Teoria da Complexidade

Cid C. de Souza / IC–UNICAMP

Universidade Estadual de Campinas
Instituto de Computação

2º semestre de 2010

Autor

Prof. Cid Carvalho de Souza
Universidade Estadual de Campinas (UNICAMP)
Instituto de Computação
Av. Albert Einstein nº 1251
Cidade Universitária Zeferino Vaz
13083-852, Campinas, SP, Brasil
Email: cid@ic.unicamp.br

Direitos autorais

Reduções entre problemas

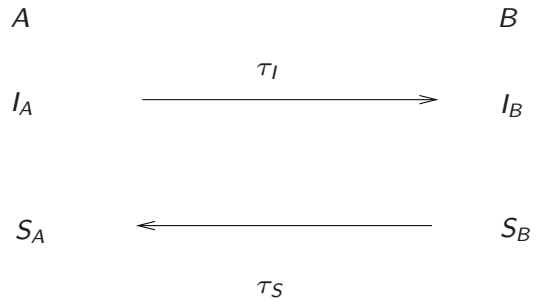
▷ Idéia básica da **redução de Turing**:

Problema A:	Problema B:
• Instância de entrada: I_A;	• 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**:



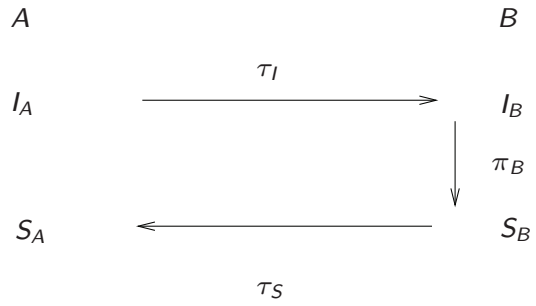
▷ Quando usar **reduções** ?

- **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 $\mathcal{I}_\bullet$;

▷ **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 $O(f(n))$.

▷ **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:

$$A = a_0 a_1 \ldots a_{n-1} \text{ e } B = b_0 b_1 \ldots b_{n-1}.$$

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:
$A = a_0 a_1 \ldots a_{n-1}$ e $B = b_0 b_1 \ldots b_{m-1}$, 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:

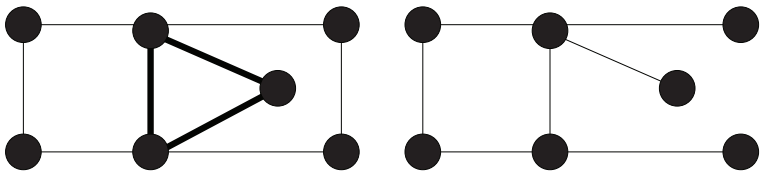
- $I_{CSM} = (acgtact, gtactac, 6)$;
- $I_{SM} = (acgtactacgtact, 12, gtactac, 6)$;
- $S_{SM} = S_{CSM} = \{k = 2\}$.

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:



(e) Com $\triangle$. (f) Sem $\triangle$.

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 $a_{ij}^2 = \sum_{k=1}^{n} a_{ik}.a_{kj}$. Logo:

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

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

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

◇ **Redução:** PET $\propto_{n^2}$ MM

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

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

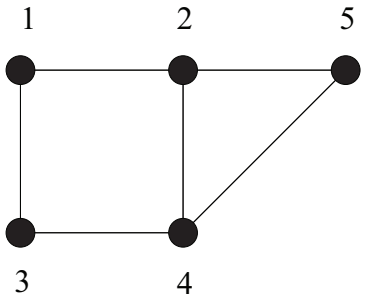
Portanto, τ_I é $O(n^2)$.

- 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

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

  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).

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

- 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.)

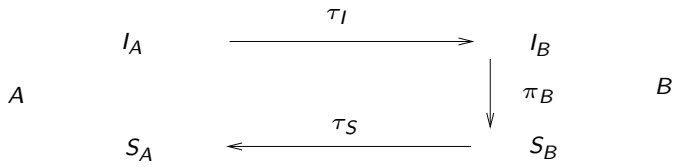
- ▷ **Exercício:** Dada uma atribuição de valores paras as variáveis, descreva um algoritmo **polinomial** que confirma se $\mathcal{F}$ é verdadeira ou falsa para esta atribuição.
- ▷ Não se conhece algoritmo eficiente para SAT !
- ▷ Característica do problema SAT comum a diversos problemas encontrados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

- ▷ **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



- ▷ 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 !
- Vamos introduzir um *novo modelo de computação* que nos ajudará a identificá-los.

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: A primeira fase, pode usar o comando não-determinístico **Escolha**, e **constrói** uma proposta de solução. A segunda fase, *só usa comandos determinísticos*, e **verifica** 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 ← Escolha(1, ..., n);
    (* Fase de verificação *)
    Se A[j] = x então retornar Aceitar;
    se não retornar Rejeitar;
```

▷ Qual a complexidade deste algoritmo ?

Algoritmos não-determinísticos (cont.)

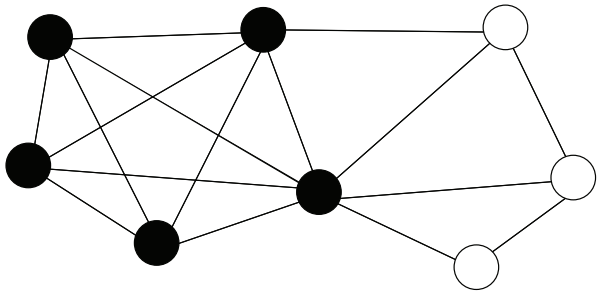
▷ **Definição:** *a complexidade de um algoritmo não-determinístico executado sobre uma instância qualquer é o número mínimo de passos necessários para que ele retorne* `Aceitar` *caso exista uma seqüência de* **Escolha**s *que leve a essa conclusão. Se o algoritmo retornar* `Rejeitar` *o seu tempo de 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.



Algoritmos não-determinísticos (cont.)

- Um algoritmo não-determinístico para CLIQUE seria:

```
CliqueND(G,n,k);
    (* Fase de construção *)
    S ← V;
    C ← {}; (* vértices da clique proposta *)
    Para i = 1 até k faça
        u ← Escolha(S);
        S ← S − {u};
        C ← C ∪ {u};
    fim-para
    (* Fase de verificação *)
    Para todo par de vértices distintos (u, v) em C faça
        Se (u, v) ∉ 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`.

Simulando máquinas não-determinísticas (cont.)

▷ Um outro algoritmo não-determinístico para CLIQUE:

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


Simulando máquinas não-determinísticas (cont.)

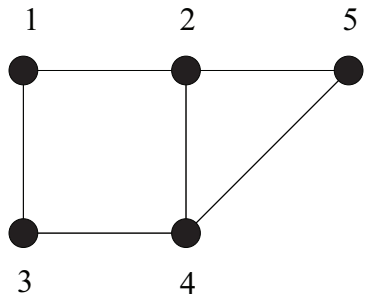


Figura: determinar se há uma CLIQUE de tamanho 3

▷ 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.)

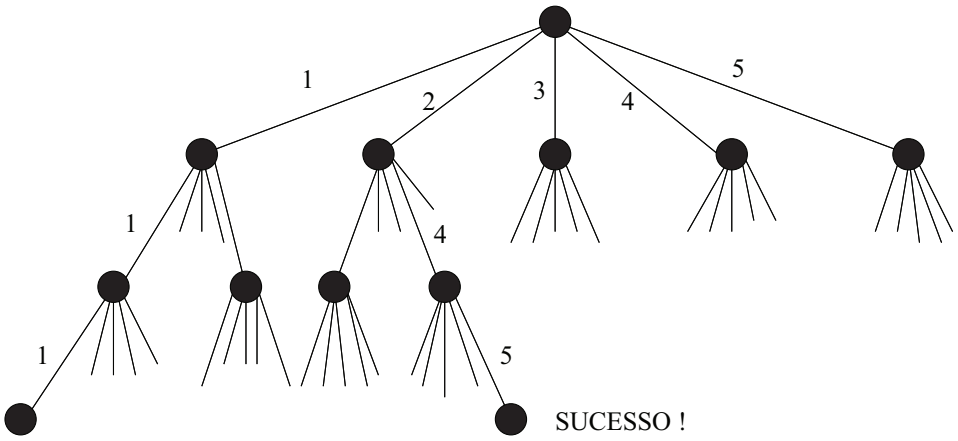


Figura: Fluxo de execução de CliqueND2.

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 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

- ▷ *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 !

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.

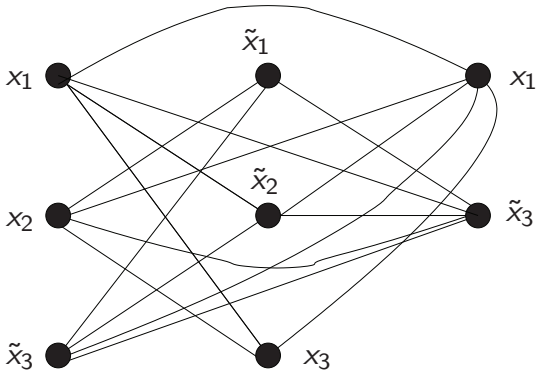
▷ É fácil mostrar que a fórmula $\mathcal{F}$ é satisfeita por alguma atribuição de variáveis se e somente se o grafo c-partido G tem uma clique de tamanho c.

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

▷ Exemplo da redução: seja

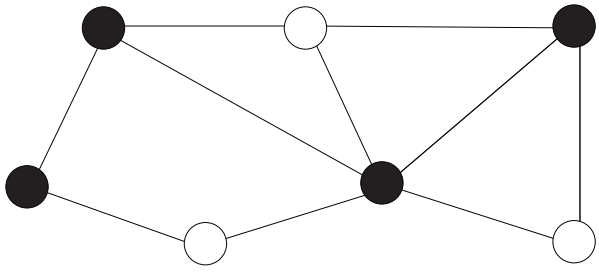
$$\mathcal{F} = (x_1 + x_2 + \overline{x}_3).(\overline{x}_1 + \overline{x}_2 + x_3).(x_1 + \overline{x}_3).$$

O grafo correspondente à instância de CLIQUE é dado por:



Provas de $\mathcal{NP}$-completude: 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 $G \iff$
$\overline{U} = V - U$ é uma cobertura de vértices de tamanho $n - k$ em $\overline{G}$.

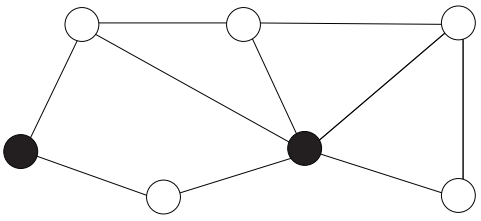
◇ Portanto, $\overline{G}$ tem uma cobertura de tamanho $\ell = n - k$ se e somente se G tem uma clique de tamanho k. □

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

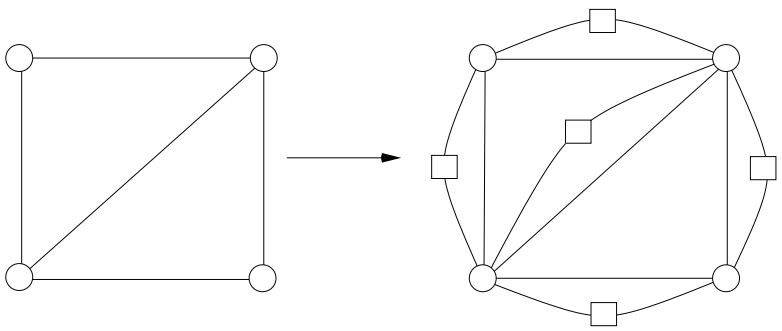
1. DS está $\mathcal{NP}$. *(Exercício !)*
2. CV $\propto_{\text{poli}}$ DS

   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 anteriormente 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 !)
- ▷ Os dois resultados anteriores completam a demonstração. □

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 $\mathcal{F}$ (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 $\mathcal{F}$ 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 $\mathcal{F}$ de SAT em uma instância $\mathcal{F}_3$ de 3SAT:

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

- se $k = 3$, coloque C em $\mathcal{F}_3$.
- se $k = 2$, coloque a cláusula

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

em $\mathcal{F}_3$, 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 $\mathcal{F}_3$, 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 $\mathcal{F}_3$, criando assim $k - 3$ novas variáveis.

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 $\mathcal{F}$ tem m cláusulas e n variáveis, $\mathcal{F}_3$ terá $O(nm)$ cláusulas e variáveis.
- Por construção, $\mathcal{F}$ é SAT se e somente se $\mathcal{F}_3$ 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.