Teoria da Complexidade

Cid C. de Souza / IC–UNICAMP

Universidade Estadual de Campinas
Instituto de Computação

1^{o} semestre de 2012

Revisado por Zanoni Dias

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

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 <u>verifica</u>** 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

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

- 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:

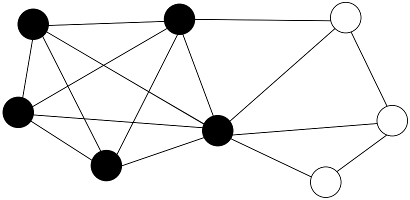
  ```
  BuscaND(A,x);
      (* Fase de construção *)
      j ← Escolha(1, ..., n);
      (* Fase de verificação *)
      Se A[j] = x então retornar Aceitar;
      se não retornar Rejeitar;
  ```

- 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* **Escolhas** *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 $\Omega(n)$!

- 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 *)
  $S \leftarrow V$;
  $C \leftarrow \{\}$; (* vértices da clique proposta *)
  Para $i = 1$ **até** k **faça**
    $u \leftarrow \text{Escolha}(S)$;
    $S \leftarrow S - \{u\}$;
    $C \leftarrow C \cup \{u\}$;
  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`.

Simulando máquinas não-determinísticas (cont.)

▷ 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**
    $u \leftarrow \text{Escolha}(V)$;
    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.)

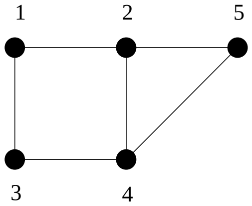


Figura: determinar se há uma CLIQUE de tamanho 3

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

Simulando máquinas não-determinísticas (cont.)

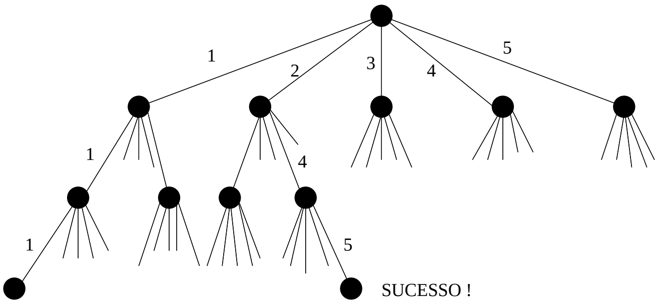


Figura: Fluxo de execução de CliqueND2.

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 <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. $A \in \mathcal{NP}$ <u>e</u>
  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!

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.

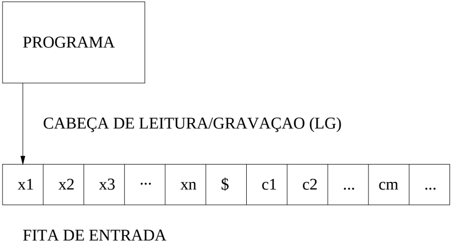
- Portanto, a fase de construção pode ser vista como uma seqüência de *escolhas* não-determinísticas que vai codificando, posição a posição, a cadeia que representa $c(x)$. Cada uma destas *escolhas* é feita sobre o alfabeto que descreve o *sistema de codificação*.

- 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 $\mathcal{F}$ de tamanho polinomial no tamanho da entrada da **Máquina de Turing** tal que $\mathcal{F}$ 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* $\mathcal{F}$ 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.

- 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 $\mathcal{F}$ nas variáveis anteriores da forma:

$$\mathcal{F}(z, y) = U(z, y).S(z, y).W(z, y).E(z, y)$$

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.

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

- 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,

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

- 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:

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

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 $\mathcal{F}$: $O(p^3(|x|) \log \; p(|x|))$.
- $\mathcal{F}(z,y)$ 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.

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.
- $\boxed{\text{Lembre-se:}}$

  Para provar que um problema A está em $\mathcal{NP}$-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.

- 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, $i \in \{1, \ldots, t\}$.

- 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.)

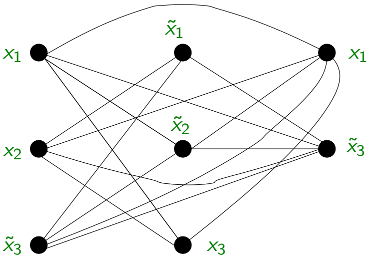
- O número de vértices de G é $O(c.v)$ enquanto o número de arestas é $O(c^2v^2)$. Fazendo-se $k = c$, teremos construído uma instância de CLIQUE em tempo polinomial no tamanho da entrada de SAT.
- É fácil mostrar que a fórmula $\mathcal{F}$ é satisfeita por alguma atribuição de variáveis se e somente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

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

- Exemplo da redução: seja

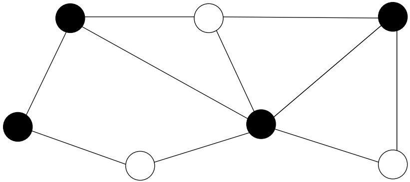
$$\mathcal{F} = (x_1 + x_2 + \overline{x}_3).(\overline{x}_1 + \overline{x}_2 + x_3).(x_1 + \overline{x}_3).$$

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.

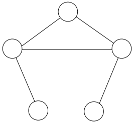


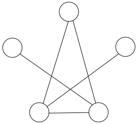
- CV: dado um grafo não-orientado $G = (V, E)$ e um valor inteiro $\ell \in \{1, \ldots, n\}$, onde $n = |V|$, pergunta-se: G possui uma *cobertura* com ℓ vértices?

▷ *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.)

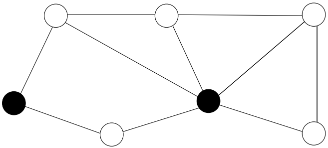
- Transformando uma instância CLIQUE em uma instância CV: Seja $G = (V, E)$ o grafo dado na entrada de CLIQUE e k o tamanho da clique procurada. A instância de CV será o grafo complementar $\overline{G}$ e o parâmetro ℓ é dado por $n - k$, onde $n = |V|$.
- A instância de entrada de CV é construída em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

- **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.

- **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 $k \in \{1, \dots, n\}$, onde $n = |V|$, pergunta-se: G possui um *conjunto dominante* com k vértices?

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

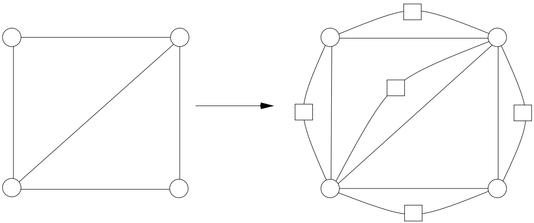
▷ *Teorema*: DS $\in \mathcal{NP}$-completo.

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.

- 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. $\square$

- **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:

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

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

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

Prova do Lema:

($\Rightarrow$): existe um i para o qual $x_i = 1$. Se fizermos $y_j = 1$ para todo $j = 1, \dots, i-2$ e $y_j = 0$ para todo $j = i-1, \dots, k-3$, 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

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

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. $\square$

- **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?

- **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?

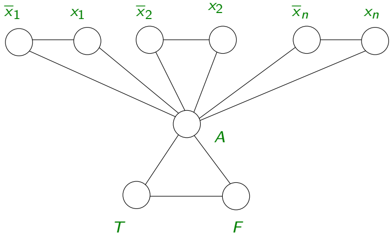
Mais provas de $\mathcal{NP}$-completude:

- **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

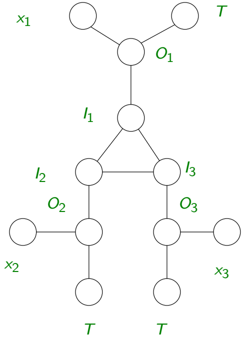
- **Definição**: Uma **coloração** de um grafo é uma atribuição de cores aos seus vértices tal que dois vértices adjacentes tenham cores distintas.
- Um grafo é k colorável se é possível colori-lo com k cores.
- **Problema da 3-coloração** (3COL): dado um grafo $G = (V, E)$, deseja-se saber se G pode ser colorido com 3 cores.
- *Teorema*: 3COL $\in \mathcal{NP}$-completo.
  1. 3COL está em $\mathcal{NP}$.
  2. 3SAT $\propto_{\text{poli}}$ 3COL

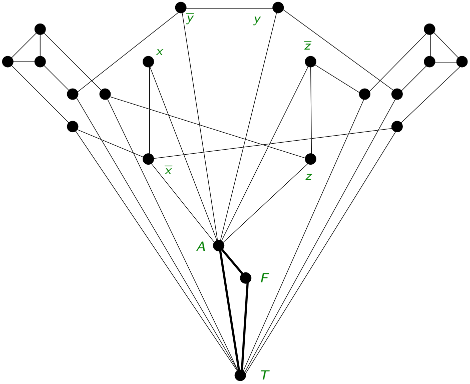
- Estrutura central:



- Estrutura da cláusula:

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





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

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

- **Problema dos K subconjuntos distintos** (KSD): 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 $\sum_{c_i \in S_j} c_i = L$ para todo $j = 1, \ldots, K$?
- KSD está em $\mathcal{NP}$? Pouco provável que exista um certificado conciso (polinomial), já que $1 \leq K \leq 2^n$.
- KSD pertence a NP-Difícil.
- PAR $\propto_{\text{poli}}$ KSD.

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 $X \in \mathcal{NP}$-completo 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 $f(n) \geq \log n$, $\mathcal{NSPACE}(f(n)) \subseteq \mathcal{PSPACE}(f(n)^2)$.

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

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 $< P >$ para com a entrada x e retorna `NÃO` caso contrário.
  Observação: esta subrotina seria capaz de testar se um programa qualquer entraria em *loop infinito*.
- Usando H, posso escrever o programa D representado a seguir cujo objetivo é decidir se um programa P para quando a sua própria codificação for passada na entrada.

O problema da Parada

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

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

Analisando:

$$D(<D>) \begin{cases} \text{para,} & \text{se } H(<D>,<D>) \text{ retornar } \texttt{NÃO} \\ \text{não para,} & \text{se } H(<D>,<D>) \text{ retornar } \texttt{SIM} \end{cases}$$

- O que está errado? O que podemos concluir?