Teoria da Complexidade

Cid C. de Souza – IC-UNICAMP

13 de outubro de 2005

Autor

Prof. Cid Carvalho de Souza
Universidade Estadual de Campinas (UNICAMP)
Instituto de Computação
Cx. Postal 6176, 13083-970, Campinas, SP, Brasil
Email: `cid@ic.unicamp.br`

Direitos autorais

Reduções entre problemas

▷ Idéia básica da **redução de Turing**:

Problema A:	Problema B:
• Instância de entrada: I_A; • Solução: S_A.	• Instância de entrada: I_B; • Solução: S_B.

▷ **Definição**: uma **redução** do problema A **para** o problema B é um par de transformações τ_I e τ_S tal que, dada uma instância qualquer I_A de A:

- τ_I transforma I_A em uma instância I_B de B **e**
- τ_S transforma a solução S_B de I_B em uma solução S_A de I_A.

▷ **Esquema**:

$$\begin{array}{ccc} A & & B \\ I_A & \xrightarrow{\tau_I} & I_B \\ S_A & \xleftarrow[\tau_S]{} & S_B \end{array}$$

▷ Quando usar **reduções** ?

- **Situação 1**: quero encontrar um algoritmo para A e conheço um algoritmo para B. Ou seja, vou determinar uma *cota superior* para o problema A.
- **Situação 2**: quero encontrar uma *cota inferior* para o problema B e conheço uma *cota inferior* para o problema A.

▷ Exemplo:

- Desejo resolver um sistema linear da forma $Ax = b$.
- Disponho de um programa que resolve sistemas lineares quando a matriz de entrada A é simétrica (i.e., $a_{ij} = a_{ji}$).
- O meu sistema linear não satisfaz esta propriedade.

◦ O que fazer ?

- Transformar a instância do meu problema numa instância que é resolvida pelo algoritmo implementado pelo programa.
- Notar que todo x que é solução de $A^T Ax = A^T b$ também é solução de $Ax = b$ e que $A^T A$ é *simétrica*.

▷ Resolver sistemas lineares da forma $Ax = b$ quando A é **simétrica** é *pelo menos tão difícil quanto* resolver um sistema linear onde A é uma matriz qualquer ?

Formalizando ...

$\mathcal{I}_\bullet$: conjunto de todas instâncias do problema $\bullet$;

$\mathcal{S}_\bullet$: conjunto de todas as soluções das instâncias em $\mathcal{I}_\bullet$;

▷ **Definição**: Um problema A é *redutível ao problema* B em tempo $f(n)$ se existe a redução esquematizada abaixo

$$\begin{tikzcd}
A & B \\
I_A \arrow[r, "\tau_I"] & I_B \arrow[d, "\pi_B"] \\
S_A & S_B \arrow[l, "\tau_S"']
\end{tikzcd}$$

onde: $n = |I_A|$ e τ_I e τ_S são $O(f(n))$.

▷ **Notação**: $A \propto_{f(n)} B$.

▷ **Observações**:

1. Conhecendo um algoritmo π_B para B, temos imediatamente um algoritmo π_A que resolve *instâncias genéricas* de A:
$$\pi_A \doteq \tau_S \circ \pi_B \circ \tau_I.$$
A **complexidade de** π_A será dada pela soma das complexidades de τ_I, π_B e τ_S. Ou seja, temos uma *cota superior* para A.

2. Se π_B tem complexidade $g(n)$ e $g(n) \in \Omega(f(n))$ então temos que $g(n)$ também é cota superior para A.
   - *Se $g(n) \notin \Omega(f(n))$, a cota superior de $g(n)$ ainda vale ?*

3. Se $\Omega(h(n))$ é uma cota inferior para o problema A e $f(n) \in o(h(n))$, então $\Omega(h(n))$ também é cota inferior para o problema B.
   - *Por quê exigir que $f(n) \in o(h(n))$? O que aconteceria se não fosse ?*
   - *Lembrete: $o(h(n))$ e $\Omega(h(n))$ são **mutuamente excludentes !***

Exemplos de Reduções

▷ **Problema do casamento cíclico de cadeias de caracteres (CSM)**

Entrada: Alfabeto Σ e duas cadeias de caracteres de tamanho n:

$$A = a_0a_1 \ldots a_{n-1} \text{ e } B = b_0b_1 \ldots b_{n-1}.$$

Pergunta: B é um *deslocamento cíclico* de A ?

Ou seja, existe $k \in \{0, \ldots, n-1\}$ tal que $a_{(k+i) \mod n} = b_i$ para todo $i \in \{0, \ldots, n-1\}$?

◦ Exemplo: para $A = acgtact$ e $B = gtactac$ temos $n = 7$ e $k = 2$.

- Como se resolve este problema ?

◇ **Problema do Casamento de Cadeias (SM)**:

Entrada: Alfabeto Σ e duas cadeias de caracteres: $A = a_0a_1\ldots a_{n-1}$ e $B = b_0b_1\ldots b_{m-1}$, sendo $m \leq n$.

Pergunta: Encontrar a primeira ocorrência de B em A ou concluir que B não é subcadeia de A.

Ou seja, determinar o menor $k \in \{0, \ldots, n-1\}$ tal que $a_{k+i} = b_i$ para todo $i \in \{0, \ldots, m-1\}$ ou retornar $k = -1$.

∘ Exemplo: para $A = acgttaccgtacccg$ e $B = tac$ ($n = 15$ e $m = 3$) tem-se $k = 4$.

- **Observação**: O problema SM pode ser resolvido em tempo $O(m + n)$ através do algoritmo de Knuth, Morris e Pratt.

◇ **Redução:** CSM $\propto_n$ SM

- Instância de CSM: $I_{CSM} = (A, B, n)$;
- τ_I constrói a instância de SM:

$$I_{SM} = (A', 2n, B, n), \text{ onde } A' = A \parallel A.$$

  Portanto, τ_I é $O(n)$.

- Se k é a solução de SM para I_{SM}, então k também é solução de I_{CSM}. Logo, τ_S é $O(1)$ e a redução é $O(n)$.

◇ Exemplo:

- $I_{CSM} = (acgtact, gtactac, 6)$;
- $I_{SM} = (acgtactacgtact, 12, gtactac, 6)$;
- $S_{SM} = S_{CSM} = \{k = 2\}$.

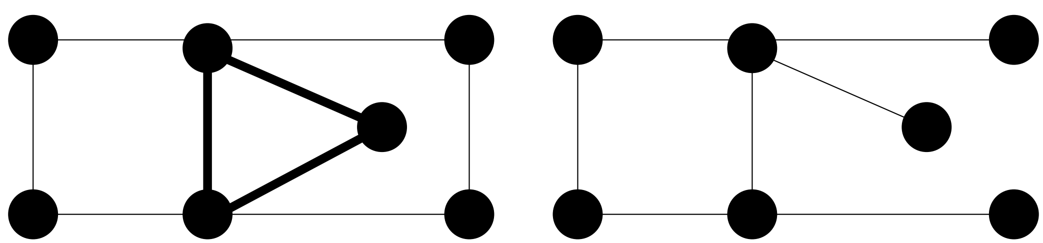
Exemplos de Reduções (cont.)

▷ **Problema da existência de um triângulo em um grafo conexo não orientado (PET):**

Entrada: grafo conexo não orientado $G = (V, E)$, sem auto-laços, onde $|V| = n$ e $|E| = m$.

Pergunta: G possui um ciclo de comprimento 3, ou seja, um triângulo ?

∘ Exemplos:



(a) Com △. (b) Sem △.

Observações:

- Algoritmo trivial: verificar todas as triplas de vértices (complexidade=$O(n^3)$).
- Existe algoritmo $O(mn)$ que é muito bom para *grafos esparsos*.
- Supor que o grafo é dado pela sua *matriz de adjacências* $A(G)$.
- Se $A^2(G) = A(G) \times A(G)$, então $a^2_{ij} = \sum_{k=1}^{n} a_{ik}.a_{kj}$. Logo:

$$a^2_{ij} > 0 \Leftrightarrow \exists\, k \in \{1, \ldots, n\} \text{ tal que } a_{ik} = a_{kj} = 1.$$

- Portanto, o triângulo (i, j, k) existirá se e somente se $a^2_{ij} > 0$ e $a_{ij} = 1$.
- *Observação*: $a_{ii} = 0$ pois não há auto-laços.

$\diamond$ **Problema da multiplicação de matrizes quadradas (MM):**

Entrada: Duas matrizes quadradas de números inteiros A e B de ordem n.

Pergunta: qual é a matriz P resultante do produto $A \times B$?

$\diamond$ **Observação**: MM pode ser resolvido em tempo $O(n^{\log 7 = 2.81\ldots})$ através do algoritmo de Strassen.

$\diamond$ **Redução:** PET $\propto_{n^2}$ MM

- $I_{PET} = A(G)$;
- τ_I constrói a instância de MM:

$$I_{MM} = (A, A, n), \text{ onde } A = A(G).$$

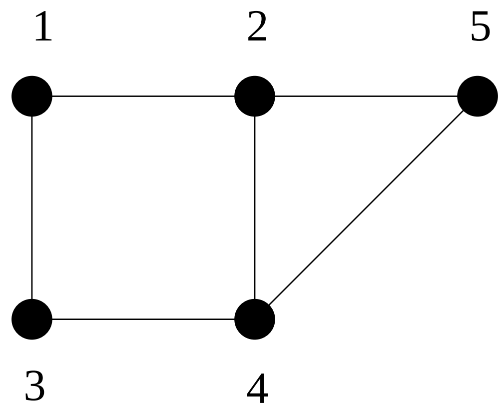
Portanto, τ_I é $O(n^2)$.

- Se $S_{MM} = P$ é a solução de MM para I_{MM}, então a solução de I_{PET} pode ser obtida através do algoritmo τ_S a seguir:

> **Para** $i = 1$ **até** n **faça**
>   **Para** $j = 1$ **até** n **faça**
>   **Se** ($p_{ij} > 0$ **e** $a_{ij} = 1$), **retorne** `Verdadeiro`.
> **Retorne** `Falso`.

▷ A complexidade de τ_S, assim como aquela da redução, é $O(n^2)$.

- Exemplo: PET $\propto$ MM.



$A(G)$

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

$P = A(G) \times A(G)$

	1	2	3	4	5
1	2	0	0	2	1
2	0	3	2	1	1
3	0	2	2	0	1
4	2	1	0	3	1
5	1	1	1	1	2

Exemplos de Reduções (cont.)

▷ **Multiplicação de Matrizes Simétricas (MMS):**

Entrada: 2 matrizes simétricas A e B de números inteiros de ordem n.

Pergunta: qual é a matriz P resultante do produto $A \times B$?

◇ Problema MMA: obter a matriz produto de duas matrizes arbitrárias (não necessariamente simétricas)

◇ MMS é mais fácil do que MMA ?

◇ **Observações**:

- MMS é um caso particular de MMA: a redução $MMS \propto MMA$ é imediata e tem complexidade $O(n^2)$. Portanto *MMA é pelo menos tão difícil quanto MMS.*
- Será que *MMS é pelo menos tão difícil quanto MMA ?* (menos intuitivo)

◇ **Redução:** MMA $\propto_{n^2}$ MMS

- $I_{MMA} = (A, B, n)$;
- τ_I constrói a instância de MMS: $I_{MM} = (A', B', 2n)$, onde

$$A' = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \quad \text{e} \quad B' = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$$

  Portanto, τ_I é $O(n^2)$.

- Suponha que a solução de MMS é dada por:

$$P' = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$$

  Se P é a solução de MMA, então τ_S pode ser implementada através do seguinte algoritmo:

Para $i = 1$ **até** n **faça**
  Para $j = 1$ **até** n **faça**
    $p_{ij} = p'_{ij}$.

▷ A complexidade da redução é $O(n^2)$.

▷ Pela redução acima, **se** todo algoritmo de MMA está em $\Omega(h(n))$, **então** todo algoritmo para MMS está em $\Omega(h(n))$ também.

- $h(n)$ está em $\Omega(n^2)$. *(Por quê ?)*

▷ se $T(n)$ é a complexidade de um algoritmo para MMS e $T(2n) \in O(T(n))$[a], então pela redução acima, tem-se um algoritmo de complexidade $O(T(n) + n^2)$ para resolver MMA.

[a]função suave: qualquer polinômio satisfaz.

Erros comuns ao se usar reduções

1. Usar redução na ordem inversa: por exemplo ao fazer a redução $A \propto B$ e concluir que A é pelo menos tão difícil quanto B.
2. Dada a redução $A \propto B$ achar que toda instância de B tem que ser mapeada numa instância de A (o mapeamento só vai numa direção).
3. Usar o algoritmo produzido por uma redução sem se preocupar com a existência de um outro algoritmo mais eficiente.

   Exemplo:

   - redução do **problema inteiro da mochila (IKP)** ao **problema binário da mochila (BKP)**.
   - A redução pode levar *a uma instância de entrada do BKP de tamanho muito grande !*

Reduções polinomiais

Definição: Se $A \propto_{f(n)} B$ e $f(n) \in O(n^k)$ para algum valor k real, então a redução de A para B é **polinomial**.

Observações:

- No caso de obtenção de uma cota superior para A, a importância das reduções polinomiais é óbvia pois, havendo um algoritmo polinomial para B, a redução leva imediatamente a um algoritmo *eficiente* para A.
- Todas reduções vistas anteriormente são polinomiais.
- A existência de uma redução polinomial do problema A para o problema B é denotada por $A \propto_{\text{poli}} B$.

Classes de Problemas

- $\triangleright$ Problemas para os quais são conhecidos **algoritmos eficientes**:

  ordenação de vetores, obtenção da mediana de um vetor, árvore geradora mínima de um grafo, caminhos mais curtos em grafos, multiplicação de matrizes, etc.

- $\triangleright$ Existem inúmeros problemas para os quais *não são conhecidos algoritmos eficientes !*

- $\triangleright$ Considere o problema de satisfazer uma fórmula lógica F na *forma normal conjuntiva* (**SAT**, ou *Satisfiability*):
  - Variáveis: $x_1, \ldots, x_n$ (mais suas negações: $\overline{x}_i$ para todo i);
  - Operadores lógicos: "+" e "." (OU e E lógicos);
  - Cláusulas: $C_1, C_2, \ldots, C_m$ da forma $C_i = (x_{i1} + x_{i2} + \ldots)$;
  - Fórmula: $F = C_1.C_2. \ldots .C_m$.

Classes de Problemas (cont.)

- ▷ **Pergunta:** Existe alguma atribuição das variáveis $x_1, \ldots, x_n$ para a qual F seja verdadeira, i.e., $F = 1$?
- ▷ Exemplo:

$$F = (x_1 + x_2 + \overline{x}_3).(\overline{x}_1 + \overline{x}_2 + x_3).(x_1 + \overline{x}_3).$$

Se $x_1 = 1$ e $x_2 = x_3 = 0$ tem-se que $F = 1$. Ou seja, a resposta ao problema **SAT** para esta instância é **SIM**.

- ▷ **Exercício:** Encontre um algoritmo para SAT. O seu algoritmo tem complexidade polinomial ?

Classes de Problemas (cont.)

- **Exercício:** Dada uma atribuição de valores paras as variáveis, descreva um algoritmo **polinomial** que confirma se F é verdadeira ou falsa para esta atribuição.
- Não se conhece algoritmo eficiente para SAT !
- Característica do problema SAT comum a diversos problemas encontrados em Computação:

> *É difícil encontrar um algoritmo polinomial que resolve o problema mas* ***existe um algoritmo polinomial que verifica*** *se uma* ***proposta de solução*** *resolve de fato o problema.*

Classes de Problemas (cont.)

▷ **Idéia**: catalogar os problemas como estando em pelo menos duas classes:

- a classe dos problemas para os quais se conhece um algoritmo eficiente para **resolução**.
- a classe dos problemas para os quais se conhece um algoritmo eficiente de **verificação**.

▷ O estudo de classes de complexidade é feito tradicionalmente para **problemas de decisão**, ou seja, aqueles em que a resposta é da forma "SIM" ou "NÃO".

Tipos de reduções e classes de Problemas

$$\begin{array}{ccccc} & I_A & \xrightarrow{\quad\tau_I\quad} & I_B & \\ A & & & \Big\downarrow \pi_B & B \\ & S_A & \xleftarrow{\quad\tau_S\quad} & S_B & \end{array}$$

- Redução de Turing (ou de Cook): admite-se que o algoritmo π_B seja usado múltiplas vezes. Assim, se a redução é polinomial, e o número de chamadas para π_B é limitado a um polinômio no tamanho da entrada de A, pode-se afirmar que A é resolvido em tempo polinomial, desde que π_B tenha tempo polinomial.

- Redução de Karp: usada para provas de pertinência de problemas de decisão às diferentes classes de problemas que veremos a seguir. Neste tipo de redução, π_B só pode ser chamado <u>uma única vez</u>. Além disso, π_B deve responder *SIM* para I_B se e somente se I_A for uma instância *SIM* para o problema A.

Tipos de reduções e classes de Problemas (cont.)

▷ A redução de Karp é um caso particular da redução de Turing. Se nas definições de classes de problemas usássemos a redução de Turing em vez da redução de Karp as classes não seriam menores, mas, ainda é uma questão em aberto se elas seriam maiores.

Classes de Problemas (cont.)

▷ Exemplo de um problema de decisão:

> *Dado um grafo conexo não-orientado $G = (V, E)$, pesos inteiros w_e para cada aresta $e \in E$ e um valor inteiro W, pergunta-se: G possui uma árvore geradora de peso menor que W ?*

▷ **Observação**: já conhecemos a **versão de otimização** deste problema, a qual pode ser resolvido eficientemente pelos algoritmos de Kruskal e de Prim.

▷ Em geral é fácil encontrar uma **redução polinomial (de Turing)** do problema de otimização para o problema de decisão, ou seja:

$$\text{OTM} \propto_{\text{poli}} \text{DEC}.$$

A redução inversa é trivial.

Algoritmos não-determinísticos

- Em um algoritmo **determinístico** o resultado de cada operação é definido de maneira **única**.
- No modelo de computação não-determinístico, além dos comandos determinísticos usuais, um algoritmo pode usar o comando **Escolha(**S**)** o qual retorna um elemento do conjunto S.
- Não existe regra que especifique o funcionamento do comando **Escolha(**S**)**. Existem $|S|$ resultados possíveis para esta operação e o comando retorna **aleatoriamente** um deles.
- Os algoritmos não-determinísticos são divididos em duas fases. Na primeira fase, que pode fazer uso do comando não-determinístico **Escolha**, **constrói-se** uma proposta de solução.
  Na segunda fase, onde só são usados comandos determinísticos, **verifica-se** se a proposta de solução resolve de fato o problema.

Algoritmos não-determinísticos (cont.)

- Ao final da fase de verificação, os algoritmos não-determinísticos sempre retornarão o resultado **Aceitar** ou **Rejeitar**, dependendo se a solução proposta resolve ou não o problema.
- A proposta de solução gerada ao final da fase de construção do algoritmo não determinístico é chamada de um *certificado*.
- A complexidade de execução do comando **Escolha** é $O(1)$.
- Uma **máquina não-determinística** é aquela que é capaz de executar um algoritmo não-determinístico. *É uma abstração !*

Algoritmos não-determinísticos (cont.)

▷ Exemplo: determinar se um valor x pertence a um vetor A de n posições.

Um algoritmo não-determinístico seria:

BuscaND(A,x);
  (* Fase de construção *)
  $j \leftarrow$ Escolha$(1, \ldots, n)$;
  (* Fase de verificação *)
  Se $A[j] = x$ **então** retornar `Aceitar`;
  se não retornar `Rejeitar`;

▷ Qual a complexidade deste algoritmo ?

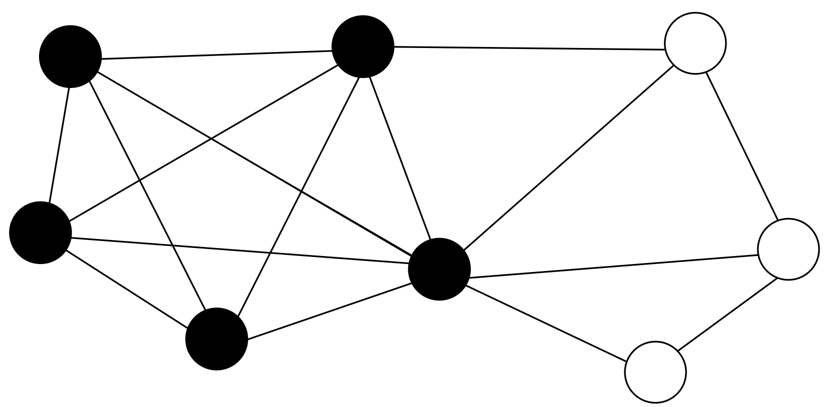
Algoritmos não-determinísticos (cont.)

- **Definição**: *a complexidade de um algoritmo não-determinístico executado sobre uma instância qualquer é o número mínimo de passos necessários para que ele retorne* `Aceitar` *caso exista uma seqüência de* **Escolha***s que leve a essa conclusão. Se o algoritmo retornar* `Rejeitar` *o seu tempo de execução é* $O(1)$.
- Um algoritmo não-determinístico tem complexidade $O(f(n))$ se existem constantes positivas c e n_0 tais que para toda instância de tamanho $n \geq n_0$ para o qual ele resulta em `Aceitar`, o tempo de execução é limitado a $cf(n)$.
- Assim, o algoritmo `BuscaND` tem complexidade $O(1)$. Note que qualquer algoritmo determinístico para este problema é $\Omega(n)$!

Algoritmos não-determinísticos (cont.)

▷ Outro exemplo: **CLIQUE**

- *Enunciado*: dado um grafo conexo não-orientado $G = (V, E)$ e um valor inteiro $k \in \{1, \ldots, n\}$, onde $n = |V|$ pergunta-se: G possui uma *clique* com k vértices ?
- Uma *clique* é um subgrafo completo de G.



▷ Um algoritmo não-determinístico para Clique seria:

CliqueND(G,n,k);
  (* Fase de construção *)
  $S \leftarrow V$;
  $C \leftarrow \{\}$; (* vértices da clique proposta *)
  Para $i = 1$ **até** k **faça**
    $u \leftarrow$ Escolha(S);
    $S \leftarrow S - \{u\}$;
    $C \leftarrow C \cup \{u\}$;
  fim-para
  (* Fase de verificação *)
  Para todo par de vértices distintos (u, v) em C **faça**
    Se $(u, v) \notin E$ retornar `Rejeitar`;
  fim-para
  Retornar `Aceitar`;

▷ Complexidade (não-determinística): $O(k + k^2) \subseteq O(n^2)$.

▷ Não se conhece algoritmo determinístico polinomial para CLIQUE.

Simulando máquinas não-determinísticas

- ▷ Podem ser imaginadas como sendo máquinas determinísticas com *infinitos* processadores, os quais se comunicam entre si de modo *instântaneo*, ou seja, uma mensagem vai de um processador ao outro em tempo *zero*.

- ▷ O fluxo global de execução de um algoritmo não-determinístico pode ser esquematizado através de uma *árvore*. Cada caminho na árvore iniciando na raiz corresponde a uma seqüência de escolhas e, portanto, a um possível fluxo de execução do programa. Em um dado vértice, $|S|$ filhos serão criados ao se executar o comando `Escolha(S)`, cada um correspondendo a um possível resultado retornado por esta operação, alocando-se então um novo processador para continuar a operação a partir deste ponto.

Simulando máquinas não-determinísticas (cont.)

- ▷ Pode-se imaginar que a *árvore de execução* é percorrida em largura e que, ao ser atingido o primeiro nível onde uma execução do algoritmo retorna `Aceitar`, o processador que chegou a este estado comunica-se instantaneamente com todos os demais, interrompendo o algoritmo.
- ▷ Exemplo: um outro algoritmo não-determinístico de complexidade $O(n^2)$ para CLIQUE: *(próxima transparência)*
- ▷ Note que existem seqüências de escolhas que podem não deixar que o laço *enquanto* termine ! Mas, a complexidade não-determinística só se interessa pelo número **mínimo** de passos que leva a uma conclusão de `Aceitar`.

▷ Um outro algoritmo não-determinístico para CLIQUE:

CliqueND2(G,n,k);
(* Fase de construção *)
$j \leftarrow 0$;
$C \leftarrow \{\}$; (* vértices da clique proposta *)
Enquanto $j < k$ **faça**
$u \leftarrow$ Escolha(V);
Se $u \notin C$ **então**;
$C \leftarrow C \cup \{u\}$;
$j \leftarrow j + 1$;
fim-se
enquanto
(* Fase de verificação *)
Para todo par de vértices distintos (u, v) em C **faça**
Se $(u, v) \notin E$ retornar `Rejeitar`;
fim-para
Retornar `Aceitar`;

Simulando máquinas não-determinísticas (cont.)

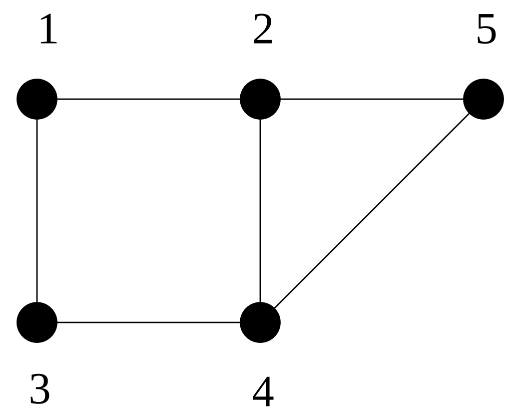


Figura 1: Determinar se há clique de tamanho 3

- ▷ Árvore de simulação determinística: *(próxima tranparência)*
- ▷ **Exercício** Desenvolva um algoritmo não-determinístico polinomial para SAT. Qual a complexidade do seu algoritmo ?

Simulando máquinas não-determinísticas (cont.)

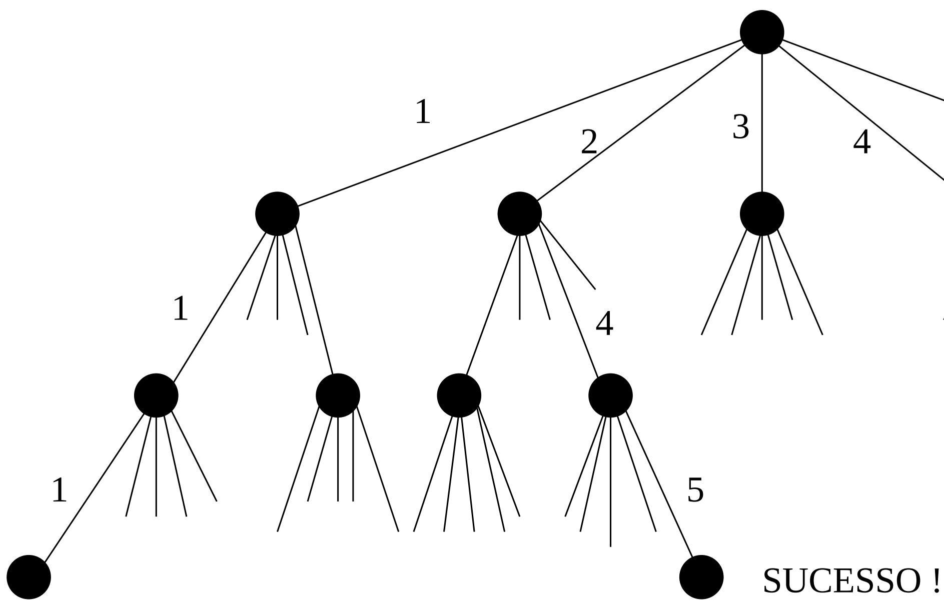


Figura 2: Fluxo de execução de CliqueND2.

As classes $\mathcal{P}$ e $\mathcal{NP}$

- ▷ **Definição**: $\mathcal{P}$ é o conjunto de problemas que podem ser resolvidos por um **algoritmo determinístico** polinomial.
- ▷ **Definição**: $\mathcal{NP}$ é o conjunto de todos os problemas que podem ser resolvidos por um **algoritmo <u>não</u>-determinístico** polinomial.
- ▷ Como todo algoritmo determinístico é um caso particular de um algoritmo não-determinístico, segue que

$$\mathcal{P} \subseteq \mathcal{NP}.$$

- ▷ Assim, todos os problemas que possuem algoritmos polinomiais estão em $\mathcal{NP}$. Além disso, como visto anteriormente, CLIQUE e SAT estão em $\mathcal{NP}$.

As classes $\mathcal{P}$ e $\mathcal{NP}$ (cont.)

▷ **Questão fundamental da Teoria da Computação**:

$$\boxed{\mathcal{P} = \mathcal{NP}\ ?}$$

▷ Em geral, os algoritmistas acreditam que a *proposição é falsa !*

▷ Como mostrar que a proposição é falsa ?

Encontrar um problema $A \in \mathcal{NP}$ e mostrar que <u>nenhum</u> algoritmo determinístico polinomial pode resolver A.

▷ Como mostrar que a proposição é verdadeira ?

Mostrar que <u>para todo</u> problema $B \in \mathcal{NP}$ existe um algoritmo determinístico polinomial que o resolve.

As classes $\mathcal{NP}$-difícil e $\mathcal{NP}$-completo

- Será que existe um problema A em $\mathcal{NP}$ tal que, se A está em $\mathcal{P}$ então todo problema em $\mathcal{NP}$ também está em $\mathcal{P}$?
- Que característica deveria ter este problema A para que a propriedade acima se verificasse facilmente ?
- "Basta" encontrar um problema A em $\mathcal{NP}$ tal que, para **todo** problema B em $\mathcal{NP}$ existe uma **redução polinomial (de Karp)** de B para A.
- Nota: daqui em diante o termo *"redução"* será usado para referir-se à **redução de Karp**.
- **Definição**: A é um problema $\mathcal{NP}$-difícil se todo problema de $\mathcal{NP}$ se reduz polinomialmente a A.

As classes $\mathcal{NP}$-difícil e $\mathcal{NP}$-completo (cont.)

▷ **Definição**: A é um problema $\underline{\mathcal{NP}\text{-completo}}$ se

1. $A \in \mathcal{NP}$ $\underline{\text{e}}$
2. $A \in \mathcal{NP}$-difícil.

▷ **Observações**:

1. Por definição, $\mathcal{NP}$-completo $\subseteq \mathcal{NP}$-difícil.
2. Se for encontrado um algoritmo polinomial para um problema qualquer em $\mathcal{NP}$-difícil então ficará provado que $\mathcal{P} = \mathcal{NP}$.

▷ **Definição**: dois problemas P e Q são **polinomialmente equivalentes** se $P \propto_{\text{poli}} Q$ e $Q \propto_{\text{poli}} P$.

Todos problemas de $\mathcal{NP}$-completo são polinomialmente equivalentes !

Provas de $\mathcal{NP}$-completude

- *Lema*: Seja A um problema em $\mathcal{NP}$-difícil e B um problema em $\mathcal{NP}$. Se existir uma redução polinomial de A para B, ou seja $A \propto_{\text{poli}} B$ então B está em $\mathcal{NP}$-completo.
- Dificuldade: encontrar um problema que esteja em $\mathcal{NP}$-completo.
- Será que existe ?
- Cook provou que SAT é $\mathcal{NP}$-completo !

Teorema de Cook: redefinindo a classe $\mathcal{NP}$

Se A é um *problema de decisão* em $\mathcal{NP}$ e π é um algoritmo *não-determinístico* polinomial que resolve A, então:

- Como a fase de verificação de π só realiza operações determinísticas e tem complexidade polinomial, o *certificado* $c(x)$ gerado pela fase de construção de π tem tamanho polinomial no tamanho da instância de entrada. Ou seja:

$$|c(x)| \leq p(|x|),$$

  onde $p(.)$ é o polinômio que descreve a complexidade de π.

- Portanto, a fase de construção pode ser vista como uma seqüência de *escolhas* não-determinísticas que vai codificando, posição a posição, a cadeia que representa $c(x)$. Cada uma destas *escolhas* é feita sobre o alfabeto que descreve o *sistema de codificação*.

Teorema de Cook: redefinindo a classe $\mathcal{NP}$ (cont.)

▷ Pelas observações anteriores, pode-se *redefinir* a classe $\mathcal{NP}$ como sendo o conjunto dos problemas de decisão para os quais existe um algoritmo **determinístico** π tal que, dados uma instância e um certificado, π **verifica** em tempo polinomial **no tamanho da instância** se o certificado resolve o problema.

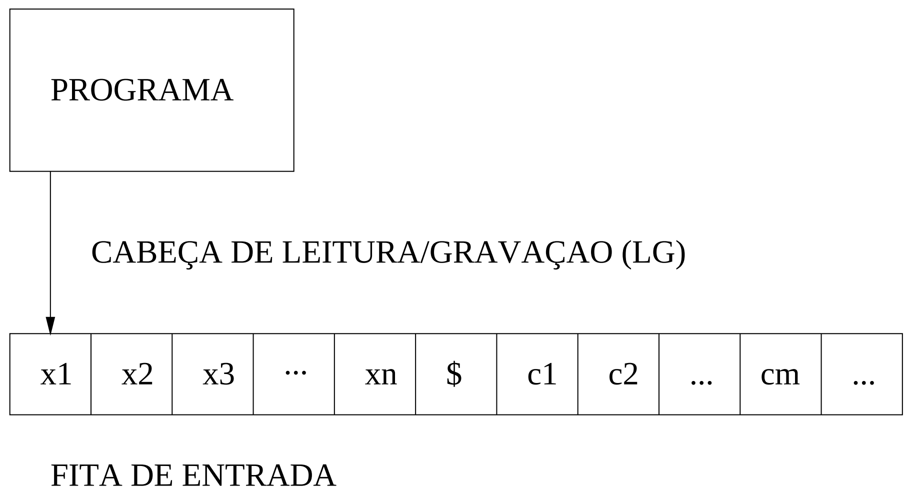
▷ Para mostrar que todo problema de $\mathcal{NP}$ se reduz polinomialmente a SAT, deve-se usar uma característica comum a todos os problemas desta classe.

▷ Essa característica é a existência de um *algoritmo verificador* **determinístico polinomial** !

Teorema de Cook: principais idéias da prova

- Todo algoritmo eficiente pode ser descrito por um **modelo de computação** conhecido como uma **Máquina de Turing**. Em particular, para qualquer problema de $\mathcal{NP}$ o algoritmo verificador pode ser descrito por este modelo.
- Mostrar que existe uma fórmula booleana F de tamanho polinomial no tamanho da entrada da **Máquina de Turing** tal que F pode ser satisfeita *se e somente se* a **Máquina de Turing** encerra sua execução retornando `Aceitar`.
- Isso equivale a dizer que, se a **Máquina de Turing** descreve o algoritmo verificador para um problema A de $\mathcal{NP}$, então x é uma instância para qual A tem resposta `SIM` *se e somente se* F tem resposta `SIM` para SAT.

Teorema de Cook: uma Máquina de Turing



Máquina de Turing

▷ Cada instrução intermediária do programa é da forma:

$$\boxed{\ell : \text{ se } \sigma \text{ então } (\sigma', o, \ell'),}$$

onde ℓ e ℓ' são números de instruções, σ e σ' são símbolos do alfabeto Σ usado pelo sistema de codificação e $o \in \{-1, 0, 1\}$.

Teorema de Cook: Máquina de Turing (cont.)

▷ O significado da instrução anterior é o seguinte:

> se o símbolo lido na fita de entrada é σ, escreva σ' no seu lugar, mova a cabeça de LG o posições para direita e depois execute a instrução de número ℓ'. Caso contrário, vá para a instrução $\ell + 1$.

▷ A última instrução do programa é:

$$\boxed{t : \texttt{Aceitar},}$$

onde t é o número de instruções do programa.

▷ No início da computação, a cabeça de LG encontra-se na posição mais à esquerda da fita de entrada.

Teorema de Cook: Máquina de Turing (cont.)

▷ Uma cadeia $x\$c(x)$ é **aceita** por um algoritmo verificador π de complexidade $p(|x|)$ se este alcançar a última instrução depois de no máximo $p(|x|)$ passos.

▷ Se π não alcançar a última instrução neste número de passos ou a cabeça de LG estiver fora de uma posição que descreve a cadeia de entrada, então a cadeia é **`rejeitada`**.

▷ Portanto, fazem parte da classe $\mathcal{NP}$ os problemas de decisão para os quais existe um algoritmo verificador π de complexidade polinomial $O(p(n))$, tal que x é uma instância de entrada `SIM` *se e somente* se <u>existe</u> uma cadeia $c(x)$, com $|c(x)| \leq p(|x|)$ tal que π **aceita** $x\$c(x)$.

Teorema de Cook

▷ *Teorema*: SAT é $\mathcal{NP}$-completo.

▷ *Esboço da prova*:

◇ SAT está em $\mathcal{NP}$ (exercício);

◇ Considere um problema genérico $A \in \mathcal{NP}$, x uma instância de entrada para A, $c(x)$ um certificado para x e π um algoritmo verificador de complexidade $O(p(|x|))$ para A contendo t instruções.

◇ Definir as variáveis booleanas a seguir:

∘ $z_{ij\sigma}$ para todo $0 \leq i, j \leq p(|x|)$ e todo $\sigma \in \Sigma$, onde $z_{ij\sigma} = 1$ *se e somente* se no instante i, a j-ésima posição da cadeia na fita de entrada contém o símbolo σ.

Teorema de Cook: prova (cont.)

◇ Definição das variáveis (cont.):

∘ $y_{ij\ell}$ para todo $0 \leq i \leq p(|x|)$, para todo $0 \leq j \leq p(|x|)+1$ e todo $1 \leq \ell \leq t$, onde $y_{ij\ell} = 1$ *se e somente se* no instante i, a cabeça de LG está na j-ésima posição da cadeia na fita de entrada e a ℓ-ésima instrução do programa está sendo executada.

Observação: se $j = 0$ ou $j = p(|x|)+1$ a cabeça de LG terá caído fora da cadeia de entrada e a computação irá ser *rejeitada*.

▷ A partir da **Máquina de Turing** correspondente a π com uma entrada dada por $x\$c(x)$, construir uma fórmula booleana F nas variáveis anteriores da forma:

$$F(z,y) = U(z,y).S(z,y).W(z,y).E(z,y)$$

Teorema de Cook: prova (cont.)

- Se $U(z,y)$ for verdadeiro, estará garantido que a cada instante de tempo, cada posição da cadeia de entrada contém um único símbolo, que a cabeça de LG estará sobre uma única posição e que o programa executa uma única instrução.

$$
\begin{aligned}
U(z,y) \quad = \quad & \left(\prod_{\substack{0 \le i,j \le p(|x|) \\ \sigma \neq \sigma'}} \left(\overline{z}_{ij\sigma} + \overline{z}_{ij\sigma'} \right) \right) \cdot \left(\prod_{\substack{0 \le i \le p(|x|) \\ j \neq j' \text{ ou } \ell \neq \ell'}} \left(\overline{y}_{ij\ell} + \overline{y}_{ij'\ell'} \right) \right) \cdot \\
& \left(\prod_{\substack{0 \le i \le p(|x|) \\ 1 \le \ell \le t}} \left(\overline{y}_{i0\ell} \cdot \overline{y}_{i,p(|x|)+1,\ell} \right) \right) \cdot \\
& \left(\prod_{0 \le i \le p(|x|)} \left(\left(\prod_{1 \le j \le p(|x|)} \sum_{\sigma \in \Sigma} z_{ij\sigma} \right) \cdot \sum_{\substack{1 \le j \le p(|x|) \\ 1 \le \ell \le t}} y_{ij\ell} \right) \right)
\end{aligned}
$$

Teorema de Cook: prova (cont.)

- Se $S(z,y)$ for verdadeiro, estará garantido que a Máquina de Turing está inicializada corretamente. Ou seja: os $|x|+1$ símbolos mais à esquerda na fita correspondem à codificação de $x\$$, a cabeça de LG está na posição mais à esquerda da fita de entrada e que a primeira instrução do programa a ser executada será a instrução número 1. Ou seja,

$$S(z,y) = (\prod_{j=1}^{|x|} z_{0jx(j)}).z_{0,|x|+1,\$}.y_{011}.$$

Teorema de Cook: prova (cont.)

- Se $W(z, y)$ for verdadeiro, estará garantido que o algoritmo π realiza corretamente as instruções contidas no programa. Ou seja, ao executar a instrução

$$\ell: \text{ se } \sigma \text{ então } (\sigma', o, \ell'),$$

o símbolo que é gravado na posição corrente é σ' ou permanece inalterado, a cabeça de LG se movimenta para o posições à direita da posição corrente ou fica na mesma posição e a próxima instrução a ser executada é a instrução ℓ' ou a instrução $\ell + 1$. Além disso, deverá ser garantido que, se j é a posição corrente da cabeça de LG, no instante seguinte, todos os símbolos nas demais posições permanecem inalterados.

Teorema de Cook: prova (cont.)

- $W(z,y)$ é uma conjunção de fórmulas $W_{ij\sigma\ell}$.
  Para cada $0 \leq, i \leq p(|x|)$, $1 \leq j \leq p(|x|)$, $\sigma \in \Sigma$, $1 \leq \ell < t$, considere a ℓ $\underline{\text{a}}$ instrução dada por:

$$\boxed{\ell: \text{ se } \sigma \text{ então } (\sigma', o, \ell'),}$$

- A fórmula $W_{ij\sigma\ell}$ para uma instrução intermediária é:

$$W_{ij\sigma\ell}(z,y) = (\overline{z}_{ij\sigma} + \overline{y}_{ij\ell} + z_{i+1,j,\sigma'}) \cdot (\overline{z}_{ij\sigma} + \overline{y}_{ij\ell} + y_{i+1,j+o,\ell'}) \cdot \prod_{\tau \neq \sigma} ((\overline{z}_{ij\tau} + \overline{y}_{ij\ell} + z_{i+1,j,\tau}) \cdot (\overline{z}_{ij\sigma} + \overline{y}_{ij\ell} + y_{i+1,j,\ell+1})$$

Teorema de Cook: prova (cont.)

- A fórmula $W_{ij\sigma t}$ para a última instrução é:

$$W_{ij\sigma t}(z,y) = (\overline{z}_{ij\sigma} + \overline{y}_{ijt} + y_{i+1,j,t})$$

Nota: se o algoritmo atinge a instrução t, ele permanece lá.

- Resta garantir que se a cabeça de LG está numa posição diferente de j, esta permanecerá inalterada:

$$\prod_{\substack{0 \leq i \leq p(|x|) \\ \sigma \in \Sigma, j \neq j' \\ 1 \leq \ell \leq t}} (\overline{z}_{ij\sigma} + \overline{y}_{ij'\ell} + z_{i+1,j,\sigma'})$$

Teorema de Cook: prova (cont.)

- Se $E(z, y)$ é verdadeiro, estará garantido que t é a última instrução executada pelo algoritmo π. Ou seja:

$$E(z, y) = \sum_{j=1}^{p(|x|)} y_{p(|x|),j,t}.$$

- Complexidade da construção de F: $O(p^3(|x|) \log\ p(|x|))$.
- $F(z, y)$ tem resposta `SIM` para SAT se e somente se x tem resposta `SIM` para A. $\square$
- Detalhes da prova: *Combinatorial Optimization: Algorithms and Complexity*, C.H. Papadimitriou e K. Steiglitz, Dover, 1982.
- Máquinas de Turing: *The Design and Analysis of Computer Algorithms* (cap. 1), A.V. Aho, J.E. Hopcroft e J.D. Ullman, Addison-Wesley, 1974.

$\boxed{\textbf{Provas de } \mathcal{NP}\textbf{-completude}}$

▷ Depois que Cook (1971) provou que SAT estava em $\mathcal{NP}$-completo Karp (1972) mostrou que outros 24 problemas famosos também estavam em $\mathcal{NP}$-completo.

▷ $\boxed{\text{Lembre-se:}}$

Para provar que um problema A está $\mathcal{NP}$-completo é necessário:

1. Provar que A está em $\mathcal{NP}$;
2. Provar que A está em $\mathcal{NP}$-difícil: pode ser feito encontrando-se uma redução polinomial de um problema B qualquer em $\mathcal{NP}$-difícil para A.

Provas de $\mathcal{NP}$-completude: CLIQUE

▷ CLIQUE: dado um grafo não-orientado $G = (V, E)$ e um valor inteiro $k \in \{1, \ldots, n\}$, onde $n = |V|$, pergunta-se: G possui uma *clique* com k vértices ?

▷ *Teorema*: CLIQUE $\in \mathcal{NP}$-completo.

1. CLIQUE está $\mathcal{NP}$.
2. SAT $\propto_{\text{poli}}$ CLIQUE

◇ **Definição**: um grafo $G = (V, E)$ é *t-partido* se o conjunto de vértices pode ser particionado em t subconjuntos $V_1, V_2, \ldots, V_t$ tal que **não** existam arestas em E ligando dois vértices em um mesmo subconjunto V_i, $i \in \{1, \ldots, t\}$.

Provas de $\mathcal{NP}$-completude: CLIQUE (cont.)

◇ Transformação de uma instância SAT em uma instância CLIQUE:

Seja $F = C_1.C_2.\ldots.C_c$ uma fórmula booleana nas variáveis $x_1, \ldots, x_v$. Construa o grafo c-partido $G = ((V_1, V_2, \ldots, V_c), E)$ tal que:

- Em um subconjunto V_i existe um vértice associado a cada variável que aparece na cláusula C_i de F;
- A aresta (a, b) está em E *se e somente se* a e b estão em subconjuntos distintos **e**, além disso, a e b não representam simultaneamente uma variável e a sua negação.

Provas de $\mathcal{NP}$-completude: CLIQUE (cont.)

- ▷ O número de vértices de G é $O(c.v)$ enquanto o número de arestas é $O(c^2v^2)$. Fazendo-se $k = c$, teremos construído uma instância de CLIQUE em tempo polinomial no tamanho da entrada de SAT.

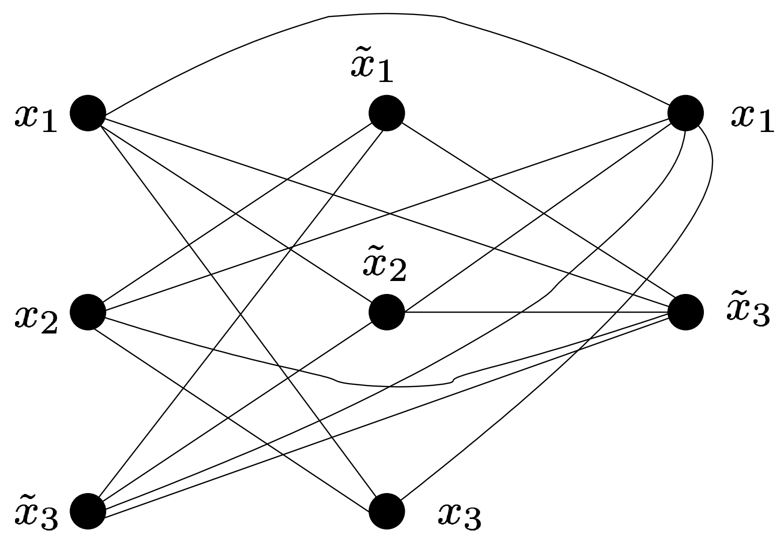
- ▷ É fácil mostrar que a fórmula F é satisfeita por alguma atribuição de variáveis se e somente se o grafo c-partido G tem uma clique de tamanho c.

Provas de $\mathcal{NP}$-completude: CLIQUE (cont.)

▷ Exemplo da redução: seja

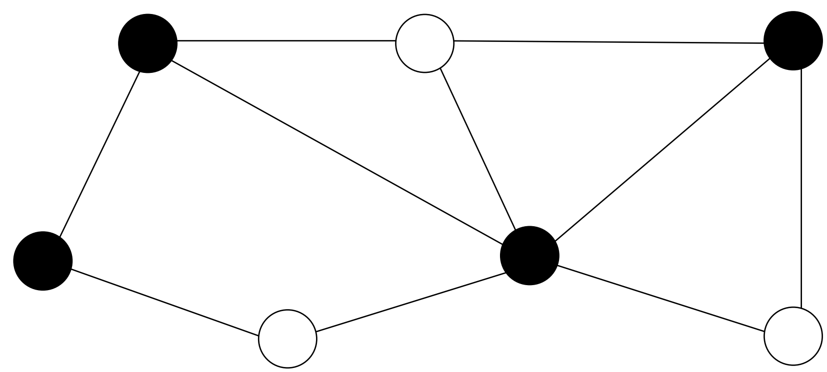
$$F = (x_1 + x_2 + \overline{x}_3).(\overline{x}_1 + \overline{x}_2 + x_3).(x_1 + \overline{x}_3).$$

O grafo correspondente à instância de CLIQUE é dado por:



Provas de $\mathcal{NP}$-completude: Cobertura de vértices (CV)

- **Definição**: dado um grafo não-orientado $G = (V, E)$, diz-se que um subconjunto de vértices U é uma *cobertura* se toda aresta de E tem pelo menos uma das extremidades em U.



- CV: dado um grafo não-orientado $G = (V, E)$ e um valor inteiro $\ell \in \{1, \ldots, n\}$, onde $n = |V|$, pergunta-se: G possui uma *cobertura* com ℓ vértices ?

Provas de $\mathcal{NP}$-completude: CV (cont.)

▷ *Teorema*: CV $\in \mathcal{NP}$-completo.

1. CV está $\mathcal{NP}$. *(Exercício !)*
2. CLIQUE $\propto_{\text{poli}}$ CV

◇ Dado um grafo não-orientado $G = (V, E)$ define-se o seu grafo complementar $\overline{G}$ com o mesmo conjunto de vértices mas tal que uma aresta está em G se e somente se ela não está em $\overline{G}$.

Provas de $\mathcal{NP}$-completude: CV (cont.)

- Transformando uma instância CLIQUE em uma instância CV:

  Seja $G = (V, E)$ o grafo dado na entrada de CLIQUE e k o tamanho da clique procurada. A instância de CV será o grafo complementar $\overline{G}$ e o parâmetro ℓ é dado por $n-k$, onde $n = |V|$.

- A instância de entrada de CV é construída em tempo $O(n^2)$.

- Pode-se mostrar que G é uma instância SIM de CLIQUE se e somente se $\overline{G}$ é uma instância SIM de CV usando o seguinte resultado:

  U é uma clique de tamanho k em $G \Longleftrightarrow \overline{U} = V - U$ é uma cobertura de vértices de tamanho $n - k$ em $\overline{G}$.

- Portanto, $\overline{G}$ tem uma cobertura de tamanho $\ell = n - k$ se e somente se G tem uma clique de tamanho k. $\square$

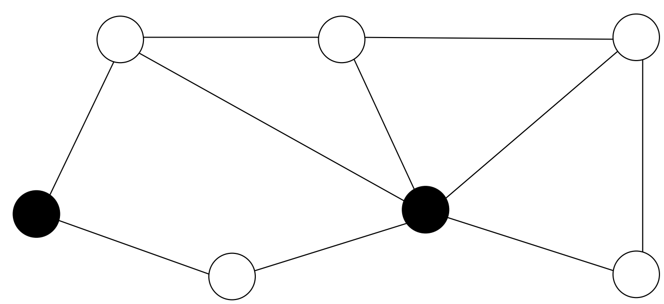
Provas de $\mathcal{NP}$-completude: Conjunto Independente (IS)

▷ **Exercício**:

- um *conjunto independente* (ou *estável*) em um grafo não-orientado $G = (V, E)$ é um subconjunto de vértices U para o qual, dado um par qualquer de elementos u e v em U, a aresta (u, v) **não** está em E.
- Problema do conjunto independente (IS): dado um grafo não-orientado $G = (V, E)$ e um valor inteiro $\ell \in \{1, \ldots, n\}$, onde $n = |V|$, deseja-se saber se G possui um *conjunto independente* com ℓ vértices ?
- Mostre que IS está em $\mathcal{NP}$-completo.

Provas de $\mathcal{NP}$-completude: Conjunto Dominante (DS)

▷ **Definição**: dado um grafo não-orientado $G = (V, E)$, um *conjunto dominante* em G é um subconjunto de vértices U com a propriedade de que, para todo vértice $z \in V$, ou z está em U ou existe um vértice x em um U tal que a aresta (x, z) está em E.



▷ DS: dado um grafo não-orientado $G = (V, E)$ e um valor inteiro $k \in \{1, \ldots, n\}$, onde $n = |V|$, pergunta-se: G possui um *conjunto dominante* com k vértices ?

Provas de $\mathcal{NP}$-completude: Conjunto Dominante (cont.)

▷ *Teorema*: DS $\in \mathcal{NP}$-completo.

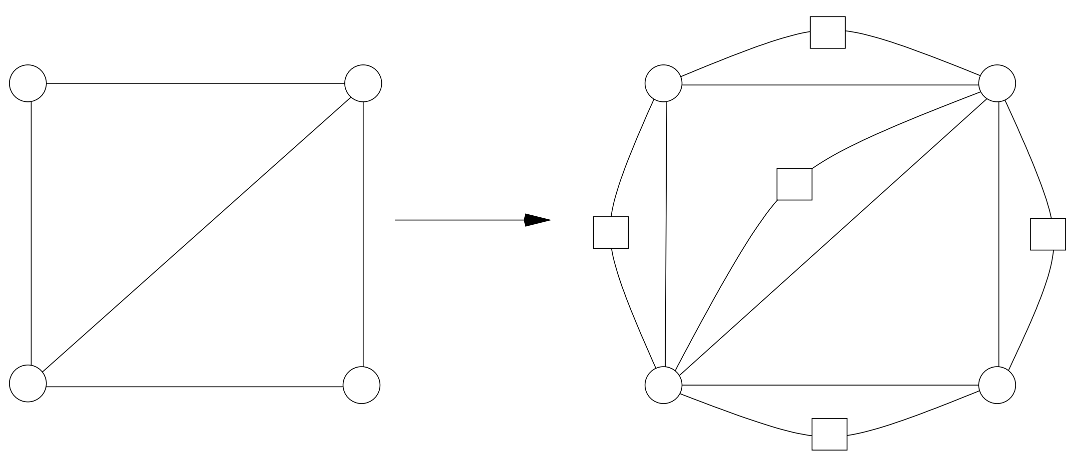
1. DS está $\mathcal{NP}$. *(Exercício !)*
2. CV $\propto_{\text{poli}}$ DS

   Seja $G = (V, E)$ o grafo dado na entrada de CV e ℓ o tamanho da cobertura procurada. A instância de DS será dada por $k = \ell$ e pelo grafo $G' = (V', E')$ construído a partir de G da seguinte forma:

   - Todo vértice de G também é vértice de G'.
   - Para cada aresta (i, j) em E, cria-se um vértice z_{ij} em G'. Se Z é o conjunto de vértices criados desta forma, tem-se que $V' = V \cup Z$.
   - Para cada vértice z_{ij} cria-se as arestas (z_{ij}, i) e (z_{ij}, j). Seja E_z o conjunto das arestas criadas desta forma.

Provas de $\mathcal{NP}$-completude: Conjunto Dominante (cont.)

- O conjunto das arestas de G' é composto pelas arestas em E_z e pela arestas de E, ou seja, $E' = E \cup E_z$.
- Portanto, se $|V| = n$ e $|E| = m$, a instância de entrada de DS é obtida em $O(n + m)$.
- Exemplo de redução de CV para DS:



Provas de $\mathcal{NP}$-completude: Conjunto Dominante (cont.)

▷ *Proposição*: G é uma instância SIM de CV se e somente se G' é uma instância SIM de DS.

◇ *Lema*: Se U é um conjunto dominante de G' e $|U| \leq n$, então é possível construir um conjunto dominante W de G' tal que $|W| = |U|$ e $W \cap Z = \emptyset$, ou seja, $W \subseteq V$.

◇ *Proposição*: o conjunto $W \subseteq V$ obtido acima é uma cobertura de vértices para G.

- Como W é um conjunto dominante e W não tem vértices em Z, para cada aresta de E pelo menos uma das suas extremidades está em W.

◇ *Proposição*: se W é uma cobertura de vértices em G, W também é um conjunto dominante em G'. (e de G !)

$\square$

Provas de $\mathcal{NP}$-completude: problema binário da mochila

▷ **Definição** (BKP):

São dados: um conjunto $U = \{u_1, u_2, \ldots, u_n\}$ de n elementos, dois valores inteiros positivos w_i e c_i (respectivamente o **peso** e o **custo**) associados a cada elemento u_i de U e dois valores inteiros positivos W e C. Deseja-se saber se existe um subconjunto Z de U tal que $\sum_{u_i \in Z} w_i \leq W$ e $\sum_{u_i \in Z} c_i \geq C$?

▷ **Definição**: o *problema da partição* (PAR):

São dados um conjunto finito $V = \{v_1, v_2, \ldots, v_n\}$ de n elementos e um valor inteiro positivo f_i associado a cada elemento v_i de V. Deseja-se saber se existe um subconjunto X de V tal que $\sum_{v_i \in X} f_i = \sum_{v_i \in\, V-X} f_i$?

Provas de $\mathcal{NP}$-completude: BKP (cont.)

▷ *Teorema*: PAR está em $\mathcal{NP}$-completo. *(conhecido !)*

▷ *Teorema*: BKP está em $\mathcal{NP}$-completo:

1. BKP está em $\mathcal{NP}$. *(Exercício !)*
2. PAR $\propto_{\text{poli}}$ BKP.

Transformando uma instância I de PAR para uma instância I' de BKP:

- Faça $U = V$ e $w_i = c_i = f_i$ para todo elemento u_i de U.
- Faça $W = C = \frac{\sum_{v_i \in X} w_i}{2}$.
- A instância de BKP é criada em $O(n)$.
- I é uma instância SIM de PAR se e somente se I' é uma instância SIM de BKP.

□

Provas de $\mathcal{NP}$-completude: 3SAT

- ▷ **Definição**: dada uma fórmula booleana F (na forma normal conjuntiva) onde cada cláusula contém exatamente 3 literais, deseja-se saber se é possível fazer uma atribuição de valores às variáveis de modo que F se torne verdadeira.

- ▷ *Teorema*: 3SAT está em $\mathcal{NP}$-completo.
  1. 3SAT está em $\mathcal{NP}$. *(Exercício !)*
  2. SAT $\propto_{\text{poli}}$ 3SAT.

Provas de $\mathcal{NP}$-completude: 3SAT (cont.)

◇ Transformando uma instância F de SAT em uma instância F_3 de 3SAT:

Suponha que $F = C_1.C_2.\ldots.C_m$. Considere uma cláusula $C = (x_1 + x_2 + \ldots + x_k)$ de F com k literais.

- se $k = 3$, coloque C em F_3.
- se $k = 2$, coloque a cláusula

$$C' = (x_1 + x_2 + z).(x_1 + x_2 + \overline{z})$$

em F_3, criando assim mais uma variável. É claro que uma atribuição de valores às variáveis irá satisfazer C se e somente se ela satisfizer também a C'.

Provas de $\mathcal{NP}$-completude: 3SAT (cont.)

- se $k = 1$, coloque a cláusula

$$C' = (x_1 + y + z).(x_1 + \overline{y} + z).(x_1 + y + \overline{z}).(x_1 + \overline{y} + \overline{z})$$

em F_3, criando assim mais duas variáveis. Pode-se mostrar que uma atribuição de valores as variáveis irá satisfazer C se e somente se ela satisfizer também a C'.

- se $k \geq 4$, coloque a cláusula:

$$\begin{aligned} C' \quad = \quad & (x_1 + x_2 + y_1).(x_3 + \overline{y}_1 + y_2).(x_4 + \overline{y}_2 + y_3).\ldots. \\ & \ldots.(x_{k-2} + \overline{y}_{k-4} + y_{k-3}).(x_{k-1} + x_k + \overline{y}_{k-3}) \end{aligned}$$

em F_3, criando assim $k - 3$ novas variáveis.

$\boxed{\textit{Lema:}}$ C é SAT se e somente se C' é SAT.

Provas de $\mathcal{NP}$-completude: 3SAT (cont.)

Prova do Lema:

($\Rightarrow$): existe um i para o qual $x_i = 1$. Se fizermos $y_j = 1$ para todo $j = 1, \ldots, i-2$ e $y_j = 0$ para todo $j = i-1, \ldots, k-3$, teremos uma atribuição para a qual C' é SAT.
($\Leftarrow$): se C' é SAT, existe uma atribuição onde pelo menos um x_i vale 1. Caso contrário, C' seria equivalente a

$$C' = (y_1).(\overline{y}_1 + y_2).(\overline{y}_2 + y_3). \ldots .(\overline{y}_{k-4} + y_{k-3}).(\overline{y}_{k-3})$$

que obviamente não é SAT. $\square$

- Portanto, se F tem m cláusulas e n variáveis, F_3 terá $O(nm)$ cláusulas e variáveis.

- Por construção, F é SAT se e somente se F_3 é SAT. $\square$

Outros problemas em $\mathcal{NP}$-completo

▷ **Caminho Hamiltoniano em Grafos Não-Orientados:**

Definição: Um *caminho hamiltoniano* em um grafo não orientado G é um caminho que passa uma única vez por todos vértices de G.

Instância: Um grafo não orientado $G = (V, E)$.
Questão: G tem um caminho hamiltoniano ?

▷ **Ciclo Hamiltoniano em Grafos Não-Orientados:**

Definição: Um *ciclo hamiltoniano* em um grafo não orientado G é um ciclo que passa uma única vez por todos vértices de G.

Instância: Um grafo não orientado $G = (V, E)$.
Questão: G tem um ciclo hamiltoniano ?

Outros problemas em $\mathcal{NP}$-completo (cont.)

▷ **Caixeiro Viajante:** *(TSP)*

Definição: Um *tour* em um conjunto de cidades é uma viagem que começa e termina em uma mesma cidade e que passa por todas demais cidades do conjunto **exatamente uma vez**.

Instância: V um conjunto de cidades, distâncias $d_{ij} \in \mathbb{Z}_+$ entre todos os pares de cidades em V **e** um inteiro positivo D.
Questão: Existe um *tour* das cidades em V cuja distância total é menor do que D ?

Mais provas de $\mathcal{NP}$-completude:

- ▷ **Seqüenciamento com janelas de tempo** (SJT): dado um conjunto T de n tarefas e, para cada $t \in T$, um prazo de início $r(t)$, uma duração $\ell(t)$ e um prazo de conclusão $d(t)$, sendo $r(t)$, $d(t)$ e $\ell(t)$ inteiros não-negativos, deseja-se saber se existe um *seqüenciamento viável* para as tarefas em T.
- ▷ **Definição**: um *seqüenciamento viável* é um mapeamento $\sigma : T \to \mathbb{Z}^+$ tal que $\sigma(t) \geq r(t)$ e $\sigma(t) + \ell(t) \leq d(t)$ para todo $t \in T$ e, para todo par (t, t') de T, $\sigma(t) + \ell(t) \leq \sigma(t')$ ou $\sigma(t') + \ell(t') \leq \sigma(t)$.
- ▷ **Exercício**: Mostre que SJT $\in \mathcal{NP}$-completo.

Mais provas de $\mathcal{NP}$-completude: 3COL

▷ **Definição**: Uma **coloração** de um grafo é uma atribuição de cores aos seus vértices tal que dois vértices adjacentes tenham cores distintas.

▷ Um grafo é k colorável se é possível colorí-lo com k cores.

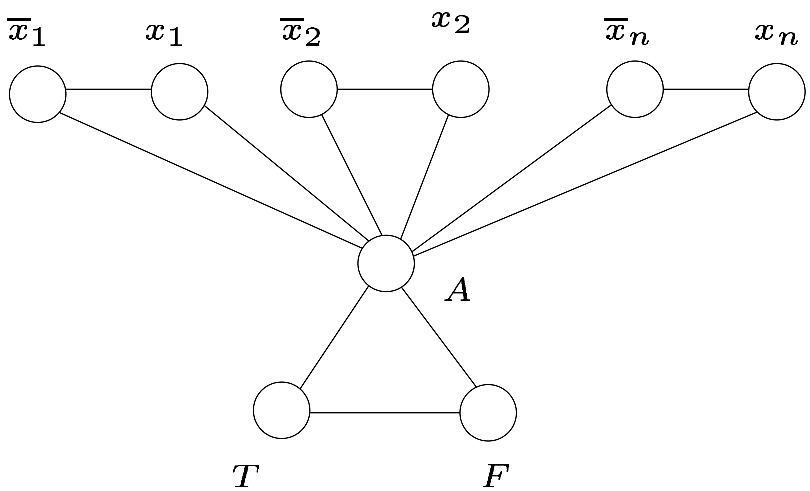
▷ **Problema da 3-coloração** (3COL): dado um grafo $G = (V, E)$, deseja-se saber se G pode ser colorido com 3 cores.

▷ *Teorema*: 3COL $\in \mathcal{NP}$-completo.

1. 3COL está em $\mathcal{NP}$.
2. 3SAT $\propto_{\text{poli}}$ 3COL

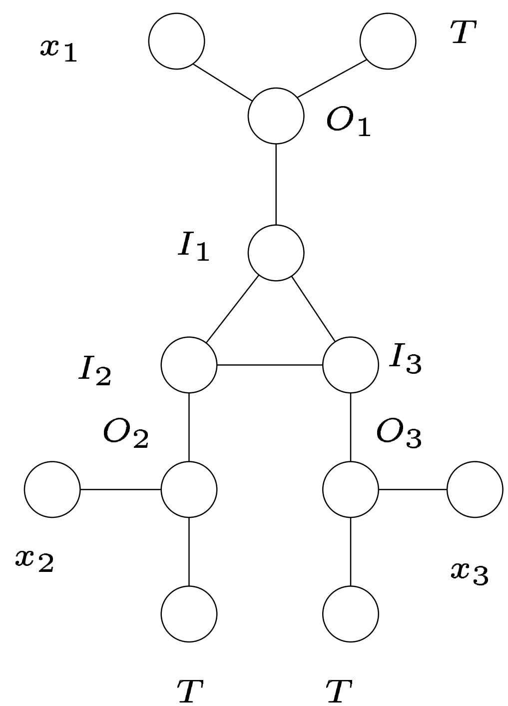
Mais provas de $\mathcal{NP}$-completude: 3COL (cont.)

- Estrutura central:

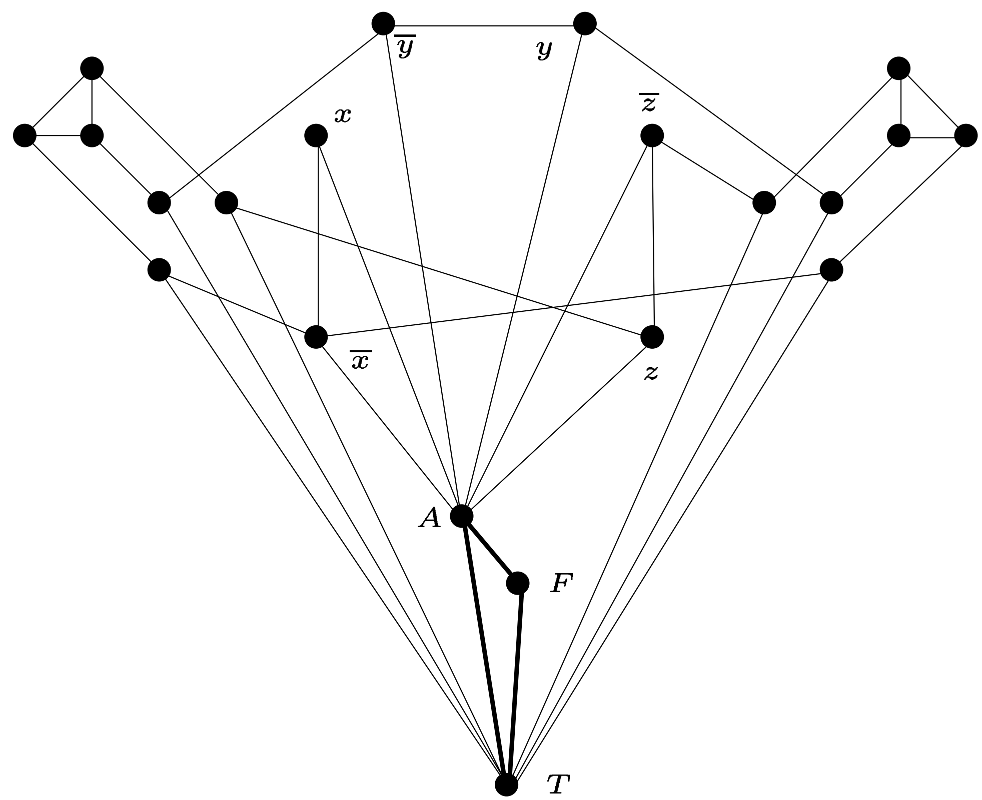


- Estrutura da cláusula:

$$C = (x_1 + x_2 + x_3).$$



Mais provas de $\mathcal{NP}$-completude: 3COL (cont.)



$$C = (\overline{x} + y + \overline{z}) \cdot (\overline{x} + \overline{y} + z)$$

$\boxed{\mathcal{NP}\textbf{-difícil} \times \mathcal{NP}\textbf{-completo ?}}$

- ▷ **Problema dos K subconjuntos mais pesados** (KSMP): dados um conjunto $C = \{c_1, \ldots, c_n\}$ de números inteiros, um inteiro K e um inteiro L, existem subconjuntos *distintos* $S_1, \ldots, S_K$ de C tal que $\sum_{c_i \in S_j} c_i \geq L$ para todo $j = 1, \ldots, K$?
- ▷ KSMP está em $\mathcal{NP}$?
  Pouco provável que haja certificado conciso !
- ▷ *Teorema*: Se KSMP estiver em $\mathcal{P}$ então $\mathcal{P} = \mathcal{NP}$.

  Prova:

  - Existe uma **redução polinomial de Turing** de PARTIÇÃO para o KSMP. PAR $\propto_{\text{poli}}$ KSMP.
  - Redução composta ... $\square$

Indo além de $\mathcal{NP}$: A classe co-$\mathcal{NP}$

▷ Complemento de um problema A: é o problema $\overline{A}$ cujas instâncias `SIM` são exatamente as instâncias `NÃO` de A e vice-versa. Exemplo:

`CiH`: Dado um grafo G, G é hamiltoniano ?

$\overline{\texttt{CiH}}$: Dado um grafo G, G é não-hamiltoniano ?

▷ Não está claro que $\overline{\texttt{CiH}}$ esteja em $\mathcal{NP}$.
(O que seria um certificado conciso para este problema ?)

▷ Outro exemplo:

`AGM`: Dado um grafo $G = (V, E)$, com pesos inteiros w_e para todo $e \in E$, existe uma árvore geradora em G com peso $\leq W$?

$\overline{\texttt{AGM}}$: Dado um grafo $G = (V, E)$, com pesos inteiros w_e para todo $e \in E$, toda árvore geradora em G tem peso $\geq W + 1$?

Indo além de $\mathcal{NP}$: A classe co-$\mathcal{NP}$(cont.)

- **Teorema**: Se $A \in \mathcal{P}$ então $\overline{A} \in \mathcal{P}$.
- **Definição**: co-$\mathcal{NP}$é a classe de todos os problemas que são complementares de problemas que estão em $\mathcal{NP}$.
- *Questão fundamental*: co-$\mathcal{NP} = \mathcal{NP}$?
- É mais provável que co-$\mathcal{NP} \neq \mathcal{NP}$ ...
  Novamente, os problemas de $\mathcal{NP}$-completo parecem ser a chave da questão !
- **Teorema**: Se $X \in \mathcal{NP}$-co e $\overline{X} \in NP$ então co-$\mathcal{NP} = \mathcal{NP}$.
- Como esta teoria pode ser usada para sugerir a existência ou não de um algoritmo eficiente para um problema ?

Indo além de $\mathcal{NP}$: Complexidade de espaço

- **Definição**: um problema tem **complexidade de espaço** $f(n)$ se existir um algoritmo que, para toda instância de tamanho n, use $O(f(n))$ espaço (**memória**) para resolvê-lo.
- **Definição**: $\mathcal{PSPACE}$ é a classe dos problemas que admitem algoritmos **determinísticos** que usam **espaço polinomial** no tamanho da entrada.
- Fatos:
  - $\mathcal{P} \in \mathcal{PSPACE}$.
  - $\mathcal{NP} \in \mathcal{PSPACE}$.
  - co-$\mathcal{NP} \in \mathcal{PSPACE}$.
- **Definição**: $\mathcal{NPSPACE}$ é a classe dos problemas que admitem algoritmos **não-determinísticos** que usam **espaço polinomial** no tamanho da entrada.

Indo além de $\mathcal{NP}$: Complexidade de espaço (cont.)

▷ *Questão fundamental*: $\mathcal{PSPACE} = \mathcal{NPSPACE}$?

▷ É claro que $\mathcal{PSPACE} \subset \mathcal{NPSPACE}$.

▷ *Observação*: não-determinismo representa vantagem quando se trata de complexidade de tempo pois o tempo não pode ser recuperado. Já a memória pode ser reaproveitada ...

Teorema de Savitch: Para toda função $f : \mathbb{N} \to \mathbb{N}$ onde $f(n) \geq \log n$, $\mathcal{NSPACE}(f(n)) \subseteq \mathcal{SPACE}(f(n)^2)$.

▷ Conseqüência: $\mathcal{PSPACE} = \mathcal{NPSPACE}$!

Indo além de $\mathcal{NP}$: Indecidibilidade
O problema da Parada

▷ Suponha que você concebeu uma subrotina H muito especial que realiza a seguinte tarefa. Dado um programa P implementado por uma codificação $< P >$ e uma entrada x, H retorna `SIM` se $< P >$ pàra com a entrada x e retorna `NÃO` caso contrário.

Observação: seria maravilhoso ter esta subrotina pois você saberia quando os seus programas do `LABs` entrariam em um *laço infinito*. :)

▷ Usando H, posso escrever o programa D representado a seguir cujo objetivo é decidir se um programa P pàra quando a sua própria codificação for passada na entrada.

Indo além de $\mathcal{NP}$: Indecidibilidade
O problema da Parada (cont.)

Programa $D(< P >)$;
`a`: **Se** $H(< P >, < P >) =$ `SIM` **então** repita `a`;
se não `PARE`.

- ▷ O que acontece se passarmos D como entrada para ele mesmo ?

  Analisando:

$$D(< D >) \begin{cases} \text{pàra,} & \text{se } H(< D >, < D >) \text{ retornar } \texttt{NÃO}, \\ & \text{ou seja, se } D \text{ ... não parar !} \\ \text{não pàra,} & \text{se } H(< D >, < D >) \text{ retornar } \texttt{SIM}, \\ & \text{ou seja, se } D \text{ ... parar !} \end{cases}$$

- ▷ O que está errado ?!?!?!? O que podemos concluir ?

Tratamento de problemas $\mathcal{NP}$-difíceis: *backtracking*.

▷ Casos de aplicação:

- Problemas cujas soluções podem ser representadas por tuplas (vetores) de tamanho fixo ou variável da forma $(x_1, \ldots, x_n)$.
- Solucionar o problema equivale a encontrar <u>uma</u> tupla que otimiza uma *função critério* $P(x_1, \ldots, x_n)$ **ou** encontrar todas as tuplas que satisfaçam $P(x_1, \ldots, x_n)$.

▷ Restrições:

- *Explícitas*: especificam os domínios (finitos) das variáveis na tupla.
- *Implícitas*: relações entre as variáveis da tupla que especificam quais delas respondem ao problema.

Backtracking: conceitos básicos

▷ **Espaço de soluções**: conjunto de todas as tuplas satisfazendo as restrições *explícitas*.

▷ **Estados do problema**: conjunto de todas as subseqüências das tuplas do *espaço de soluções*.

▷ **Algoritmo *Força Bruta***: enumera todas tuplas do espaço de soluções e verifica quais delas satisfazem às restrições *implícitas*.

▷ **Algoritmo *Backtracking***:

- busca sistemática no espaço de estados do problema que é organizado segundo uma *estrutura de árvore*, denominada **árvore de espaço de estados**.
- uso de *funções limitantes* para restringir a busca na árvore.

Backtracking: conceitos básicos (cont.)

▷ Métodos de exploração do espaço de estados (EE):

- *nós ativos*: aqueles que ainda têm filhos a serem gerados.
- *nós amadurecidos*: aqueles em que todos os filhos já foram gerados ou não devam ser mais expandidos de acordo com a *função limitante*.
- *nó corrente*: aquele que está sendo explorado.

▷ *Backtracking*: busca no EE é feita em *profundidade*.

▷ *Branch-and-Bound*: durante a busca no EE a geração de todos os filhos do nó corrente assim como o cálculo da função limitante em cada um deles é feita de uma vez só (e.g., busca em *largura*).

Backtracking: o algoritmo (recursivo)

BACK(k);
    (* Entrada: $x_1, x_2, \ldots, x_{k-1}$ (já escolhidos). *)
    (* Saída: todas as soluções serão impressas. *)

    (* Obtém o domínio de x_k e o seu tamanho. *)
    $(T, \ell) \leftarrow$ Domínio($x_1, x_2, \ldots, x_{k-1}$);
    Para $i = 1$ **até** ℓ **faça** (* testa todos valores do domínio *)
        $x_k \leftarrow T[i]$;
        Se $(x_1, \ldots, x_k)$ satisfaz às restrições implícitas **então**
            `IMPRIMA`($x_1, \ldots, x_k$); (* solução encontrada *)
        (* Verifica se busca deve prosseguir nesta subárvore *)
        Se $B_k(x_1, \ldots, x_k)$ é verdadeiro **então** BACK($k + 1$);
    fim-para.
fim.

***Backtracking*: problema da soma de subconjuntos (SOS)**

- ▷ SOS: dado um conjunto $S = \{w_1, \ldots, w_n\}$ de n valores inteiros positivos e um valor inteiro positivo W, existe $U \subseteq \{1, \ldots, n\}$ tal que $\sum_{i \in U} w_i = W$?
- ▷ Exemplo: Se $n = 4$, $S = \{11, 13, 24, 7\}$ e $W = 31$ tem-se $U = \{1, 2, 4\}$ e $U = \{3, 4\}$.
- ▷ SOS é $\mathcal{NP}$-completo. (PAR $\propto_{\text{poli}}$ SOS)
- ▷ Representação das soluções:
  1. tupla de tamanho variável: como no exemplo acima.
  2. tupla de tamanho fixo n: $x_i = 1$ se $i \in U$ e $x_i = 0$ caso contrário.

Backtracking: problema SOS (cont.)

▷ *Funções Limitantes*:

1. $B_k(x_1, \ldots, x_k) = \texttt{true} \Leftrightarrow \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq W$.
2. Suponha que $0 < w_1 \leq w_2 \leq \ldots \leq w_n$. Então $(x_1, \ldots, x_k)$ não pode levar a uma nova solução se $\sum_{i=1}^{k} w_i x_i + w_{k+1} > W$. Logo, uma outra função limitante seria:

$$\begin{aligned} B'_k(x_1, \ldots, x_k) = \texttt{true} \quad \Leftrightarrow \quad & B_k(x_1, \ldots, x_k) = \texttt{true}\ \textbf{e} \\ & \textstyle\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq W. \end{aligned}$$

▷ O algoritmo:

- Tuplas de tamanho n (fixo) onde todo x_i está em $\{0, 1\}$.
- Hipóteses: $0 < w_1 \leq w_2 \leq \ldots \leq w_n < W$ e $\sum_{i=1}^{n} w_i \geq W$.
- Os valores $s = \sum_{i=1}^{k-1} w_i x_i$ e $r = \sum_{i=k+1}^{n} w_i$ são passados como parâmetro da k-ésima chamada recursiva.

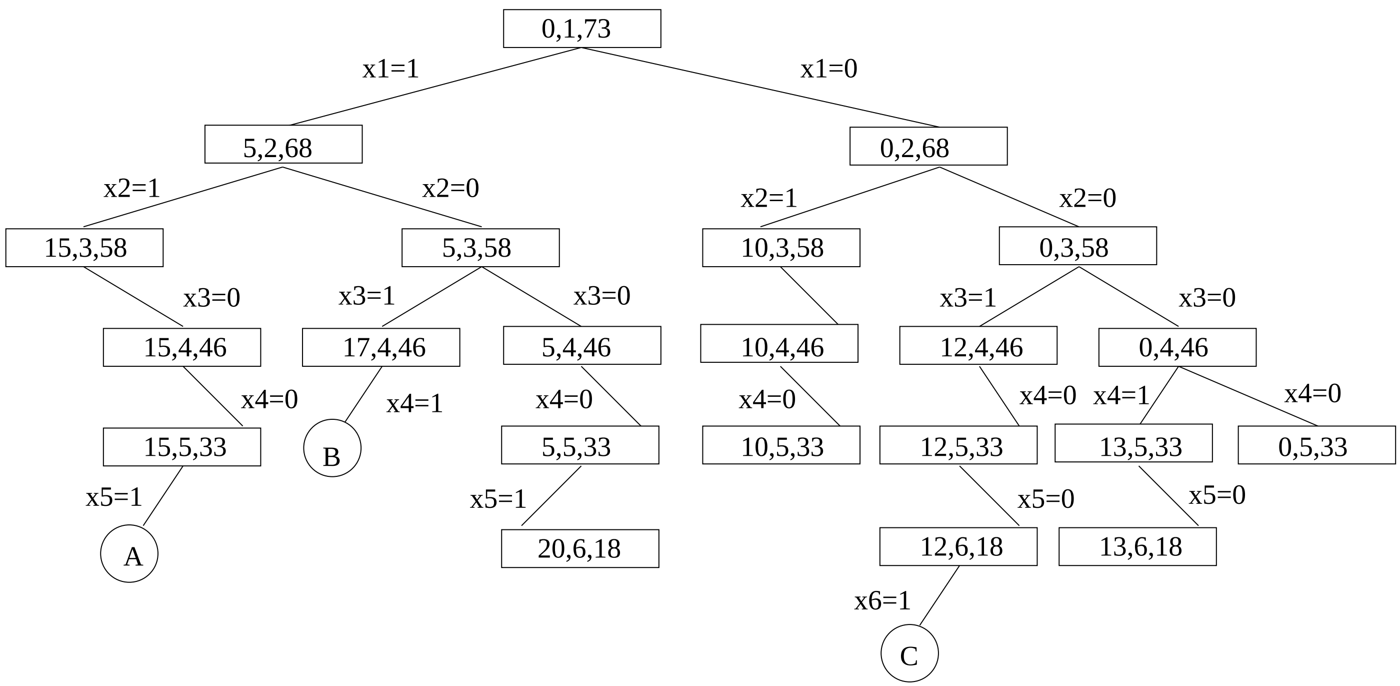
***Backtracking*: problema SOS (cont.)**

```
SOS(s, k, r);   (* Domínio de x_k será sempre {0,1}. *)
    x_k ← 1;   (* Caso x_k = 1. *)
    Se (s + w_k = W) então
        IMPRIMA (x_1, ..., x_k, 0, ..., 0) (* não precisa de recursão *)
    se não
        Se (s + w_k + w_{k+1} ≤ W) então SOS(s + w_k, k + 1, r − w_k);
    fim-se
    x_k ← 0;   (* Caso x_k = 0. *)
    Se (s + r − w_k ≥ W) e (s + w_{k+1} ≤ W) então
        SOS(s, k + 1, r − w_k);
    fim-se
fim.
```

***Backtracking*: problema SOS (cont.)**

- ▷ Exemplo: $n = 6$, $W = 30$, $S = \{5, 10, 12, 13, 15, 18\}$.
- ▷ Espaço de estados completo: $2^{6+1} - 1 = 127$.
- ▷ Parte da árvore de espaço de estados gerada por `SOS(0,1,73)` (próxima transparência ...)
- ▷ Legenda:
  - triplas (s, k, r);
  - $\bigcirc$ são os nós-resposta.

Backtracking: problema SOS (cont.)



***Backtracking*: p-coloração de grafos (COR)**

- COR: dado um grafo não-orientado G com n vértices e representado por sua matriz de adjacências A, encontrar todas as colorações de G com p cores ou menos.
- Representação das soluções: tuplas de tamanho n (fixo) onde $x_i \in \{0, 1, \ldots, p\}$ representa a cor do vértice i.
- A cor *zero* significa que o vértice ainda não está colorido.
- O algoritmo inicializará a tupla com *zeros*.
- *Função Limitante*: recursão só é interrompida se não for possível alocar uma cor para o k-ésimo vértice.

***Backtracking*: p-coloração de grafos (COR)**

Domínio($x_1, \ldots, x_{k-1}$);

Para $i = 1$ **até** p **faça** $\texttt{pode}[i] = 1$;

Para $j = 1$ **até** $k - 1$ **faça**

Se ($A[k, j] = 1$ e $x_j \neq 0$) **então**

$\texttt{pode}[x_j] \leftarrow 0$; (* não pode ter cor igual a um vizinho *)

fim-para;

$\ell \leftarrow 0$; (* Constrói o domínio *)

Para $i = 1$ **até** p **faça**

Se ($\texttt{pode}[i] = 1$) **então**

$\ell \leftarrow \ell + 1$; $T[\ell] \leftarrow i$;

fim-se;

fim-para;

Retornar (T, ℓ).

fim.

***Backtracking*: p-coloração de grafos (COR)**

`COR`(k);

$(T, \ell) \leftarrow$ Domínio$(x_1, x_2, \ldots, x_{k-1})$;

Para $i = 1$ **até** ℓ **faça**

$x_k \leftarrow T[i]$;

Se $k = n$ **então** (* todos vértices estão coloridos *)

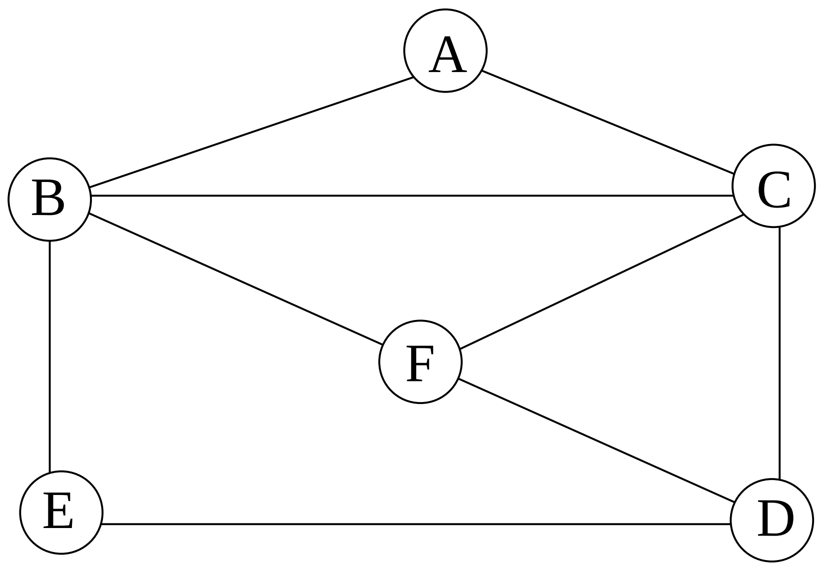
`IMPRIMA`$(x_1, \ldots, x_k)$

se não `COR`$(k+1)$;

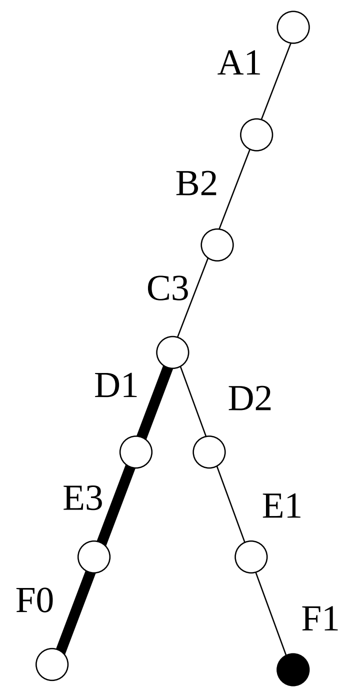
fim-para.

Backtracking: p-coloração de grafos (COR)

▷ Exemplo: $p = 3$.



Backtracking: p-coloração de grafos (COR)



Exemplo de *backtracking*: árvore de espaço de estados

Tratamento de problemas $\mathcal{NP}$-difíceis: *branch&bound*

- ▷ Aplicado a problemas onde se quer otimizar uma *função objetivo*.
- ▷ Exploração do espaço de estados: todos os filhos de um nó da árvore de espaço de estados são gerados ao mesmo tempo.
- ▷ Estratégia do melhor limitante (*best bound*): próximo nó a ser explorado é indicado por uma *função classificadora*.
- ▷ Em cada nó da árvore, a *função classificadora* estima o melhor valor da função objetivo no subespaço das soluções representadas por aquele nó.
- ▷ Os nós são *amadurecidos* por: **(1)** <u>inviabilidade</u> (não satisfazer as restrições implícitas); **(2)** <u>limitante</u> (função classificadora indica que ótimo não pode ser atingido naquela subárvore) ou **(3)** <u>otimalidade</u> (ótimo da subárvore já foi encontrado).

Algoritmo genérico de *Branch&bound*

`B&B`(k); (* considerando problema de **maximização** *)
 `Nós-ativos` $\leftarrow$ {nó raiz}; `melhor-solução` $\leftarrow \{\}$; $\underline{z} \leftarrow -\infty$;
 Enquanto (`Nós-ativos` não está vazia) **faça**
  Escolher um nó k em `Nós-ativos` para ramificar;
  Remover k de `Nós-ativos`;
  Gerar os filhos de k: $n_1, \ldots, n_q$ e computar os $\overline{z}_{n_i}$ correspondentes;
    (* $\overline{z}_{n_i} \leftarrow -\infty$ se restrições implícitas não são satisfeitas *)
  Para $j = 1$ **até** q **faça**
   se ($\overline{z}_{n_i} \leq \underline{z}$) **então** amadurecer o nó n_i;
   se não
    Se (n_i representa uma solução completa) **então**
     $\underline{z} \leftarrow \overline{z}_{n_i}$; `melhor-solução` $\leftarrow$ {solução de n_i};
    se não adicionar n_i à lista `Nós-ativos`.
   fim-se
  fim-para
 fim-enquanto
fim.

***Branch&bound*: mochila binária (BKP)**

▷ Dados n itens com pesos positivos $w_1, \ldots, w_n$ e valores positivos $c_1, \ldots, c_n$, encontrar um subconjunto de itens de **valor máximo** e cujo peso não exceda a capacidade da mochila dada por um valor positivo W.

▷ *Função classificadora*: como estimar o valor da *função objetivo* ?

▷ *Relaxação*: posso levar qualquer fração de um item.

▷ Algoritmo para o problema relaxado quando os itens estão ordenados de forma que $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}$.

▷ *Por quê funciona ?*

Branch&bound: mochila binária (BKP)

```
Calcula_z̄(W, C, k); (* função classificadora para BKP *)
    j ← k + 1;
    W' ← W;
    C' ← C;
    Enquanto W' ≠ 0 faça
        x_j ← min{W'/w_j, 1};
        W' ← W' − w_j x_j;
        C' ← C' + c_j x_j;
        j ← j + 1;
    enquanto
    Retornar C';
fim
```

***Branch&bound*: mochila binária (BKP)**

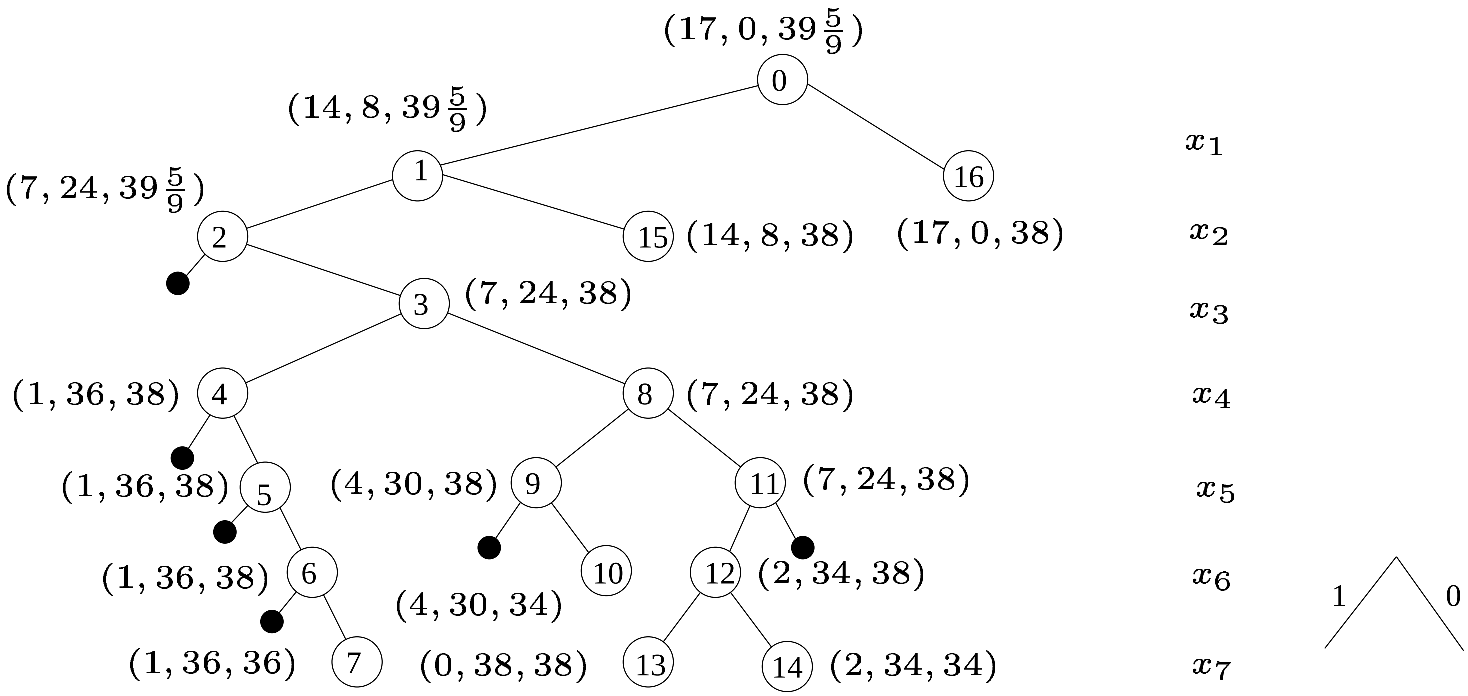
▷ Exemplo:

$$
\begin{array}{lll}
\max & 8x_1 + 16x_2 + 20x_3 + 12x_4 + 6x_5 + 10x_6 + 4x_7 & \\
& 3x_1 + \ \ 7x_2 + \ \ 9x_3 + \ \ 6x_4 + 3x_5 + \ \ 5x_6 + 2x_7 & \leq 17 \\
& x_i \in \{0,1\}, i = 1, \ldots, n. &
\end{array}
$$

▷ Parte explorada da árvore de espaço de estados (próxima transparência).

▷ Legenda: $(W', C, \overline{z}_{n_i})$ onde W' é a capacidade restante na mochila, C é o custo da solução parcial correspondente ao nó e $\overline{z}_{n_i}$ é o valor do limitante obtido pela função classificadora no nó.

Branch&bound: problema da mochila (cont.)



Ordem de geração dos nós: $0, 1, 16, 2, 15, 3, 4, 8, 5, 6, 7, 9, 11, 10, 12, 13, 14$

Branch&bound: Flowshop Scheduling (FSP)

▷ **Dados de entrada**: conjunto de n tarefas $J_1, \ldots, J_n$ cada uma delas composta de duas subtarefas sendo que a primeira deve ser executada na máquina 1 e a segunda na máquina 2, somente após encerrada a execução da primeira. O tempo de processamento da tarefa J_j na máquina i é dado por t_{ij}.

▷ **Definição**: o *tempo de término* da tarefa J_j na máquina i é dado por f_{ij}.

▷ **Pede-se**: encontrar uma seqüência de execução das subtarefas nas máquinas de modo que a soma dos tempos de término na máquina 2 seja mínima. Ou seja, a função objetivo é:

$$\min f = \sum_{j=1}^{n} f_{2j}.$$

***Branch&bound*: FSP (cont.)**

▷ **Resultados conhecidos para o FSP**:

- a versão de decisão de FSP é $\mathcal{NP}$-completo.
- Existe um escalonamento ótimo no qual a seqüência de execução das tarefas é a mesma nas duas máquinas (***permutation schedules***) e no qual não há tempo ocioso *desnecessário* entre as tarefas.

▷ Exemplo: $n = 3$.

t_{ij}	Máquina 1	Máquina 2
Tarefa 1	2	1
Tarefa 2	3	1
Tarefa 3	2	3

***Branch&bound*: (FSP) (cont.)**

▷ *Permutation Schedule* ótimo: $f = 18$

Máquina 1	1	1	3	3	2	2	2	
Máquina 2			1		3	3	3	2

▷ Outro *Permutation Schedule*: $f = 21$

Máquina 1	2	2	2	3	3	1	1		
Máquina 2				2		3	3	3	1

***Branch&bound*: FSP (cont.)**

- Representação da solução: como existe uma soluçao ótima que é um *permutation schedule*, o natural seria utilizar uma tupla $(x_1, \ldots, x_n)$ de tamanho fixo onde x_i é o número da i-ésima tarefa da permutação.
- Suponha que num dado nó da árvore as tarefas de um subconjunto M de tamanho r tenham sido escalonadas. Seja t_k, $k = 1, \ldots, n$, o índice da k-ésima tarefa em qualquer escalonamento que possa ser representado por um nó na subárvore cuja raiz é o nó corrente. O custo deste escalonamento será:

$$f = \sum_{i \in M} f_{2i} + \sum_{i \notin M} f_{2i}.$$

Branch&bound: FSP (cont.)

▷ Como o primeiro termo da soma já está definido, as seguintes *funções classificadoras* poderiam estimar o valor do outro termo:

$$S_1 = \sum_{k=r+1}^{n} [f_{1,t_r} + (n-k+1)t_{1,t_k} + t_{2,t_k}],$$

na qual assume-se que cada tarefa começa a ser executada na máquina 2 imediatamente após a sua conclusão na máquina 1, e

$$S_2 = \sum_{k=r+1}^{n} [\max(f_{2,t_r}, f_{1,t_r} + \min_{i \notin M} t_{1i}) + (n-k+1)t_{2,t_k}],$$

na qual assume-se que cada tarefa começa na máquina 2 imediatamente depois que a tarefa precedente termina sua execução na máquina 2.

***Branch&bound*: FSP (cont.)**

- A minimização de S_1 pode ser obtida ordenando-se as tarefas na ordem crescente dos valores de t_{1,t_k}.
- A minimização de S_2 pode ser obtida ordenando-se as tarefas na ordem crescente dos valores de t_{2,t_k}.
- Se $\hat{S}_1$ e $\hat{S}_2$ são os mínimos acima, tem-se um *limitante inferior* facilmente calculado por:

$$f \geq \sum_{i \in M} f_{2i} + \max(\hat{S}_1, \hat{S}_2).$$

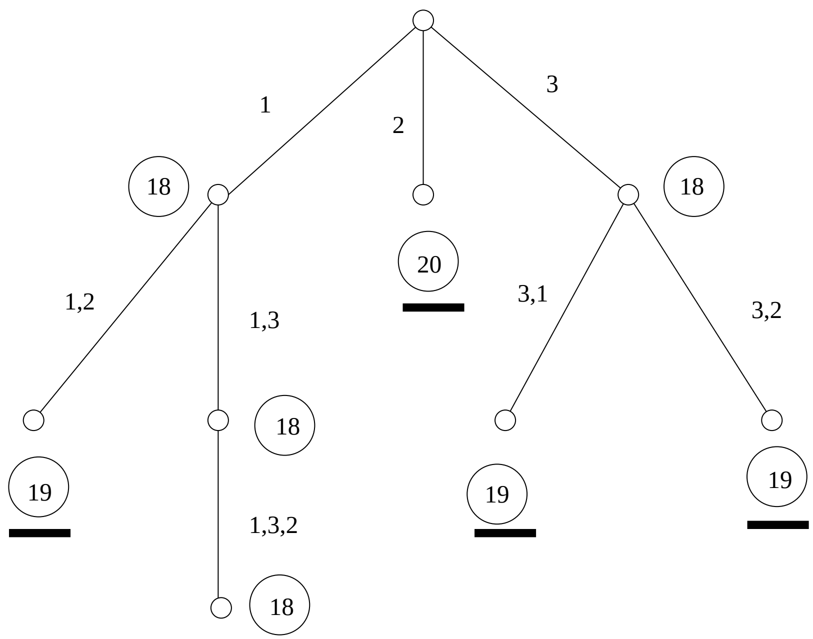
Branch&bound: FSP (cont.)

▷ Exemplo (continuação): os valores computados para estimar f para os três nós filhos da raiz seriam:

$$f = \begin{cases} 18 & \text{se a tarefa 1 for escalonada primeiro;} \\ 20 & \text{se a tarefa 2 for escalonada primeiro;} \\ 18 & \text{se a tarefa 3 for escalonada primeiro.} \end{cases}$$

▷ Parte da árvore de espaços gerada: próxima transparência.

Branch&bound: FSP (cont.)



Tratamento de problemas $\mathcal{NP}$-difíceis: Programação Linear

▷ Um problema de Programação Linear é expresso como:

$$\begin{array}{rrcll} \min & z = \sum_{j=1}^{n} c_j x_j & & & \\ \text{Sujeito a} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_i, & i = 1, \ldots, m \\ & x_j & \geq & 0, & j = 1, \ldots, n \end{array}$$

▷ Ou na forma matricial por:

$$\begin{array}{rrcl} \min & z = cx & & \\ \text{Sujeito a} & Ax & \leq & b \\ & x & \in & \mathbb{R}^n_+ \end{array}$$

onde A é uma matriz $m \times n$ e b e c são vetores de m e n posições respectivamente.

Programação Linear: Exemplo

Um fabricante de ligas de alumínio deseja produzir 200 T de certa liga especial a custo mínimo, satisfazendo certas restrições quanto à composição da mesma. Para tanto, há disponível no mercado alguns tipos de sucata, com preços e composições conhecidas. As condições que devem ser obedecidas pela liga e os demais dados estão resumidos na tabela a seguir. Equacionar o problema como um PL.

sucata →	1	2	3	Liga	
composição ↓				min(T)	max(T)
FE	0.20	0.09	0.10	—-	30
MN	0.04	0.04	0.02	—-	5
MG	0.04	0.06	—-	—-	3
AL	0.70	0.75	0.80	—-	150
SI	0.04	0.06	0.08	5	40
Custo (Us$ / T)	130	180	210		
Disponibilidade (T)	150	150	200		

Programação Linear: Exemplo (cont.)

- *Variáveis*: x_j quantidade a ser adquirida da sucata j em toneladas. Assim, se d_j é a quantidade disponível da sucata j no mercado então, $x_j \leq d_j$ e $x_j \geq 0$ para $j = 1, 2, 3$.

- Se c_j é o custo da tonelada da sucata j então a *função objetivo* é

$$\min \sum_{j=1}^{3} c_j x_j.$$

- Sendo M_i e m_i respectivamente as quantidades máxima e mínima do componente i na liga, e a_{ij} a fração de i na sucata j, as seguintes restrições devem ser atendidas:

$$\sum_{j=1}^{3} a_{ij} x_j \leq M_i \quad \text{e} \quad \sum_{j=1}^{3} a_{ij} x_j \geq m_i, \quad \forall i \in \{\text{FE,MN,MG,AL,SI}\}.$$

- O total a ser produzido da liga é 120 T, ou seja, $\sum_{j=1}^{n} x_j = 120$.

Programação Linear (cont.)

▷ Hipóteses da PL:

- *Proporcionalidade*: a contribuição de uma variável na função objetivo e numa restrição dobra se o valor da variável dobrar.
- *Aditividade*: as contribuições individuais das variáveis se somam na função objetivo e nas restrições e são independentes.
- *Divisibilidade*: as variáveis podem ser divididas em qualquer fração.
- *Determinismo dos coeficientes*: todos coeficientes dos vetores c e b assim como os elementos da matriz A são dados por constantes conhecidas.

Programação Linear (cont.)

▷ Alguns truques algébricos:

$$\min z \Longleftrightarrow \max(-z)$$

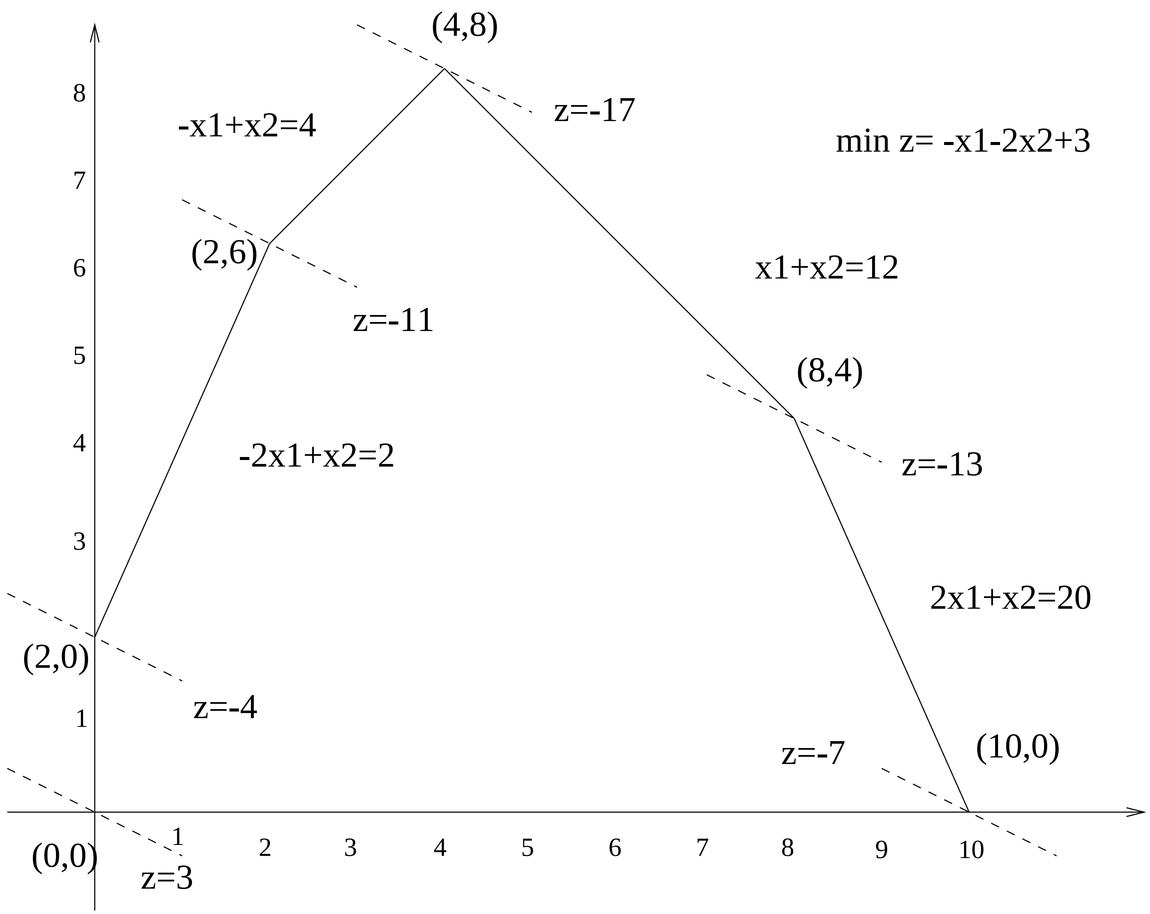
$$ax \leq b \Longleftrightarrow ax + s = b, s \geq 0$$

$$ax \geq b \Longleftrightarrow ax - s = b, s \geq 0$$

$$ax = b \Longleftrightarrow \begin{cases} ax \leq b \\ ax \geq b \end{cases}$$

$$x \in \mathbb{R} \Longleftrightarrow x = x' - x'', x' \geq 0, x'' \geq 0.$$

Programação Linear: Resolução gráfica



Programação Linear: algoritmos

- ▷ Método *Simplex*: complexidade **exponencial** mas muito eficiente na prática !
- ▷ Método dos *Elipsóides* (Kachian, 1979): primeiro algoritmo polinomial para PL. Pouca utilizado na prática devido a dificuldades de implementação !
- ▷ Métodos de *Pontos Interiores* (Karmarkar, 1984): algoritmo polinomial e bastante eficiente na prática !

Programação Linear Inteira (PLI)

▷ **Programação Linear Inteira** (PLI):

Problema PLI *Puro*:

$$\begin{array}{rrcl} \min & z = cx & & \\ \text{Sujeito a} & Ax & \leq & b \\ & x & \in & \mathbb{Z}^n_+ \end{array}$$

Problema PLI *Misto*:

$$\begin{array}{rrcl} \min & z = cx + dy & & \\ \text{Sujeito a} & Ax + By & \leq & b \\ & x & \in & \mathbb{R}^p_+ \\ & y & \in & \mathbb{Z}^n_+ \end{array}$$

▷ Versão de *decisão*: dada uma matriz inteira $A : m \times n$, dois vetores inteiros $c : 1 \times n$ e $b : m \times 1$ e um valor inteiro q, determinar se existe $x \in \mathbb{Z}^n$ tal que $Ax \leq b, x \geq 0$ e $cx \leq q$.

Programação Linear Inteira (PLI)

▷ *Teorema*: PLI $\in \mathcal{NP}$.

É possível provar que se o sistema tem solução então existe uma solução (vetor) cujo valor de cada componente é limitado polinomialmente pelo tamanho entrada, ou seja, existe um certificado sucinto para PLI.

▷ *Teorema*: PLI $\in \mathcal{NP}$-difícil.

Basta provar que um problema de $\mathcal{NP}$-difícil se reduz polinomialmente a PLI. Mas isso é ***equivalente a formular o problema usando programação linear inteira !***

Programação Linear Inteira: formulações

- ▷ Exemplo 1: SAT $\propto_{\text{poli}}$ PLI.
- ▷ Instância do SAT: $F = (x + y + \overline{z}).(\overline{x} + \overline{y} + z).(y + \overline{z})$ com $m = 3$ cláusulas e $n = 3$ variáveis.
- ▷ Formulação PLI: criar 6 variáveis binárias $x, y, z, \overline{x}, \overline{y}$ e $\overline{z}$ que terão valor *um* se as literais correspondentes na fórmula F forem *verdadeiras* e terão valor *zero* caso contrário.

$$
\begin{array}{rlcllcl}
\min & w = x & & \text{(* qualquer função linear serve ! *)} & & & \\
\text{Sujeito a} & x + y + \overline{z} & \geq & 1, & \overline{x} + \overline{y} + z & \geq & 1, \\
 & y + \overline{z} & \geq & 1, & x + \overline{x} & = & 1, \\
 & y + \overline{y} & = & 1, & z + \overline{z} & = & 1, \\
 & \multicolumn{6}{l}{x, y, z, \overline{x}, \overline{y}, \overline{z} \in \{0, 1\}}
\end{array}
$$

Programação Linear Inteira: formulações (cont.)

- Exemplo 2: CLIQUE $\propto_{\text{poli}}$ PLI.
- Instância de CLIQUE: grafo $G = (V, E)$ com $|V| = n$ e $|E| = m$.
- Formulação PLI: para cada vértice $u \in V$ cria-se uma variável binária x_u que vale *um* se e somente se o vértice u está na clique.
- Função objetivo (CLIQUE de maior tamanho): $\max \sum_{u \in V} x_u$.
- Restrições: para cada aresta (u, v) que não está em E pelo menos um dos vértices não pode estar na clique, ou seja, $x_u + x_v \leq 1$.
- Logo a formulação se reduz a:

$$\begin{array}{rrcll} \max & z = \sum_{u \in V} x_u & & & \\ \text{Sujeito a} & x_u + x_v & \leq & 1, & \forall (u, v) \notin E \\ & x_u & \in & \{0, 1\} & \forall u \in V. \end{array}$$

Programação Linear Inteira: formulações (cont.)

- ▷ Exemplo 3 (cobertura de vértices): CV $\propto_{\text{poli}}$ PLI.
- ▷ Instância de CV: grafo $G = (V, E)$ com n vértices e m arestas.
- ▷ Formulação PLI: para cada $u \in V$ cria-se uma variável binária x_u que vale *um* se e somente se o vértice u está na cobertura.
- ▷ Função objetivo (cobertura de menor tamanho): $\min \sum_{u \in V} x_u$.
- ▷ Restrições: para cada aresta (u, v) de E pelo menos um dos seus vértices extremos está na cobertura, ou seja, $x_u + x_v \geq 1$.
- ▷ Logo a formulação se reduz a:

$$
\begin{array}{rrcll}
\min & z = \sum_{u \in V} x_u & & & \\
\text{Sujeito a} & x_u + x_v & \geq & 1, & \forall (u,v) \in E \\
 & x_u & \in & \{0,1\} & \forall u \in V.
\end{array}
$$

Programação Linear Inteira: formulações (cont.)

▷ Exemplo 4 (3 coloração): 3COLOR $\propto_{\text{poli}}$ PLI.

▷ Instância de 3COLOR: grafo $G = (V, E)$ com $|V| = n$ e $|E| = m$.

▷ Formulação PLI (variáveis):

- uma variável binária x_{uk} para cada vértice $u \in V$ e cada cor $k \in \{1, 2, 3\}$ tal que $x_{uk} = 1$ *se e somente se* o vértice u foi colorido com a cor k.
- uma variável binária y_k para toda cor $k \in \{1, 2, 3\}$ cujo valor será *um* se e somente se algum vértice receber a cor k.

▷ Função objetivo (minimizar o número de cores usadas):

$$\min \sum_{k=1}^{3} y_k.$$

Programação Linear Inteira: formulações (cont.)

▷ Restrições (3COLOR):

- Todo vértice deve receber exatamente uma cor, ou seja,

$$\sum_{k=1}^{3} x_{uk} = 1, \forall u \in V.$$

- Se um vértice recebe uma cor k, esta cor tem que ser usada:

$$x_{uk} \leq y_k, \forall u \in V, k = 1, 2, 3.$$

- Uma cor só pode ser usada se algum vértice tiver aquela cor:

$$y_k \leq \sum_{u \in V} x_{uk}, k = 1, 2, 3.$$

- Os vértices extremos de uma aresta não podem ter a mesma cor: $x_{uk} + x_{vk} \leq 1, \forall (u, v) \in E, k = 1, 2, 3.$

Programação Linear Inteira: formulações (cont.)

- ▷ Exemplo 5 (*Scheduling* com janela de tempo): SJT $\propto_{\text{poli}}$ PLI.
- ▷ Instância de SJT: um conjunto T de n tarefas e, para cada $t \in T$, um prazo de início r_t, uma duração ℓ_t e um prazo de conclusão d_t, sendo r_t, d_t e ℓ_t inteiros não-negativos. Decidir se existe um *seqüenciamento viável* das tarefas de T em uma máquina.
- ▷ Variáveis naturais: para todo $t \in T$ o instante de início de execução da tarefa é dado por σ_t.
- ▷ Função objetivo: qualquer função linear serve, e.g., $\min \sigma_1$.
- ▷ Restrições envolvendo um única tarefa:

$$\begin{array}{lcll} \sigma_t & \geq & r_t, & \forall t \in T \quad \text{(início da tarefa)} \\ \sigma_t + \ell_t & \leq & d_t,, & \forall t \in T \quad \text{(fim da tarefa)} \end{array}$$

Programação Linear Inteira: JST (cont.)

- Variáveis binárias: para poder representar corretamente a relação entre os tempos de início de duas tarefas t e t' em T é necessário que se saiba qual delas irá ser executada antes.

$$y_{tt'} = \begin{cases} 1, \text{se } t \text{ antecede } t' \\ 0, \text{caso contrário} \end{cases}, \text{ para todo par } \{t, t'\} \in T.$$

- Restrições envolvendo pares de tarefas:

$$\begin{array}{rcll} y_{tt'} + y_{t't} & = & 1, & \forall \{t, t'\} \in T \\ \sigma_t + \ell_t & \leq & \sigma_{t'} + (1 - y_{tt'})M, & \forall \{t, t'\} \in T \\ \sigma_{t'} + \ell_{t'} & \leq & \sigma_t + y_{tt'}M, & \forall \{t, t'\} \in T \end{array}$$

onde M é um valor suficientemente grande. Por exemplo, M poderia ser $\max_{t \in T}\{d_t\} - \min_{t \in T}\{r_t\}$.

Programação Linear Inteira: formulações (cont.)

▷ Exemplo 5 (Problema de Transporte): uma grande empresa de consultoria possui m escritórios e n clientes espalhados em todo Brasil. No escritório i estão baseados a_i consultores e cada cliente j, para $j = 1, \ldots, n$, contratou b_j consultores. O custo de deslocar um consultor do escritório i para o cliente j é c_{ij}.

Equacionar este problema como um PLI.

▷ Variáveis: para todo par (escritório i, cliente j), define-se a variável **inteira** x_{ij} que representa o número de consultores que serão deslocados do escritório i para o cliente j.

Programação Linear Inteira: formulações (cont.)

▷ A formulação do problema como um PLI é dada por:

$$
\begin{array}{llcll}
\min & z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} & & & \\
\text{Sujeito a} & \sum_{j=1}^{n} x_{ij} & \leq & a_i, & i = 1, \ldots, m \\
 & \sum_{i=1}^{m} x_{ij} & = & b_j, & j = 1, \ldots, n \\
 & x_{ij} & \in & \mathbb{Z}_+^n, & i = 1, \ldots, m \text{ e} \\
 & & & & j = 1, \ldots, n.
\end{array}
$$

▷ *Observação*: este problema pode ser resolvido em tempo polinomial !

Programação Linear Inteira: formulações (cont.)

- Exemplo 6 (Problema de Localização de Facilidades **Não Capacitado** (UFL)): Dado um conjunto $N = \{1, \ldots, n\}$ de locais potenciais para instalação de depósitos e um conjunto $M = \{1, \ldots, m\}$ de clientes, suponha que f_j seja o custo de instalar o depósito em j e que c_{ij} seja o custo de transportar toda demanda de mercadorias do depósito j para o cliente i. Decidir quais depósitos instalar e que fração da demanda de cada cliente deve ser atendida por cada depoósito.
- Variáveis:

$$y_j = \begin{cases} 1, \text{se for instalado um depósito em } j \\ 0, \text{caso contrário} \end{cases}$$

Programação Linear Inteira: formulações (cont.)

▷ Variáveis (cont.):

$x_{ij} \in [0,1]$: fração da demanda do cliente i atendida pelo depósito j.

▷ Restrições:

- satisfação da demanda: $\sum_{j \in N} x_{ij} = 1, \forall\, i \in M$.
- uso do depósito j: $\sum_{i \in M} x_{ij} \leq m y_j, \forall\, j \in N$.

▷ Função objetivo: $\min z = \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} + \sum_{j \in N} f_j y_j$.

Programação Linear Inteira: formulações (cont.)

▷ Exemplo 7 (Problema do planejamento da produção **capacitado** (CLS)): decidir as quantidades a produzir de um certo produto em um horizonte de planejamento de n períodos de tempo. Os dados de entrada são:

f_t: **custo fixo** de produção no período t;

p_t: custo unitário de produção no período t;

h_t: custo unitário de estocagem no período t;

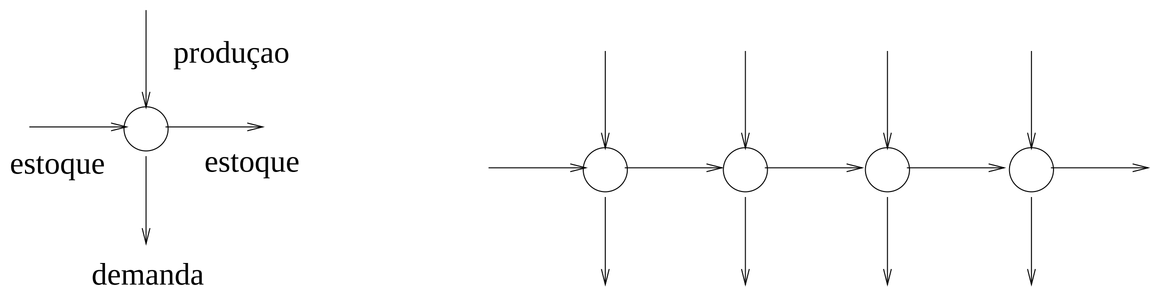
d_t: demanda no período t;

C_t: a capacidade de produçao no período t;

s_0, s_n: os estoques inicial e final do produto.

Programação Linear Inteira: formulações (cont.)

▷ Um modelo gráfico:



▷ Variáveis:

x_t: quantidade produzida no período t;

s_t: estoque no período t;

$$y_t : \begin{cases} 1, & \text{se for decidido produzir no período } t; \\ 0, & \text{caso contrário.} \end{cases}$$

Programação Linear Inteira: formulações (cont.)

▷ Formulação:

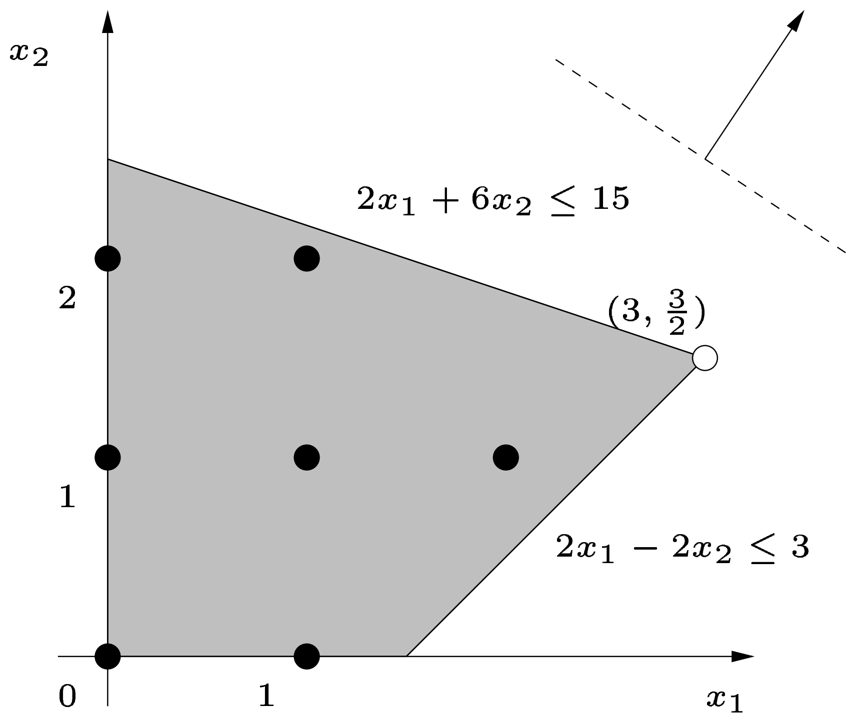
$$
\begin{array}{rrcll}
\min & z = \sum_{t=1}^{n}(p_t x_t + h_t s_t + f_t y_t) & & & \\
\text{Sujeito a} & s_{t-1} + x_t & = & d_t + s_t, & \text{para } t = 1, \ldots, n \quad (1)\\
 & x_t & \leq & C_t y_t, & \text{para } t = 1, \ldots, n \quad (2)\\
 & \multicolumn{4}{l}{s_t \geq 0, x_t \geq 0, \text{para } t = 1, \ldots, n,}\\
 & \multicolumn{4}{l}{y_t \in \{0,1\}, \text{ para } t = 1, \ldots, n.}
\end{array}
$$

onde (1) representa a conservação de fluxo no período t e (2) restringe a produção no período t a C_t ou a *zero* dependendo se a decisão foi de produzir ou não naquele período.

Prog. Linear Inteira: *branch&bound*

- *Função classificadora*: usar a relaxação linear, ou seja, resolver o problema como se todas as variáveis fossem reais.
- Explorar o nó com **melhor limitante** (*best bound)*.
- No caso de variáveis binárias, substituir $x \in \{0,1\}$ por $0 \leq x \leq 1$.
- *Divisão do espaço de soluções*: mais comum é usar a *regra da variável "mais fracionária"*, onde dada a solução ótima x^* da relaxação linear, encontra-se a variável x cujo máximo das diferenças $(x - \lfloor x^* \rfloor)$ e $(\lceil x^* \rceil - x)$ seja o mais próximo de 0.5 e cria-se dois PLIs a partir do PLI corrente acrescentando em um deles a restrição $x \leq \lfloor x^* \rfloor$ e no outro a restrição $x \geq \lceil x^* \rceil$.

Prog. Linear Inteira: *branch&bound* (exemplo)



$PL0$:

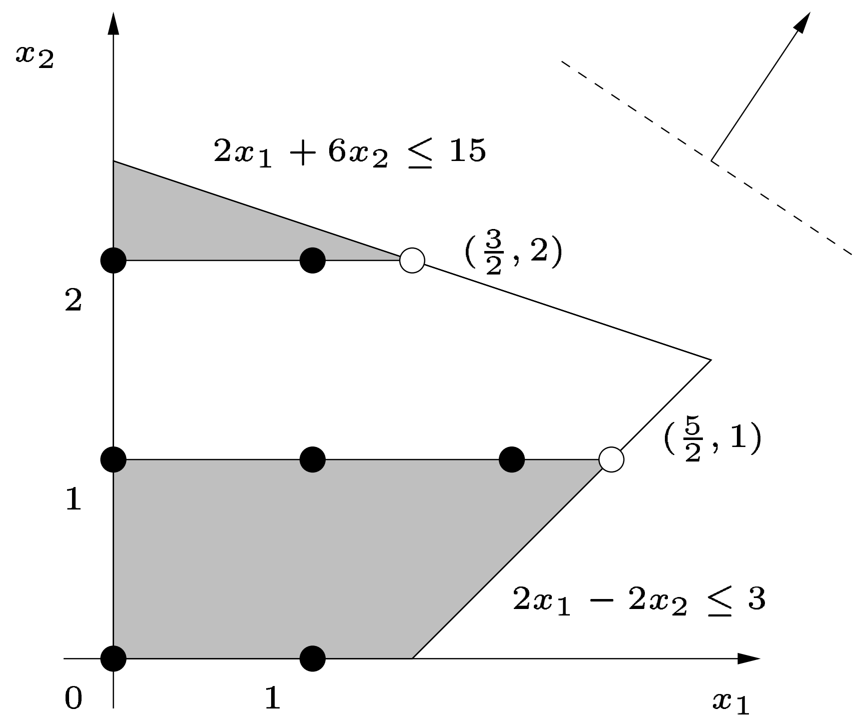
$\max 2x_1 + 3x_2$

$2x_1 + 6x_2 \leq 15$

$2x_1 - 2x_2 \leq 3$

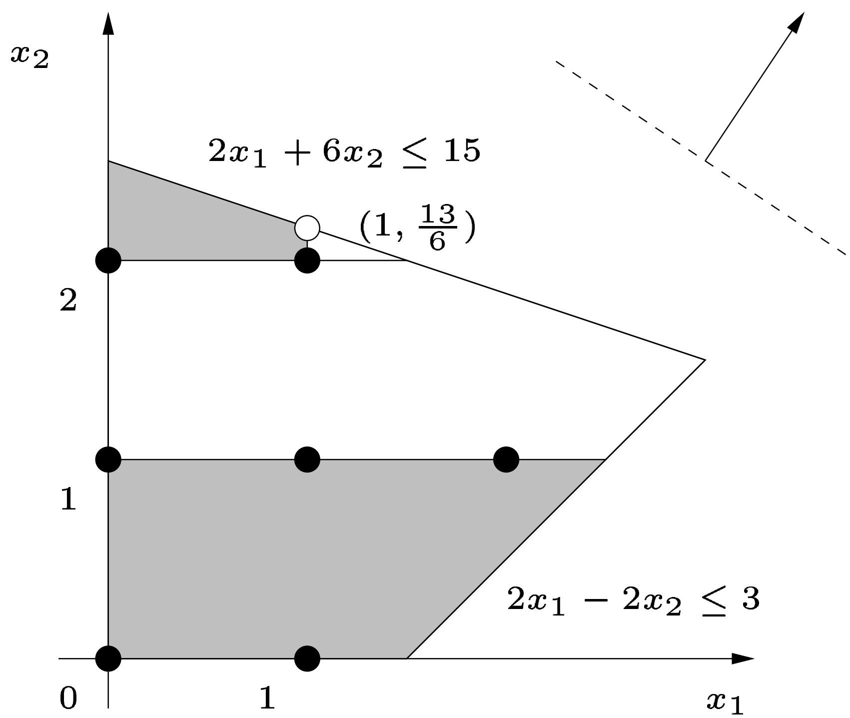
$x_1 \geq 0, x_2 \geq 0$

Prog. Linear Inteira: *branch&bound*(exemplo)



$$PL1 = PL0 + \{x_2 \leq 1\}, \ (z_1 = 8)$$
$$PL2 = PL0 + \{x_2 \geq 2\}, \ (z_2 = 9)$$

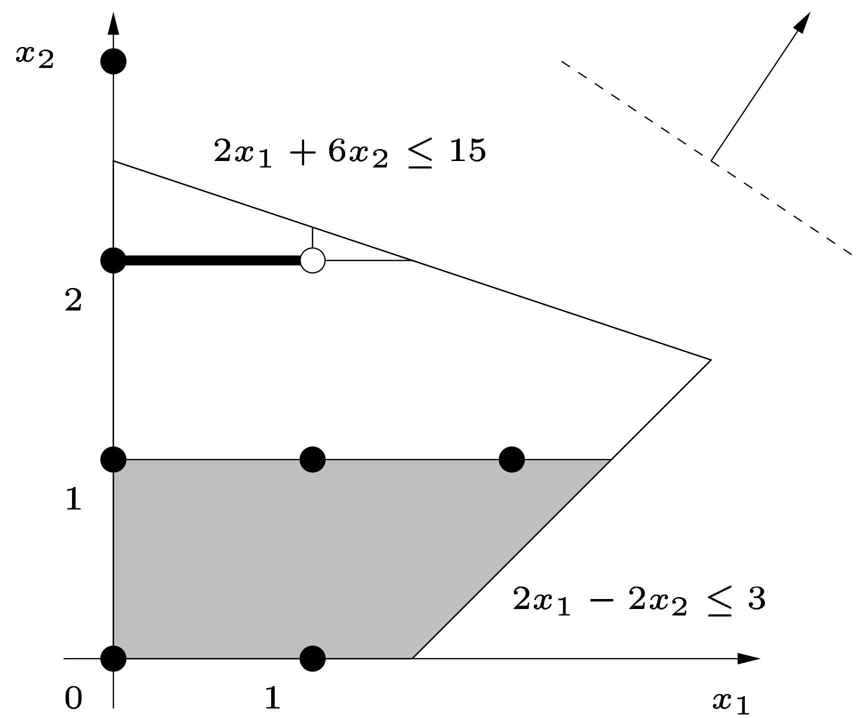
Prog. Linear Inteira: *branch&bound* (exemplo)



$PL3 = PL2 + \{x_1 \leq 1\},\ (z_3 = 8.5)$

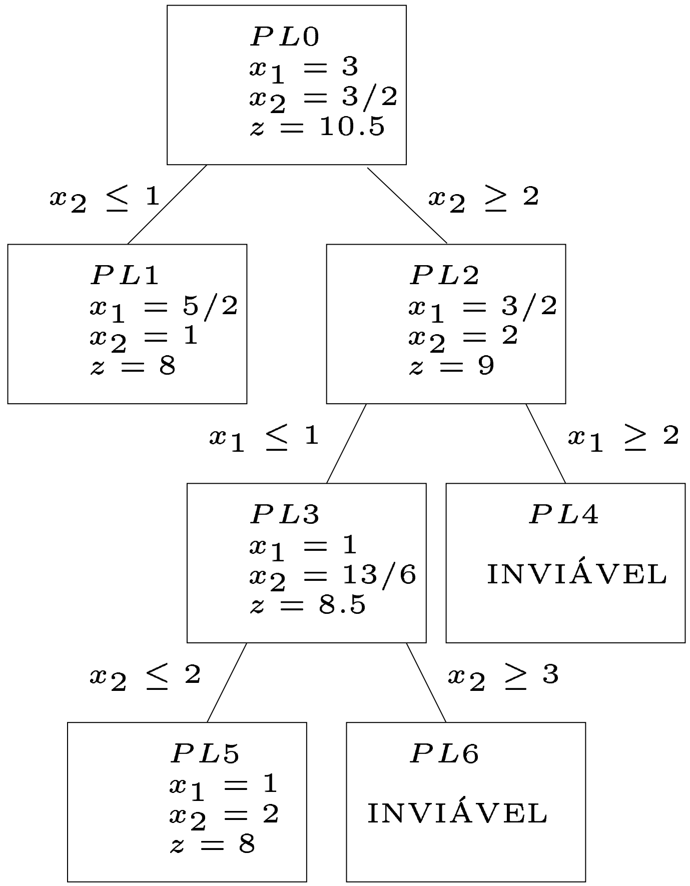
$PL4 = PL2 + \{x_1 \geq 2\}$, (inviável)

Prog. Linear Inteira: *branch&bound* (exemplo)



$PL5 = PL3 + \{x_2 \leq 2\}, \ (z_5 = 8)$
$PL6 = PL3 + \{x_2 \geq 3\}$, (inviável)

Prog. Linear Inteira: *branch&bound* (exemplo)



Tratamento de problemas $\mathcal{NP}$-difíceis: Heurísticas

▷ Heurísticas são algoritmos que geram soluções viáveis para quais *não se pode dar garantias de qualidade*. Ou seja, não se sabe o quão *distante* a solução gerada está de uma solução ótima (5% ?, 10% ?, 50% ?, 100% ?, ...).

▷ Tipos de heurísticas:

- *construtivas*: normalmente adotam estratégias *gulosas* para construir as soluções. Tipicamente são aplicadas a problemas onde é fácil obter uma solução viável.
- *de busca local*: partem de uma solução inicial e, através de transformações bem definidas, visitam outras soluções até atingir um *critério de parada* pré-definido.

Heurísticas Construtivas (TSP)

▷ Exemplo 1: TSP em um grafo não orientado completo.

```
Vizinho-Mais-Próximo(n, d)      (* d: matriz de distâncias *)
    Para i = 1 até n faça visitado[i] ← Falso ;
    visitado[1] ← Verdadeiro;
    ciclo ← {}, comp ← 0 e k ← 1;
    Para i = 1 até n − 1 faça
        j* ← argmin{d[k, j] : visitado[j] = Falso};
        visitado[j*] ← Verdadeiro ;
        ciclo ← ciclo ∪ {(k, j*)};     comp ← comp + d[k, j*];
        k ← j*;
    fim-para
    ciclo ← ciclo ∪ {(k, 1)};     comp ← comp + d[k, 1];
Retorne comp.
```

▷ Complexidade: $O(n^2)$

Heurísticas Construtivas (TSP)

▷ Exemplo 2: heurística para o TSP ≡ algoritmo de Kruskal para AGM.

`TSP-Guloso`(n, d) (* d: matriz de distâncias *)
  $\mathcal{L} \leftarrow$ lista das arestas ordenadas crecentemente pelo valor de d;
  Para $i = 1$ **até** n **faça** grau$[i] \leftarrow 0$; componente$[i] = i$ **fim-para**
  $k \leftarrow 0$; ciclo $\leftarrow \{\}$; comp $\leftarrow 0$;
  Enquanto $k \neq n$ **faça**
    $(u, v) \leftarrow$ `Remove-primeiro`$(\mathcal{L})$;
    Se (grau$[u] \leq 1$ **e** grau$[v] \leq 1$ **e** componente$(u) \neq$ componente(v))
    ou (grau$[u] =$ grau$[v] = 1$ **e** $k = n - 1$) **então**
      ciclo $\leftarrow$ ciclo $\cup \{(u, v)\}$; comp $\leftarrow$ comp $+ d[u, v]$;
      `Unir-componentes`(u, v);
      grau$[u]++$; grau$[v]++$; $k++$;
    fim-se
  fim-enquanto
Retorne comp.

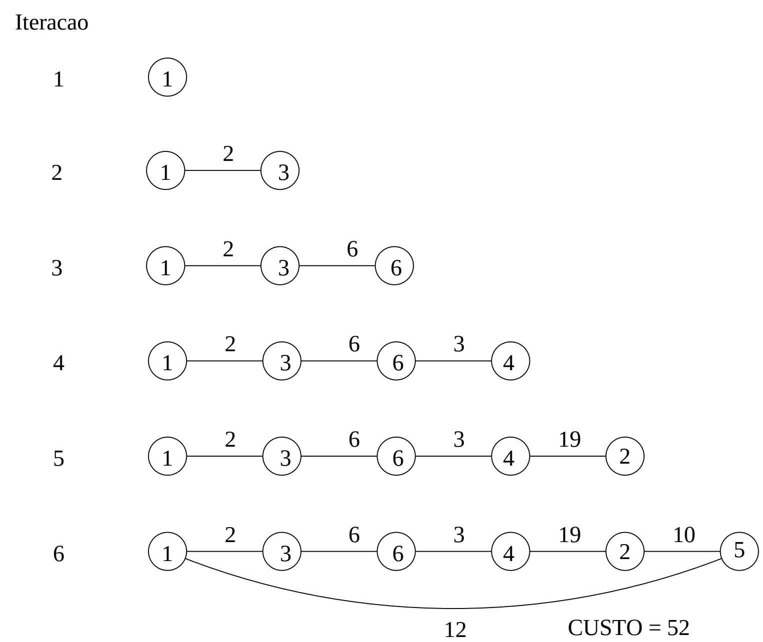
▷ Complexidade: $O(n^2 \log n)$ (usar *compressão de caminhos* para *união de conjuntos disjuntos*).

Heurísticas Construtivas (TSP)

Aplicação das heurísticas para o TSP:

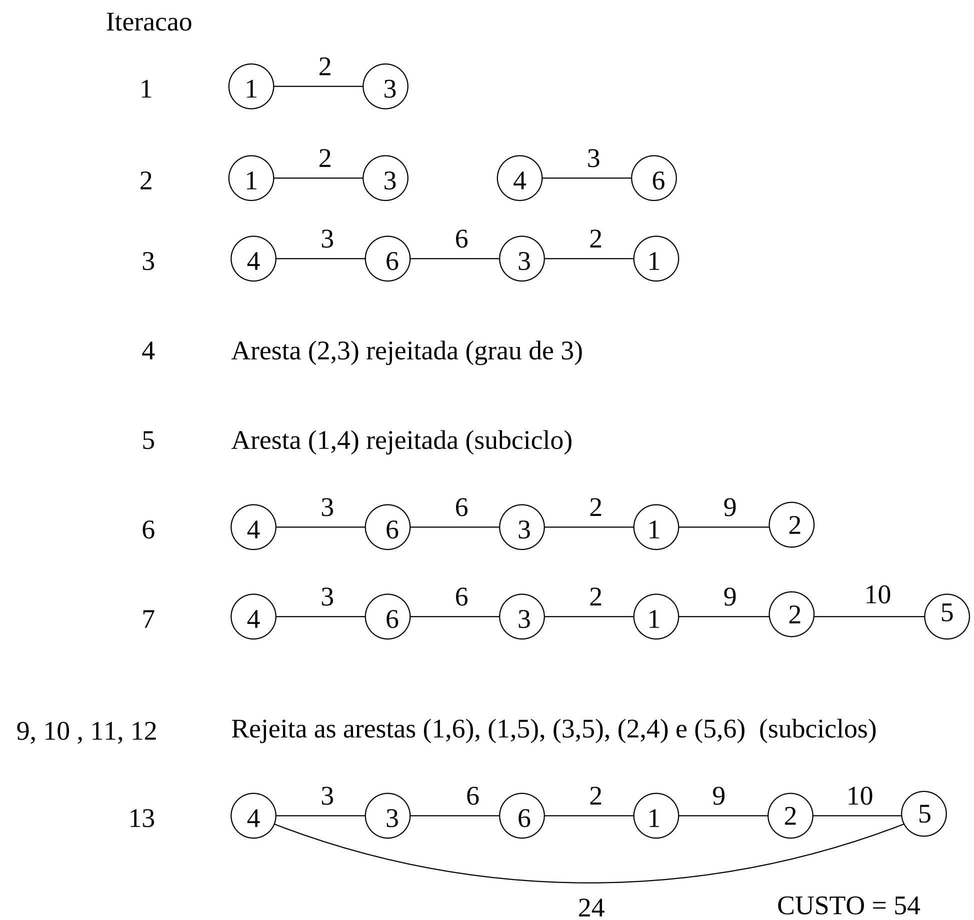
$$d = \begin{bmatrix} - & 9 & 2 & 8 & 12 & 11 \\ 9 & - & 7 & 19 & 10 & 32 \\ 2 & 7 & - & 29 & 18 & 6 \\ 8 & 19 & 29 & - & 24 & 3 \\ 12 & 10 & 18 & 24 & - & 19 \\ 11 & 32 & 6 & 3 & 19 & - \end{bmatrix}$$

Vizinho-Mais-Próximo



Heurísticas Construtivas (TSP)

TSP-GULOSO



Heurísticas Construtivas (Mochila)

▷ Exemplo 2: Problema da Mochila.

`Mochila-guloso`(c, w, W)
  Ordenar itens segundo a razão $\frac{c_i}{w_i}$;
  (* assuma que $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}$ *)

  $\overline{W} \leftarrow W$; $S \leftarrow \{\}$;
  Para $i = 1$ **até** n **faça**
    Se $w_i \leq \overline{W}$ **então**
      $\overline{W} \leftarrow \overline{W} - w_i$;
      $S \leftarrow S \cup \{i\}$;
    fim-se
  fim-para
Retorne S.

▷ Complexidade: $O(n \log n)$.

Heurísticas Construtivas (Mochila)

▷ Aplicação da heurística `Mochila-guloso`.

$$
\begin{array}{ll}
\text{maximize} & 8x_1 + 16x_2 + 20x_3 + 12x_4 + 6x_5 + 10x_6 + 4x_7 \\
\text{Sujeito a} & 3x_1 + \ 7x_2 + \ 9x_3 + \ 6x_4 + 3x_5 + \ 5x_6 + 2x_7 \leq 17, \\
 & x \in \mathbb{B}^7.
\end{array}
$$

Observação: $\frac{8}{3} \geq \frac{16}{7} \geq \frac{20}{9} \geq \frac{12}{6} \geq \frac{6}{3} \geq \frac{10}{5} \geq \frac{4}{2}$

▷ Solução gulosa: $S = \{1, 2, 4\}$, custo $= 36$.

▷ Solução ótima: $S = \{1, 2, 6, 7\}$, custo $= 38$.

Heurísticas Construtivas

▷ Soluções gulosas podem ser *arbitrariamente ruins* !

▷ `Mochila-guloso` é arbitrariamente ruim.

▷ Instância: $W = n$, $c_1 = 3/n$, $w_1 = 2/n$ e, para todo $i = 2, \ldots, n$, $c_i = n - (1/n)$ e $w_i = n - (1/n)$.
Observação: $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} = \ldots = \frac{c_n}{w_n}$.

▷ Solução gulosa: $S = \{1\}$, custo $= 3/n$.

▷ Solução ótima: $S = \{2\}$, custo $= n - (1/n)$.

▷ $\lim_{n \to \infty} \frac{(3/n)}{n-(1/n)} = 0$.
Ou seja, aumentando o valor de n nesta instância, a solução gulosa pode se afastar tanto quanto eu quiser da solução ótima !

Heurísticas Construtivas

- ▷ `Vizinho-Mais-Próximo` para o TSP é arbitrariamente ruim.
- ▷ Instância: matriz simétrica de distâncias d onde, para $i < j$, tem-se:

$$d[i,j] = \begin{cases} n^2, & \text{se } i = n-1 \text{ e } j = n, \\ 1, & \text{se } j = i+1, \\ 2, & \text{caso contrário.} \end{cases}$$

- ▷ Solução gulosa: ciclo $= \{1, 2, \ldots, n-1, n\}$ e comp $= n^2 + n$.
- ▷ Solução ótima: ciclo $= \{1, 2, \ldots, n-3, n, n-2, n-1\}$ e comp $= n+3$.
- ▷ $\lim_{n\to\infty} \frac{n+3}{n^2+n} = 0$.
  Novamente, aumentando o valor de n nesta instância, a solução gulosa pode se afastar tanto quanto eu quiser da solução ótima !

Heurísticas de Busca Local

- Como nos algoritmos de *branch&bound* e *backtracking*, as soluções são representadas por tuplas.
- Sendo $\mathcal{F}$ o conjunto de todas as possíveis tuplas e $t \in \mathcal{F}$, a *vizinhança* da solução t, $N(t)$, é o subconjunto de tuplas de $\mathcal{F}$ que podem ser obtidas ao se realizar um conjunto de transformações pré-determinadas sobre t.
- *Complexidade da vizinhança*: número de tuplas na vizinhança.
- Exemplo 1:

  A tupla é um vetor binário de tamanho n.

  $N_1(t)$: conjunto de todas as tuplas obtidas de t "flipando" uma de suas componentes.

  Complexidade: $\Theta(n)$.

Heurísticas de Busca Local

▷ Exemplo 2:

A tupla é um vetor representando uma permutação de $\{1, \ldots, n\}$.

$N_2(t)$: conjunto de todas as tuplas obtidas trocando-se as posições de dois elementos da permutação.

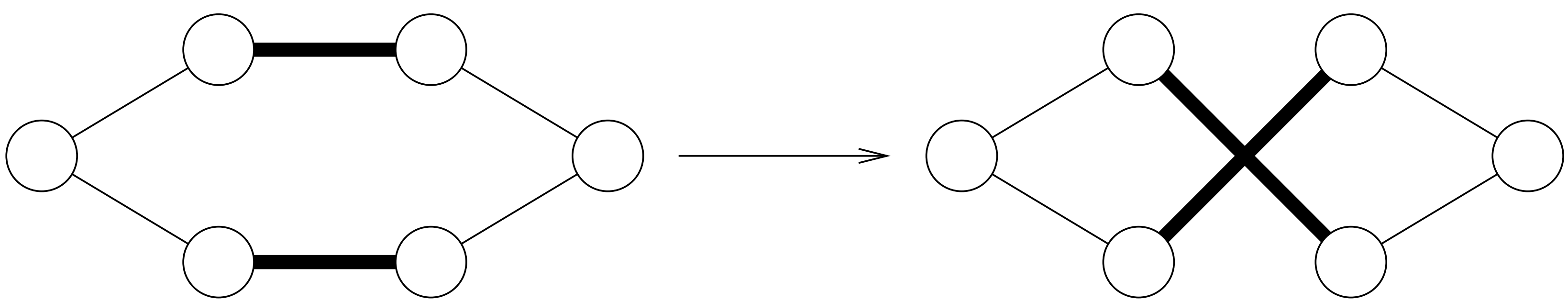
Complexidade: $\Theta(n^2)$.

▷ Algoritmo de busca local (problema de minimização):

- Encontrar uma solução inicial t.
- Encontrar t' em $N(t)$ com menor custo.
- Se o custo de t' é menor que o custo de t, fazer $t \leftarrow t'$ e repetir o passo anterior. Se não, retorne t e pare.

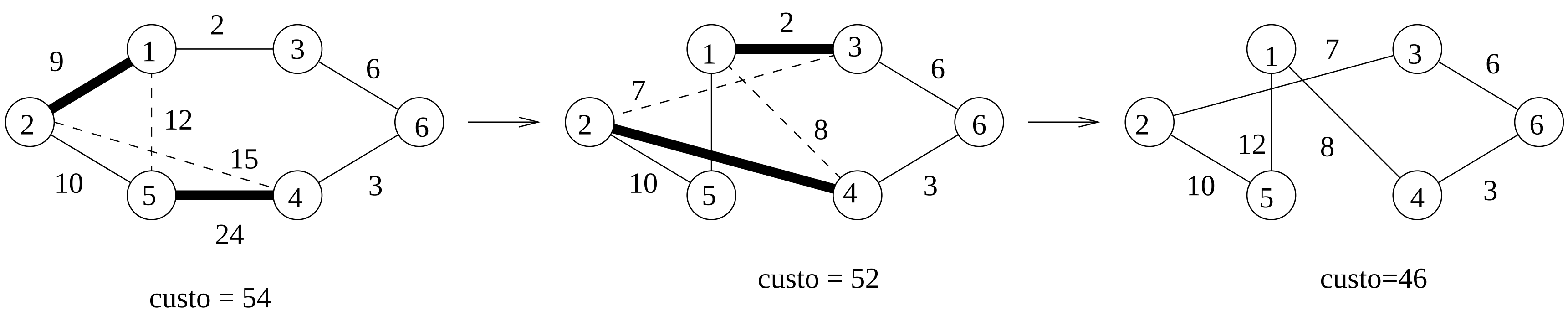
Heurísticas de Busca Local (TSP)

- ▷ Heurística da 2-troca para o TSP (Lin e Kernigham).
- ▷ Ciclo representado por uma permutação dos n vértices.
- ▷ Vizinhança: substituir pares de arestas.



- ▷ Complexidade: $\Theta(n^2)$.

Heurísticas de Busca Local (TSP)



- ▷ Tuplas: vetor de permutações de 1 até n.
- ▷ Vizinhança:
  inverte seqüência entre posições i e $j \pmod{n}$ $(j \geq i+2)$.
- ▷ No exemplo:
  $(1, 3, 6, \underline{4}, 5, 2, \underline{1}) \Longrightarrow (\underline{1}, 3, 6, 4, \underline{2}, 5, 1) \Longrightarrow (1, 4, 6, 3, 2, 5, 1)$

Heurísticas de Busca Local (Partição de Grafos)

- ▷ Entrada: grafo não orientado $G = (V, E)$, com $|V| = 2n$, e custos c_{ij} para toda aresta $(i, j) \in E$.
- ▷ Saída: um subconjunto $V' \subseteq V$, com $|V'| = n$ e que minimize o valor de $\sum_{i \in V'} \sum j \notin V' c_{ij}$.
- ▷ Solução representada por um vetor a de $2n$ com os valores de 1 até $2n$. Nas n primeiras posições estão os vértices de V' e nas n seguintes os vértices de $\overline{V'}$.
- ▷ Vizinhança: todas as trocas possíveis de pares de vértices $(a[i], a[j])$, onde $1 \leq i \leq n$ e $(n + 1) \leq j \leq 2n$.
- ▷ Complexidade: $\Theta(n^2)$.

Heurísticas de Busca Local (Partição de Grafos)

▷ Exemplo: grafo completo com 6 vértices (K_6).

$$c = \begin{bmatrix} - & 9 & 2 & 8 & 12 & 11 \\ 9 & - & 7 & 19 & 10 & 32 \\ 2 & 7 & - & 29 & 18 & 6 \\ 8 & 19 & 29 & - & 24 & 3 \\ 12 & 10 & 18 & 24 & - & 19 \\ 11 & 32 & 6 & 3 & 19 & - \end{bmatrix}$$

Solução inicial:
$a = \{1, 4, 6, 2, 3, 5\}$.

-

vizinhos	$(1,2)$	$(1,3)$	$(1,5)$	$(4,2)$	$(4,3)$	$(4,5)$	$(6,2)$	$(6,3)$	$(6,5)$
ganho	-29	-12	-7	$\underline{-66}$	-15	-40	-22	-43	-32

- Nova solução: $a = \{1, 2, 6, 4, 3, 5\}$.

-

vizinhos	$(1,4)$	$(1,3)$	$(1,5)$	$(2,4)$	$(2,3)$	$(2,5)$	$(6,4)$	$(6,3)$	$(6,5)$
ganho	37	34	23	66	51	26	44	59	54

- *Ótimo Local !*

Heurísticas de Busca Local

- Pode ser vantajoso que a busca local passe por soluções inviáveis !
- Nesses casos a função objetivo é composta de duas parcelas:

$$g(.) = f(.) + \alpha h(.),$$

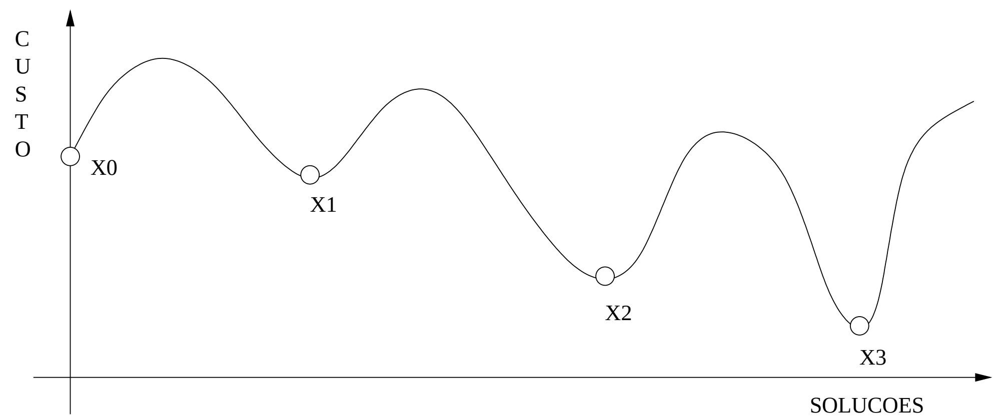
  onde f é função original, h é uma função que mede quão inviável é a solução e α é um fator de penalização.
- Exemplo: no problema da partição de grafos, considere a vizinhança onde só um vértice muda de V' para $\overline{V'}$ ou vice-versa.
- Penalizar as soluções inviáveis usando $\alpha > 0$ grande e definindo:

$$h(V', \overline{V'}) = ||V'| - |\overline{V'}||^2.$$

- Se acabar em uma solução inviável, aplicar um algoritmo guloso que rapidamente restaura a viabilidade.

Heurísticas de Busca Local

▷ Busca local retorna solução que é ótimo local.



▷ Escapando de ótimos locais: mover para melhor vizinho mesmo se o custo for pior.

▷ *Metaeurísticas*: Busca Tabu, Simulated Annealing, Algoritmos genéticos,etc.

Heurísticas de Busca Local (Busca Tabu)

- ▷ Inserir na busca local uma lista de movimentos **tabu** que impedem, por algumas iterações, que um determinado movimento seja realizado.

  Objetivo: evitar que uma solução seja revisitada.

  Exemplo: no caso da equipartição de grafos, pode-se impedir que a troca de dois vértices por t iterações.
- ▷ Repetir a busca local básica por α iterações ou se nenhuma melhora foi obtida nas últimas β iterações.
- ▷ Os parâmetros α e β são fixados *a priori*.
- ▷ Parâmetros a ajustar: tamanho da lista tabu t, α e β.

Heurísticas de Busca Local (*Simulated Annealing*)

▷ Baseado na equivalência entre o processo físico de formação de cristais e a otimização de um problema combinatório Π.

$$\begin{array}{lcl} \text{Solução de } \Pi & \equiv & \text{Estado da matéria} \\ \text{Custo da Solução} & \equiv & \text{Energia do Estado} \end{array}$$

Observação: *cristal* é um estado físico de energia mínima !

▷ Obtenção de cristais (Física): colocar matéria em alta temperatura e resfriá-la lentamente. No início, com temperatura alta, a matéria pode mudar para estados de mais alta *ou* mais baixa energia. O processo é caótico !

No final, com temperatura baixa, praticamente só é possível passar de um estado para outro que tenha energia menor. Esse processo se encerra quando o sistema está "*congelado*", ou seja, foi atingido um estado de *mínimo local de energia*.

Heurísticas de Busca Local (*Simulated Annealing*)

- ▷ Para simular esse processo, o algoritmo de busca local passa de uma solução para outra na sua vizinhança com uma probabilidade que é maior para as soluções de menor custo e que, após um número elevado de iterações, tende a zero para as soluções que provoquem aumento de custo.

- ▷ O comportamento do algoritmo assemelha-se àquele de uma busca aleatória no início e àquele de uma busca local determinística no final.

- ▷ Parâmetros a serem ajustados: A temperatura inicial do processo (T_0); a taxa de resfriamento ($\alpha \in (0,1)$), o número máximo de iterações (L) e a medida de "congelamento" (número de iterações sem melhoria).

Heurísticas de Busca Local (*Simulated Annealing*)

`Simulated-Annealing` (* problema de minimização *)
- $T \leftarrow T_0$; $S^* \leftarrow S$; custo$^* \leftarrow$ custo(S);
- $S \leftarrow$ `Gera-Solucao-Inicial`; congelado $\leftarrow$ Falso;
- **Enquanto** `NOT`(congelado) **faça**
  - **Para** $i = 1$ **até** L **faça**
    - $S' \leftarrow$ `Escolha-Vizinho-Aleatorio`(S);
    - $\delta \leftarrow$ custo(S') − custo(S);
    - **Se** $\delta \leq 0$ **então** $S \leftarrow S'$
    - **se não**
      - Fazer $S \leftarrow S'$ com probabilidade $e^{-(\delta/T)}$;
      - **Se** custo(S) < custo(S^*) **então**
        - $S^* \leftarrow S$ e custo$(S^*) \leftarrow$ custo(S);
    - **fim-se**
  - **fim-para**
  - $T \leftarrow \alpha T$; congelado $\leftarrow$ ($T <$ `TEMPERATURA-MINIMA`);
- **fim-enquanto**

Retornar S^*.

Tratamento de problemas $\mathcal{NP}$-difíceis: Aproximações

- *Algoritmos aproximados* encontram uma solução **com garantia de qualidade** em tempo polinomial.

- Nomenclatura:

P	problema $\mathcal{NP}$-difícil
H	algoritmo aproximado
I	instância de P
$z^*(I)$	valor ótimo da instância I
$z^H(I)$	valor da solução obtida por H para a instância I

- *Aproximação absoluta*: para algum $k \in \mathbb{Z}_+$ tem-se que

$$|z^*(I) - z^H(I)| \leq k, \text{ para todo } I.$$

Aproximação Absoluta

▷ Exemplo 1: alocação de arquivos em discos (MFA).

Dados n arquivos de tamanhos $\{\ell_1, \ldots, \ell_n\}$ e **dois** discos de capacidade L, qual o maior número de arquivos que podem ser armazenados nos discos ?

▷ *Teorema*: MFA $\in \mathcal{NP}$-completo. (**Exercício**)

▷ Algoritmo: supor que $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n$.

```
Aprox-MFA(n, ℓ);
    L' ← L;    j ← 1;
    Enquanto L' ≥ ℓj faça
        L' ← L' − ℓj;    Colocar(j, 1);    j + +;
    fim-enquanto;
    L' ← L;
    Enquanto L' ≥ ℓj faça
        L' ← L' − ℓj;    Colocar(j, 2);    j + +;
    fim-enquanto;
Retornar  j − 1.
```

Aproximação Absoluta

Teorema: $|z^*(I) - z^H(I)| \leq 1$.

<u>*Prova*</u>: Seja p o número de arquivos que o algoritmo `Aprox-MFA` consegue armazenar em um grande disco com capacidade $2L$. Além disso, seja $j = \text{argmax}\{\sum_{i=1}^{j} \ell_i \leq L\} \leq p$.

1. $p \geq z^*(I)$; (a)
2. $\sum_{i=1}^{p} \ell_i \leq 2L$; (b)
3. $\sum_{i=j+1}^{p-1} \ell_i \leq \sum_{i=j+2}^{p} \ell_i \leq L$, devido a (b) e à definição de j que implica que $\sum_{i=1}^{j+1} \ell_i > L$. (c)

$$\ldots \stackrel{(c)}{\Longrightarrow} z^H(I) \geq p - 1 \stackrel{(a)\wedge(c)}{\Longrightarrow} z^H(I) \geq z^*(I) - 1. \qquad \square$$

Aproximação Absoluta

- ▷ Exemplo 2: coloração de *grafos planares* (CGP).
- ▷ Resultados conhecidos:
  - ∘ CGP $\in \mathcal{NP}$-completo.
  - ∘ Todo grafo planar tem pelo menos um vértice de grau menor que 6.
  - ∘ Um *grafo é bipartido* se e somente se ele não tem ciclos ímpares.

`6-cores`(G); (* $G = (V, E)$ *)
    Se $|V| = 0$ **então Retornar** 0; **Se** $|E| = 0$ **então Retornar** 1;
    Se G é bipartido **então Retornar** 2;
    se não
        Escolher v com grau(v) ≤ 5; $G' \leftarrow G - v$; $k \leftarrow$ `6-cores`(G');
        Seja $x \in \{1, 2, 3, 4, 5, 6\}$ uma cor diferente daquela dos vizinhos de v;
        Se $(x > k)$ **então** $k \leftarrow k + 1$; $x \leftarrow k$; **fim-se**;
        cor$[v] \leftarrow x$;
    fim-se
Retornar k.

Aproximação Absoluta

▷ **Teorema**: $|z^*(I) - z^H(I)| \leq 3$.

Prova: Se $|V| = 0$, $|E| = 0$ ou o grafo é bipartido então a coloração feita por `6-cores` é ótima e o resultado é imediato.

Caso contrário, G tem pelo menos um *ciclo ímpar*. Logo qualquer coloração precisará de pelo menos três cores. *(justifique !)*

Como o número de cores usadas por `6-cores` é ≤ 6 e a solução ótima requer pelo menos 3 cores, tem-se que

$$|z^*(I) - z^H(I)| \leq |3 - 6| = 3.$$

$\square$

▷ **Observações**:

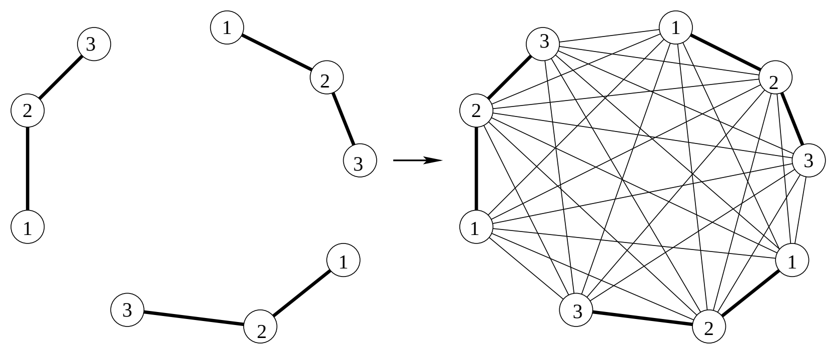
- Todo grafo planar admite uma 4-coloração.
- São poucos problemas que tem aproximação absoluta.

Aproximação Absoluta $\times$ Questão $\mathcal{P} = \mathcal{NP}$

▷ **Teorema**: Não existe uma aproximação absoluta para `CLIQUE` com complexidade polinomial a menos que $\mathcal{P} = \mathcal{NP}$.

Prova: Suponha que $\mathcal{P} \neq \mathcal{NP}$ e que existe um algoritmo polinomial H para `CLIQUE` tal que $|z^*(I) - z^H(I)| \leq k \in \mathbb{Z}_+$.

Seja G^{k+1} o grafo composto de $k+1$ cópias de G mais todas as arestas ligando pares de vértices em diferentes cópias.



Observação: se α é o tamanho da maior clique de G então a maior clique de G^{k+1} tem $\alpha(k+1)$ vértices.

Aproximação Absoluta $\times$ Questão $\mathcal{P} = \mathcal{NP}$

▷ *Prova*: (cont.)

Executando-se H para G^{k+1} tem-se que

$$z^*(G^{k+1}) - z^H(G^{k+1}) \leq k \Longrightarrow z^H(G^{k+1}) \geq (k+1)z^*(G) - k.$$

Se C é a clique encontrada por H em G^{k+1}, existe uma cópia de G tal que $C' = V \cap C$ e $|C'| \geq |C|/(k+1)$. Logo

$$|C'| \geq \frac{(k+1)z^*(G) - k}{k+1} = z^*(G) - \frac{k}{k+1}.$$

Portanto, $|C'| \geq z^*(G)$, ou seja C' é uma clique máxima de G.

Absurdo ! $\square$

α-Aproximação

▷ Um algoritmo H para um problema P é uma *α-aproximação* se

- P é um problema de **minimização** e $\dfrac{z^H(I)}{z^*(I)} \leq \alpha \;\; \forall\, I$,

ou

- P é um problema de **maximização** e $\dfrac{z^*(I)}{z^H(I)} \leq \alpha \;\; \forall\, I$.

Observação: α é sempre maior ou igual a 1.

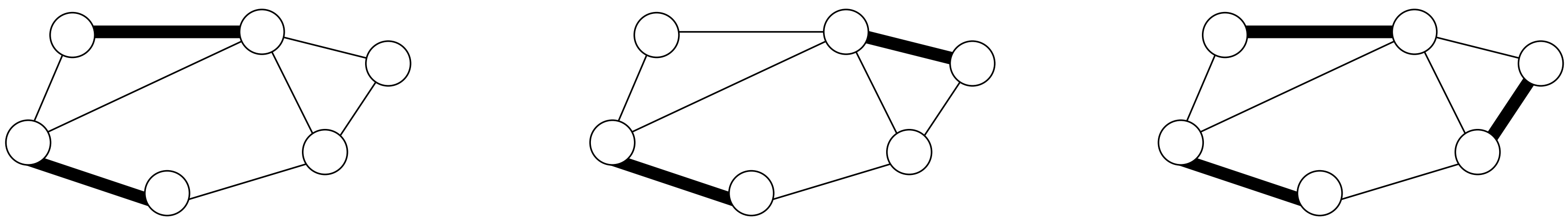
▷ Um algoritmo H é uma *α-aproximação relativa* para um problema P se

$$\left| \frac{z^*(I) - z^H(I)}{z^*(I)} \right| \leq \alpha, \text{ para todo } I.$$

α-**Aproximação**

▷ Exemplo 1: Cobertura mínima de vértices (CV).

▷ *Definições*: emparelhamento em grafos.



MAXIMAL MAXIMO

▷ Algoritmo:

`CV-2-Aprox`(G); (* $G = (V, E)$ *)
  $C \leftarrow \{\}$;
  Construir um emparelhamento maximal M^* em G;
  Para todo $(u, v) \in M^*$ **faça** $C \leftarrow C \cup \{u, v\}$;
Retornar C.

α-Aproximação

▷ **Teorema**: $\frac{z^H(I)}{z^*(I)} \leq 2$.

Prova:

Parte I: C é uma cobertura de vértices pois, se existisse uma aresta (u, v) não coberta então M^* não seria maximal.

Parte II: $|C| \leq 2z^*(I)$.

Se C' e M' são respectivamente uma cobertura e um emparelhamento qualquer de G então $|C'| \geq |M'|$. Logo:

$$z^*(I) \geq |M^*| = \frac{|C|}{2} = \frac{z^H(I)}{2}.$$

$\square$

α-Aproximação

▷ Exemplo 2: *bin packing* unidimensional.

Dados n arquivos de tamanhos $\{t_1, \ldots, t_n\}$ e disketes de capacidade de armazenamento C, qual o menor número de disketes necessários para fazer o *backup* de todos os arquivos ?

Observação: supor que $t_i \leq C$ para todo $i = 1, \ldots, n$.

▷ Algoritmo básico:

```
Bin-Aprox(t, n, C);
    Preprocessamento(t, n);    Disketes-em-uso ← {};    k ← 0;
    Para i = 1 até n faça
        j ← Escolher-diskette(Disketes-em-uso, i);
        Se j = 0 então    (* arquivo não cabe nos disketes em uso *)
            k++;    Disketes-em-uso ← Disketes-em-uso ∪ {k};    j ← k;
        fim-se
        Armazenar(i, j);
    fim-para
Retornar k.
```

α-Aproximação

▷ Descrição dos procedimentos do algoritmo `Bin-Aprox`:

- `Preprocessamento`: retorna uma nova permutação dos arquivos.
- `Escolher-diskete`: retorna o número do diskete em uso onde será armazenado o arquivo i ou *zero* caso não encontre diskete com capacidade residual de armazenamento suficiente.
- `Armazenar`: registra que o arquivo i será alocado ao j-ésimo diskete, atualizando a sua capacidade residual de armazenamento.

α-Aproximação

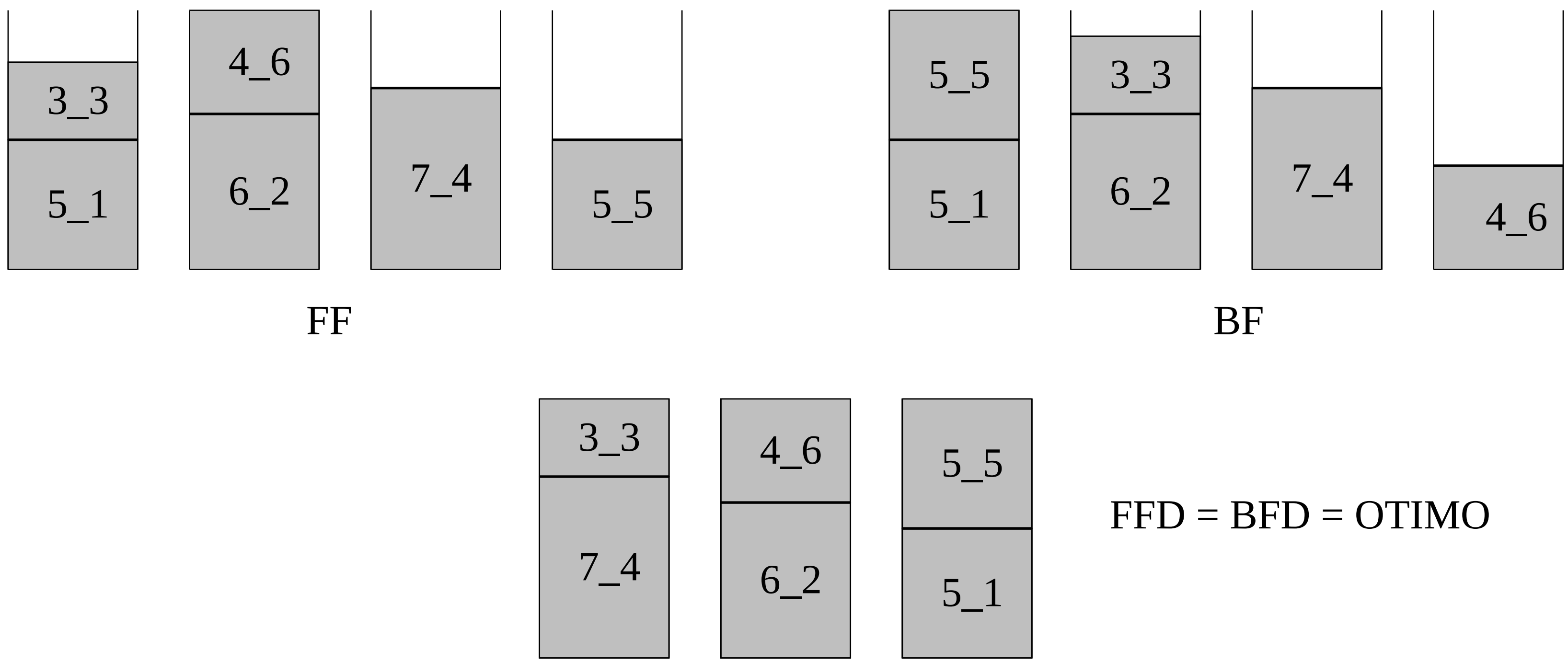
▷ Estratégias alternativas:

- *First Fit* (FF): `Preprocessamento` mantém ordem dos arquivos de entrada e `Escolher-diskete` procura o diskete em uso de *menor índice* aonde cabe o arquivo corrente.
- *Best Fit* (BF): `Preprocessamento` mantém ordem dos arquivos de entrada e `Escolher-diskete` procura o diskete em uso de *menor capacidade residual de armazenamento* aonde cabe o arquivo corrente.
- *First Fit Decrease* (FFD): variante do algoritmo FF onde o `Preprocessamento` ordena os arquivos em ordem decrescente de tamanho.
- *Best Fit Decrease* (BFD): variante do algoritmo BF onde o `Preprocessamento` ordena os arquivos em ordem decrescente de tamanho.

α-Aproximação

▷ Exemplo de aplicação dos algoritmos para *bin packing*:

$C = 10$, $n = 6$, $t = \{5_1, 6_2, 3_3, 7_4, 5_5, 4_6\}$ (notação: $i_j \Longrightarrow t_j = i$).



α-Aproximação

▷ **Teorema**: FF é um algoritmo 2-aproximado para *bin packing*.

Prova: seja b o valor retornado por FF e b^* o valor ótimo.

Suponha que os disketes estão ordenados decrescentemente pela sua capacidade residual. Note que a capacidade residual dos $b-1$ primeiros disketes da solução de FF é $\leq C/2$. Caso contrário, se dois disketes tivessem capacidade residual $\geq C/2$ os seus arquivos teriam sido armazenados em um único diskete. Como o total armazenado no diskete b é maior que a capacidade residual dos demais disketes, tem-se

$$S = \sum_{i=1}^{n} t_i \geq b\,\frac{C}{2}.$$

Como $b^* \geq \lceil \frac{S}{C} \rceil \geq \frac{S}{C}$, a equação acima implica que $b^* \geq \frac{1}{2}\, b$. $\square$

α-Aproximação

▷ **Teorema**: para toda instância I do *bin packing* tem-se que

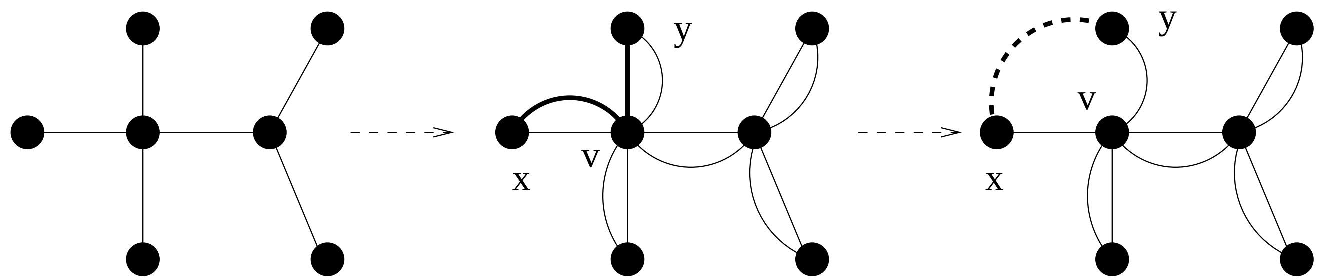
$$z^{xx}(I) \leq \frac{17}{10}\, z^*(I) + 2 \quad \text{e} \quad z^{xxD}(I) \leq \frac{11}{9}\, z^*(I) + 2,$$

onde $xx \in \{FF, BF\}$.

▷ Exemplo 3: 2-aproximação para o TSP-*métrico*, ou seja, quando as distâncias obedecem à *desigualdade triangular*.

`TSP-Aprox`(G); (* $G = (V, E)$ e completo *)
  Construir T, uma árvore geradora minima de G;
  Construir o grafo C duplicando-se todas as arestas de T;
  Enquanto houver vértices de grau ≥ 2 em C **faça**
    $v \leftarrow$ vértice de grau ≥ 2;
    Encontre x e y distintos com (x, v) e (y, v) em C;
    Faça $C \leftarrow (C \cup (x, y)) - \{(x, v), (y, v)\}$; (+)
  fim-enquanto
Retorne C;

α-Aproximação



▷ **Teorema**: `TSP-Aprox` é uma 2-aproximação para o TSP-*métrico*.

<u>*Prova*</u>: se z^* é o custo mínimo de um ciclo hamiltoniano em G,

$$\text{custo}(T) \leq z^* \Rightarrow 2\ \text{custo}(T) \leq 2z^*.$$

Por outro lado, devido aos custos obedecerem à desigualdade triangular, o comando (+) só pode diminuir o custo de C ao longo das iterações. Logo

$$\text{custo}(C) \leq 2\ \text{custo}(T) \leq 2z^*. \qquad \square$$

α-Aproximação $\times$ Questão $\mathcal{P} = \mathcal{NP}$

▷ **Teorema**: Não existe uma α-aproximação para `TSP` (genérico) com complexidade polinomial a menos que $\mathcal{P} = \mathcal{NP}$.

Prova: Suponha que $\mathcal{P} \neq \mathcal{NP}$ e que existe um algoritmo polinomial H tal que $\frac{z^H(I)}{z^*(I)} \leq \alpha \in \mathbb{Z}_+$.

Seja G o grafo dado como entrada do problema de decisão do ciclo hamiltoniano (HAM). Construa o grafo G' completando com as arestas que faltam. Atribua custo um às arestas originais e custo αn àquelas que foram inseridas no passo anterior.

Se G tem um ciclo hamiltoniano, então o valor ótimo do TSP é $z^*(G) = n$. Como H é α-aproximado para o TSP

$$\frac{z^H(G)}{z^*(G)} \leq \alpha \;\Rightarrow\; z^H(G) \leq \alpha\, n.$$

α-**Aproximação** $\times$ **Questão** $\mathcal{P} = \mathcal{NP}$

▷ <u>*Prova*</u> (cont.):

Assim, quando G tem um ciclo hamiltoniano, o ciclo encontrado por H para o TSP só terá arestas originais de G !

Por outro lado, se G não tem ciclo hamiltoniano, $z^H(G) \geq 1 + \alpha\, n$.

Portanto, G tem um ciclo hamiltoniano se somente se $z^H(G) \leq \alpha\, n$, ou seja, H resolve HAM em tempo polinomial.

▷ Absurdo, já que, por hipótese, $\mathcal{P} \neq \mathcal{NP}$. $\square$