

Problemas Intratáveis

ou

computação eficiente

X

computação ineficiente

Problemas Tratáveis

- Os problemas que podem ser resolvidos em tempo polinomial em um computador típico são exatamente os mesmos problemas que podem ser resolvidos em tempo polinomial em uma MT.
- Problemas práticos que exigem tempo polinomial quase sempre podem ser resolvidos em um período de tempo tolerável, enquanto aqueles que exigem tempo exponencial* em geral não podem ser resolvidos, exceto para instâncias pequenas.

* *Qualquer tempo de execução maior que todos os tempos polinomiais*

Objetivo

- Teoria da intratabilidade: técnicas para mostrar problemas que não podem ser resolvidos em tempo polinomial (*intratáveis*).
- Vejamos o problema específico (SAT):
 - A questão de saber se uma expressão booleana pode ser *satisfeita*, isto é, tornar-se verdadeira para alguma atribuição de valores verdade TRUE e FALSE a suas variáveis.

O problema da satisfatibilidade

- Esse problema desempenha aqui o mesmo papel que a L_u 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: 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”*

$P \neq NP$

- Enquanto que os resultados da Teoria da Indecidibilidade são irrefutáveis, os da Teoria da Intratabilidade são baseados numa suposição fortemente aceita, porém nunca provada, **a hipótese $P \neq NP$** .
- Essa hipótese nos diz que a classe de problemas que podem ser resolvidos por MTND em tempo polinomial (NP) inclui pelo menos alguns problemas que não podem ser resolvidos por MTD em tempo polinomial (P), mesmo considerando maior grau do polinômio. Ou seja, embora na forma não determinística o tempo é polinomial, na forma determinística (i.e, um programa de computador), o tempo é exponencial.

$$P \neq NP$$

- Há milhares de problemas que *parecem* estar nessa categoria, pois eles podem ser resolvidos facilmente por uma MTND de tempo polinomial, ainda que nenhum programa de computador (ou seja, uma MTD) de tempo polinomial seja conhecido.
- **Consequência:** todos esses problemas ou têm soluções determinísticas de tempo polinomial (que ainda não foram encontradas), ou então nenhum deles tem essas soluções; i.e. todos eles realmente exigem tempo exponencial.

A classe P

- Problemas que podem ser resolvidos em tempo polinomial
 - Uma MT M é dita de *complexidade de tempo* $T(n)$ (ou *que tem tempo de execução* $T(n)$) se, sempre que M recebe uma entrada w de comprimento n , M pára depois de efetuar no máximo $T(n)$ movimentos, independentemente do fato de M aceitar ou não w .
- Dizemos que uma linguagem L está na classe P se existe algum polinômio $T(n) = c_1.n^k + c_2.n^{k-1} + \dots + c_{k-1}.n + c_k$, tal que $L = L(M)$ para alguma MT determinística M de complexidade de tempo $T(n)$.

A classe P

- A grande maioria dos problemas com que nos deparamos está na classe P , ou seja, tem solução em tempo polinomial.

A classe NP

- Formalmente, dizemos que uma linguagem L está na classe NP (polinomial não determinística) se existe uma MTND M e uma complexidade de tempo polinomial $T(n)$ tais que $L=L(M)$ e, quando M recebe uma entrada de comprimento n , não existe nenhuma sequência de mais de $T(n)$ movimentos de M .

A classe NP

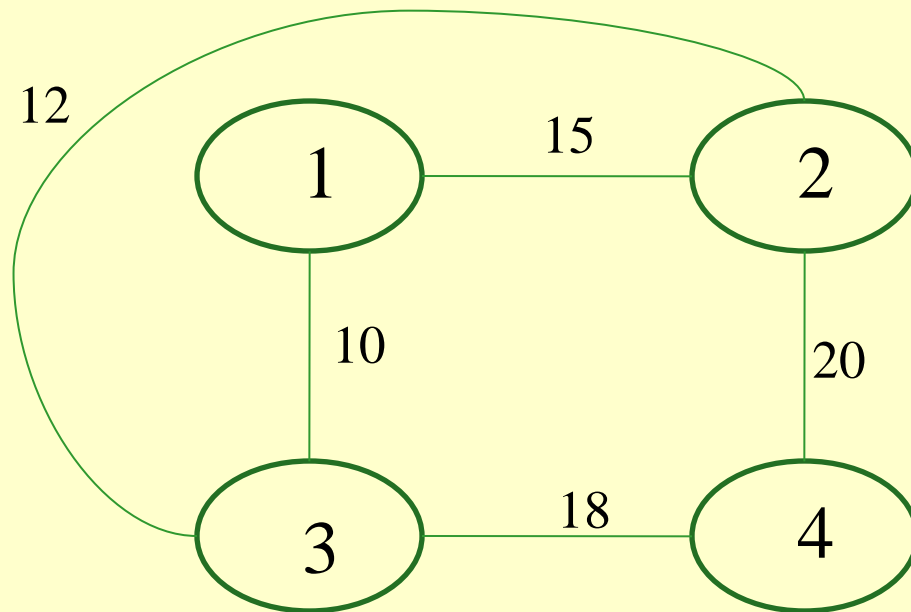
- Considerando que toda MT determinística é uma MTND com no máximo uma opção de movimento, então $P \subseteq NP$.
- Porém, parece que NP contém muitos problemas que não estão em P . A razão intuitiva é que uma MTND funcionando em tempo polinomial tem a habilidade de pressupor um número exponencial de soluções possíveis para um problema e verificar cada uma em tempo polinomial, “*em paralelo*”.

$$P \stackrel{?}{=} NP$$

- Uma das principais questões em aberto na Matemática é se $P = NP$, isto é, se de fato tudo o que pode ser feito em tempo polinomial por uma MTND pode realmente ser feito por uma MTD em tempo polinomial, talvez com um polinômio de grau mais alto.

Exemplo em *NP*: o problema do caixeiro viajante (PCV)

- Dado um conjunto de cidades, encontrar um circuito que inclua cada cidade uma única vez, e que represente o trajeto mais econômico.
- **Entrada:** (a) um grafo com m vértices ou nós, e e arestas, com pesos inteiros nas arestas; cada nó representa uma cidade, e (b) um limite de peso (custo) W .
- **Pergunta** adaptada à MT: esse grafo tem solução para o PCV com peso total no máximo W ?
- **Solução:** verificar a existência de *circuitos hamiltonianos*, ou seja, conjuntos de arestas de caminhos cíclicos onde cada nó do grafo ocorre exatamente uma vez; e verificar o peso total (soma dos pesos das arestas) de cada um deles, e comparar com W .



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, há no máximo 2 soluções possíveis (*verifique*).

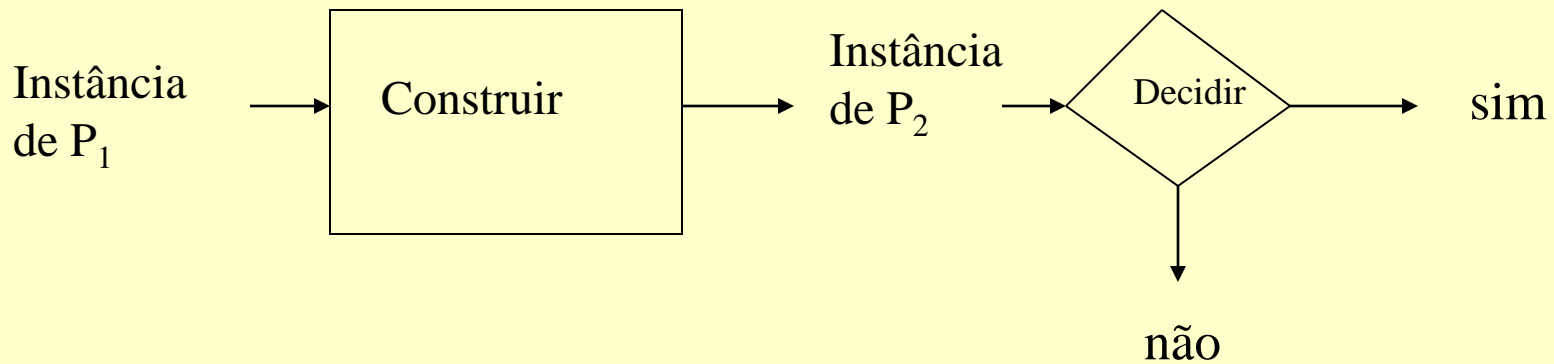
Em grafos de m nós, o número de ciclos distintos cresce como $O(m!)$, que é maior que 2^{cm} para qualquer constante c .

- A solução do PCV envolve a determinação de todos os ciclos hamiltonianos e o cálculo de seus pesos totais.
- Embora algumas escolhas ruins possam ser descartadas, temos que examinar um número exponencial de ciclos para (i) concluir que não existe nenhum com o limite de peso desejado W , ou (ii) descobrir um, dependendo da ordem em que os ciclos são considerados. Num computador determinístico, isso levaria portanto um tempo exponencial.
- Já, se tivermos um computador não-determinístico (MTND), poderíamos, *para cada permutação* possível dos m nós ($m!$ *permutações*), verificar se (a) consiste num ciclo hamiltoniano; e (b) calcular seu peso e comparar com W .

- Cada processamento de verificação de uma permutação gastaria tempo polinomial: (a) $O(e)$ para verificar se é ciclo + $O(m)$ para verificar se é hamiltoniano; (b) $O(e)$ para calcular o peso.
- Ou seja, $O(m+e)$ (onde $m+e$ nesse caso representa o tamanho da entrada n) é necessário para verificar cada solução-candidata.
- Se cada solução-candidata fosse analisada em paralelo (ou seja, como uma MTND), teríamos um tempo polinomial.
- Sendo assim, o PCV está em NP .

Redução de tempo polinomial

- Mostramos que um problema P_2 não pode ser resolvido em tempo polinomial, ou seja, que não está em P , por meio da redução de um problema P_1 , que “sabemos” que não está em P , a P_2 .

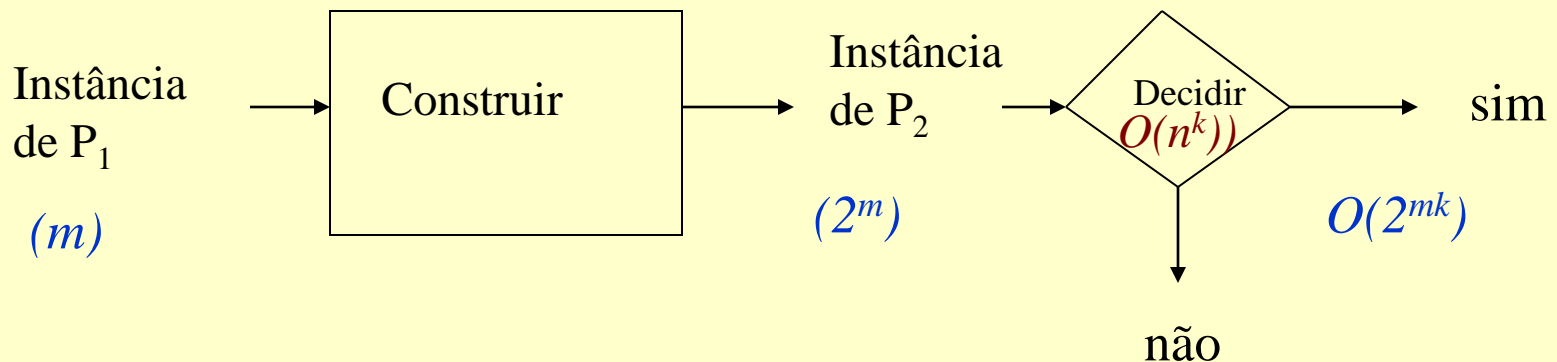


Suponha que queremos provar que “se P_2 está em P , então P_1 também está”. Como P_1 *não* está em P , poderíamos então afirmar que P_2 também não está em P .

No entanto, só a existência do algoritmo não é suficiente....

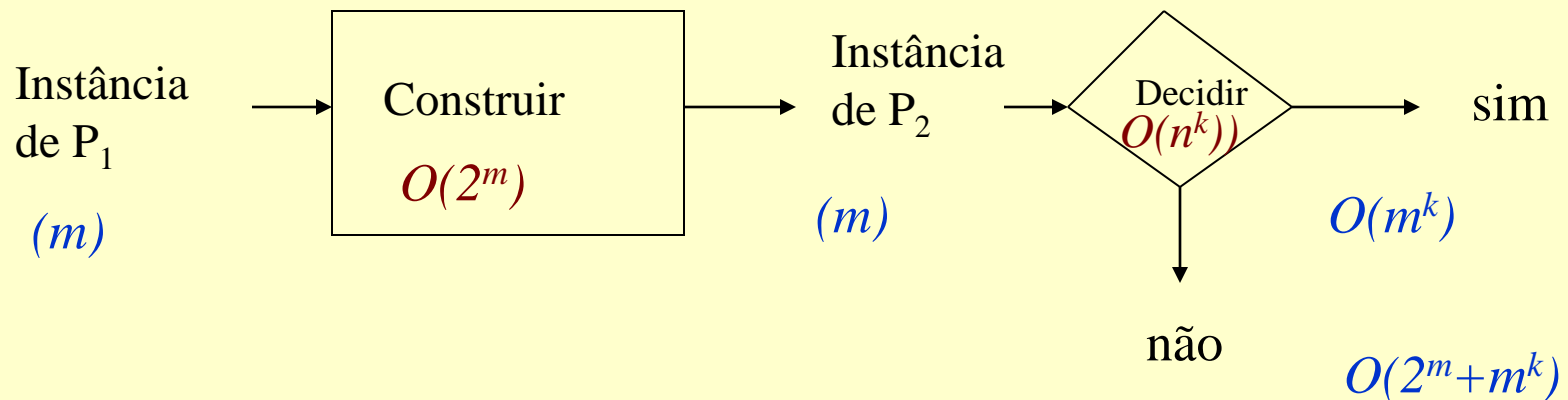
Se “Decidir” é polinomial ($O(n^k)$) e:

- “Construir” produz, para uma instância de P_1 de comprimento m , uma saída de comprimento 2^m . Então “Decidir” vai ser exponencial $O(2^{mk})$. Assim, o algoritmo de decisão para P_1 demora, quando recebe uma entrada de comprimento m , um tempo exponencial em m . Isso é consistente com a situação em que P_2 está em P e P_1 não está em P .



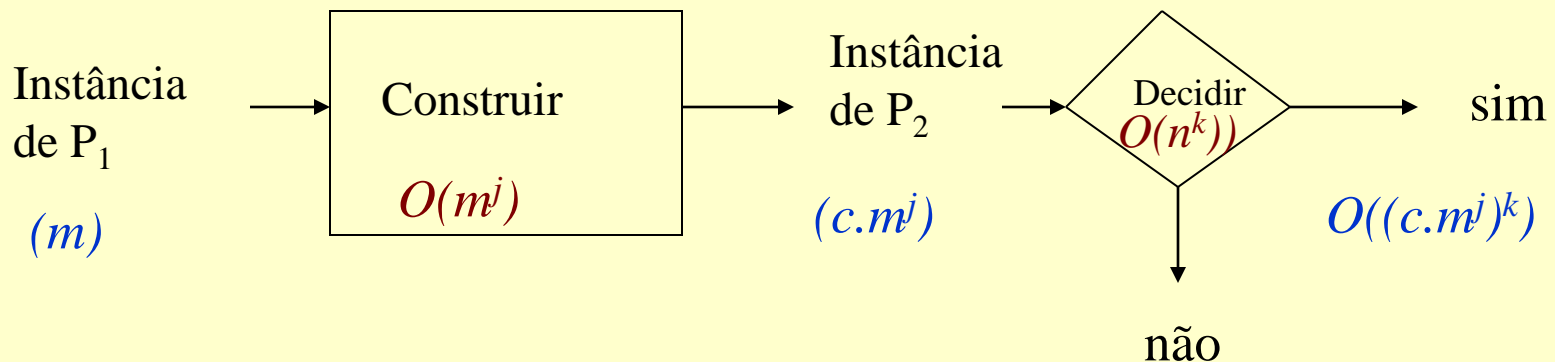
Se “Decidir” é polinomial ($O(n^k)$) e:

- “Construir” produz, para uma instância de P_1 de comprimento m , uma saída também de comprimento m , mas demore para isso um tempo exponencial, digamos $O(2^m)$. Então mesmo se “Decidir” é polinomial em m , a única implicação é que um algoritmo de decisão para P_1 vai demorar $O(2^m + m^k)$ sobre a entrada de comprimento m . Isso é novamente consistente com a situação em que P_2 está em P e P_1 não está.



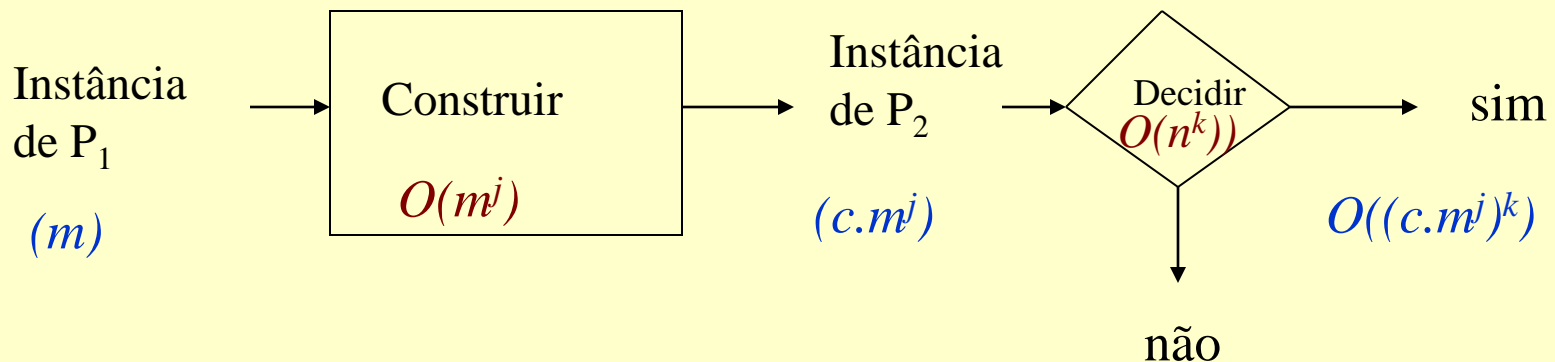
Conclusão

- A simples existência do algoritmo de conversão não nos garante que P_2 herda a propriedade de intratabilidade de P_1 .
- Na teoria da intratabilidade, a redução só permite a herança das propriedades de P_1 para P_2 **se o tempo de conversão (“Construir”) das instâncias de P_1 para P_2 for polinomial no comprimento de sua entrada.**
- Note que, se a conversão leva o tempo $O(m^j)$ sobre a entrada de comprimento m , então a instância de saída de P_2 não pode ser mais longa que o número de etapas tomadas, ou seja, ela é no máximo $c.m^j$ para alguma constante c .



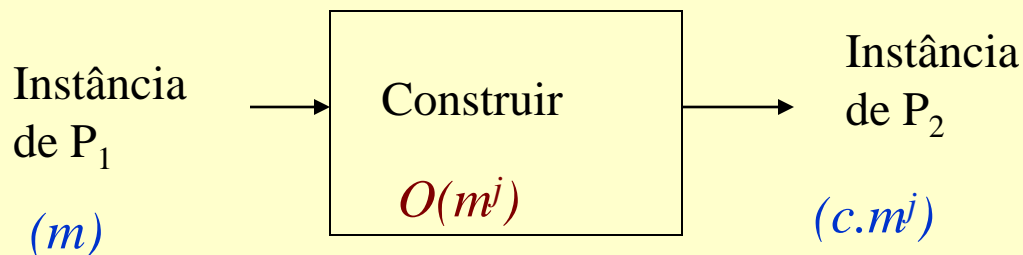
Agora podemos provar:

- “se P_2 está em P , então P_1 também está”. Como P_1 *não* está em P , poderíamos então afirmar que P_2 também não está em P .
- Suponha que “Decidir” leva $O(n^k)$ para decidir sobre uma entrada de comprimento n . Então, podemos resolver a pertinência a P_1 de uma cadeia de comprimento m no tempo $O(m^j + (c \cdot m^j)^k)$, onde m^j define o tempo para realizar a conversão, e o termo $(c \cdot m^j)^k$ é o tempo para resolver a instância resultante de P_2 .
 - $O(m^j + cm^{jk})$, com c, j, k constantes, é um polinômio em m , e concluímos que P_1 está em P (Absurdo!).



Redução de tempo polinomial

- Uma redução de P_1 a P_2 é em tempo polinomial se ela leva um tempo igual a algum polinômio no comprimento da instância de P_1 .
- **Consequência:** a instância de P_2 terá um comprimento polinomial no comprimento da instância de P_1 .



Problemas NP-completos

- Quais linguagens estão em NP e não estão em P?
- Seja L uma linguagem (um problema) em NP. Dizemos que L é *NP-completa* se:
 1. L está em NP.
 2. Para toda linguagem L' em NP, existe uma redução de tempo polinomial de L' a L .

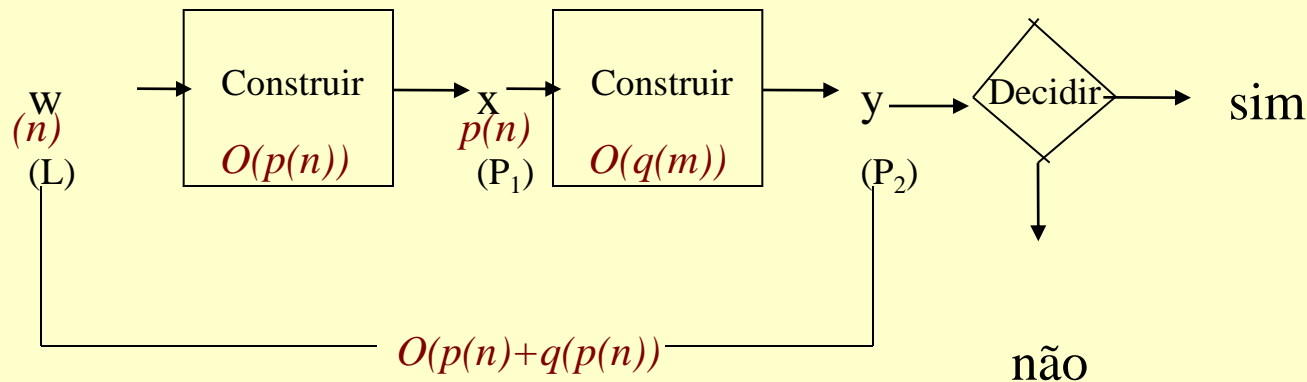
- Como parece que $P \neq NP$ e, em particular, todos os problemas NP-completos estão em $NP - P$, em geral, provar que um problema é *NP-completo* consiste de uma prova de que o problema não está em P .
- **Teorema 1:** Se P_1 é *NP-completo*, P_2 está em NP e existe uma redução de tempo polinomial de P_1 a P_2 , então P_2 é *NP-completo*.
- Prova: precisamos mostrar que toda linguagem L de NP se reduz em tempo polinomial a P_2 (*pela definição de NP-completo*)

Continuação da prova....

Como P_1 é *NP-completo*, sabemos que existe uma redução de tempo polinomial ($p(n)$) de L (qualquer) a P_1 . Assim, uma cadeia w em L de comprimento n é convertida numa cadeia x em P_1 de comprimento no máximo igual a $p(n)$.

Por hipótese, existe uma redução de tempo polinomial ($q(m)$) de P_1 a P_2 . Então essa redução transforma x em alguma cadeia y de P_2 em no máximo tempo $q(p(n))$. Assim, a transformação de w em y demora um tempo no máximo igual a $p(n)+q(p(n))$, que é um polinômio em n .

Concluimos que L é redutível em tempo polinomial a P_2 . Como L pode ser qualquer linguagem de *NP*, mostramos que tudo que está em *NP* se reduz em tempo polinomial a P_2 ; isto é, P_2 é *NP-completo*.



- **Teorema 2:** Se algum problema *NP-completo* P estiver em P então $P=NP$.
- Ou seja, se qualquer problema de *NP* estiver em P , então todos os problemas de *NP* também estarão em P .
- **Prova:** Suponha que P seja *NP-completo* e ao mesmo tempo esteja em P . Então todas as linguagens L em *NP* se reduzem em tempo polinomial a P . Se P está em P , então L também está em P .
- Como se acredita que $P \neq NP$, ou seja, que existem muitos problemas em *NP* que não estão em P , consideramos uma prova de que um problema é *NP-completo* equivalente a uma prova de ele não ter nenhum algoritmo polinomial (i.e. não pertence a P) e, portanto, nenhuma boa solução por computador.

Teorema de Cook: O problema SAT é NP-completo.
(veja demonstração na bibliografia)

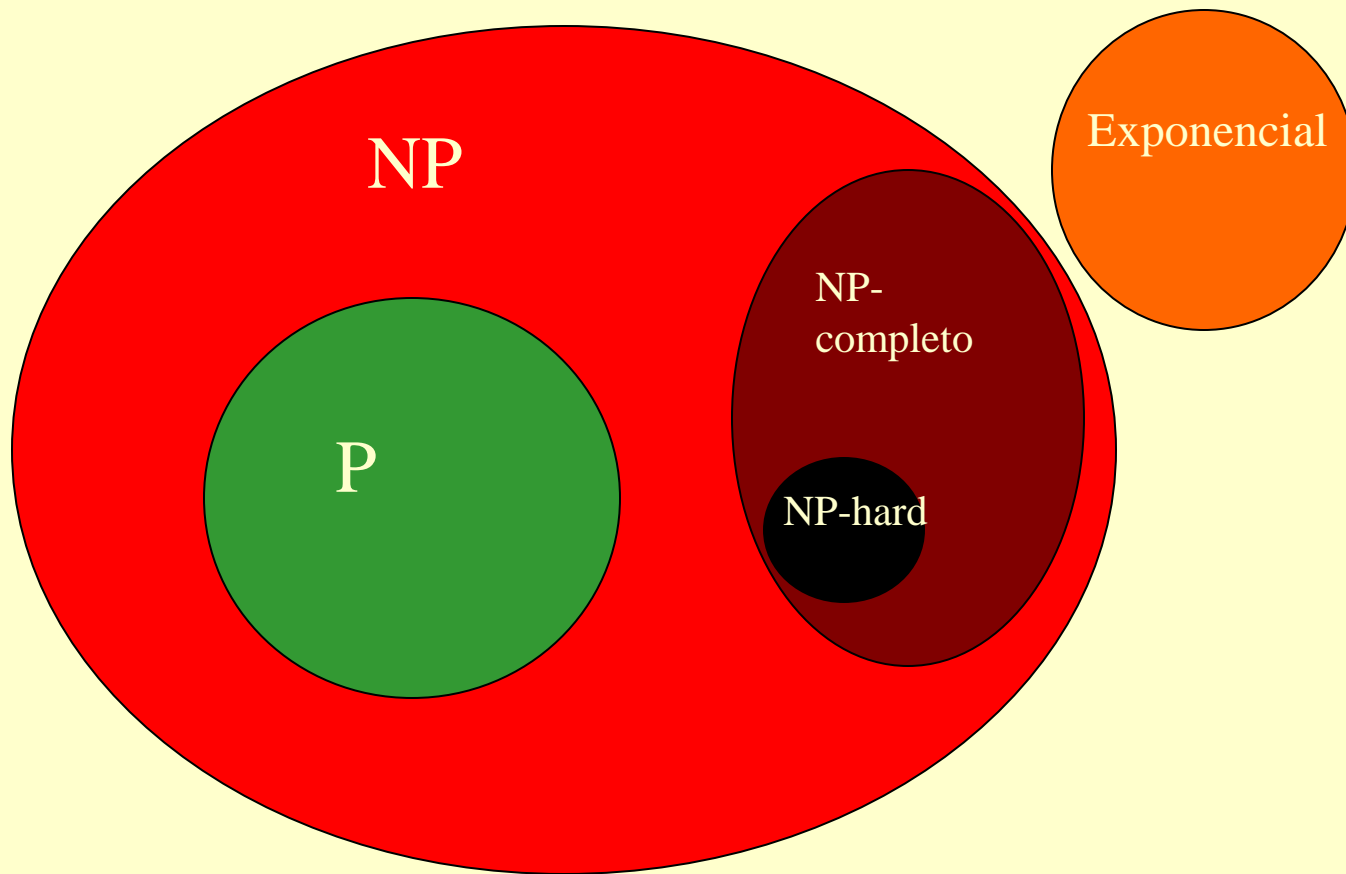
O problema SAT é usado para demonstrar que determinados problemas são *NP-completos*. Faz-se isso por meio da redução de SAT a esses problemas, e invocando o

Teorema 1: *Se P_1 é NP-completo, P_2 está em NP e existe uma redução de tempo polinomial de P_1 a P_2 , então P_2 é NP-completo.*

Problemas NP-difíceis (NP-hard)

- Alguns problemas L são tão difíceis que, embora possamos provar a condição (2) da definição de *NP-completo* (toda linguagem em *NP* se reduz a L em tempo polinomial), não podemos provar a condição (1): que L está em *NP* (*existe uma MTND*). Nesse caso, dizemos que L é *NP-difícil* (*NP-hard*) (pode-se usar o termo “intratável” no sentido de *NP-difícil*).
- Para provar que L é *NP-difícil*, é suficiente mostrar que L muito provavelmente exige tempo exponencial, ou pior.

Problemas comprovadamente Exponenciais



MTND que executa um número de passos exponencial em relação ao tamanho da entrada

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”.
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 num computador. 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.

Problemas sobre Grafos

NP-completos

- 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 $T_i < T_j$ 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 $T_i < T_j$, então T_i é atribuída a uma unidade de tempo anterior a T_j ?*

Resumo

- *As classes P e NP .* P consiste de todas as linguagens ou problemas aceitos por alguma MT que funciona em algum período de tempo polinomial, como uma função do comprimento de sua entrada. NP é a classe de linguagens ou problemas que são aceitos por MTND com um limite polinomial sobre o tempo que leva qualquer sequência de escolhas não-determinísticas.
- *A questão $P=NP$.* Não se sabe se P e NP são realmente as mesmas classes de linguagens, embora existam fortes suspeitas de que existem linguagens em NP que não estão em P .
- *Reduções de tempo polinomial.* Se podemos transformar instâncias de um problema em tempo polinomial em instâncias de um segundo problema que tem a mesma resposta – sim ou não – dizemos que o primeiro problema é redutível em tempo polinomial ao segundo.

Resumo

- *Problemas NP-completos:* Uma linguagem é *NP-completa* se está em *NP* e se existe uma redução de tempo polinomial de cada linguagem em *NP* à linguagem em questão. O fato de ninguém jamais ter encontrado um algoritmo de tempo polinomial para qualquer um dos milhares de problemas *NP-completos* conhecidos dá força à crença de que nenhum problema *NP-completo* está em *P*.
- *Problemas de satisfatibilidade NP-completos:* O Teorema de Cook mostrou o primeiro problema *NP-completo* – se uma expressão booleana é satisfatível – pela redução de todos os problemas em *NP* ao problema SAT em tempo polinomial.
- *Outros Problemas NP-completos:* Existe uma vasta coleção de problemas *NP-completos* conhecidos (caixeiro viajante, circuito hamiltoniano, cobertura de nós, etc.); provou-se que cada um deles é *NP-completo* por meio de uma redução de tempo polinomial de algum problema *NP-completo* conhecido.