x_{ij} = 1. \text{ Cada tarefa está alocada em uma única máquina.}$$

- $\sum_{i=1}^n t_i x_{ij} \leq L_{\max} \quad j = 1, \dots, m.$  O *makespan* tem que ser  $\geq$  à carga de qualquer máquina.

- Domínio das variáveis:

- $x_{ij} \in \{0, 1\} \quad i = 1, \dots, n \quad j = 1, \dots, m$
- $L_{\max} \in \mathbb{R}$

É um problema de Programação Linear misto, porque tem variáveis inteiras ( $x_{ij}$ ) e contínuas ( $L_{\max}$ ) junto.

## Aula 4 Problema da Colocação de Grafos

- Variáveis:

- $x_{uk} \quad u \in V \quad k = 1, \dots, \Delta(G) + 1.$  Indica se o vértice  $u$  tem cor  $k$ .  $\Delta(G)$  é o grau máximo no grafo.
- $y_k \quad k = 1, \dots, \Delta(G) + 1.$  Indica se a cor  $k$  é usada.

- Função objetivo:  $\min \sum_{k=1}^{\Delta(G)+1} y_k.$  Minimizar o número de cores usadas.

- Restrições:

- $\sum_{k=1}^{\Delta(G)+1} x_{uk} = 1 \quad \forall u \in V.$  Cada vértice tem apenas uma cor.
- $x_{uk} + x_{vk} \leq 1 \quad \forall (u, v) \in E \quad k = 1, \dots, \Delta(G) + 1.$  Vértices adjacentes tem cores diferentes.
- $x_{uk} \leq y_k \quad \forall u \in V \quad k = 1, \dots, \Delta(G) + 1.$  Um vértice só pode usar uma cor que esteja disponível (cujo  $y_k = 1$ ).

- Domínio das variáveis:

- $x_{uk} \in \{0, 1\} \quad u \in V \quad k = 1, \dots, \Delta(G) + 1$
- $y_k \in \{0, 1\} \quad k = 1, \dots, \Delta(G) + 1$

O número máximo de cores é  $\Delta(G) + 1$  porque o pior caso é quando o nó com maior grau tem todos os vizinhos com cores diferentes, então ele precisa ser colorido com uma cor nova.

## Aula 5 Problema da Localização de Instalações sem Capacidades

- Variáveis:

- $y_i \quad i \in F.$  Indica se a instalação  $i$  foi aberta.

- $x_{ij} \quad i \in F \quad j \in D$ . Indica se cliente  $j$  se conecta à instalação  $i$ .
- Função objetivo:  $\min \sum_{i \in F} f_i y_i + \sum_{j \in D} \sum_{i \in F} d_{ji} x_{ij}$ . Minimizar o custo de abertura mais custo de conexão.
- Restrições:
  - $\sum_{i \in F} x_{ij} = 1 \quad \forall j \in D$ . Cada cliente se conecta a apenas uma instalação.
  - $x_{ij} \leq y_i \quad \forall i \in F \quad \forall j \in D$ . Cliente só pode se conectar a uma instalação aberta.
- Domínio das variáveis:
  - $x_{ij} \in \{0, 1\} \quad \forall i \in F \quad \forall j \in D$
  - $y_i \in \{0, 1\} \quad i \in F$

## Aula 6 Problema da Cobertura por Conjuntos

- **Entrada:** conjunto de elementos  $U = \{e_1, e_2, \dots, e_n\}$ , uma coleção de subconjuntos  $S_1, S_2, \dots, S_m$ , cada subconjunto  $S_j \subseteq U$  com peso  $w_j$ .
- **Soluções viáveis:** uma coleção de subconjuntos que cobre  $U$ , i.e., encontrar  $I \subseteq \{1, 2, \dots, m\}$  tal que  $\bigcup_{j \in I} S_j = U$ .
- **Função objetivo:** custo total dos subconjuntos em  $I$ , i.e.,  $\sum_{j \in I} w_j$ .
- **Objetivo:** encontrar coleção de custo mínimo.

### 6.1 Formulação em PLI

- Variáveis:  $x_j \quad j = 1, \dots, m$ . Indica se o conjunto  $j$  foi escolhido.
- Função objetivo:  $\min \sum_{j=1}^m w_j x_j$ . Minimizar o custo dos subconjuntos escolhidos.
- Restrição:  $\sum_{j: e_i \in S_j} x_j \geq 1 \quad i = 1, \dots, n$ . Todo elemento deve ser coberto por ao menos um subconjunto.
- Domínio das variáveis:  $x_j \in \{0, 1\} \quad j = 1, \dots, m$

## Aula 7 Resolvedores de PLI

Ferramenta para fazer a modelagem de PLI: OR-tools. Gratuito.

Resolvedores mais famosos: Gurobi, CPLEX. São resolvedores proprietários.

## 7.1 OR-Tools

Usar o `pywraplp` do `ortools.linear_solver`. Usar um resolvido `pywraplp.Solver`. O que tem que fazer é criar as variáveis em uma lista.

```
x = list()
for j in range(0, num_sets):
    # Variaveis inteiras entre 0 e 1 = binarias
    # Toda variavel tem um nome x[j]
    x.append(solver.IntVar(0.0, 1.0, 'x[{}]' .format(j)))
```

Depois, criamos as restrições (*constraints*) para cada elemento  $e_i \in U$ .

```
constraintType1 = list()
for i in range(0, num_elements):
    # Um constraint tem dois limitantes (inferior e superior)
    constraintType1.append(solver.Constraint(1, solver.
infinity()))
```

Na parte direita da inequação ( $\sum_{j:e_i \in S_j}$ ), nós levamos em consideração todos os conjuntos que possuem o elemento  $e_i$ . Então temos que passar por todos os conjuntos e seus elementos para indicar a restrição, aplicar aquele *constraint* para aquele conjunto em específico (colocar o coeficiente como 1).

```
for s in set_list:
    for i in s.elements:
        constraintType1[i].Setcoefficient(x[s.index], 1)
```

Também temos que definir a função objetivo.

```
objective = solver.Objective()
objective.SetMinimization() # A funcao eh de minimizacao
for j in range(0, num_sets):
    objective.SetCoefficient(x[j], set_list[j].cost)
```

Depois de modelado, podemos apenas resolver usando `solver.Solve()`

## Aula 8 Problema do corte máximo

- Variáveis:

- $x_u \quad \forall u \in V$ . Indica se o vértice  $u$  esta no corte  $S$ .
- $y_e \quad \forall e \in E$ . Indica se a aresta atravessa o corte  $S$ .

- Função objetivo:  $\max \sum_{e \in E} y_e$ . Maximizar o número de arestas no corte.
- Restrições:
  - $y_e \leq x_u - x_v + M \times a_e \quad \forall e = (u, v) \in E$ . A aresta só atravessa o corte se os extremos estão em grupos distintos (direção  $u \rightarrow v$ ).
  - $y_e \leq x_v - x_u + M \times (1 - a_e) \quad \forall e = (u, v) \in E$ . Direção  $v \rightarrow u$ .
- Domínio das variáveis:
  - $x_u \in \{0, 1\} \quad \forall u \in V$
  - $y_e \in \{0, 1\} \quad \forall e \in E$
  - $a_e \in \{0, 1\} \quad \forall e \in E$

Usa o  $M$ , um número grande, e uma variável auxiliar  $a_e$  binária associada a cada aresta. Isso seleciona apenas uma das restrições a funcionar em cada tempo (tipo o módulo, mas módulo **não** é função linear). São duas restrições complementares. Por ser uma variável, o resolvedor vai escolher os melhores valores para  $a_e$  para garantir o resultado.

## Aula 9 Problema de Steiner

- **Entrada:**  $G = (V, E)$ , com  $V = R \cup S$ , sendo  $R$  terminais e  $S$  vértices de Steiner, e função  $w$  de peso nas arestas.
- **Soluções viáveis:** árvores que conectam todos os vértices em  $R$ .
- **Função objetivo:** soma dos pesos das arestas na árvore.
- **Objetivo:** encontrar uma árvore de peso mínimo.

### 9.1 Formulação em PLI

- Variáveis:  $x_e \quad \forall e \in E$ . Indica se a aresta  $e$  está na árvore.
- Função objetivo:  $\min \sum_{e \in E} w_e x_e$ . Minimizar o custo da árvore.
- Restrições:  $\sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S : S \cap R \neq \emptyset, S \cap R \subset R$  Qualquer corte  $S$  que separa os terminais deve ter aresta atravessando.
- Domínio das variáveis:  $x_e \in \{0, 1\} \quad \forall e \in E$ .

# PROGRAMAÇÃO LINEAR - CONTINUAÇÃO

Garante a solução ótima, mas sem garantir o tempo polinomial.

## Aula 1 Programação linear

Diferente do PLI, aqui as variáveis podem ser **reais**, não necessariamente inteiras.

As variáveis são sempre positivas. Se precisar de variáveis que possam ser negativas, usa-se duas variáveis na verdade, sempre em dupla ( $x_i^+ - x_i^-$ ). A combinação das duas variáveis podem gerar os valores positivos ou negativos necessários.

### 1.1 Exepmlo de PL para produção

Dois tipos de produto. Cada produto do tipo 1 dá 1 de lucro, e cada produto do tipo 2 dá 2 de lucro.

Maximizar:  $x_1 + 2x_2$

Mas existem insumos (restrições).

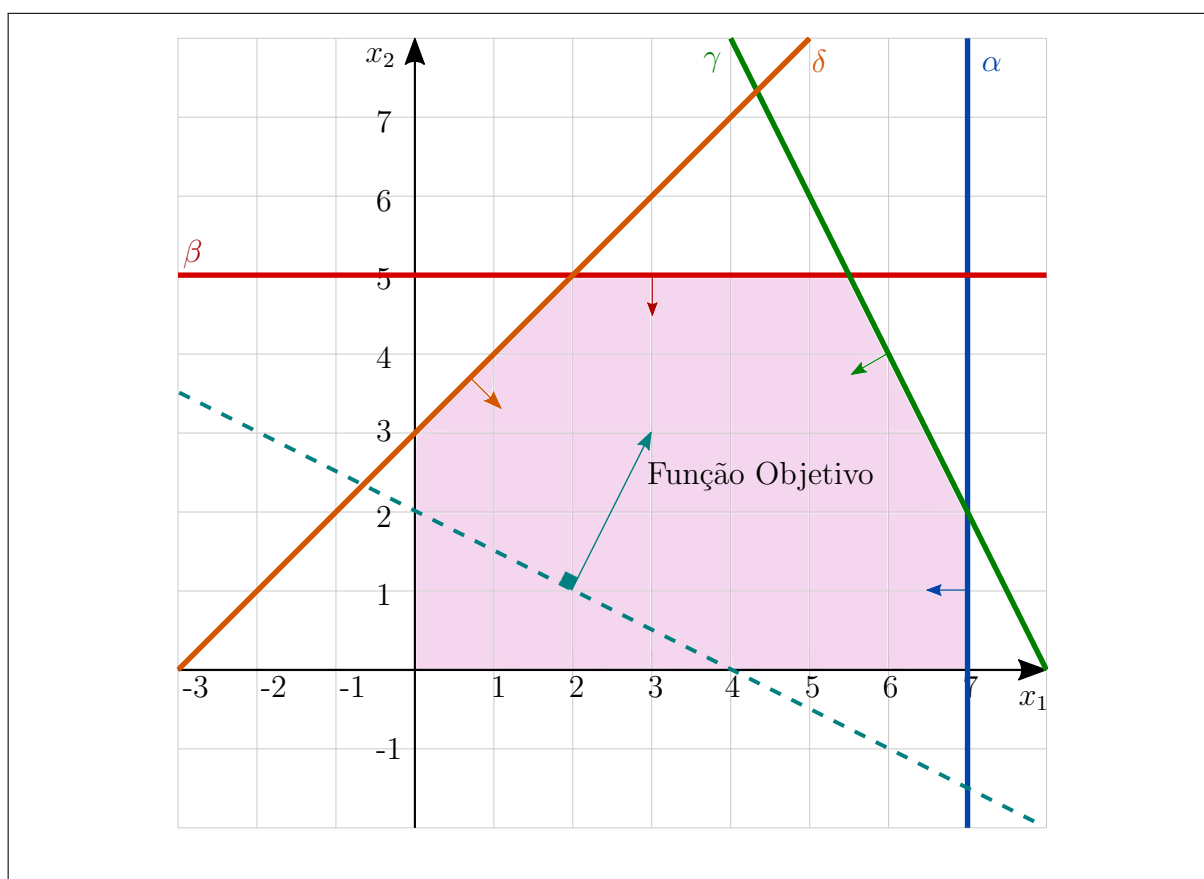
- $x_1 \leq 7 \rightarrow \alpha$
- $x_2 \leq 5 \rightarrow \beta$
- $2x_1 + x_2 \leq 16 \rightarrow \gamma$
- $-x_1 + x_2 \leq 3 \rightarrow \delta$

A produção também não pode ser negativa.

- $x_1 \geq 0$
- $x_2 \geq 0$

É possível visualizar as restrições no plano.

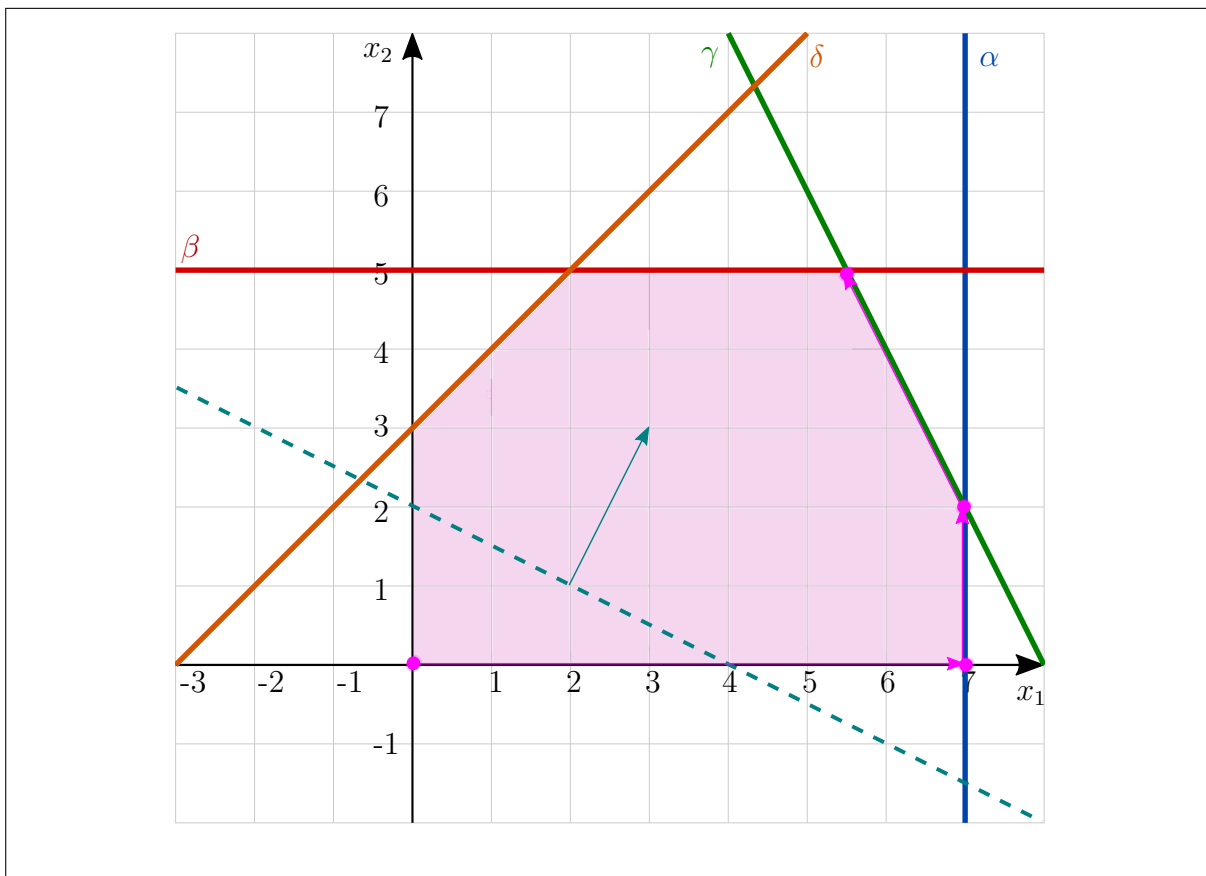




A função objetivo é representada por um vetor (o que importa é o comprimento, direção e sentido). Esse vetor é ortogonal a uma reta em que  $x_1 + 2x_2$  é constante. Ao longo dessa reta, o valor da função objetivo não muda.

## Aula 2 Intuição geométrica do Simplex

A ideia é migrar de um vértice para um vértice vizinho do poliedro (a área dentro das restrições) sempre melhorando a função objetivo. → **Busca local**



Iniciando no ponto  $(0,0)$ , vamos aumentando o valor de  $x_1$  e, em seguida, o valor de  $x_2$ , já que o vetor da função objetivo indica que o valor aumenta conforme aumenta o  $x_1$  e, mais ainda, o  $x_2$ .

Depois nós voltamos, porque, apesar de diminuir o valor de  $x_1$ ,  $x_2$  contribui mais para o valor da função objetivo. E chegamos então ao máximo local. Os únicos dois vizinhos são o ponto de onde veio  $(7,2)$  e o ponto seguinte  $(2,5)$ . Mas os dois têm valores piores que o ponto atual.

## 2.1 Convexidade

Essa estratégia funciona pois o poliedro é convexo (qualquer segmento de reta ligando dois pontos do conjunto está dentro do conjunto).

Todo conjunto definido pelas restrições lineares (chamado de semi-espço) é convexo. Porque o formato das restrições são sempre “os pontos que estão acima (ou abaixo) desse plano”. E a intersecção de duas ou mais restrições também é convexa. Então **todo** poliedro resultante é convexo.

Como o poliedro é convexo, **todo ótimo local é ótimo global**. E isso também garante que **sempre existe um ótimo em algum vértice**.

## 2.2 Algoritmo Simplex

Precisamos primeiramente colocar o PL na forma padrão, usando **variáveis de folga**. O método Simplex **não** trabalha com inequações, então precisa dessas variáveis de folga.

Exemplo:  $x_1 \leq 7 \Rightarrow x_1 + s_\alpha = 7$ . Quem vai fazer o papel da desigualdade, é a variável de folga  $s_\alpha$ .

- Função objetivo:  $x_1 + 2x_2$

- Restrições:

- $x_1 + s_\alpha = 7$
- $x_2 + s_\beta = 5$
- $2x_1 + x_2 + s_\gamma = 16$
- $-x_1 + x_2 + s_\delta = 3$
- $x_1, x_2, s_\alpha, s_\beta, s_\gamma, s_\delta \geq 0$

Cria uma solução básica inicial. Cada vértice corresponde a uma solução básica (só tem **uma** variável diferente de 0 para cada restrição).

Enquanto houver uma variável  $x_j$  fora da base (com valor 0), com coeficiente  $c_j$  positivo na função objetivo, faça:

- seja  $i$  uma restrição com  $a_{ij}$  negativo que minimiza  $\frac{b_i}{-a_{ij}}$ , sendo  $b_i$  o lado direito dessa restrição e  $a_{ij}$  o coeficiente de  $x_j$  na restrição
  - $a_{ij}$  deve ser negativo, para que  $x_j$  seja positivo
  - $\frac{b_i}{-a_{ij}}$  deve ser mínimo para que  $x_j$  pare de aumentar ao chegar em uma restrição. Caso contrário, alguma variável ficará negativa.
- Faça o pivoteamento. Insira  $x_j$  na solução básica no lugar da variável  $i$ , faça eliminação de Gauss e atualize a função objetivo.

Solução básica inicial:

- $x_1 = 0, x_2 = 0$
- $s_\alpha = 7, s_\beta = 5, s_\gamma = 16, s_\delta = 3$

Escolhe uma variável não básica com coeficiente positivo na função objetivo:  $x_1$ .

Primeiro nós isolamos as variáveis de folga das restrições.

$$\begin{aligned}s_\alpha &= 6 - x_1 \\ s_\beta &= 5 - x_2 \\ s_\gamma &= 16 - 2x_1 - x_2 \\ s_\delta &= 3 + x_1 - x_2\end{aligned}$$

O  $a_{ij}$  deve ser negativo. Se pegarmos, por exemplo, a restrição  $\delta$ , que tem coeficiente +1 para  $x_1$ , temos:  $x_1 = -3 + s_\delta + x_2 = -3$ . Isso ocorre, porque estamos “tirando”  $s_\delta$  da solução (seu valor passa a ser 0), para que possamos colocar  $x_1$  na base. Lembrando que no máximo um valor em cada restrição pode ser diferente de 0 para que nós nos matenhamos em um vértice do poliedro. Mas essa solução (com o  $x_1 = -3$ ) não é viável, pois as variáveis não podem ter valores negativos.

Outra condição necessária é que  $\frac{b_i}{-a_{ij}}$  deve ser mínimo. Para as restrições com  $x_1$  de coeficiente negativo ( $\alpha$  e  $\gamma$ ), temos:  $\frac{7}{-(-1)} = 7 \rightarrow \alpha$  e  $\frac{16}{-(-2)} = 8 \rightarrow \gamma$ .

Tomando a restrição  $\gamma$ , **errada**, pois não minimiza, temos:  $x_1 = \frac{16-s_\gamma-x_2}{2} = 8$ .  $x_1$  não ficou com um valor inviável por ser negativa, mas passa a desrespeitar a restrição  $\alpha$  ( $x_1 \leq 7$ ). Isso porque, com a variável de folga:  $s_\alpha = 7 - x_1 = 7 - 8 = -1$ . A variável de folga fica com valor negativo, o que é errado, pois **nenhuma** variável pode ser negativa.

Escolhendo, **de forma correta**, a restrição  $\alpha$ , com  $\frac{7}{-(-1)} = 7$  mínimo, temos:  $x_1 = 7 - s_\alpha = 7$ . E devemos atualizar o valor das outras variáveis.

$$\begin{aligned}s_\alpha &= 0 \\ s_\beta &= 5 - x_2 = 5 \\ s_\gamma &= 16 - 2(7 - s_\alpha) - x_2 = 2 + s_\alpha - x_2 = 2 \\ s_\delta &= 3 + 7 - s_\alpha - x_2 = 10 \\ \text{E atualizamos a função objetivo: } &7 - s_\alpha + 2x_2 = 7\end{aligned}$$

Temos  $n$  variáveis (ignorando as variáveis de folga) e  $m$  restrições (ignorando as restrições de não negatividade). As variáveis de folga indicam a distância do ponto atual até a borda da restrição correspondente. No caso do exemplo, temos  $n = 2$  e  $m = 4$ .

O espaço em que temos é  $n$ -dimensional, ou seja, o número de dimensões é igual ao número de variáveis. Se uma solução repousa em uma intersecção de restrições, essa solução é um vértice  $\rightarrow$  É necessário que o número de restrições dentro da base seja igual ao número de variáveis para garantir que teremos um ponto. Por exemplo, a intersecção de 2 planos em um espaço tridimensional é uma reta, mas com um terceiro plano, temos um ponto de intersecção.

A mudança do ponto se dá, quando deslocamos esse ponto ao longo de uma restrição que esteja **fora da base**, aquela que tem variável de folga igual a zero.

### Aula 3 Algoritmo Simplex - Continuação

Paramos, na iteração anterior, no caso em que:

$$x_1 = 7, x_2 = 0$$

$$x_1 = 7 - s_\alpha$$

$$s_\beta = 5 - x_2$$

$$s_\gamma = 2 + 2s_\alpha - x_2$$

$$s_\delta = 10 - s_\alpha - x_2$$

Calculando  $\frac{b_i}{-a_{i2}}$  para cada restrição, temos:

$$\beta \rightarrow \frac{5}{-(-1)} = 5$$

$$\gamma \rightarrow \frac{2}{-(-1)} = 2$$

$$\delta \rightarrow \frac{10}{-(-1)} = 10$$

O valor que minimiza é o da restrição  $\gamma$ . Dessa forma, tiramos  $s_\gamma$  da base e colocamos  $s_2$  na base.

$$\text{Função objetivo: } 7 - s_\alpha + 2(2 + 2s_\alpha - s_\gamma) = 11 + 3s_\alpha - 2s_\gamma = 11$$

$$x_1 = 7 - s_\alpha = 7$$

$$s_\beta = 5 - (2 + 2s_\alpha - s_\gamma) = 3 - 2s_\alpha + s_\gamma = 3$$

$$x_2 = 2 + 2s_\alpha - s_\gamma = 2$$

$$s_\delta = 10 - s_\alpha - (2 + 2s_\alpha - s_\gamma) = 8 - 3s_\alpha + s_\gamma = 8$$

Iniciando uma nova iteração. Na função objetivo  $(11 + 3s_\alpha - 2s_\gamma)$ , uma variável fora da base que tenha coeficiente positivo é  $s_\alpha$ . Vamos então colocar  $s_\alpha$  na base, naquelas restrições que possuem coeficiente negativo.

$$1 \rightarrow \frac{7}{-(-1)} = 7$$

$$\beta \rightarrow \frac{3}{-(-2)}$$

$$\delta \rightarrow \frac{8}{-(-3)}$$

O que minimiza é o valor  $\frac{3}{2}$ , então tiramos  $s_\beta$  da base e colocamos  $s_\alpha$ . Com isso, temos:

$$\begin{aligned}
\text{Função objetivo: } 11 + 3\left(\frac{3}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2}\right) - 2s_\gamma &= \frac{31}{2} - \frac{1}{2}s_\gamma - \frac{3}{2}s_\beta = \frac{31}{2} \\
x_1 = 7 - \left(\frac{3}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2}\right) &= \frac{11}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2} = \frac{11}{2} \\
s_\alpha = \frac{3}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2} &= \frac{3}{2} \\
x_2 = 2 + 2\left(\frac{3}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2}\right) - s_\gamma &= 2 + 3 + s_\gamma - s_\beta - s_\gamma = 5 - s_\beta = 5 \\
s_\delta = 8 - 3\left(\frac{3}{2} + \frac{s_\gamma}{2} - \frac{s_\beta}{2}\right) + s_\gamma &= 8 - \frac{9}{2} - \frac{3}{2}s_\gamma + \frac{3}{2}s_\beta + s_\gamma = \frac{7}{2} - \frac{1}{2}s_\gamma + \frac{3}{2}s_\beta = \frac{7}{2}
\end{aligned}$$

Todos os coeficientes das variáveis na função objetivo são negativos. Então estamos em um ótimo local/global. Isso se dá porque, a partir desse ponto, a direção do vetor aponta para uma região fora da área de soluções viáveis. O resultado final é, portanto:

$$\begin{aligned}
x_1 &= \frac{11}{2}, x_2 = 5 \\
s_\alpha &= \frac{3}{2} \\
s_\beta &= 0 \\
s_\gamma &= 0 \\
s_\delta &= \frac{7}{2}
\end{aligned}$$

## Aula 4 Observações adicionais ao Simplex

Cada variável ao modelo de PL adiciona uma dimensão ao espaço de busca, então esses modelos podem ficar bastante complexos, já que existem casos em que podemos ter milhares/milhões de variáveis (ou dimensões).

Temos também que um vértice pode corresponder a mais de uma solução básica, quando mais de  $n$  restrições se coincidirem naquele mesmo ponto. Nesses casos, pode-se trocar a solução básica, mas sem melhorar função objetivo. É preciso lidar com isso, às vezes trocando a base sem melhorar a função objetivo para que a restrição que melhora o resultado possa ser percorrida.

Nem sempre é direto se obter uma solução viável. Mas existe uma maneira de garantir isso usando o Simplex. Adiciona-se uma variável a mais (de viabilidade) nas restrições. Cria-se uma função objetivo “virtual” usando apenas as variáveis de viabilidade com coeficientes negativos. Ao maximizar essa função, temos todas as variáveis de viabilidade em 0, isso faz o algoritmo encontrar uma solução viável (mas não ótima), e usamos essa solução como inicialização do Simplex de otimização. Como as variáveis de viabilidade são 0, elas não são mais utilizadas.

A implementação do Simplex é feita utilizando operações sobre matrizes (pivoteamento...) em uma matriz chamada *tableau*.

Esse é um algoritmo bastante clássico, mas não é polinomial no pior caso, é exponencial, mas na prática ele tende a rodar em tempo polinomial normalmente. Algoritmos polinomiais no pior caso são: elipsoide e algoritmo dos pontos interiores.

## SEMANA 6

### BRANCH AND BOUND PARA PLI E LIMITANTES

Resolvedores de PLI usam uma combinação do método de *Branch and Bound* junto com a resolução de Programas Lineares (usando o Simplex por exemplo).

#### Aula 1 Branch and Bound - Introdução

A ideia básica é fazer uma busca exaustiva inteligente das soluções através de enumeração (branch) e poda (bound) das soluções ruins.

A enumeração das soluções é feita de acordo com os valores das variáveis inteiras. A ordem para se percorrer a árvore pode afetar a eficiência do algoritmo, normalmente percorre-se a árvore de acordo com o valor do limitante dual dos nós. O limitante dual indica que “a partir desse ramo não é possível obter uma solução melhor que esse valor”.

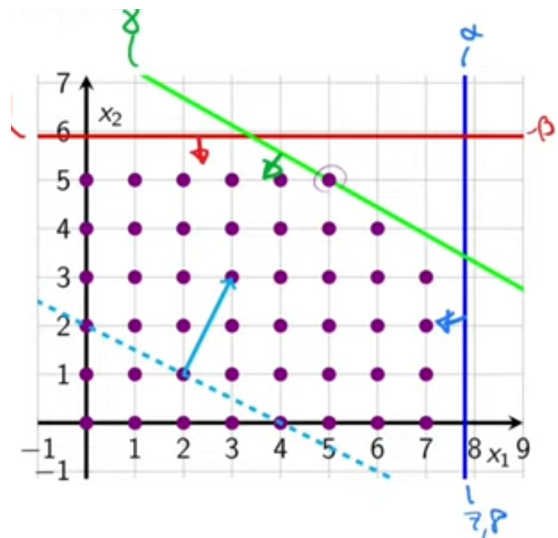
A poda dos ramos é feita de acordo com a viabilidade ou com a qualidade das soluções. A viabilidade é indicada pelas restrições do PLI. Para verificar se um nó é promissor ou não, vemos se o limitante dual do nó é pior que a melhor solução já encontrada  $\rightarrow$  se o melhor do ramo for pior que o melhor que eu já tenho, ele não é promissor.

#### Aula 2 Exemplo de Branch and Bound

- Função objetivo: maximizar  $x_1 + 2x_2$

- Restrições:

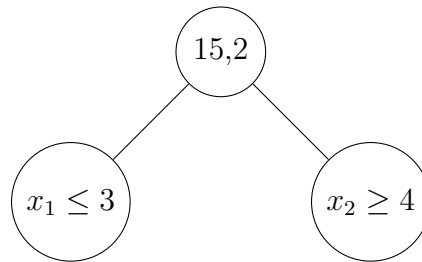
- $x_1 \leq 7, 8 \rightarrow \alpha$
- $x_2 \leq 5, 9 \rightarrow \beta$
- $9x_1 + 16x_2 \leq 125 \rightarrow \gamma$
- $x_1, x_2 \in \mathbb{Z}^+$



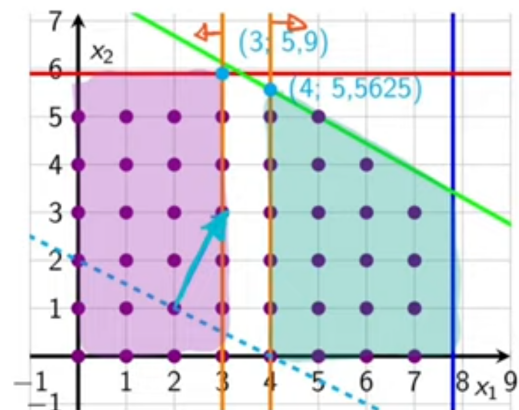
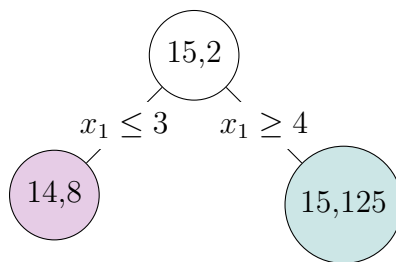
Para resolver o problema, primeiro relaxamos a integralidade do PLI ( $x_1, x_2 \in \mathbb{R}^+$ ) para encontrar o limitante dual das soluções através da solução de um problema de PL (através do Simplex, por exemplo).

Temos, por exemplo, como solução o ponto  $(3, 4; 5, 9)$  com valor da função objetivo  $15,2$ . Com isso, a melhor solução (do PL) tem função objetivo  $15,2$  como limitante dual (nenhuma solução do PLI vai ser melhor que isso).

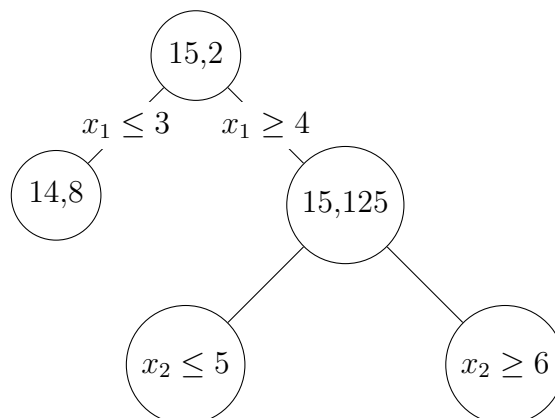
Fazemos primeiro o *branch* em torno da variáveis mais fracionária (em que o PL tem “mais dúvida”).



Agora temos dois subproblemas (dois novos polígonos) e repetimos o procedimento para cada novo subproblema (usamos o PL para calcular o limitante dual).

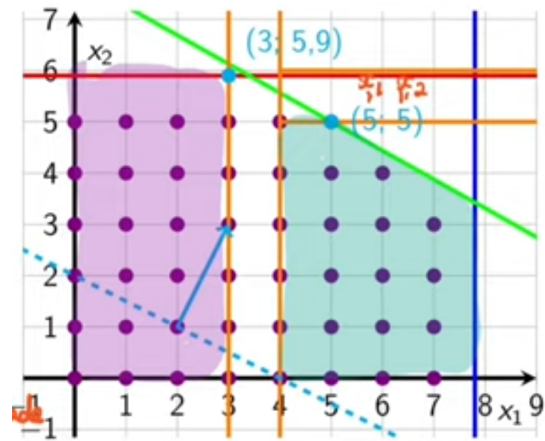
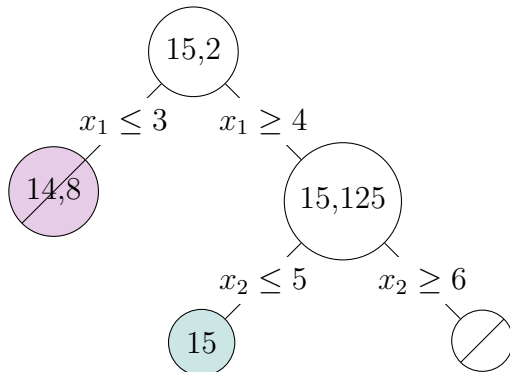


A partir daqui, podemos explorar seguindo aquele ramo que possui “maior espaço para melhoria”. Ou seja, aquele cujo limitante dual esteja mais próximo do limitante dual original, nesse caso o ramo da direita. temos então como solução limitante  $(4; 5, 5625)$ .  $x_1$  não está fracionária, então fazemos a ramificação com a variável  $x_2$ .





Com isso, temos:



O ramo da direita não é mais explorado (é podado), pois temos uma inviabilidade, já que  $x_2 \geq 6$  desrespeita a restrição  $\beta$ . Também não precisamos continuar explorando o ramo da esquerda, porque a solução do limitante dual já é inteira no ponto  $(5; 5)$ , então é a melhor solução inteira até o momento.

Deveríamos então continuar a explorar o subproblema em que  $x_1 \leq 3$ . Porém, sabemos que seu limitante dual é 14, 8, o que é menor que o valor da melhor solução inteira obtida (15), então não é necessário percorrer esse ramo, pois nunca terá uma solução melhor.

Como não temos mais nenhum nó ativo para ser percorrido, o algoritmo termina e temos como solução o ponto  $(5; 5)$  com valor da função objetivo 15.

### Aula 3 Pseudocódigo do Branch and Bound

Enquanto houver **nós ativos**, escolher aquele com melhor (maximização/minimização) limitante dual.

- Se o limitante dual for pior que a melhor solução já encontrada: eliminar o nó.
- Se a solução do PL desse nó for inteira, temos uma nova solução.
  - Finalize o nó e atualize a melhor solução encontrada.
- Caso contrário, ramifique o nó em torno da variável  $x$  com valor  $v$  mais fracionário (cujas partes decimais estão mais perto de 0,5) na solução do PL.
  - Crie dois subproblemas com  $x \leq \lfloor v \rfloor$  e  $x \geq \lceil v \rceil$
  - Calcule o limitante dual do nó correspondente a cada subproblema
    - \* Pode, por inviabilidade, podar aqueles que não tiverem solução

Antes de entrar no laço, calculamos o limitante dual do problema original e fazemos deste o primeiro nó.

## Aula 4 Problema da Mochila

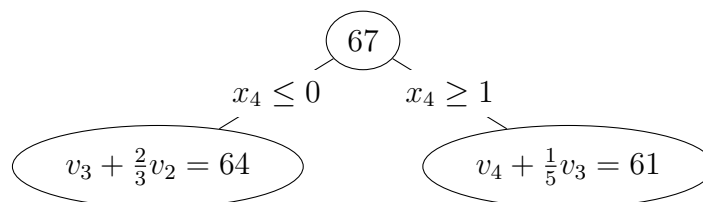
- Variáveis:  $x_i \in \{0, 1\} \quad \forall i \in \{1, \dots, n\}$
- Função objetivo:  $\max \sum_{i=1}^n v_i x_i$
- Restrição:  $\sum_{i=1}^n w_i x_i \leq W$

Obtemos o PL, relaxando as restrições de integralidade  $\rightarrow$  mochila fracionária.

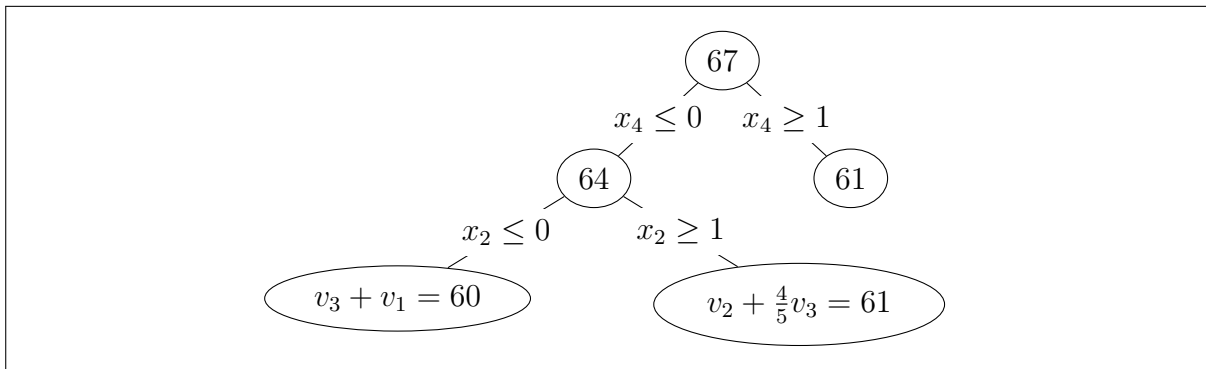
O problema da mochila fracionária obtém a melhor solução através de um algoritmo guloso. Então podemos usar ele ao invés do Simplex.

- $w_1 = 2 \quad v_1 = 10 \quad \frac{v_1}{w_1} = 5$
- $w_2 = 3 \quad v_2 = 21 \quad \frac{v_2}{w_2} = 7$
- $w_3 = 5 \quad v_3 = 50 \quad \frac{v_3}{w_3} = 10$
- $w_4 = 7 \quad v_4 = 51 \quad \frac{v_4}{w_4} = 8,5$
- $W = 7$

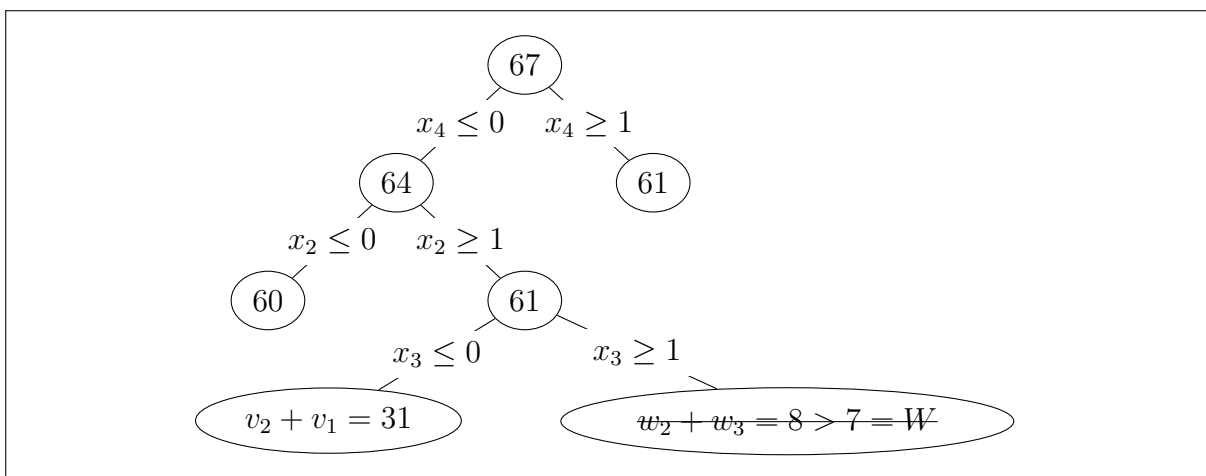
Obtendo a melhor solução para o PL, usando o algoritmo guloso, temos que o limitante dual é:  $v_3 + \frac{2}{6}v_4 = 50 + 17 = 67$ . A solução não é inteira, pois  $x_4 = \frac{1}{3}$ . Ramificamos na variável mais fracionária ( $x_4$ ).



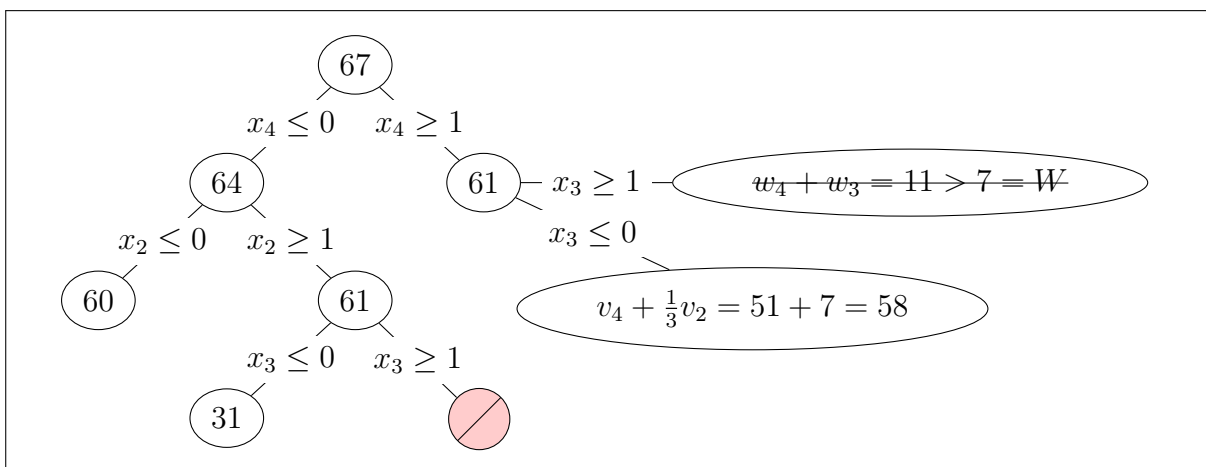
Escolhemos o melhor limitante dual (como o problema é de maximização, pega-se o maior) e ramificamos de novo na variável mais fracionária ( $x_2$ ). A restrição de que  $x_4 \leq 0$  continua valendo durante essa ramificação.



Ramificamos novamente, o nó com limitante dual 61.

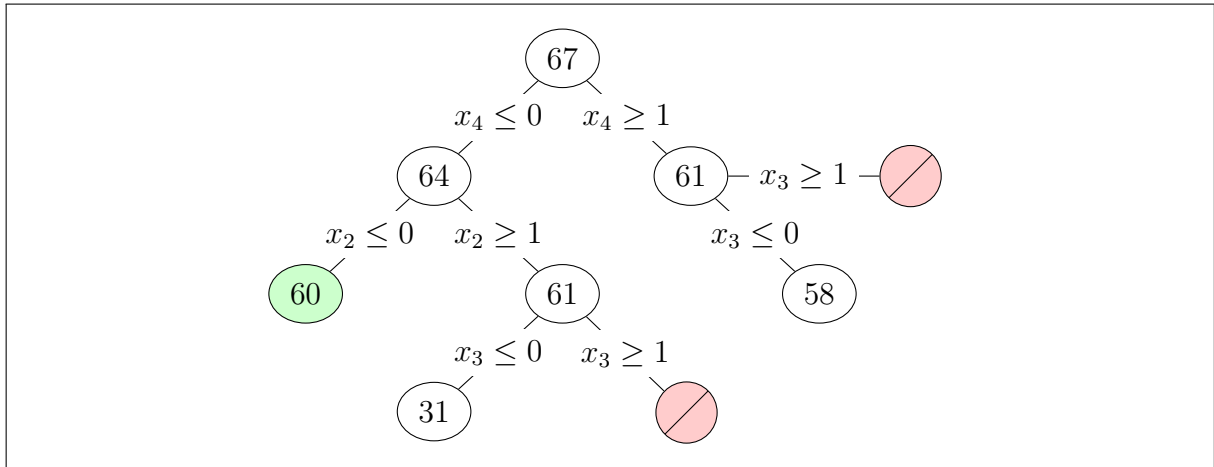


Removemos o nó em que  $x_3 \geq 1$  porque ao ter  $x_2$  e  $x_3$  inteiramente na mochila, o peso é excedido, então a solução não é viável. Seguimos ramificando no nó com o melhor limitante dual (61), em que  $x_4 \geq 1$ .

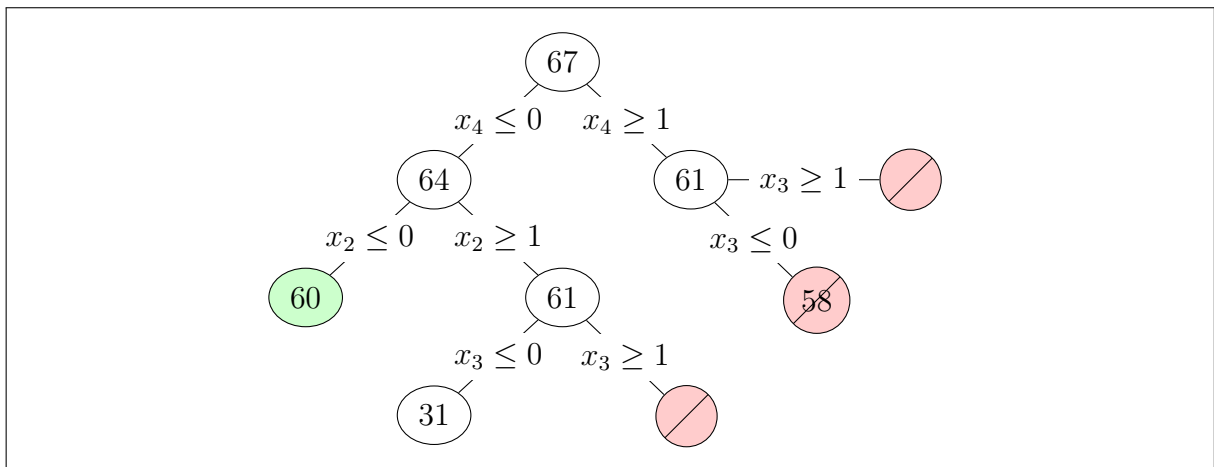


Removemos mais um nó por inviabilidade. Continuamos explorando os nós ativos com maior limitante dual. Nesse caso, o de valor 60, em que  $x_4 \leq 0, x_2 \leq 0$ . Nesse caso, a solução existente já é inteira. Com  $x_3 = 1, x_1 = 1$  e limitante dual 60. Portanto, essa é

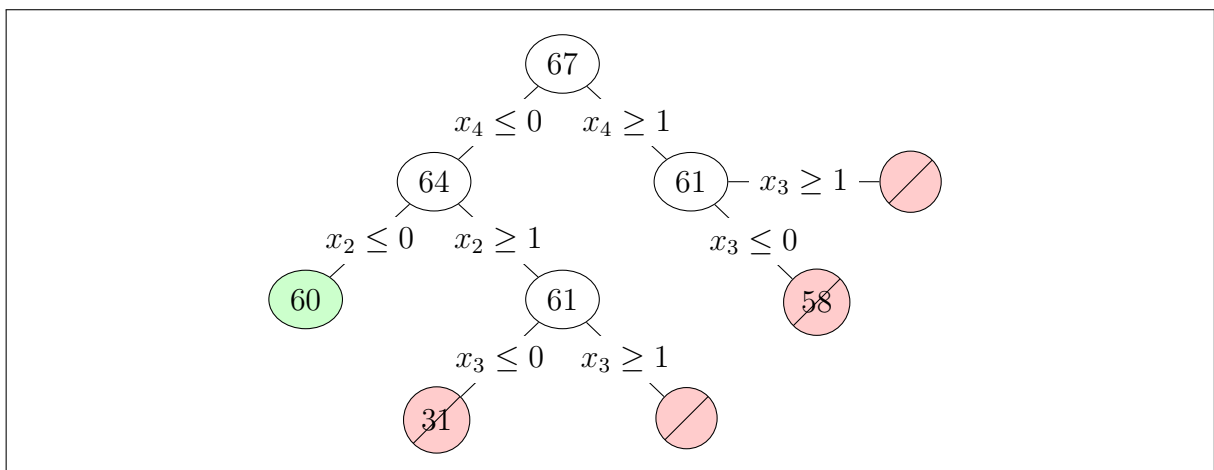
a melhor solução obtida até o momento, essa solução é chamada de **limitante primal**. Ele é desativado, por ser o melhor possível.



Seguimos explorando os outros nós ativos em ordem. Seguindo o nó com limitante dual 58. Como esse limitante é menor que o limitante primal encontrado anteriormente, o nó é apenas podado por limitante.



Em seguida, o nó com valor limitante 31 também é podado por limitante.



## Aula 5 Eficiência do Branch and Bound

A eficiência depende do número de podas, que depende de:

- limitante primal (melhor solução encontrada até o momento)
- limitante dual (solução do PL relaxado para o nó sendo avaliado)

Então devemos encontrar **limitantes mais fortes**.

**Gap:** diferença entre o limitante primal e o limitante dual “mais frouxo” (melhor resultado) dos nós ativos. → `RELATIVE_MIP_GAP`

### 5.1 Fortalecendo limitantes primais

Pode usar heurísticas e passar essa informação para o resolvidor de PLI. → `SetHint`

Também é possível arredondar a solução do PL para obter uma solução inteira. Tem que prestar atenção no sentido das desigualdades para saber se arredonda para cima ou para baixo. → Fazer isso abre mão de encontrar a solução ótima

É comum que resolvidores tenham como setar um *callback* para chamar depois de encontrar a solução da relaxação. Pode usar isso para resolver o arredondamento. → Não tem no Or-tools, mas tem no Gurobi (pago)

Em geral, o valor de uma variável indica o quanto ela é “interessante” para a solução do problema.

### 5.2 Fortalecendo limitantes duais

O limitante dual é obtido da solução do PL vindo da relaxação. → Temos que melhorar a formulação do PL

Devemos encontrar **novas desigualdades** que **não** eliminam nenhuma solução **inteira** válida, mas que elimine soluções fracionárias da região viável. “Diminui a área da região viável, aproximando as desigualdades do PL aos pontos inteiros”

Envoltória convexa, fecho convexo ou *convex hull* → Menor conjunto convexo que possui todos os pontos de um conjunto. **Todo vértice é inteiro, desde que o conjunto só tenha pontos inteiros.**

Se encontrarmos a envoltória convexa do problema e executar a relaxação nesta área, vamos sempre obter a solução ótima. → Encontrar a envoltória convexa é NP-Difícil

## Aula 6 Problema do Conjunto Independente

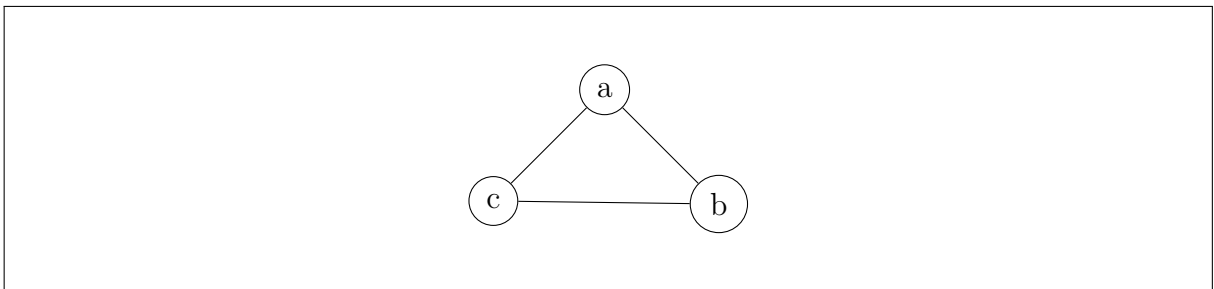
- **Entrada:** grafo  $G = (V, E)$

- **Soluções viáveis:** um conjunto  $S \subseteq V$  tal que toda aresta  $(u, v) \in E$  tem no máximo um extremo em  $S$
- **Função objetivo:** tamanho de  $S$
- **Objetivo:** encontrar solução de valor máximo

### 6.1 Formulação em PLI para o Conjunto Independente

- Variáveis:  $x_u \in \{0, 1\} \quad \forall u \in V$
- Função objetivo:  $\max \sum_{u \in V} x_u$
- Restrição:  $x_u + x_v \leq 1 \quad \forall (u, v) \in E$

Na relaxação:  $0 \leq x_u \leq 1$ .



Se no exemplo  $x_a = 0,5; x_b = 0,5; x_c = 0,5$ . O limitante dual é 1,5, enquanto o limitante primal é 1,0.

Sabendo que o grafo é um triângulo (as arestas vão ter que ficar sozinhas em  $S$ ), podemos criar uma outra restrição  $x_a + x_b + x_c \leq 1$ . Essa restrição não elimina nenhuma solução inteira, mas melhora as soluções do PL relaxado.

Isso pode servir a qualquer subgrafo completo (todos os nós conectados).  $\rightarrow$  Clique

Como usar escolhas aleatórias para melhorar os algoritmos. Alguns dos métodos mais avançados (algoritmos de aproximação e meta-heurísticas) costumam usar a aleatoriedade (Ver [última aula da Semana 3](#)).

Trocar decisões **arbitrárias** (determinísticas) por escolhas **aleatórias**.

- Escolha do vértice inicial
- Ordem de se percorrer uma lista
- Construir uma solução viável inicial
- Escolher o vizinho em uma busca local
- Critério de desempate

Assumindo que exista um algoritmo aleatorizado **AlgAleat**, o algoritmo genérico (para um problema de maximização) é:

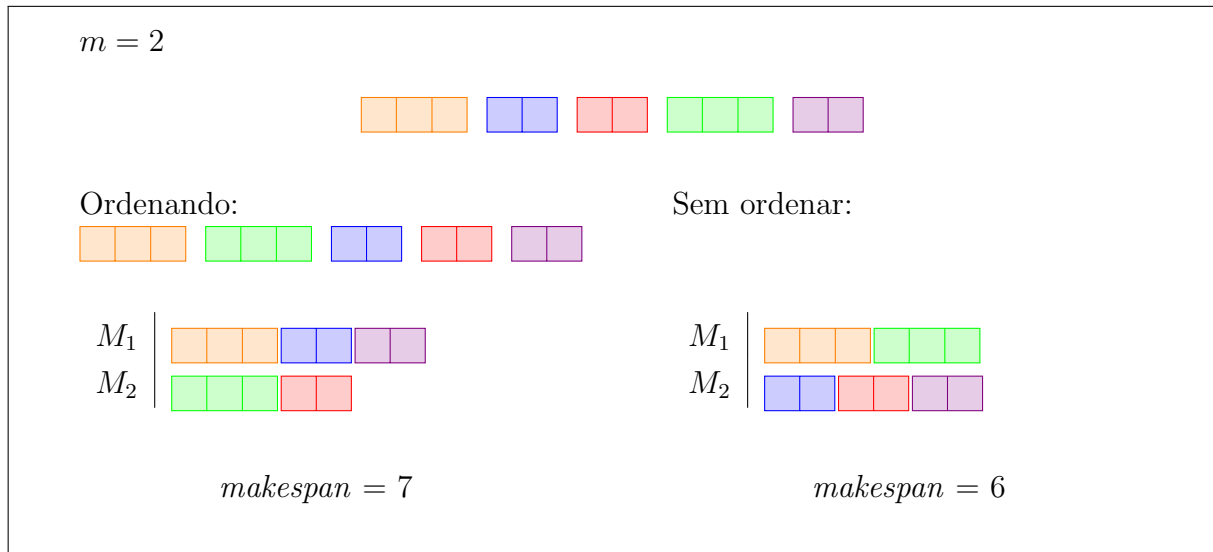
```
Função MultiStart( $I, k$ )  
  best_sol  $\leftarrow -\infty$ ;  
  para  $i \leftarrow 1$  até  $k$  faça  
    curr_sol  $\leftarrow$  AlgAleat( $I$ );  
    se curr_sol  $>$  best_sol então  
      | best_sol  $\leftarrow$  curr_sol;  
    fim  
  fim  
  retorna best_sol  
fim
```

É possível também fazer um algoritmo sensível à melhoria. Por exemplo, fazer parar o algoritmo quando passar  $k$  iterações **sem** melhoria ao invés de ser fixo por  $k$  iterações. Continua procurando enquanto melhorar.

### Aula 1 Problema do Escalonamento

Usaremos o algoritmo **EscalonaGusloso2** visto na [Semana 2](#). No exemplo, encontramos um caso em que é melhor não ordenar as tarefas (ao ordenar o resultado é determinístico), ou seja, podemos criar um algoritmo aleatorizado que pode chegar a um caso não ordenado que me dê um melhor resultado.



**Função EscalonaAleat( $n, t, m$ )**

```

para  $j \leftarrow 1$  até  $m$  faça  $M_j \leftarrow \emptyset$ ;
Escolha uma permutação aleatória dos itens e renomeie de acordo;
para  $i \leftarrow 1$  até  $n$  faça
    seja  $j$  uma máquina em que  $\sum_{i \in M_j} t_i$  é mínimo;
     $M_j \leftarrow M_j \cup \{i\}$ ;
fim
retorna  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$ 
fim

```

**1.1 Heurística de Busca Local**

Podemos aplicar também a aleatoriedade para heurísticas de busca local. Como exemplo, temos a função EscalonaBuscaLocal da [Semana 3](#).

**Função EscalonaBuscaLocal( $n, t, m$ )**

```

 $\mathcal{M} \leftarrow$  um escalonamento inicial;
enquanto houver um item  $i'$  na máquina mais carregada  $j'$  e uma máquina  $j$ 
tal que  $l(j) + t_{i'} < l(j')$  faça
     $M_{j'} \leftarrow M_{j'} \setminus \{i'\}$ ;
     $M_j \leftarrow M_j \cup \{i'\}$ ;
fim
retorna  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$ 
fim

```

Podemos ter mais de uma máquina  $j$  que satisfaça o critério de  $l(j) + t_{i'} < l(j')$ . A escolha de qual máquina vai ser efetivamente usada pode ser feita de forma gulosa (p.

ex. aquela que mais diminui o *makespan*). Mas podemos fazer essa escolha de forma aleatorizada também, escolhendo de forma aleatória que máquina vai receber o item  $i'$ .

## Aula 2 Problema do Corte Máximo

Usando como base o algoritmo `CorteMaximoGuloso` visto na [Semana 2](#), podemos aleatorizá-lo.

```

Função CorteMaximoAleat( $G = (V, E)$ )
     $S \leftarrow \emptyset$ ;
     $\bar{S} \leftarrow \emptyset$ ;
     $A \leftarrow \{s\}$ ; /* Vértices alcançados, existe caminho */
    para  $v \in A$  com maior grau faça
        se  $|\{(v, u) \in \delta(v) : u \in S\}| \leq |\{(v, u) \in \delta(v) : u \in \bar{S}\}|$  então
             $S \leftarrow S \cup \{v\}$ ;
        senão
             $\bar{S} \leftarrow \bar{S} \cup \{v\}$ 
        fim
    fim
fim

```

O algoritmo usa um vértice inicial arbitrário  $s$  e percorre os vértices alcançados também de maneira arbitrária (seguindo os de maior grau). Podemos escolher o vértice inicial aleatoriamente.

É importante salientar que: o número de vértices é linear dado a entrada ( $|V|$ ), então é um número não muito grande, pode valer a pena testar para todos e pegar o melhor, fazendo uma escolha determinística.

Para juntar isso no algoritmo base `MultiStart`, podemos fazer permutações aleatórias dos vértices e percorrê-las, usando o vértice atual da permutação como o inicial do algoritmo `CorteMaximoAleat`. Dessa forma, ele vai continuar procurando em todos os vértices, mas de uma maneira aleatória e podemos colocar um critério de parada para não percorrer **todo** o conjunto de vértices quando ele for muito grande.

O critério de percorrer a lista de vértices alcançados também é arbitrário (guloso). Podemos também aleatorizar esse processo.

### 2.1 Heurística de Busca Local

Com relação à busca local, podemos usar como base o algoritmo da [Semana 3](#). O algoritmo `CorteMaximoBuscaLocal` usa como base um “corte inicial”. Esse corte inicial pode ser totalmente arbitrário, mas também pode ser construído aleatoriamente. Por exemplo, podemos sortear apenas se cada vértice está nesse corte inicial.

Já no caso desse corte inicial, o número de possibilidades de escolha é muito maior ( $2^{|V|-1} - 1$ ), não sendo viável testar todas as possibilidades.

### Aula 3 Embaralhamento de Knuth

Normalmente precisamos percorrer listas nos algoritmos. Uma maneira de aleatorizar é percorrer a lista permutada de maneira uniforme.

Sorteamos, com probabilidade uniforme, um item do sufixo do vetor (parte do vetor não processada) e depois adicionamos o item na posição corrente.

```
| 1  2  3  4  5
Escolhemos do sufixo o número 3.
3 | 2  1  4  5
Escolhemos do sufixo o número 5.
3  5 | 1  4  2
Escolhemos do sufixo o número 1.
3  5  1 | 4  2
Escolhemos do sufixo o número 2.
3  5  1  2 | 4
Permutação final: 3  5  1  2  4
```

O algoritmo é  $O(n)$ . E há a garantia de que a probabilidade é uniforme para todas as permutações possíveis.

### Aula 4 Problema do Caixeiro Viajante

Usamos como base o algoritmo TSP-Guloso4 visto na [Semana 2](#).

```
Função TSP-Guloso4( $G = (V, E), w$ )
  Seja  $v$  um vértice qualquer;
   $C \leftarrow (v)$ ;
  enquanto  $C$  não contém todos os vértices faça
    Seja  $C = (v_1, \dots, v_i)$ ;
    Escolha um vértice  $x \notin C$  que minimiza  $w(v_i, x)$ ;
    Insira  $x$  no final de  $C$ ;
  fim
  retorna  $C$ 
fim
```

O vértice inicial  $v$  pode ser escolhido aleatoriamente.

Podemos também aleatorizar o critério guloso. Por exemplo, se houver empate, podemos escolher aleatoriamente qual o vértice que será incluso no caminho.

Para aumentar a aleatoriedade, podemos “relaxar” o critério guloso para aumentar os empates que temos. Por exemplo, ao invés de pegar um vértice que minimiza o caminho, podemos escolher os vértices próximos.

Algoritmo GRASP. Se temos  $x_{\min} \notin C$  mais próximo de  $v_i$  e  $x_{\max} \notin C$  mais distante de  $v_i$ . Podemos criar um critério segundo um determinado intervalo de distância:  $w(v_i, x) \leq w(v_i, x_{\min}) + \alpha (w(v_i, x_{\max}) - w(v_i, x_{\min}))$ , sendo  $\alpha$  um valor entre 0 e 1 que indica a “força” da aleatoriedade (seleciona o quão longe estão os vértices que podemos considerar).

## 4.1 Heurística de Busca Local

Também podemos aleatorizar as heurísticas de busca local, por exemplo, o algoritmo TSP-2-OPT.

O circuito hamiltoniano inicial pode ser apenas uma permutação aleatória dos vértices (só funciona se o grafo for completo).

Também podemos colocar a aleatoriedade no critério de busca, ao invés de escolher arbitrariamente uma troca que traz um ganho, pode escolher aleatoriamente qual é a troca que vai ser feita.

Para fazer as permutações em um grafo não completo, podemos incluir as arestas “que faltam” com pesos **muito grandes**. Como o critério da busca quer encontrar trocas que diminuam o custo, ele vai acabar descartando essas arestas e ficamos com a solução do mesmo jeito. E assim, não precisamos nos preocupar em fazer um algoritmo mais complexo para gerar a permutação.

## SEMANA 8

---

# VISÃO GERAL DE MÉTODOS DE OTIMIZAÇÃO

Visão geral de métodos mais avançados que não foram vistos durante o curso

## Aula 1 Métodos exatos

- Programação dinâmica
- Programação por restrições (Constraint programming) → Baseado em apresentar as restrições que a solução tem que ter. Não é necessariamente linear. Existem resolvedores para isso.

### 1.1 Programação dinâmica

Resolver um problema dividindo a entrada em menores do mesmo tipo. → Divisão e conquista. Mas o que difere é que os resultados dos subproblemas são armazenados para não ter que recomputar.

- O problema tem que ter uma subestrutura ótima. → A solução ótima é composta de soluções ótimas dos subproblemas
- Os problemas tem que compartilhar subproblemas (para que valha a pena armazenar os resultados).

#### 1.1.1 Fibonacci

```

Função FibonacciRec(n)
  | se  $n \leq 2$  então
  |   | retorna 1
  | fim
  | retorna FibonacciRec(n-1) + FibonacciRec(n-2)
fim

```

O problema com esse algoritmo recursivo básico, é que muitos subproblemas são constantemente recalculados. → O método fica ineficiente

Para solucionar esse problema, podemos usar a memória para armazenar os valores já calculados em uma tabela.

```

Função FibonacciPD(n)
  Seja  $F$  um vetor de tamanho  $n$ ;
   $F[1] \leftarrow F[2] \leftarrow 1$ ;
  para  $i \leftarrow 3$  até  $n$  faça
     $F[i] \leftarrow F[i-1] + F[i-2]$ ;
  fim
  retorna  $F[n]$ 
fim

```

A tabela nesse caso é unidimensional. Mas pode ter várias outras dimensões, depende do número de índices que variam no problema. A ideia básica é, porém, preencher a tabela conforme vai resolvendo os subproblemas.

### 1.1.2 Problema da Mochila Binária

Seja  $S^* \subseteq \{1, 2, \dots, n\}$  uma solução ótima para  $I = (\{1, 2, \dots, n\}, v, w, W)$ . Temos duas possibilidades:

- $n \notin S^*$ , caso em que  $S^*$  é solução ótima de  $(\{1, 2, \dots, n-1\}, v, w, W) \rightarrow$  O item  $n$  não está na mochila, então a solução ótima é a mesma do caso em que ele não estivesse lá para ser escolhido, mantendo o peso e os valores.
- $n \in S^*$ , caso em que  $S^* \setminus \{n\}$  é solução ótima de  $(\{1, 2, \dots, n-1\}, v, w, W - w_n) \rightarrow$  Como  $n$  está na mochila, podemos encontrar uma solução ótima com os outros itens e temos também que retirar a capacidade dele da mochila.

Sendo  $V_{i,T}$  o custo de uma solução ótima que escolhe itens no subconjunto  $\{1, 2, \dots, i\}$  para preencher uma mochila de capacidade  $T$ , temos:

$$V_{i,T} = \begin{cases} \max \{V_{i-1,T}, V_{i-1,T-w_i} + v_i\} & \text{se } w_i \leq T \\ V_{i-1,T} & \text{se } w_i > T \end{cases}$$

Temos que verificar se o item  $i$  cabe na mochila ( $w_i \leq T$ ). Se couber, verificamos se vale a pena colocá-lo ou não (max).

$$w_1 = 1, v_1 = 1$$

$$w_2 = 3, v_2 = 2$$

$$w_3 = 2, v_3 = 1$$

$$w_4 = 1, v_4 = 2$$

$$w_5 = 4, v_5 = 6$$

Capacidade da mochila:  $W = 4$

Caso base: a mochila não tem capacidade, não podemos considerar nenhum item. Então o valor vai ser necessariamente 0. Também tem o caso em que não há nenhum item para ser verificado ( $i = 0$ ), como não podemos pegar nenhum item, o valor também é 0.

5	0					
4	0					
3	0					
2	0					
1	0					
0	0	0	0	0	0	
$i$	$T$	0	1	2	3	4

Começamos incrementando o  $i$  e o  $T$ . Por exemplo, no caso em que  $i = 1, T = 1$ , temos que o item  $i$  cabe na mochila ( $w_i \leq T$ ). Então, calculamos os valores da expressão de maximização. Preenchendo a linha toda:

5	0					
4	0					
3	0					
2	0					
1	0	1	1	1	1	
0	0	0	0	0	0	
$i$	$T$	0	1	2	3	4

Preenchendo a linha em que consideramos o item 2, temos os casos em que ele não cabe na mochila e simplesmente pegamos os valores anteriores (abaixo, em que pegamos só o item 1).

5	0					
4	0					
3	0					
2	0	1	1			
1	0	1	1	1	1	
0	0	0	0	0	0	
$i$	$T$	0	1	2	3	4

Mas agora temos o caso em que  $T = 3$ , ou seja, o item 2 cabe na mochila. Nesse caso, vemos se vale a pena pegá-lo ou não. O caso de não pegá-lo, temos o valor

$V_{i-1,T} = 1$ . O caso de pegá-lo, temos  $V_{i-1,T-w_i} + v_i = V_{1,0} + 2 = 2$ . Então vale mais a pena pegar o item 2.

Já no caso em que  $T = 4$ , temos que  $V_{i-1,T-w_i} + v_i = V_{1,1} + 2$ , ou seja, ainda há espaço livre de valor 1, em que cabe o item 1, resultando então no valor final de 3.

5	0					
4	0					
3	0					
2	0	1	1	2	3	
1	0	1	1	1	1	
0	0	0	0	0	0	
$i$	$T$	0	1	2	3	4

Continuando, incluindo os outros itens:

5	0	2	3	3	6	
4	0	2	3	3	4	
3	0	1	1	2	3	
2	0	1	1	2	3	
1	0	1	1	1	1	
0	0	0	0	0	0	
$i$	$T$	0	1	2	3	4

A solução final (valor final da mochila) está armazenada na última posição, em que  $i = n$  e  $T = W$ .



```

Função MochilaPD( $\underline{n}, \underline{w}, v, W$ )
  seja  $M$  uma matriz de tamanho  $(n + 1) \times (W + 1)$ ;
  para  $T \leftarrow 0$  até  $W$  faça
    |  $M[0][T] \leftarrow 0$ ;
  fim
  para  $i \leftarrow 1$  até  $n$  faça
    | para  $T \leftarrow 0$  até  $W$  faça
      | se  $w_i > T$  então
        | |  $M[i][T] \leftarrow M[i - 1][T]$ ;
      | senão
        | |  $M[i][T] \leftarrow \max \{m[i - 1][T], M[i - 1][T - w_i] + v_i\}$ ;
      | fim
    | fim
  fim
  retorna  $M[n][W]$ 
fim

```

## Aula 2 Algoritmos de aproximação

Outra abordagem para lidar com problemas de otimização combinatória, são os algoritmos de aproximação. São um “caminho do meio” entre métodos exatos e heurísticas.

Algoritmos são algoritmos que executam em tempo polinomial, sem garantir que a solução seja ótima. Mas podemos garantir que a solução está dentro de um intervalo com relação ao custo da solução ótima.

Seja  $A$  um algoritmo polinomial para um problema de minimização,  $A(I)$  o custo da solução de  $A$  para a entrada  $I$  e  $\text{OPT}(I)$  o custo da solução ótima.

$A$  é uma  $\alpha$ -aproximação se, para toda instância  $I$ ,

$$A(I) \leq \alpha \cdot \text{OPT}(I) \rightarrow \text{Razão de aproximação}$$

Se o problema for de maximização, invertemos a desigualdade.

$$A(I) \geq \alpha \cdot \text{OPT}(I)$$

### 2.1 Problema da Mochila Binária

```

Função MochilaAprox( $\underline{n}, \underline{w}, v, W$ )
  Ordene e renomeie os itens para que  $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$ ;
  /* Cabe até o item  $q$ , mas o  $q + 1$  não cabe mais */
  Seja  $q$  um inteiro tal que  $\sum_{i=1}^q w_i \leq W$  e  $\sum_{i=1}^{q+1} w_i > W$ ;
  retorna  $\max \{v_1 + v_2 + \dots + v_q, v_{q+1}\}$ 
fim

```

Temos que o valor máximo entre dois valores é claramente maior que a média da soma dos valores:  $\max\{a, b\} \geq \frac{a+b}{2}$ . Assim, temos:

$$\begin{aligned} \text{MochilaAprox}(I) &= \max\{v_1 + v_2 + \cdots + v_q, v_{q+1}\} \\ &\geq \frac{1}{2}(v_1 + \cdots + v_q + v_{q+1}) \\ &\geq \frac{1}{2}\text{OPT}_{\text{frac}}(I) \\ &\geq \frac{1}{2}\text{OPT}(I) \end{aligned}$$

Sabemos que  $(v_1 + \cdots + v_q + v_{q+1})$  não cabe na mochila e que vai ser maior ou igual que a solução ótima da mochila fracionária, já que ordenamos os itens seguindo sua densidade e isso nos dá a solução ótima da mochila fracionária. Também sabemos que a solução ótima da mochila fracionária sempre será maior ou igual à solução da mochila inteira, pois ela vem da sua relaxação.

Com isso, temos que esse algoritmo é uma  $\frac{1}{2}$ -aproximação.

## 2.2 Problema do Escalonamento

**Função** EscalonaAprox( $n, t, m$ )

**para**  $j \leftarrow 1$  até  $m$  **faça**  $M_j \leftarrow \emptyset$ ;

**para**  $i \leftarrow 1$  até  $n$  **faça**

    Seja  $j$  uma máquina em que  $\sum_{i' \in M_j} t_{i'}$  é mínimo;

$M_j \leftarrow M_j \cup \{i\}$ ;

**fim**

**retorna**  $\max_{j=1, \dots, m} \sum_{i \in M_j} t_i$

**fim**

O algoritmo é o mesmo algoritmo guloso usado na [Semana 2](#). Mas ele é de aproximação também porque temos uma garantia de qualidade da solução.

Seja  $j$  a máquina que define o *makespan* e seja  $k$  a última tarefa que foi alocada a essa máquina. No momento em que a tarefa  $k$  foi colocada na máquina  $j$ , essa máquina era a máquina menos carregada (devido ao critério guloso do algoritmo), ou seja, naquele momento  $M_j$  tinha carga menor ou igual a qualquer outra máquina.

Seja  $l(j) = \sum_{i \in M_j} t_i$  a carga da máquina  $j$ . Qualquer máquina  $j'$  tem carga  $l(j') \geq l(j) - t_k$  (antes de colocar o item  $k$ , a máquina  $j$  é a de menor carga). Então

$$(l(j) - t_k) \cdot m \leq \sum_{j'=1}^m l(j') = \sum_{i=1}^n t_i$$

Temos que esse valor é sempre menor que o valor da carga de todas as máquinas, que no fim é apenas a carga de todas as tarefas.

Dividindo por  $m$ :

$$l(j) - t_k \leq \frac{\sum_{i=1}^n t_i}{m} = \text{OPT}_{\text{frac}}(I) \leq \text{OPT}(I)$$

Temos que a solução ótima do escalonamento fracionário é simplesmente dividir igualmente os tempos das tarefas sobre todas as máquinas.

$$l(j) - t_k \leq \text{OPT}(I)$$

Assim

$$\begin{aligned} \text{EscalonaAprox}(I) &= l(j) \\ &= (l(j) - t_k) + t_k \\ &\leq \text{OPT}(I) + t_k \\ &\leq \text{OPT}(I) + \text{OPT}(I) \\ &= 2\text{OPT}(I) \end{aligned}$$

Assim, o algoritmo é uma 2-aproximação.

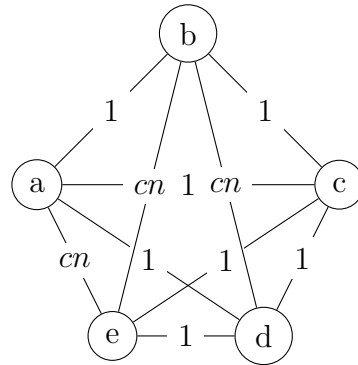
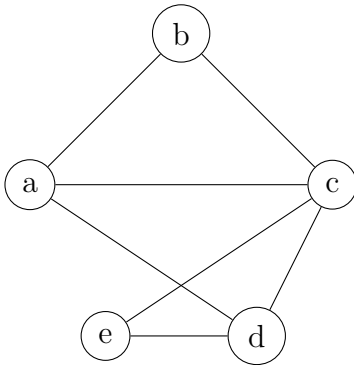
### 2.3 Problema do Caixeiro Viajante

Inaproximabilidade: não há garantia de que seja possível conseguir uma aproximação para esse problema.

Problema relacionado: problema do Circuito Hamiltoniano. “Existe um circuito que visita cada vértice apenas uma vez?” Esse é um problema NP-Completo.

Se  $P \neq NP$ , não existe algoritmo polinomial para problemas NP-Completo.

Construímos um grafo **completo**  $G' = (V, E')$  com peso  $w$  nas arestas tal que  $w(e) = 1$  se  $e \in E$  e  $w(e) = c \cdot n$  se  $e \notin E$ , para uma constante  $c$ .



Se  $G$  tem um circuito hamiltoniano, então  $G'$  também tem um circuito hamiltoniano (o mesmo) de custo  $n$  (porque as arestas têm peso 1).  $\rightarrow \text{OPT}(G', w) = n$

Se  $G$  não tem circuito hamiltoniano, então  $G'$  tem (porque é completo), mas ele tem pelo menos uma aresta de custo  $c \cdot n$ .  $\rightarrow \text{OPT}(G', w) > c \cdot n$

Relacionando com um algoritmo de aproximação: se existir um algoritmo  $A$ , que é uma  $c$ -aproximação para o TSP, então:

Para o caso de  $G$  ter um circuito hamiltoniano:

$$A(G', w) \leq c \cdot \text{OPT}(G', w) = c \cdot n$$

Para o caso de  $G$  **não** tiver um circuito hamiltoniano:

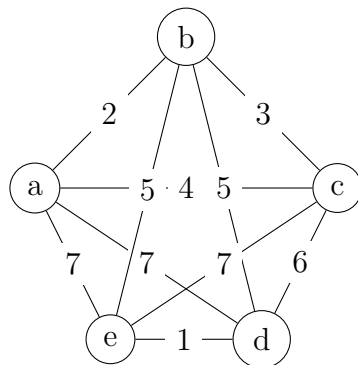
$$A(G', w) \geq \text{OPT}(G', w) > c \cdot n$$

Sabemos então que, se o custo do algoritmo sobre  $G'$  for menor ou igual a  $c \cdot n$ , sabemos que  $G$  possui um circuito hamiltoniano (porque se o valor for maior, cai no segundo caso).

Dessa forma, o algoritmo  $A$  (de aproximação e polinomial) permitiria resolver o problema de decisão do Circuito Hamiltoniano, que é NP-Completo, então esse algoritmo não existe (a não ser que  $P = NP$ , aí existiria um algoritmo polinomial que resolva os problemas NP-Completo).

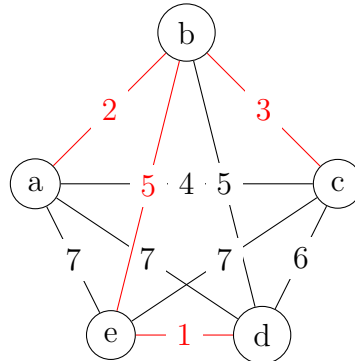
## 2.4 TSP Métrico

Um grafo  $G = (V, E)$  com peso  $w$  nas arestas é métrico se ele for completo e se, para todo trio  $u, v, x \in V$  temos que  $w(uv) \leq w(ux) + w(xv)$  (desigualdade triangular). A ideia é a de que o caminho direto entre dois pontos seja mais barato que passando por outros pontos.

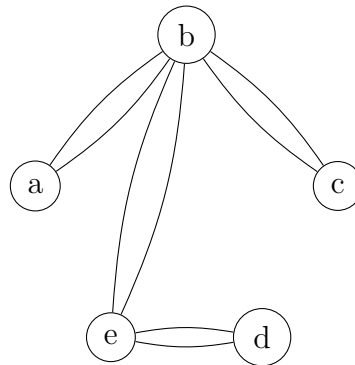


Um atalho é a troca de um caminho entre vértices  $u$  e  $v$ ,  $(u, x_1, x_2, \dots, x_k, v)$ , pela aresta  $uv$ .  $\rightarrow$  A desigualdade triangular garante que o custo nunca aumenta com isso.

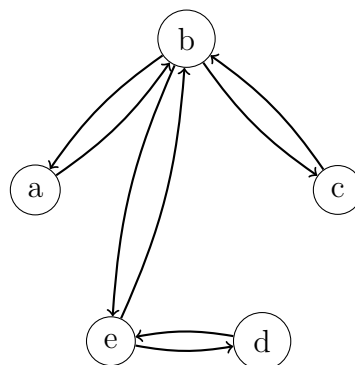
Primeiro, encontramos uma árvore geradora mínima (MST)  $T$  de  $G$  com custo  $c(T)$ .



Duplicamos as arestas de  $T$ . Esse grafo ( $T^2$ ) tem custo  $2c(T)$ .

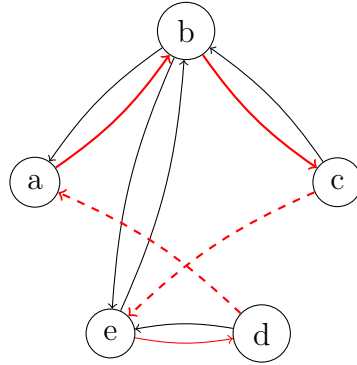


Encontre um ciclo Euleriano  $\varepsilon$  (percorre todas as arestas do grafo). Esse problema tem solução polinomial (o grau de todos os vértices tem que ser par).  $c(\varepsilon) = c(T^2) = 2c(T)$ .



$$\varepsilon = (a, b, c, b, e, d, e, b, a)$$

Percorre  $\varepsilon$  fazendo um atalho se for repetir um vértice.



$$\mathcal{C} = (a, b, c, e, d, a)$$

Temos também a garantia de que  $c(\mathcal{C}) \leq c(\varepsilon) = 2c(T)$ . Isso só é garantido por causa da desigualdade triangular.

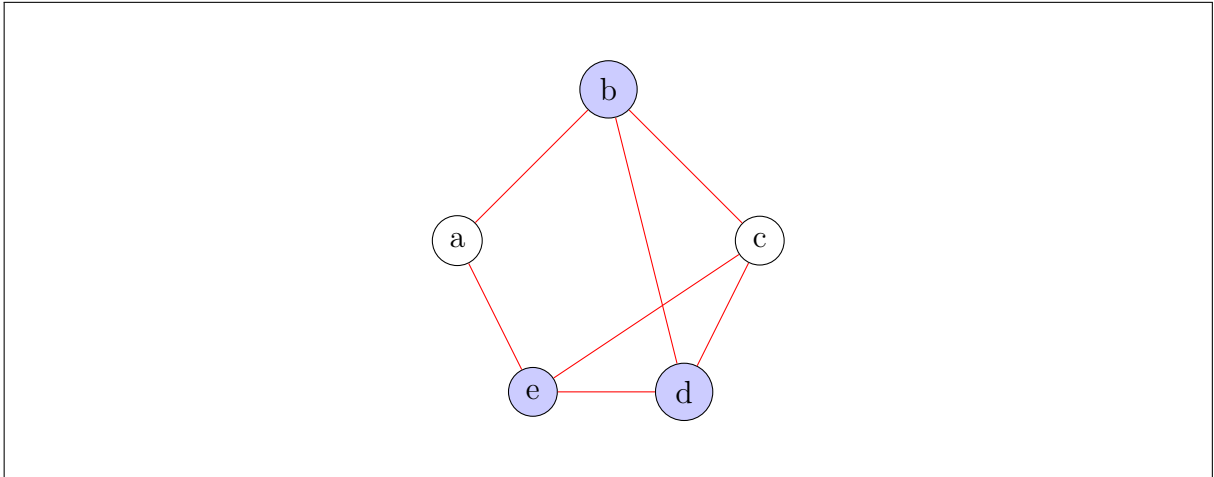
$$\begin{aligned} c(\mathcal{C}) &\leq c(\varepsilon) \\ &= c(T^2) \\ &= 2c(T) \\ &\leq 2\text{OPT}(I) \end{aligned}$$

Como  $T$  é a árvore geradora **mínima**, seu custo é menor que o de qualquer árvore geradora. O seu custo também é menor que o custo de qualquer caminho que conecte todos os vértices de  $G$  (já que é isso que uma MST faz). O custo da árvore também é menor que o de qualquer circuito que visite todos os vértices de  $G$  (porque o circuito vai ser um caminho, que já tem custo maior que a árvore, com mais uma aresta que volta pro início). Assim, o custo da árvore é inclusive menor que o da solução do TSP ( $\text{OPT}(I)$ ), que é um circuito de peso mínimo.

## 2.5 Problema da Cobertura por Vértices

- **Entrada:** grafo  $G = (V, E)$  com peso  $w$  nas arestas.
- **Soluções viáveis:** subconjunto  $S \subseteq V$  tal que para toda aresta  $uv \in E$ ,  $u \in S$  ou  $v \in S$ .
- **Função objetivo:** soma dos pesos dos vértices em  $S$ .
- **Objetivo:** encontrar solução de custo mínimo.

É um caso particular da [Cobertura por Conjuntos](#) ([Semana 4](#)). Então é possível reduzir esse problema para o de [Cobertura por Conjuntos](#).



Começamos com a formulação do problema em PLI.

- Variáveis:  $x_v \quad \forall v \in V$ . Indica se o vértice  $v$  foi escolhido.
- Função objetivo:  $\min \sum_{v \in V} w_v x_v$ . Minimizar o custo dos nós escolhidos.
- Restrição:  $x_u + x_v \geq 1 \quad \forall uv \in E$ . Para toda aresta, pelo menos uma ponta tem que estar na solução.
- Domínio das variáveis:  $x_v \in \{0, 1\} \quad \forall v \in V$ .

Programação Linear (não a inteira) pode ser utilizada para encontrar soluções aproximadas (a partir da relaxação de integralidade do problema). E a solução desse PL é um **limitante inferior** da solução do problema.

**Função** VertexCoverAprox( $G = (V, E), w$ )

```

Monte e resolva o PL, obtendo  $x^*$ ;
/* Fazemos um arredondamento determinístico do resultado do PL */
para  $v \in V$  faça
    se  $x_v^* \geq 1$  então  $x_v \leftarrow 1$ ;
    senão  $x_v \leftarrow 0$ ;
fim
retorna  $\sum_{v \in V} w_v x_v$ 
fim
```

A solução é viável, porque, como toda aresta satisfaz  $x_u^* + x_v^* \geq 1$ , pelo menos  $x_u^*$  ou  $x_v^*$  tem que ser maior 0,5 (senão a soma nunca chega em 1). Assim, pelo menos uma das variáveis em cada restrição vai ser arredondada para 1  $\rightarrow$  Garante a viabilidade da solução inteira aproximada.

Como esse algoritmo faz o arredondamento determinístico, também é possível aleatorizar.

Com relação ao custo da solução, temos alguns casos específicos:

- Se  $x_v = 1$ , então  $x_v^* \geq 0,5 \rightarrow x_v = 1 \leq 2x_v^*$ .
- Se  $x_v = 0$ , então também vale que  $x_v = 0 \leq 2x_v^*$ .

$$\begin{aligned}
 \text{VertexCoverAprox}(I) &= \sum_{v \in V} w_v x_v \\
 &\leq \sum_{v \in V} w_v (2x_v^*) \\
 &= 2 \sum_{v \in V} w_v x_v^* \\
 &= 2\text{OPT}_{\text{PL}}(I) \\
 &\leq 2\text{OPT}(I)
 \end{aligned}$$

O algoritmo é uma 2-aproximação.

É possível generalizar o algoritmo para tratar o problema da [Cobertura por Conjuntos](#). Isso depende da maneira de fazer a redução do problema e adaptação da razão para fazer o arredondamento.

### Aula 3 Parametrização

Esse é um exemplo de abordagem em que relaxamos a restrição de resolver **qualquer** instância.

Uma abordagem mais recente é a de Complexidade Parametrizada. Queremos analisar as características do problema para identificar qual é o parâmetro ( $k$ ) que influencia na complexidade dos algoritmos usados, ainda que a instância seja grande, se controlar o  $k$  podemos conseguir a solução em tempo hábil.

“Dada uma entrada  $I$  e um inteiro não negativo  $k$ ,  $I$  tem alguma propriedade que depende de  $k$ ?”

Queremos encontrar algoritmos que são exponenciais apenas com relação a  $k$  e polinomiais com relação ao tamanho  $n$  de  $I$ .  $\rightarrow O(n^c f(k))$

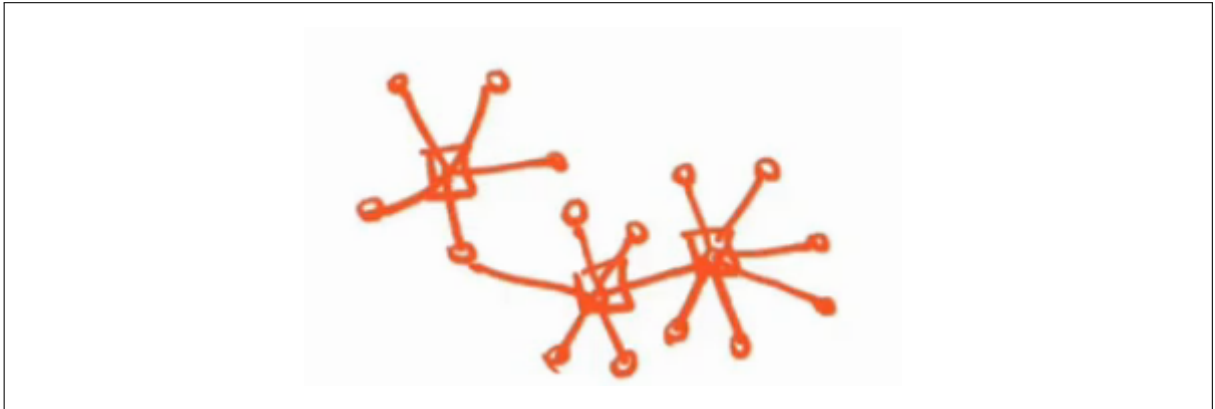
Problemas desse tipo pertencem à classe FPT (*Fixed-Parameter Tractable*), tratáveis caso o parâmetro seja fixo.

#### 3.1 Problema da k Cobertura por Vértices

- **Entrada:** grafo  $G = (V, E)$  e inteiro positivo  $k$ .
- **Questão:**  $G$  possui uma solução  $S$  tal que  $|S| \leq k$  e, para toda aresta  $uv \in E$ ,  $u \in S$  ou  $v \in S$ ?



Só estamos interessados se a cobertura for pequena (estipulado por  $k$ ), mesmo se o grafo for grande (com muitos vértices).



**Função** VertexCoverK( $G = (V, E), k$ )

```

se  $G$  não possui arestas então
|   retorna SIM
fim
se  $k = 0$  então
|   retorna NÃO
fim
Escolha uma aresta  $uv$  arbitrariamente;
/* Pelo pelo um dos extremos tem que estar na cobertura */
se VertexCoverK( $G - u, k - 1$ ) retornar SIM então
|   /* Existe uma cobertura do resto do grafo (já removemos o  $u$  e
|       todas as arestas cobertas por ele) que usa  $k - 1$  vértices (já
|       que  $u$  já foi usado para cobrir)? */
|   retorna SIM
fim
se VertexCoverK( $G - v, k - 1$ ) retornar SIM então
|   /*  $u$  não funcionou para cobrir o grafo, mas podemos tentar
|       fazendo a cobertura pelo vértice  $v$  */
|   retorna SIM
fim
retorna NÃO; /* Não conseguimos  $k$  vértices que cubram o grafo */
fim
```

A complexidade do algoritmo é  $O(|V| \cdot 2^k)$ : no pior caso, efetua duas chamadas recursivas (para o  $u$  e para o  $v$ ), mas a árvore de busca tem altura de, no máximo,  $k$ . O algoritmo é exponencial, mas a questão é que **para  $k$  pequeno** (no máximo  $\log |V|$ ), o crescimento exponencial está controlado.