Reduções entre problemas

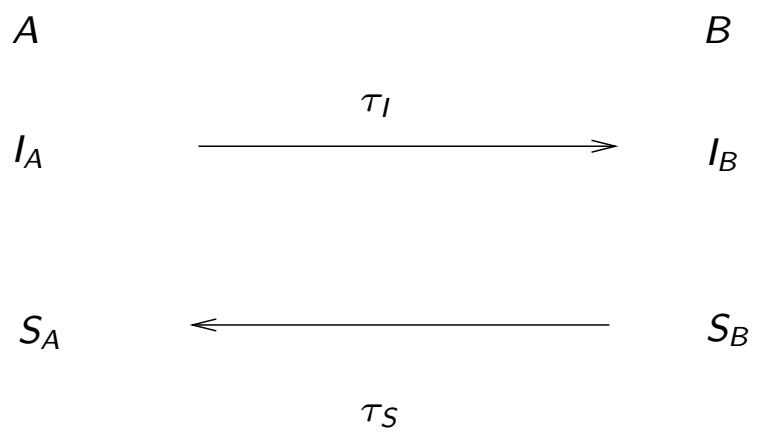
▷ Ideia básica da **redução de Turing**:

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



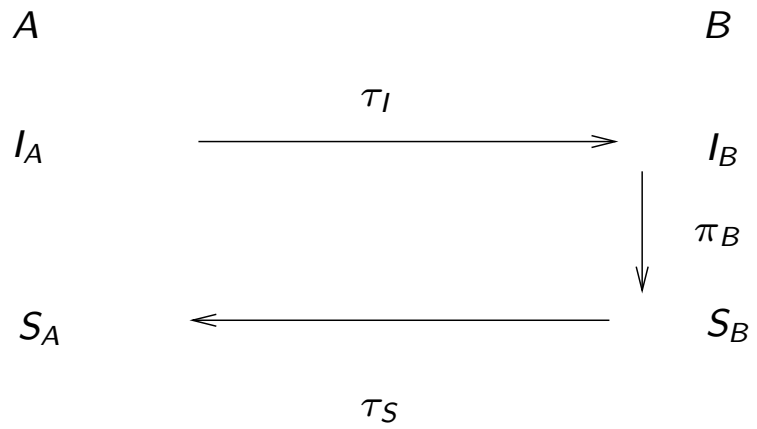
▷ Quando usar **reduções** ?

- **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.

Formalizando ...

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

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



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

$\triangleright$ **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 (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

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

  Portanto, τ_I é $O(n^2)$.

- A solução do MMS é dada por:

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

  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.

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

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

---

†: propriedade atendida por funções "suaves" (p.ex., qualquer polinômio).

Classes de Problemas

- 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.

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

- Considere o problema de satisfazer uma fórmula lógica $\mathcal{F}$ 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: $\mathcal{F} = C_1.C_2.\ \ldots.C_m$.

Classes de Problemas (cont.)

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

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

  Se $x_1 = 1$ e $x_2 = x_3 = 0$ tem-se que $\mathcal{F} = 1$. 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.)

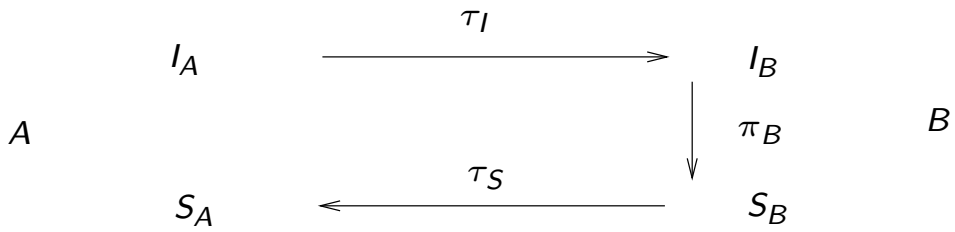
- ▷ **Exercício:** Dada uma atribuição de valores para as variáveis, descreva um algoritmo **polinomial** que confirma se $\mathcal{F}$ é verdadeira ou falsa para esta atribuição.
- ▷ Não se conhece algoritmo eficiente para SAT !
- ▷ Característica do problema SAT comum a diversos problemas encontrados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

- **Ideia**: 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



- ▷ 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 uma única vez. 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.

Classes de Problemas (cont.)

- Voltemos ao exemplo do problema SAT.
- O SAT é um problema de decisão.
- Vimos que não é fácil achar um algoritmo polinomial que resolve SAT.
- Por outro lado, dada uma proposta de solução para SAT, existe um algoritmo polinomial que verifica se essa solução de fato responde o problema.
- Além do SAT, inúmeros outros problemas compartilham dessa mesma propriedade !

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 não-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 nenhum algoritmo determinístico polinomial pode resolver A.
- Como mostrar que a proposição é verdadeira ?
  Mostrar que para todo 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 $\mathcal{NP}$-completo se
  1. $A \in \mathcal{NP}$ 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

- 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.
- 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 $\mathcal{F} = C_1.C_2.\ldots.C_c$ 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 $\mathcal{F}$;
  - 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.)

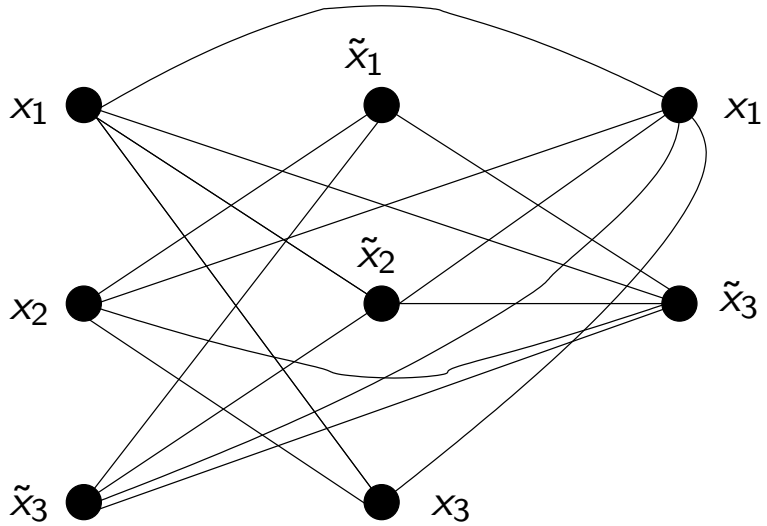
- O número de vértices de G é $O(c.v)$ enquanto o número de arestas é $O(c^2v^2)$. Fazendo-se $k = c$, teremos construído uma instância de CLIQUE em tempo polinomial no tamanho da entrada de SAT.
- É fácil mostrar que a fórmula $\mathcal{F}$ é satisfeita por alguma atribuição de variáveis se e somente se o grafo c-partido G tem uma clique de tamanho c.

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

- Exemplo da redução: seja

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

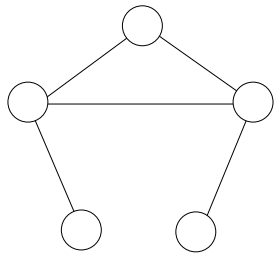
O grafo correspondente à instância de CLIQUE é dado por:

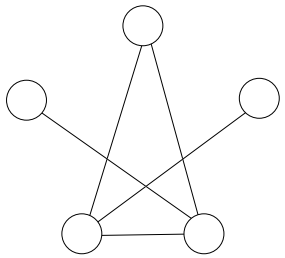


Provas de $\mathcal{NP}$-completude: CV (cont.)

- ▷ *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 $G \iff \overline{U} = V - U$ 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. □

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.

□