

As classes P, NP e co-NP e o conceito de Completude

Por que não encontramos algoritmos polinomiais para
muitos problemas?

Talvez não tenhamos AINDA encontrado ou talvez
eles sejam MESMO intrinsecamente difíceis

Introdução

- **Objetivos:**


Apresentar o conceito de NP-completude e de reduções

--- nossa ferramenta principal de comparação da dificuldade entre problemas.

- **Tópicos:**

- Algoritmo Exponencial versus Polinomial
- Problemas de Decisão
- As classes P, NP, co-NP e suas relações
- Exemplos de problemas (SAT, principalmente)
- Algoritmos Não-determinísticos
- Completude
- Redução Polinomial
- Estrutura de Prova de alguns problemas NP-completos

Algoritmos Polinomiais e Exponenciais

- Algoritmos **Exponenciais** são, em geral, simples variações de pesquisa exaustiva no espaço de soluções (força bruta). 
- Algoritmos **Polinomiais** são obtidos através de um entendimento mais profundo da estrutura do problema. Vejam como exemplo a descoberta em agosto de 2002 de um algoritmo polinomial para o **Problema de Verificar se um número é Primo**.
- Um problema é considerado **intratável** se não existe um algoritmo polinomial para resolvê-lo.
- Um problema é considerado **bem resolvido/tratável** se existe um algoritmo polinomial para o problema. Tais problemas são considerados eficientes.

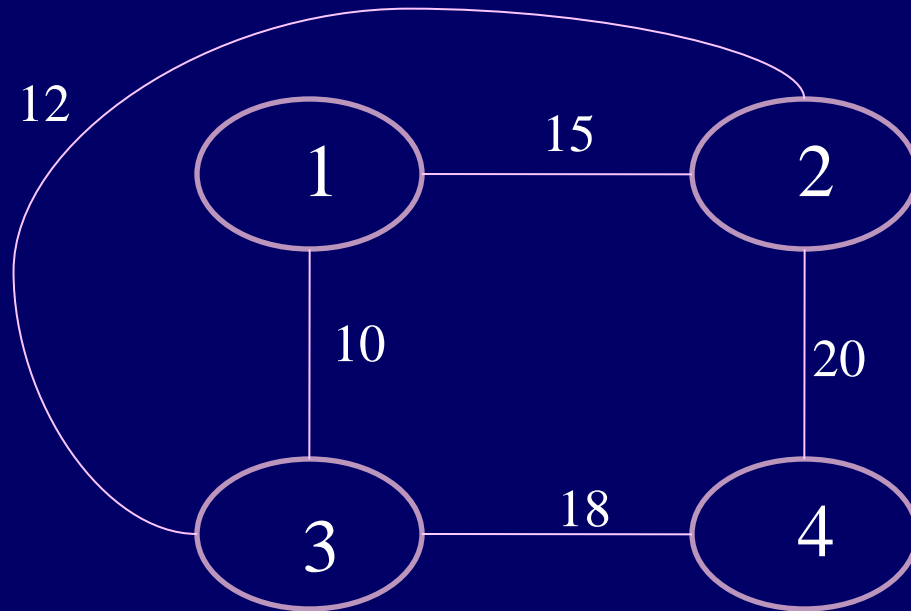


Ciclo Hamiltoniano

- Dado um grafo não dirigido $G = (V, E)$, um ciclo hamiltoniano é um ciclo simples que contém todos os vértices de G .
 - Algoritmo Ingênuo: Listar todas as permutações de vértices de G e então checar cada permutação para ver se ela é um ciclo hamiltoniano.
 - Qual é o tempo de execução deste algoritmo?

A *cycle* is a path where the last vertex is adjacent to the first. A cycle in which no vertex is repeated is said to be a *simple cycle*.

A simple cycle that passes through every vertex is said to be a *Hamiltonian cycle*.



Solução (única) para o grafo acima: o ciclo (1, 2, 4, 3, 1). O peso total desse ciclo é $15+20+18+10 = 63$.

Assim, se $W \geq 63$, a resposta é “sim”, caso contrário, é “não”.

Para grafos com 4 nós, no máximo há 2 soluções possíveis (*verifique*). Em grafos de m nós, o número de ciclos distintos cresce como $O(m!)$, que é maior que 2^{cm} para qualquer constante c .

Tempo de Execução

- Se usarmos a matriz de adjacências como uma codificação do grafo, o número m de vértices no grafo é $\Omega(\text{raiz}(n))$ onde $n = |\langle G \rangle|$ é o tamanho da codificação de G .
- Existem $m!$ **permutações** de vértices e o tempo de execução é $\Omega(m!) = \Omega(\text{raiz}(n)!) = \Omega(2^{\text{raiz}(n)})$ que não é polinomial.

$$2^n \leq n! \leq n^n \text{ para } n \geq 4$$

Notação Ω - ômega

- A notação Ω é usada para expressar o limite inferior do tempo de execução de qualquer algoritmo para resolver um problema específico
- Exemplo: O limite inferior para qualquer algoritmo de ordenação que utiliza comparação de chaves é $\Omega(n \log n)$.

Clique

- Dado um grafo não dirigido $G=(V,E)$, um clique C em G é um sub-grafo completo de G , i.e. um sub-grafo tal que todos os vértices em C estão conectados a todos os outros vértices em C . O Problema pede para determinar se G contém um clique de tamanho k , ou seja, contendo k vértices.

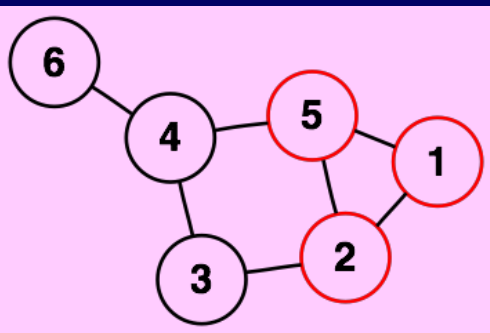
- Algoritmo Ingênuo: Listar todos os k sub-conjuntos de V e checar cada um para ver se forma um clique. O tempo de execução é:

$$\Omega(k^2 \binom{|V|}{k})$$

coeficiente binomial = número de modos de escolher k em $|V| =$

$$\text{combinações} = \frac{|V|!}{k! (|V| - k)!}$$

$$k! (|V| - k)!$$



que é polinomial se k é constante, mas em geral k pode ser proporcional a $|V|$ e neste caso roda em tempo super-polinomial.



Problemas de Decisão

- Resolver problemas de decisão podem ser pensados como "reconhecimento de linguagens formais"
- As instâncias do problema são codificadas como cadeias e uma cadeia pertence à linguagem sse a resposta ao problema de decisão é SIM!

Problemas de Decisão (Sim/Não)

- Por simplicidade, a Teoria da **NP-Compleitude** restringe sua atenção a **problemas de decisão**.
- É fácil transformar um problema genérico em um problema de decisão.
- Os problemas de otimização podem ser rephraseados pela imposição de um limite sobre o valor a ser otimizado.

Exemplo 1: Caixeiro Viajante

Dados: um conjunto de cidades $C = \{c_1, c_2, \dots, c_m\}$, uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$, e uma constante k

Problema de Otimização: achar um caminho que passe por todas as cidades e cujo tamanho seja **menor ou igual a k** .

Problema Refraseado: Existe um caminho para todas as cidades em C cujo comprimento total seja **menor ou igual a k** ? (SIM/NÃO)

Obs: pode também ser refraseado em termos de caminho hamiltoniano.

Dados: um grafo $G(V, E)$ não dirigido com peso nas arestas e um número k .

Problema Refraseado: Existe um caminho hamiltoniano cujo custo seja **no máximo k** ? (SIM/NÃO)

Exemplo 2: Caminho Mínimo

Dados: um grafo $G = (V, E)$, dois vértices $u, v \in V$ e um inteiro não negativo k .

Problema Refraseado: Existe um caminho em G entre u e v cujo comprimento seja no máximo k ?
(SIM/NÃO)

Classe P

P é a classe de problemas que podem ser resolvidos por algoritmos determinísticos em tempo polinomial.

- Podemos ser simplistas se considerarmos que
 - "estar em P significa "fácil" e "não estar em P" significa "difícil".
- Na teoria esta suposição é válida, PORÉM na prática nem sempre é verdadeira por várias razões.

Algoritmos Eficientes versus Ineficientes

- **R1:** A teoria ignora fatores constantes. Um problema que tem tempo $10^{1000} n$ está em P (tem tempo linear), mas é intratável na prática. Um problema que tem tempo $10^{-10000} 2^n$ não está em P (tem tempo exponencial), mas é tratável para valores de n acima de 1000.
- **R2:** A teoria ignora o tamanho dos expoentes. Um problema que tem tempo n^{1000} está em P, mas é intratável. Um problema com tempo $2^{n/1000}$ não está em P, mas é tratável com n acima de 1000.
- Embora, em geral, graus elevados e coeficientes muito grandes não ocorram na prática.
- **R3:** Ela só considera a análise de pior tempo.
- **R4:** Ela **não** considera **soluções probabilísticas** mesmo aquelas que admitem uma pequena probabilidade de erro.
- Um algoritmo 2^n é muito mais rápido que um algoritmo n^5 para $n \leq 22$. ($n = 22 \rightarrow 4.194.304 \times 5.153.632$)

Classe NP

Formalmente, NP é a classe de problemas que podem ser resolvidos por algoritmos não-determinísticos em tempo polinomial.

Ou, alternativamente:

NP é a classe de problemas de decisão para os quais uma dada solução pode ser verificada em tempo polinomial.

Assim, para mostrar que um problema está em NP:

- apresentamos um algoritmo não-determinístico polinomial para **RESOLVER** o problema ou
- apresentamos um algoritmo determinístico polinomial para **VERIFICAR** que uma dada solução (a solução **adivinhada**) é válida.

MTND

- Onde uma máquina de Turing normal diz a você se uma solução particular é correta,
 - uma **MT não determinística** irá dizer a você **se alguma resposta correta existe** e somente leva o tempo da maior verificação.

Caracterizando NP

As soluções para uma dada entrada podem ser geradas e verificadas:

- **Uma de cada vez:** apesar de ser possível é muito lento porque o conjunto de soluções a ser verificado é muito grande (não-polinomial).
- **Simultaneamente:** verificação de todas as soluções ao mesmo tempo (neste caso a solução poderia ser obtida em tempo polinomial).
- **Propriedade Específica:** descobrir alguma propriedade dos objetos envolvidos e obter um algoritmo que não precisa verificar todas as soluções.
 - Por exemplo, ordenação por comparação não precisa verificar cada uma das $n!$ permutações.

Caracterizando NP

Simultaneidade pode representar computação **não-determinística**.

- Um computador não-determinístico, quando diante de duas ou mais alternativas, é capaz de produzir cópias de si mesmo e continuar a computação independentemente para cada alternativa.
- **É o mesmo que computação Paralela???**
 - A computação paralela não é a resposta para tornar tratável um algoritmo exponencial. O obstáculo é a equação:
 - Trabalho = tempo de paralelismo X número de processadores
 - Assim, é impossível tornar um dos dois ou os dois componentes exponenciais. É improvável que vamos construir um computador com um número astronômico de processadores.
- Um algoritmo não-determinístico é capaz de escolher uma dentre várias alternativas possíveis a cada passo (o algoritmo é capaz de **adivinhar** a alternativa que leva a situação).

Algoritmos Não-determinísticos

- Algoritmos não-determinísticos contêm operações cujo resultado não é unicamente definido (ainda que limitado a um conjunto especificado de possibilidades):
 1. Adivinha uma atribuição para as variáveis
 2. Checa a atribuição
- Função Escolhe(C)
 - Escolhe um dos elementos de C de forma arbitrária
 - Sempre que existir um conjunto de opções que levem a um término com sucesso então este conjunto será escolhido
- A complexidade de Escolhe() é $O(1)$.



Exemplo 1 - Busca Linear

- Pesquisar/Buscar o elemento x em um conjunto de elementos $A[1..n]$, $n \geq 1$

$i \leftarrow \text{Escolhe}(A, 1, n)$

If $A[i] = x$ then sucesso

Else insucesso

- Complexidade: $O(1)$
- Para algoritmos determinísticos a complexidade da busca linear é $\Omega(n)$

Exemplo 2 - Clique

Clique: **Determinar** se um grafo $G = (V, E)$ não dirigido contém um clique de tamanho k . c é um subconjunto de vértices de tamanho k do grafo.

Entrada = $\langle G, k \rangle$

1. $c \leftarrow$ Escolhe(G, k) {escolhe não-deterministicamente um subconjunto c de k nós de G }
2. Cheque se **para todo** par $u, v \in c$, a aresta $(u, v) \in E$
3. Se sim aceite senão rejeite

Uma forma alternativa é construir um **Verificador** V para Clique.

Entrada = $\langle G, k, c \rangle$

1. Teste se c é um conjunto de k nós em G
 2. Se sim então Cheque se **para todo** par $u, v \in c$, a aresta $(u, v) \in E$
 3. Se sim então sucesso senão insucesso
- senão insucesso

O tamanho do certificado: $O(n)$ ($n = |V|$)

Complexidade: $O(n^2)$

Exemplo 3 - Satisfatibilidade

SAT: Checar se uma expressão booleana na forma normal conjuntiva (CNF) com literais x_i ou $\neg x_i$, $1 \leq i \leq n$ é satisfatível, isto é, se existe **uma** atribuição de valores lógicos (V ou F) que torne a expressão verdadeira.

Satisfatível: $(x1 \vee x2) \wedge (x1 \vee x3 \vee x2) \wedge (x3)$

$x1 = F; x2 = V; x3 = V$

Não Satisfatível: $(x1) \wedge (\neg x1)$

Várias cláusulas conectadas com \wedge . Cada cláusula contém literais conectados com \vee

Procedure Aval(E,n)

Begin

For i <- 1 to n do

$x_i \leftarrow$ Escolhe(true,false)

if $E(x_1, x_2, \dots, x_n) = \text{true}$ then sucesso

else insucesso

end

F,V,V

Certo, F,V,V satisfaz a expressão



O algoritmo obtém uma das 2^n atribuições de forma não-determinística em $O(n)$.

Importância de SAT

- Esse problema desempenha aqui na Teoria da Complexidade o mesmo papel que o A_{TM} (Problema da Parada) para problemas indecidíveis:
 - uma vez demonstrado intratável, sua redução a outros problemas nos permite concluir que esses também são intratáveis.
- A noção de redução deve ser alterada aqui:
 - a existência de um algoritmo para transformar instâncias de um problema em instâncias de outro não é mais suficiente.
 - Esse algoritmo deve demorar no máximo um **tempo polinomial**, ou então a redução não permitirá concluir que o problema de destino é intratável, mesmo que o de origem o seja.
 - Falaremos de "redução de tempo polinomial".

A Questão $P \stackrel{?}{=} NP$ E a Classe co-NP

- É uma questão aberta se $P = NP$, pois a prova parece exigir técnicas ainda desconhecidas. Mas se acredita que não são a mesma classe.

- Classe co-NP de problemas: problemas de decisão cuja solução *negativa* admite um certificado/verificação polinomial.

- Exemplo: Validade

Dado: uma expressão booleana

Problema: Decidir se a expressão é **válida** (i.e. satisfatível para todas as atribuições de valores lógicos)

Expressão Booleana Válida:

$$x \vee \neg x$$

Expressão Booleana Inválida

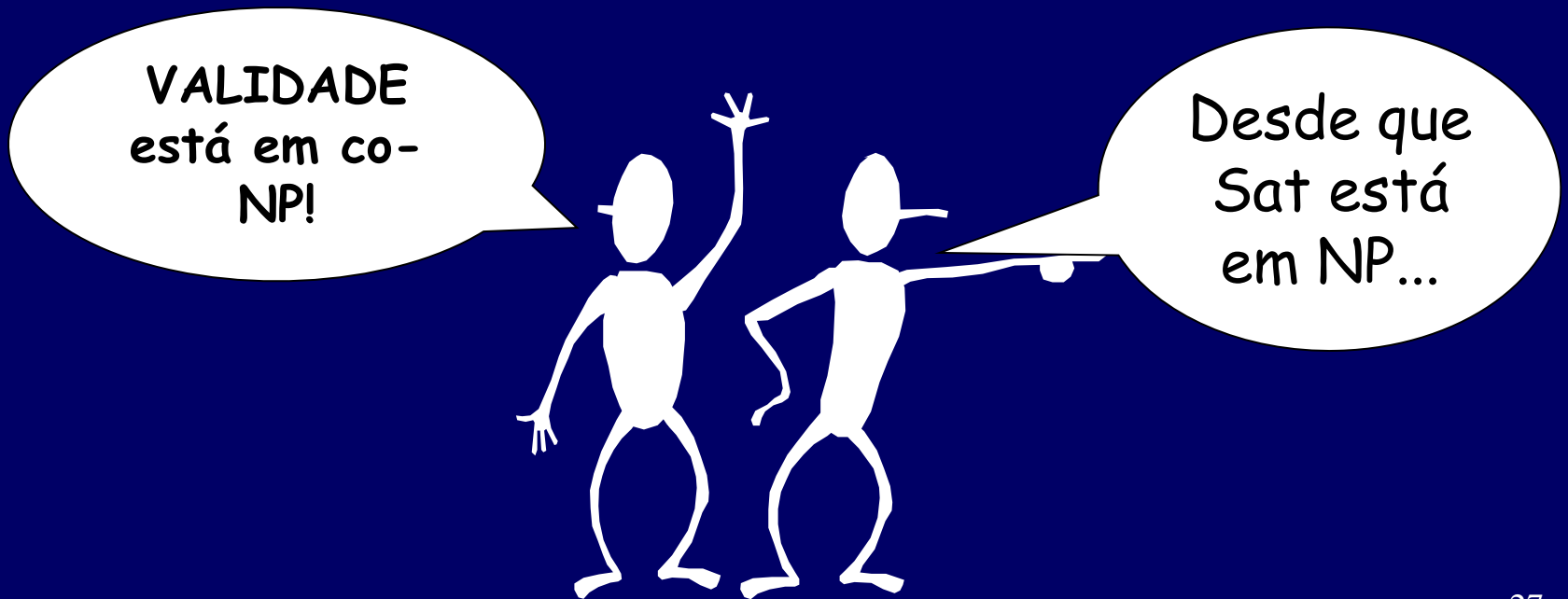
x

Validade está em co-NP

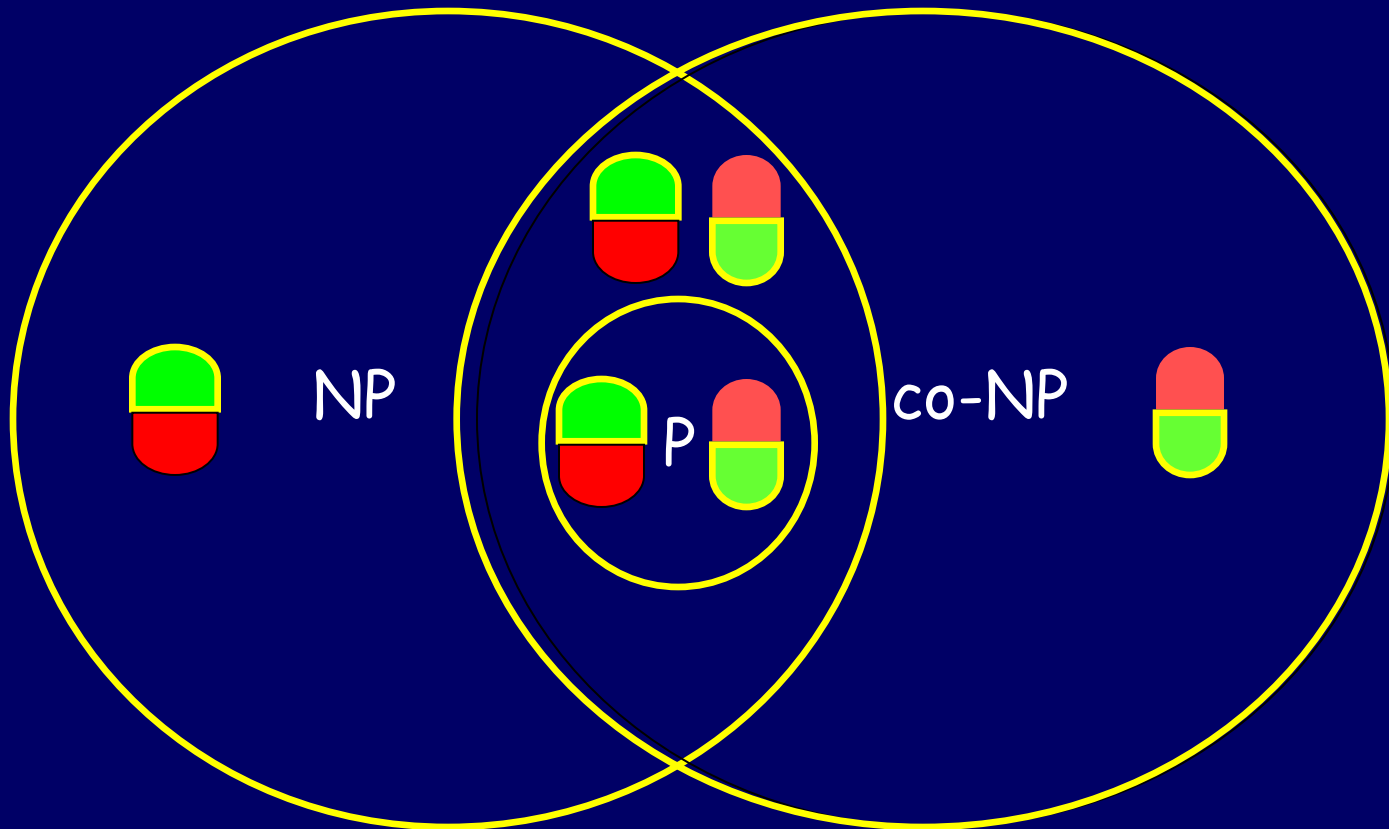
1. Escolha/Adivinhe uma atribuição de valores lógicos
2. Cheque se ela **não** satisfaz a expressão



- Por definição, o complemento de toda linguagem NP está em $co-NP$.
- O complemento de uma linguagem $co-NP$ está em NP .



NP e co-NP



$P \subseteq co-NP$: pois $co-P = P$, e $P \subseteq NP$

- Ainda é uma questão aberta se NP é fechada sobre o complemento, isto é,
 - Se $L \in \text{NP}$ implica que complemento de $L \in \text{NP}$? Podemos rephrasear a questão acima como: $\text{NP} = \text{co-NP}$?
- Sabemos que P é fechada sobre o complemento, assim $P \subseteq \text{NP} \cap \text{co-NP}$, mas não sabemos se $P = \text{NP} \cap \text{co-NP}$.

Cenários de pesquisa entre as classes de complexidade P, NP e co-NP.

$P = NP = \text{co-NP}$

$NP = \text{co-NP}$

P

$P = NP \cap \text{co-NP}$

co-NP $NP \cap \text{co-NP}$ NP
P

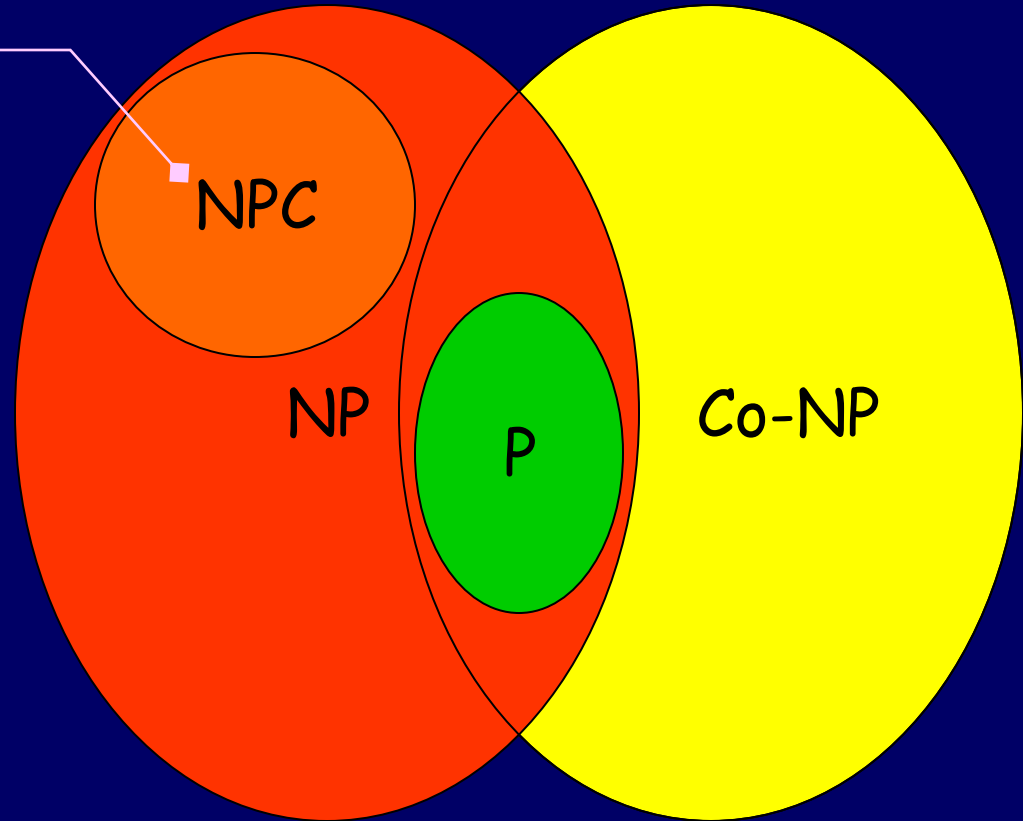
Mais provável!

NP-Compleitude

- No início dos anos 70, Cook & Levin descobriram certos problemas em NP cuja complexidade individual era relacionada com a da classe inteira.
- Estes problemas são chamados **NP-completos**.
- Eles são os mais difíceis da sua própria classe e assim podemos escolher qualquer um deles para avançar técnicas de resolução para a classe inteira.

- Se um algoritmo de tempo polinomial existir para qualquer um destes problemas, todos os problemas em NP seriam resolvidos em tempo polinomial. Talvez seja esta a razão de se acreditar que $P \neq NP$.
- Assim, a classe NP-completo tem a propriedade de que, se um problema NP-completo puder ser resolvido em tempo polinomial todos os problemas em NP tem solução polinomial e $P = NP$.
- Para definir formalmente a classe NP-completo precisamos da noção de Redução Polinomial.

Circuito hamiltoniano,
Caminho hamiltoniano,
Caixeiro viajante,
Clique, SAT com 3
ou mais literais ...

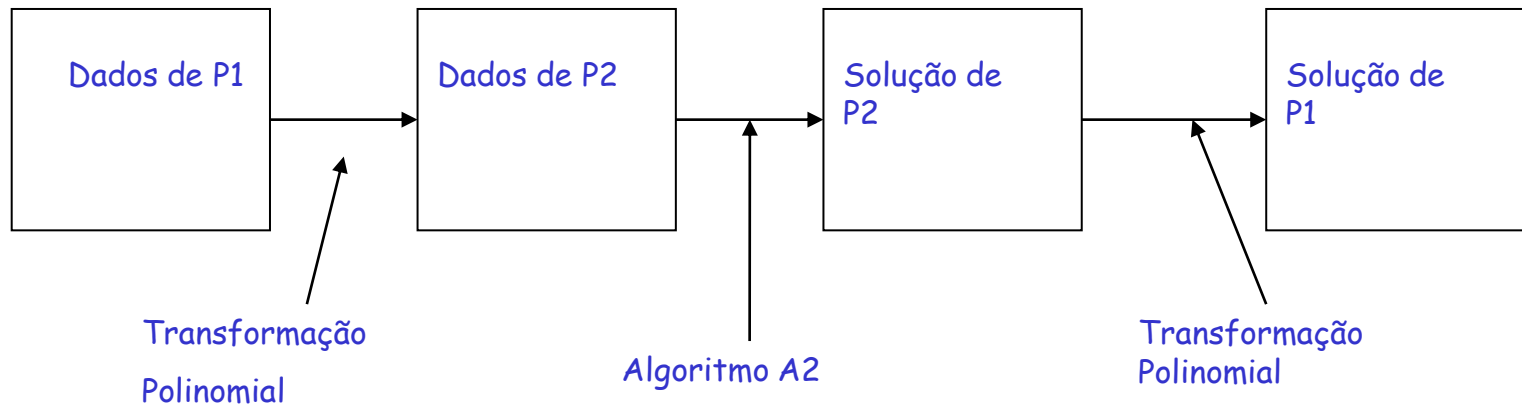


Redução/Transformação Polinomial

- Na parte do curso sobre computabilidade já definimos o conceito de reduzir um problema para outro.
- Aqui vamos usar uma redução que leva em conta a **eficiência**:

- Sejam $P1$ e $P2$ dois problemas de decisão. Suponha que exista um algoritmo $A2$ para resolver $P2$.
- Se for possível transformar $P1$ em $P2$ e sendo conhecido o processo de transformar a **solução** de $P2$ em uma **solução** de $P1$ então o algoritmo $A2$ pode ser utilizado para resolver $P1$.
- Se as transformações nos dois sentidos (entrada e saída) puderem ser realizadas em TEMPO POLINOMIAL então $P1$ é polinomialmente transformável em $P2$.
Denota-se: $P1 \leq_p P2$

Graficamente



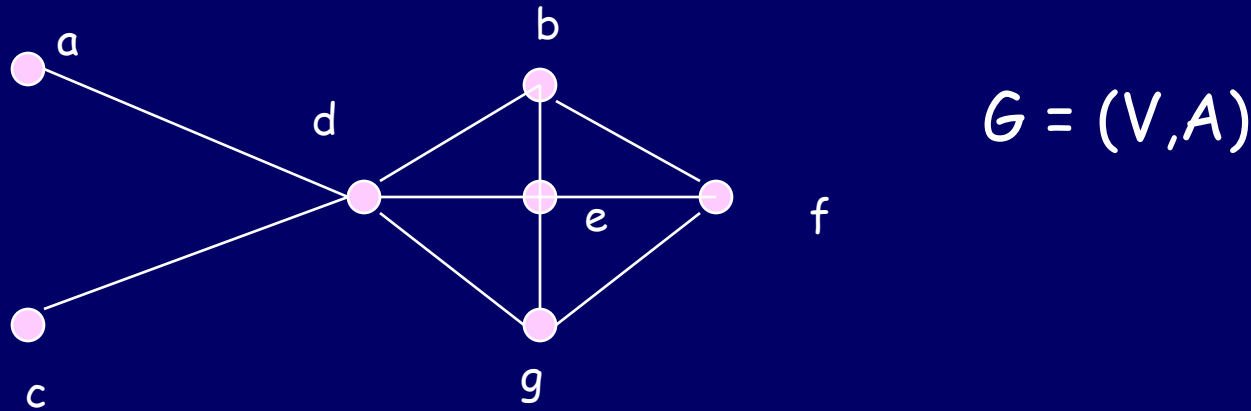
Redução - Propriedades

- **Teorema da Transitividade.** Se $P1$ é polinomialmente transformável em $P2$ e $P2$ é polinomialmente transformável em $P3$ então $P1$ é polinomialmente transformável em $P3$.

$$P1 \leq_p P2, P2 \leq_p P3 \Rightarrow P1 \leq_p P3$$

- **Definição:** Dois Problemas são Polinomialmente Equivalentes sse $P1 \leq_p P2$ e $P2 \leq_p P1$

Exemplo de Redução Polinomial



Conjunto Independente de Vértices

$V' \subseteq V \mid \forall$ par de vértices de V' é não-adjacente (sub-grafo totalmente desconexo)

$(v,w) \in V' \Rightarrow (v,w) \notin A$. Exemplo com cardinalidade 4: $\{a,c,b,g\}$

Clique

$V' \subseteq V \mid \forall$ par de vértices de V' é adjacente (sub-grafo completo)

$(v,w) \in V' \Rightarrow (v,w) \in A$. Exemplo com cardinalidade 3: $\{d,b,e\}$

Exemplo de Redução Polinomial

Instância I de Clique

Dados: grafo $G(V,A)$ e um inteiro $K > 0$

Decisão: G possui um clique de tamanho $\geq K$?

Instância $f(I)$ de Conjunto Independente de Vértices.

Considere o sub-grafo complementar GC de G e o mesmo K

f é uma **transformação polinomial** porque:

(i) GC pode ser obtido de G em tempo polinomial

(ii) G possui clique de tamanho $\geq K$ **se, e somente se**, GC possui conjunto independente de vértices de tamanho $\geq K$.

Conclusão

- Se existir um algoritmo polinomial que resolve conjunto independente de vértices este algoritmo pode ser utilizado para resolver clique também em tempo polinomial.

Clique \leq_p Conjunto Independente

- O tipo de redução comentado nestes slides é:
 - Polynomial-time **mapping** reduction
 - Polynomial-time **many-one** reduction
 - Polynomial-time **Karp** reduction
- Existem outras: Cook-reduction. Karp é um caso especial de Cook reductions.

Como reduzir?

1. Construa f .
2. Mostre que f tem complexidade polinomial.
3. Prove que f é uma redução, i.e. mostre, por exemplo, para Ciclo_H e Caminho_H :

1. Se $w \in \text{Caminho}_H$ então $f(w) \in \text{Ciclo}_H$
2. Se $f(w) \in \text{Ciclo}_H$ então $w \in \text{Caminho}_H$

Compleitude

Definição: Seja C uma classe de complexidade e L uma linguagem/problema. Nós dizemos que L é C -Completa se:

1. L está em C
2. Para toda linguagem $A \in C$, A é reduzível a L .

Se uma linguagem L satisfaz a propriedade 2 mas não necessariamente a 1, nós dizemos que L é NP-difícil.

Assim, NPC está no centro do problema de decidir se
 $P = NP$

Np-difíceis

- Apenas problemas de decisão podem ser NP-completos.
- Problemas de otimização podem ser NP-difíceis.
- Para provar que L é *NP-difícil*, é suficiente mostrar que L muito provavelmente exige tempo exponencial, ou pior.
- E problemas indecidíveis?? Podem ser NP-difíceis?

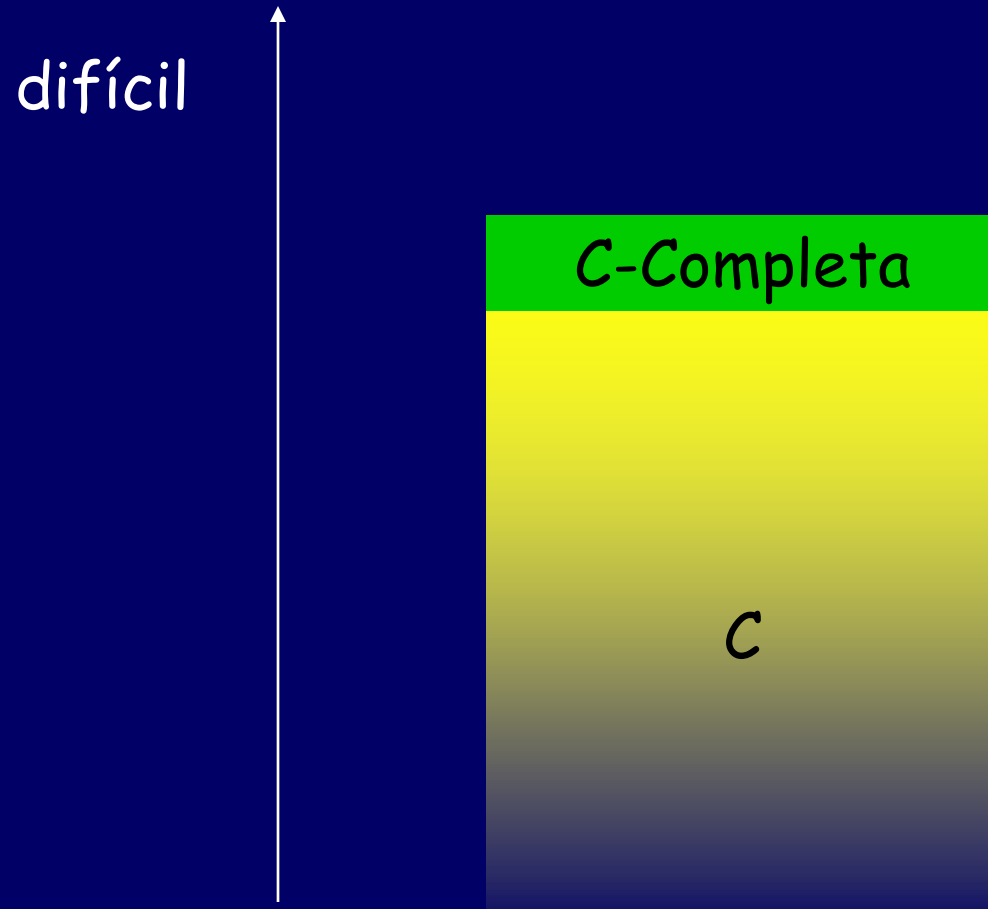
Problema da Parada

- É um exemplo de NP-difícil que não é NP-completo.
- Este problema, como sabemos, é indecidível pois não há nenhum algoritmo de nenhuma complexidade que possa resolvê-lo.

$SAT \leq_p$ Problema da Parada?

- Prova: considere o algoritmo A cuja entrada é uma expressão booleana E na forma normal conjuntiva com n variáveis
- Basta tentar 2^n possibilidades e verificar se é satisfável.
- Se for pára, senão entra em loop.
- Logo, o problema da parada é NP-difícil mas não é NP-completo.

Linguagens C-Completas



Efeitos da NP-completude

1. Quando descobrimos que um problema é *NP-completo*, ele nos diz que existe pouca chance de um algoritmo eficiente poder ser desenvolvido para resolvê-lo.
 - Somos encorajados a procurar por heurísticas, soluções parciais, aproximações ou outros meios.
 - Além disso, podemos fazer isso com a confiança de que não estamos apenas "trapaceando".

Efeitos da NP-completude

2. Cada vez que adicionamos um novo problema P *NP-completo* à lista, reforçamos a idéia de que *todos* os problemas *NP-completos* exigem tempo exponencial.
 - O esforço que foi despendido na busca de um algoritmo de tempo polinomial para o problema P foi, não intencionalmente, um esforço dedicado a mostrar que $P=NP$.
 - O resultado desse esforço é a grande evidência de que
 - a) é muito improvável que $P=NP$, e
 - b) *todos* os problemas *NP-completos* exigem tempo exponencial.

Teorema de Cook-Levin

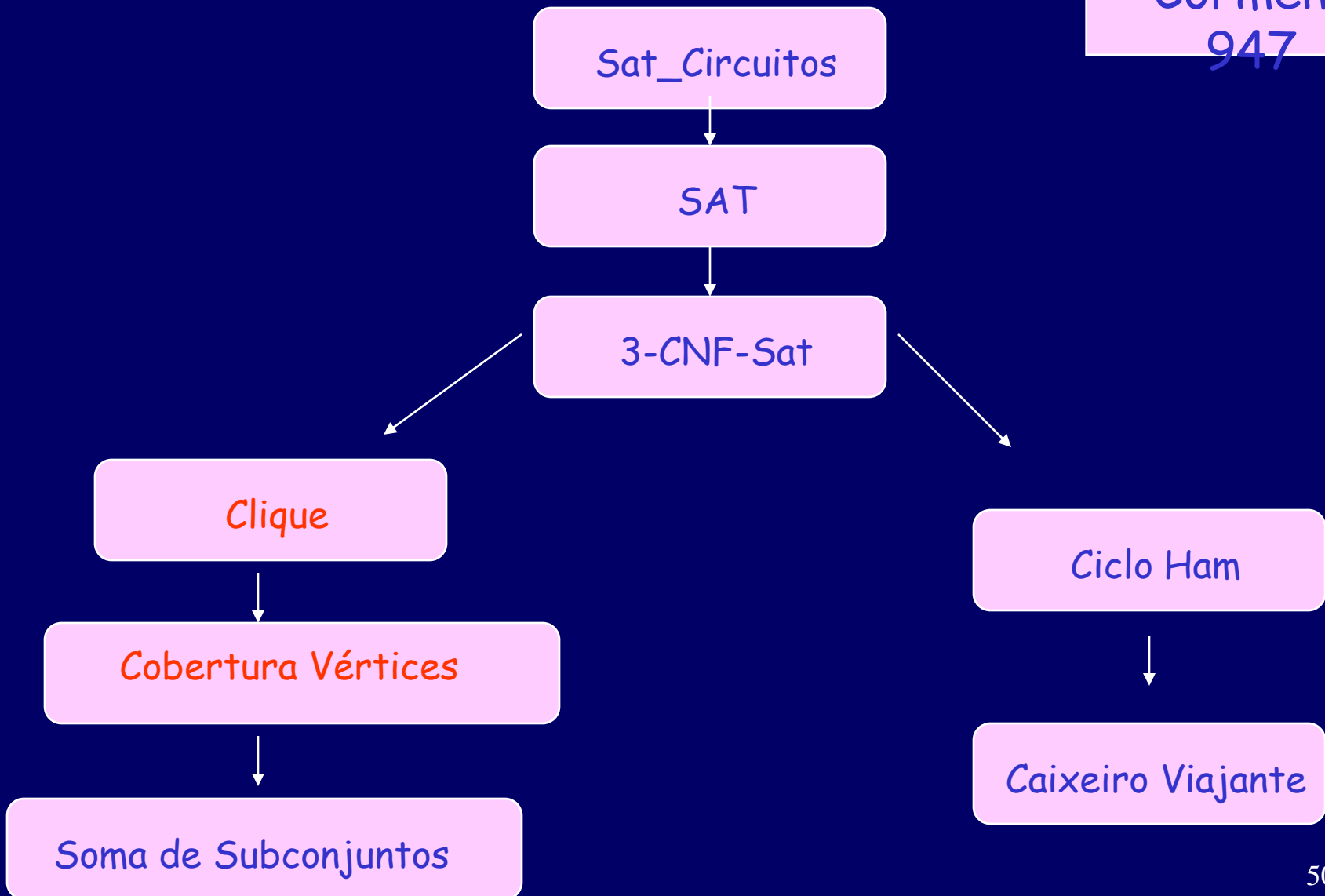
- SAT é NP-completa.
- Idéia da Prova:
 - mostrar que SAT está em NP é fácil.
 - A parte difícil é mostrar que toda linguagem em NP é polinomialmente reduzível a SAT.

SAT está em NP, desde que nós podemos **checar** o resultado de uma atribuição de valores verdade para os literais em tempo polinomial.

- Uma vez que temos um problema NP-completo, podemos obter outros por redução polinomial a partir dele. Assim, estabelecer o primeiro NP-completo é mais difícil.

Estrutura das provas de NP-completude

Cormen-
947



Exemplos de problemas NP-C e fontes de pesquisa

- Christos H. Papadimitriou, Computational Complexity, Addison-Wesley Publishing Company, 1995. (Biblioteca Central). (TENHO XEROX - PODEM EMPRESTAR)
- Cormen, T.H. Leiserson, C.E. and Rivest, R.L. Introduction to Algorithms. The Mit Press. 1990. (1ª edição). Capítulos 1, 2, 3, 4, 36, 37.
- Garey & Johnson. Computers and Intractability - a guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York, 1979.

Problemas

- Escolham problemas menos CLIQUE - CONJUNTO INDEPENDENTE de VERTICES (já citados nesses slides)

Exemplos:

- 3SAT,
- NAESAT (not-all-equal SAT),
- Cobertura de vértices,
- CAM HAM (ou CICLO HAM),
- podem também escolher CAIXEIRO VIAJANTE pela importante prática deste e de como vem sendo resolvido com heurísticas (embora seja um exemplo de CAM-CICLO HAM - as vezes a prova usa problemas diferentes para a redução),
- mochila (knapsack),
- soma de subconjuntos,
- 3-coloring (pode um mapa ser colorido com 3 cores?),
- Partição em grafos (cut), etc.

Problemas *NP-completos* sobre Grafos

- O Problema do Caixeiro Viajante (encontrar um Ciclo Hamiltoniano)
- O Problema da Cobertura de Nós: *encontrar um conjunto de nós tal que cada aresta do grafo tem pelo menos uma de suas extremidades em um nó do conjunto.*
- O Problema CLIQUE: *verificar se um grafo tem um k -clique, ou seja, um conjunto de k nós tal que existe uma aresta entre todo par de nós no clique.*
- O Problema da Coloração: *um grafo G pode ser "colorido" com k cores?*

- O Problema da Mochila: *dada uma lista de k inteiros, podemos particioná-los em dois conjuntos cujas somas sejam iguais?*

- O Problema do Escalonamento do tempo de execução unitário: *dadas k tarefas T_1, T_2, \dots, T_k , uma série de "processadores" p , um tempo limite t , e algumas restrições de precedência, da forma $T_i \prec T_j$ entre tarefas, existe um escalonamento de tarefas tal que:*
 - 1) *cada tarefa seja atribuída a uma unidade de tempo entre 1 e t ;*
 - 2) *no máximo p tarefas sejam atribuídas a qualquer unidade de tempo, e*
 - 3) *as restrições de precedência sejam respeitadas: se $T_i \prec T_j$, então T_i é atribuída a uma unidade de tempo anterior a T_j ?*