

7. A Complexidade de Problemas

7.1 Introdução

- > Nada garante que existe sempre um algoritmo eficiente (?) que resolva um problema pertencente à classe R (i.e., um problema para o qual existe uma TM determinística que decide a linguagem que lhe corresponde);
- > Informalmente, pode-se dizer que um algoritmo é eficiente se ele usa uma quantidade razoável (?) de recursos necessários à boa realização dos cálculos.
- > Geralmente, a noção de recurso é sinônima de tempo e espaço (memória).
 - > Seria possível definir um tipo de limite sobre os recursos, independentemente da tecnologia existente ?
 - > Como formalizar tal conceito ?

- > A medição dos recursos requeridos por um algoritmo é dada em termos de uma função de complexidade;
- > Tal função exprime essa quantidade de recursos em termos do tamanho da instância do problema tratado (i.e., do tamanho da palavra que representa tal instância, no caso de uma TM);
- > Uma fronteira deverá então ser estabelecida entre uma função de complexidade aceitável e inaceitável ;
- > Sendo "c" uma constante, tal fronteira será definida por meio de funções do tipo polinomial (n^c) e funções de tipo exponencial (c^n).
- > Atenção: em certos casos, algoritmos do tipo polinomial (ou seja, que têm uma função de complexidade polinomial) podem ser ineficientes: n^{10000} , por exemplo!

- > Geralmente em computação, é a complexidade em tempo que é estudada;
- > A complexidade em espaço de memória é geralmente inferior à complexidade em tempo, já que o uso de cada unidade de memória requer pelo menos uma execução (1 unidade de tempo) de uma instrução de programação;
- > Isso significa que, em geral, a resolução do problema da complexidade de tempo resolve também o da complexidade de memória (além disso, este último se torna cada vez menos preocupante com as sucessivas reduções no custo das memórias);
- > Por outro lado, resolvendo o problema da complexidade em memória, não se resolve, necessariamente a complexidade em tempo.
- > Em certos casos, contudo, é indispensável analisar-se, também, a complexidade em espaço de memória.

> Os próximos slides mostram um exemplo de análise de complexidade de tempo e de memória de algoritmos de busca da Inteligência Artificial com expansão em largura e profundidade, respectivamente.

> Tais algoritmos podem resolver problemas como o dos canibais e missionários:

Três missionários e três canibais devem atravessar um rio com um barco que pode transportar no máximo duas pessoas. Além disso, as seguintes restrições devem ser respeitadas em todos os estados do problema: em cada uma das duas margens o número de missionários não pode ser inferior ao de canibais (caso contrário, os canibais comeriam os missionários); o barco não pode atravessar o rio por si só, sem pessoas a bordo.



So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of *these* generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d . In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d).$$

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

As for space complexity: for any kind of graph search, which stores every expanded node in the *explored* set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the *explored* set and $O(b^d)$ nodes in the frontier,

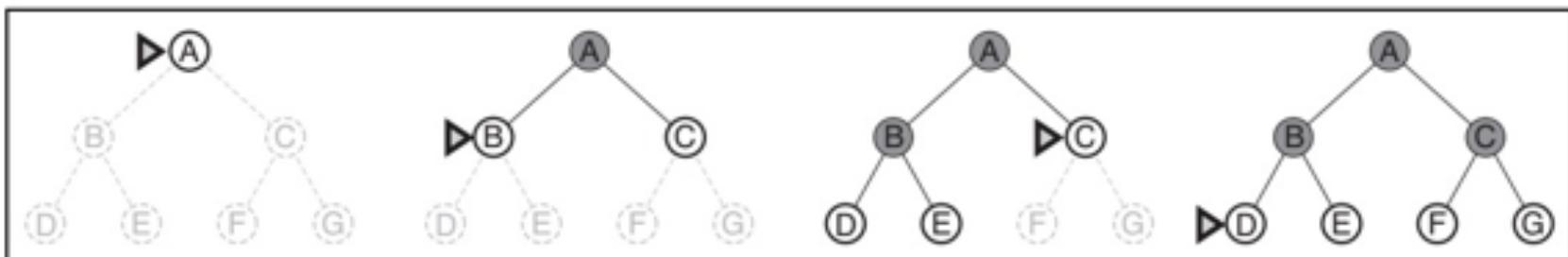


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.

An exponential complexity bound such as $O(b^d)$ is scary. Figure 3.13 shows why. It lists, for various values of the solution depth d , the time and memory required for a breadth-first search with branching factor $b = 10$. The table assumes that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

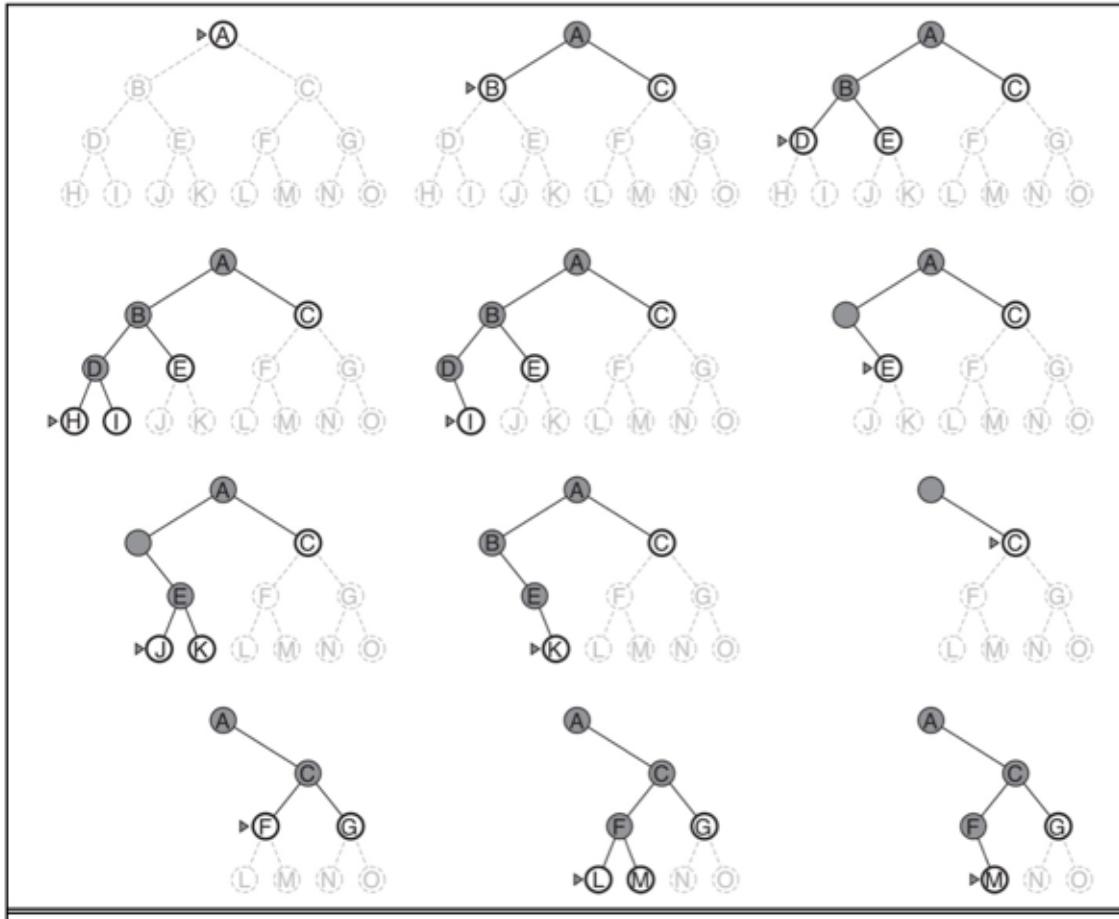


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space. Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it? The reason is the space complexity. For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. (See Figure 3.16.) For a state space with branching factor b and maximum depth m , depth-first search requires storage of only $O(bm)$ nodes. Using the same assumptions as for Figure 3.13 and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth $d = 16$, a factor of 7 trillion times less space. This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9). For the remainder of this section, we focus primarily on the tree-search version of depth-first search.

7.2 Como fazer a medição da complexidade de um problema e/ou de um algoritmo ?

- > A fim de abstrair uma avaliação da complexidade de tempo de um algoritmo independente de máquina, de linguagem de programação e de compilador, é plausível supor que o tempo de cálculo de um algoritmo sofre influência bem maior do tamanho dos dados tratados (medido pelo número de bits necessários à sua representação) do que do valor desses dados;
- > É possível que um algoritmo tenha tempo de execução diferente para dados distintos de mesmo tamanho n . Nesse caso, adota-se o maior dos tempos;
- > Assim sendo, o tempo de cálculo para dados de tamanho n corresponde ao tempo de cálculo máximo para dados de tal tamanho: análise no pior dos casos (worst case analysis).

- > A complexidade de tempo é geralmente expressa usando a notação O (ordem de grandeza) ;
- > Definição: Uma função de complexidade "g(n)" é "O(f(n))" se existem valores constantes c e n₀ tais que, para todo n > n₀, tem-se que:

$$g(n) \leq cf(n)$$

Tal definição não se estende ao comportamento das funções para valores pequenos de n, pois, nesses casos, a complexidade de tempo depende também de operações de inicialização (o que não é um problema, uma vez que valores pequenos de n normalmente não trazem inconveniente de complexidade). Logo, ela se aplica apenas aos casos de n elevado em que o tempo gasto com tais operações é desprezível e que o valor de n pode acarretar inconvenientes de complexidade.

Exemplos:

1) A função de complexidade "C . n²" (onde, obviamente, C > 0, pois a função se refere a "custo") é "O(n²)". De fato, assumindo n₀ = 1 e c = C, temos que, para todo n > 1:

$$(C \cdot n^2) \leq (C \cdot n^2);$$

2) A função de complexidade "C₁ . n² + C₂ . n" (onde C₁ e C₂ são valores positivos) é "O(n²)". De fato, assumindo n₀ = 1 e c = C₁ + C₂, temos que, para todo n > 1:

$$(C_1 \cdot n^2 + C_2 \cdot n) \leq (C_1 + C_2) \cdot n^2$$

> Que tipo de complexidade torna um algoritmo eficiente então ?

O(n) , O(n^2) , O(n^3) ?

> Não existe uma resposta clara, mas existem limites significativos que podem ser destacados:

- Uma complexidade de tipo exponencial $O(c^n)$, com $c > 1$, é quase sempre excessiva.

De fato, por exemplo, $2^{100} \approx 10^{30}$ representa um número excessivo!

Contudo, 100 não representa um valor excessivo (correspondendo, por exemplo, à quantidade de casas que um carteiro deverá visitar).

Neste caso, se o processamento de cada instrução elementar demandar 1 nanossegundo, por exemplo, no pior dos casos, uma resposta seria fornecida em mais ou menos $3 \cdot 10^{11}$ séculos !

- > Geralmente uma complexidade polinomial $O(n^k)$ para k constante representa uma complexidade de tempo aceitável;
- > Assim sendo, associa-se a noção de um algoritmo eficiente ao fato de esse algoritmo apresentar complexidade de tempo polinomial;
- > Pode-se questionar isso com base no fato, por exemplo, de que um algoritmo de complexidade polinomial $O(n^{10000})$ não é eficiente;
- > Contudo, os seguintes argumentos respaldam a ideia de delimitar a fronteira entre complexidade aceitável e inaceitável por meio do limite entre função polinomial e não polinomial: geralmente, na prática, o grau dos polinômios é inferior a 5; e, mesmo que seja questionável que "polinomial" equivale à eficácia, é inquestionável que "não polinomial" corresponde à ineficácia;
- > Assim sendo, é razoável aceitar que um problema para o qual não tenha sido encontrado um algoritmo de tipo polinomial não tenha um procedimento efetivo eficiente associado a ele.

7.3 Problemas de complexidade polinomial

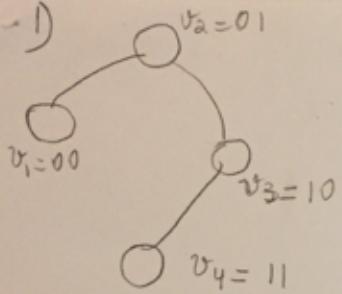
- > Também neste capítulo de Complexidade, consideramos apenas os problemas binários ou "de decisão" (cuja resposta é "sim" ou "não");
- > Conforme será mostrado, há uma correspondência entre as complexidades de problemas semelhantes mas que são propostos sob óticas diferentes, sendo um deles binário (ou "de decisão") e, o outro, não (por exemplo, o outro é "de otimização") - nesse caso, ambos terão o mesmo tipo de complexidade (polinomial ou exponencial);
- > Também aqui os problemas são representados pela linguagem de codificação de suas instâncias positivas;
- > O objetivo é então avaliar a complexidade dos problemas por meio das linguagens que os codificam;
- > Questão: variações nas codificações das instâncias positivas de um problema polinomial podem alterar a complexidade desse problema?
- > Felizmente, todas as codificações "naturais" de um problema polinomial diferem entre si de um fator polinomial.

EXEMPLO:

Considere um problema cujos dados são grafos G descritos por um conjunto V de vértices e um conjunto A de arcos não direcionados. Logo, $G = (A, V)$, onde:

- > A é um subconjunto de $V \times V$;
- > Os arcos (u,v) e (v, u) são iguais (não direcionados);
- > A seguir, dois exemplos de "codificação natural" desses dados (grafos) que, apesar de distintas, diferem entre si de um fator polinomial:

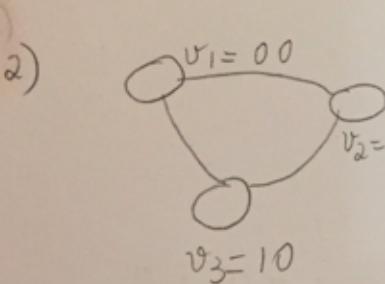
➤ Codificação 1: representa cada vértice por um inteiro binário de $\lceil \log_2(|V|) \rceil$ bits e cada arco por um par (v_i, v_j) , onde a notação $\lceil x \rceil$ corresponde ao menor inteiro superior a "x". A representação do grafo é então a lista de seus vértices seguida da lista de seus arcos. Logo, se $|V|=n$ e $|A|=m$, tal codificação tem um tamanho $O((n+m)\log_2(n))$. Exemplos:



$$A = m = 3; V = n = 4$$

$$G: ([\underbrace{00, 01, 10, 11}], [\underbrace{(v_1, v_2), (v_2, v_3), (v_3, v_4)}_{14 \text{ arcos}}])$$

$$O(3+4) \log_2 4 = 14$$



$$A = m = 3; V = n = 3$$

$$G: ([\underbrace{00, 01, 10}], [\underbrace{(v_1, v_2), (v_2, v_3), (v_3, v_1)}_{12 \text{ arcos}}])$$

$$O(3+3) \cdot \log_2 3 = 6 \cdot 2 = 12$$

> Codificação 2: expõe o número de vértices por meio de um inteiro binário de $\lceil \log_2 |V| \rceil$ bits e define os arcos por meio de uma matriz de incidência I tal que $I(i,j)=1$ se " (i,j) " é um arco do gráfico e $I(i,j)=0$, caso contrário. Tal matriz pode ser descrita por uma sequência de $|V|^2$ bits. Logo, se $|V|=n$ e $|A|=m$, tal codificação tem tamanho $O(n^2)$.

> O próximo "slide" ilustra a situações em que uma codificação "não natural" da linguagem que representa um problema polinomial pode converter-lo em um problema de complexidade exponencial:

- Codificação inadequada contendo "preenchimento" (padding):
Suponha-se um problema que em representações "naturais" de suas instâncias tenha complexidade polinomial $O(n)$.
Suponha que se faça uma codificação alternativa "não natural" e estúpida de tal problema em que se acrescente à representação de cada palavra de tamanho " n " correspondente a uma de suas instâncias uma sequência em que se repete " $2^n - n$ " vezes um mesmo símbolo desprovido de significado (codificação com "preenchimento").
- > Tal codificação "não natural" indesejavelmente transformaria a complexidade desse problema para o tipo exponencial ($O(2^n)$).

Além de evitar "preenchimentos" (paddings), uma codificação natural para produzir uma complexidade polinomial deve também respeitar os seguintes critérios:

- > deve ser possível decodificar a representação do problema em tempo polinomial. Para compreender tal restrição, usemos um exemplo extremo de codificação de um número binário onde cada dígito é representado por uma TM / M. Se M para sobre a palavra vazia, o dígito representado é "1". Caso contrário, "0". Apesar de a codificação ser perfumamente plausível (o número seria perfumamente definido por ela), não haveria um algoritmo capaz de produzir os dígitos a partir da codificação. Esse fato poderia comprometer a existência de uma solução polinomial para o problema.

> Os números não podem ser representados em base unária, mas, sim, em qualquer outra base ≥ 2 . De fato:

A relação entre o número de dígitos necessários para representar um número em base $a \geq 2$ e o número de dígitos necessários para representar o mesmo número em base $b \geq 2$ é, no máximo $\lceil \log_a(b) \rceil$, sendo limitado por uma constante.

Exemplos:

$$\underbrace{a=10}_{\text{a}=2 \therefore 2 \text{ dígitos}} = \underbrace{16_D}_{\text{b}=2 \therefore 5 \text{ dígitos}} = \underbrace{10000_B}_{\text{b}=5 \therefore 2 \text{ dígitos}} ; \frac{2}{5} = 0,4. \text{ De fato: } \lceil \log_{10} 2 \rceil = 1;$$

$$\underbrace{10000_B}_{\text{a}=2 \therefore 5 \text{ dígitos}} = \underbrace{16_D}_{\text{b}=5 \therefore 2 \text{ dígitos}} ; \frac{5}{2} = 2,5. \text{ De fato: } \lceil \log_2 10 \rceil = 4$$

Isto torna as bases ≥ 2 admissíveis. Contudo, um número de "n" dígitos em uma base " $a \geq 2$ " tem uma representação em uma base unária que pode comportar até " a^{n-1} " dígitos, o que apresenta complexidade exponencial.

Exemplos:

- Máximo decimal de 2 dígitos (i.e., "99") seria representado por " $10^2 - 1$ " = 99 sequências de dígitos "1" (cada um de seus antecessores é obtido cortando um desses dígitos "1", ou seja, 98_D seria uma sequência de 98 dígitos "1", ... 10_D seria sequência de 10 dígitos "1").

- Máximo decimal de 1 dígito (i.e., "9") : " $10^1 - 1$ " = 9 dígitos "1", ..., $1_D = 1$ dígito "1".

7.3.1 Complexidade de uma TM determinística

Definição: Seja uma TM determinística M que sempre para. A função de complexidade de tempo de M é definida por :

$$T_m(n) = \max \left\{ m \mid \exists x \in \Sigma^*, |x| = n \rightarrow \text{A EXECUÇÃO DE } M \text{ SOBRE } x \text{ TEM } m \text{ ETAPAS} \right\}$$

- > Tal função de complexidade fornece o máximo de etapas necessárias para que M decida uma palavra de comprimento n (pior caso);
- > Uma TM determinística M será de complexidade polinomial se sua função de complexidade tiver como borda superior um polinômio em n.

Definição: Uma TM determinística M é polinomial em tempo se existe um polinômio $p(n)$ tal que sua função de complexidade respeite a seguinte restrição:

$$T_M(n) \leq p(n)$$

para todo $n \geq 0$

A seguinte Tese pode então ser enunciada:

- > Se existe um algoritmo polinomial para resolver um problema, existe também uma TM determinística polinomial que decide a linguagem que codifica as instâncias positivas do problema.
- > A análise das transformações de um modelo computacional alternativo de uma TM determinística em um modelo de uma TM tradicional determinística que lhe seja equivalente mostra que tais transformações mantêm o caráter polinomial ou exponencial de ambos os modelos.

- > Por exemplo, sendo "c" uma constante, se existe uma máquina de memória a acesso direto (RAM) com função de complexidade de tempo $T(n)$ que reconhece uma linguagem, então existe também uma TM tradicional determinística de função de complexidade de tempo $(T(n))^c$ que aceita a mesma linguagem (conforme seção 5 .6 do arquivo TC-3-TM, lembre-se que o funcionamento da TM tradicional de fita única é mais complexo do que o de sua equivalente a múltiplas fitas que representa uma máquina de memória RAM - mas, ainda assim, se uma tem função de complexidade polinomial, a outra também o tem);
- > Note que, em função do valor de "c", poder existir uma diferença importante entre complexidades $T(n)$ e $(T(n))^c$. Contudo, se $T(n)$ tem complexidade polinomial, $(T(n))^c$ também tem.
- > Na prática isso mostra que a computação paralela pode diminuir a complexidade de algoritmos polinomiais complexos , mas não de algoritmos exponenciais !

7.3.2 A classe P

Definição: Uma classe é um conjunto de problemas que respeitam as restrições especificadas para aquela classe.

Definição: A classe P (polinomial) é a classe das linguagens (ou problemas) decididas por uma TM determinística polinomial.

Definição: Uma função é calculável em tempo polinomial se existe uma TM polinomial que a calcule (no sentido estabelecido pela definição apresentada na seção 5.9 do arquivo TC-3-TM).

7.4 As transformações polinomiais

- > Como mostrar que certos problemas não pertencem à classe P ?
- > Usando a noção de transformação polinomial!!
- > Antes de mostrar tais transformações, a seguir serão mostrados 2 exemplos de problemas para os quais não se conhece solução algorítmica polinomial.

7.4.1 Problema do caixeiro viajante (TS: Travelling Salesman): Versão "Problema de decisão"

- > É um problema que todas as evidências indicam não pertencer à classe P;
- > Considere um conjunto V de n cidades onde $d(v_i, v_j)$ representa a distância da cidade v_i até a cidade v_j ;
- > Considere também um valor constante b ;
- > O problema é de determinar se existe um percurso fechado (ciclo) cujo comprimento total seja menor ou igual a b (neste caso, é um "problema de decisão", pois a resposta é "sim" ou "não");
- > Trata-se então de encontrar uma permutação $v_{p_1}, v_{p_2}, v_{p_3}, \dots, v_{p_n}$ tal que :

$$\sum_{1 \leq i < n} d(v_{p_i}, v_{p_{i+1}}) + d(v_{p_n}, v_{p_1}) \leq b$$

- > Um procedimento efetivo para resolver tal problema seria: calcular o comprimento global do ciclo (circuito) de cada uma das permutações possíveis;
- > Como há "n!" permutações possíveis, tal algoritmo não é polinomial;
- > De fato, por exemplo: dobrando o valor de "n", a complexidade de tempo aumentaria exponencialmente, pois passaria a valer " $(2n)!$ " .

- > Essa versão binária ("de decisão") do problema do caixeiro viajante pode ser analisada por meio da análise de um problema parecido "de otimização": encontrar o mínimo ciclo possível para o caixeiro viajante;
- > Pode-se provar que se existe um algoritmo polinomial para o problema de otimização, então também existe um algoritmo polinomial para o problema de decisão !
- > De fato, seria imediata a obtenção de um algoritmo para o problema de decisão a partir de um algoritmo para o problema de otimização: bastaria resolver o problema de otimização e comparar o resultado " b_0 " obtido com o valor da constante " b " definida no problema de decisão (dependendo de o valor de " b_0 " ser maior ou menor, a resposta será, respectivamente, "não " ou "sim").

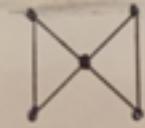
7.4.2 Problema do circuito Hamiltoniano - segundo exemplo de problema para o qual não se conhece solução polinomial:

> Considere um grafo $G = (V, A)$, sendo V o conjunto de n vértices e A o conjunto de arcos. O problema consiste em encontrar um percurso fechado (círculo) que passe por cada um dos vértices de V uma única vez.

Procedimento Efetivo para o Problema do Circuito Hamiltoniano

> Procedimento efetivo: procurar uma permutação $v_{p_1}, v_{p_2}, \dots, v_{p_n}$ de vértices tal que $(v_{p_i}, v_{p_{i+1}}) \in A$ para todo $1 \leq i < n$ e $(v_{p_n}, v_{p_1}) \in A$. Para tanto, um algoritmo que teste todas as permutações resolve o problema. Entendo, ele não é polinomial.

> Abaixo, as figuras à esquerda e à direita representam grafos que, respectivamente, "contém" e "não contém" circuito hamiltoniano.



7.4.3 Definição das transformações polinomiais

Definição: Seja uma linguagem $L_1 \subseteq \Sigma_1^*$ e uma linguagem $L_2 \subseteq \Sigma_2^*$. Uma transformação polinomial de L_1 em L_2 anotada $L_1 \propto L_2$ é uma função $f: \Sigma_1^* \rightarrow \Sigma_2^*$ que atende as seguintes restrições:

- (1) A transformação é calculável em tempo polinomial;
- (2) $f(x) \in L_2$ se, e somente se, $x \in L_1$.

- > As transformações polinomiais são também chamadas "reduções polinomiais";
- > No contexto de problemas, uma transformação é uma função f calculável em tempo polinomial que, a partir de uma instância " x " de um problema P_1 , calcula uma instância " $f(x)$ " de um problema P_2 tal que a instância " x " é positiva se, e somente se, a instância " $f(x)$ " é positiva;
- > É interessante comparar as técnicas de redução para demonstrar a indecidibilidade de problemas estudadas na seção 6.4.2 (onde um problema (ou linguagem) é reduzido a outro) com as técnicas de transformação usadas aqui no estudo de complexidade dos mesmos (onde um problema é transformado em direção a um outro): enquanto que as reduções devem ser calculáveis (recursivas), as transformações polinomiais devem ser calculáveis em tempo polinomial.

Exemplo - Existe uma transformação polinomial do problema do circuito hamiltoniano (HC) em direção ao problema do caixeiro viajante (TS):

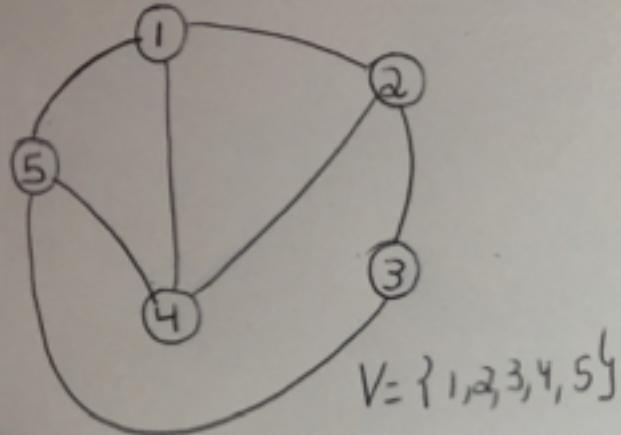
- > Provar que HC \leq TS;
- > Encontrar um algoritmo polinomial que transforma uma instância de HC em uma instância positiva de TS se, e somente se, a instância de HC é positiva.
- > A transformação (função "f") de uma instância $G = (V, A)$ de HC em uma instância $(C, d(v_i, v_j), b)$ de TS é dada a seguir:

- > O conjunto das cidades C é idêntico ao conjunto dos vértices do grafo V (ou seja, $C = V$);
- > Os comprimentos entre pares de cidades $d(v_i, v_j)$ são dados por:
 - * 1 se $(v_i, v_j) \in A$;
 - * 2 se $(v_i, v_j) \notin A$ (tornando impeditivos ao caixeiro viajante transitar por tais arcos, devido a seus elevados custos e ao baixo valor (de fato, mínimo possível) especificado abaixo para a constante "b");
- > O valor constante b é igual ao número de cidades, isto é, $b = |V| = \text{card}(V)$, o que corresponde à situação mais crítica.

EXEMPLO 1:

$HC \propto TS$

Instância "x_i" de HC (grafo)



Exemplos de permutações que comprovam que "x_i" é uma instância positiva de HC:

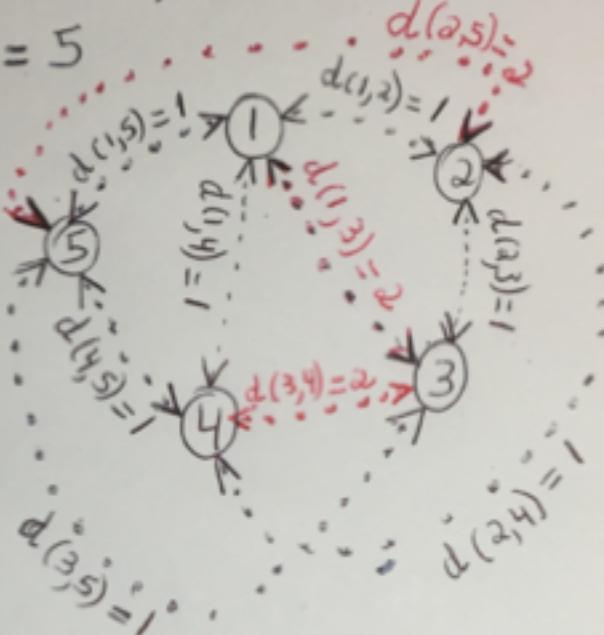
$v_3 v_5 v_4 v_1 v_2; v_1 v_2 v_3 v_4 v_5;$

$v_1 v_4 v_2 v_3 v_5$

Instância "f(x_i)" de TS (resultado da aplicação da função "f")

$$C = V = \{1, 2, 3, 4, 5\}$$

$$b = 5$$



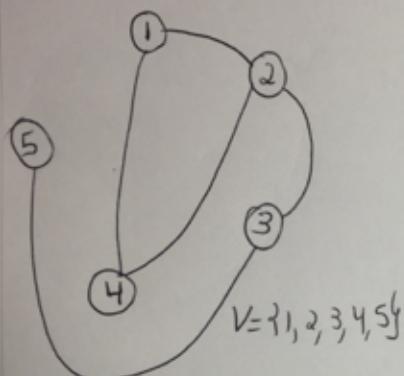
Exemplos de permutações que comprovam que "f(x_i)" é uma instância positiva de TS: $v_3 v_5 v_4 v_1 v_2; v_1 v_2 v_3 v_5 v_4; v_1 v_4 v_2 v_3 v_5$, para os quais:

$$\left(\sum_{1 \leq i \leq n} d(v_{pi}, v_{pi+1}) \right) + d(v_{pn}, v_{p1}) \leq 5$$

OBS: note que qualquer permutação em TS que apresente uma sequência " $v_i v_j$ ", tal que $(v_i, v_j) \notin A$ em HC, será rejeitada em TS, pois, neste caso, $d(v_i, v_j) = 2$ em TS e, consequentemente, o custo da permutação superará o limite de " $b = 5$ ". Obviamente, tal permutação será também rejeitada em HC.

EXEMPLO 2: $| HC \propto TS |$

Instância " x_2 " de HC (grafo):

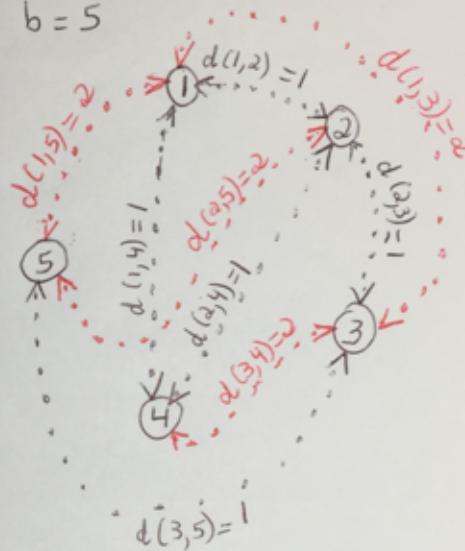


Não há permutação alguma que satisfaça as restrições de HC na instância " x_2 ". Logo " x_2 " é instância negativa de HC.

Instância " $f(x_2)$ " de TS (obtida pela transf. " f ")

$$C = V = \{1, 2, 3, 4, 5\}$$

$$b = 5$$



Não há permutação alguma que satisfaça, na instância " $f(x_2)$ ", a restrição de TS:

$$\left(\sum_{1 \leq i < n} d(v_{p_i}, v_{p_{i+1}}) \right) + d(v_{p_n}, v_{p_1}) \leq 5$$

Logo, " $f(x_2)$ " é instância negativa de TS.

Conclusão extraída da prova de que $HC \infty TS$:

- > Essa prova anterior de que existe uma transformada polinomial "f" que leva HC em direção de TS (ou seja, " $HC \infty TS$ ") permite concluir que, caso se encontre um dia um algoritmo A2 que resolva TS em tempo polinomial, então haverá também um algoritmo A3 que resolverá HC em tempo polinomial.
- > De fato, assumindo que A1 seja o algoritmo polinomial que processa tal função "f", A3 pode ser definido em dois passos:
 - 1) Aplica A1 a uma instância "x" de HC, gerando a instância "f(x)" de TS;
 - 2) Aplica A2 a "f(x)": o resultado indica se "f(x)" é ou não instância de TS. Caso seja, então "x" é instância de CH. Caso contrário, "x" não é instância de CH.

> Resumindo: Qual é a motivação para efetuar tal transformação ?

> Se a transformação $HC \propto TS$ é polinomial e se existe um algoritmo polinomial para resolver TS, então existe também um algoritmo polinomial para resolver HC;

> A transformação apresentada é claramente polinomial (o aumento de tamanho da primeira estrutura que representa HC deve implicar num aumento de mesma proporção na estrutura que representa TS);

> É evidente que se uma instância de HC é positiva, então a instância correspondente em TS também será positiva;

> Da mesma forma, se uma instância de TS é positiva, então a instância correspondente em HC também será positiva;

> O mesmo acontecerá para as instâncias negativas.

Então podemos afirmar:

$$HC \propto TS$$

Podemos provar também que $TS \propto HC$. Neste caso, a prova é menos trivial do que para a transformação $HC \propto TS$.

7.4.4 Propriedades das transformações polinomiais

Lema: Se $L_1 \leqslant L_2$ então:

- se $L_2 \in P$ então $L_1 \in P$
- se $L_1 \notin P$ então $L_2 \notin P$

> Prova: se existe um algoritmo polinomial que decide L_2 então a redução $L_1 \leqslant L_2$ produz um algoritmo polinomial que decide L_1 . A segunda conclusão pode ser provada por contradição : suponha que $L_1 \notin P$ e $L_2 \in P$; então temos uma contradição, já que se $L_2 \in P$ então $L_1 \in P$ também (decorrente da relação de "necessidade" existente na primeira implicação do lema).

> O lema acima e as transformações $HC \leqslant TS$ e $TS \leqslant HC$ permitem deduzir que $HC \in P$ se, e somente se, $TS \in P$.

Lema: as transformações polinomiais são transitivas:

Se $L_1 \propto L_2$ e $L_2 \propto L_3$ então $L_1 \propto L_3$

7.4.5 Problemas polinomialmente equivalentes

Definição: duas linguagens L_1 e L_2 são polinomialmente equivalentes se e somente se $L_1 \leq_p L_2$ e $L_2 \leq_p L_1$

$$L_1 \equiv_p L_2$$

- > A relação de equivalência polinomial entre as linguagens é transitiva, reflexiva e simétrica, permitindo assim definir "classes de equivalências polinomiais" sobre as linguagens.
- > Uma classe de equivalência polinomial é uma classe em que todo par formado por linguagens L_i e L_j que lhe pertencem é polinomialmente equivalente;
- > Logo, ou TODAS as linguagens de uma classe de equivalência têm uma solução polinomial ou, então, NENHUMA delas possui.

- > Por exemplo, HC e TS pertencem a uma mesma classe de equivalência polinomial (denominada classe NPC, conforme será visto oportunamente);
- > Para demonstrar que um outro problema X pertence à mesma classe de equivalência polinomial NPC, é necessário demonstrar que:

$HC \propto X$ ou $TS \propto X$

e

$X \propto HC$ ou $X \propto TS$

> Na prática, percebe-se que a classe de Problemas polinomialmente equivalentes a HC e TS contém numerosos problemas para os quais não foram encontradas soluções de complexidade polinomial.

> O que se pode deduzir de tal fato ?

7.5 A Classe NP

- > A ineficiência dos algoritmos conhecidos para resolver HC deve-se à quantidade de permutações (todas) a serem testadas e, não, ao processamento individual de cada uma delas, uma vez que o teste para checar se uma instância é ou não um circuito hamiltoniano é bastante rápido.
- > HC e os demais problemas que lhe são polinomialmente equivalentes, todos caracterizados pelo fato de o número de casos a explorar é que torna a solução algorítmica do problema ineficaz, permitem a definição abstrata da classe NPC que os engloba;
- > Para exprimir a complexidade de tais problemas, pode-se dizer que, caso a enumeração dos casos a serem tratados em cada instância (por exemplo, enumeração das permutações de uma instância de um grafo HC) nada custe, eles têm uma solução eficaz;
- > O interesse desse ponto de vista é que ele tem uma formalização direta por meio de uma TM não determinística (TM-ND) - ver seção 5.7 do arquivo TM-3;
- > Por exemplo, no caso de HC, podemos fazer corresponder cada permutação a ser examinada para resolver uma dada instância a uma execução de uma TM-ND (execução não determinística). Cada permutação é aceita unicamente se ela define um circuito hamiltoniano para aquela instância. Como a execução de tal TM-ND é curta em função do tamanho de suas instâncias, neste sentido, pode-se dizer que ela é eficaz;
- > Como dar uma definição abstrata de tal classe ?

7.5.1 Complexidade das TMs não determinísticas

Definição: o tempo de cálculo de uma TM-ND em uma palavra w é dado:

- pelo comprimento mais curto da execução que aceita a palavra w de entrada (caso a TM-ND a aceite);
 - pelo comprimento "1", caso a TM-ND não aceite a palavra w. A escolha desse valor "1" tem como objetivo não levar em consideração o comprimento da execução das palavras não aceitas pela TM-ND, o que permite definir a complexidade de uma TM-ND de forma análoga à de uma TM determinística.
- > Justifica-se a desconsideração das etapas de processamento de palavras "não aceitas" no cálculo de complexidade de uma TM-ND apresentado a seguir pelo fato de não existirem linguagens decididas por tais máquinas (apenas aceitas).

Definição: seja M uma TM-ND. A complexidade em tempo de M é dada pela função :

$$T_M(m) =$$

$$\max \{ m \mid \exists x \in \Sigma^*, |x| = m \\ \text{e o tempo de cálculo} \\ \text{de } M \text{ sobre } x \text{ é } m \}$$

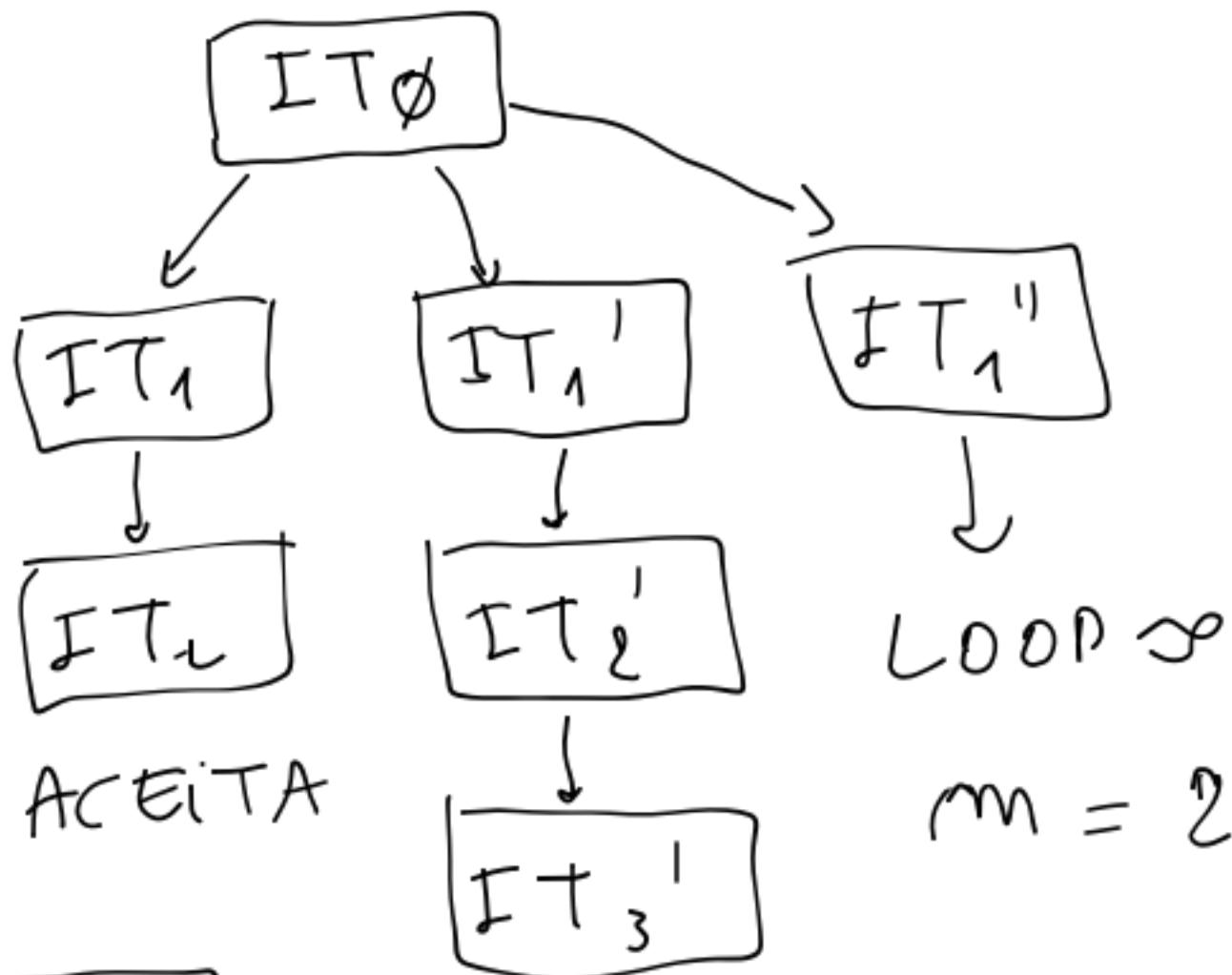
> Isso significa que, considerando as palavras de comprimento n, a complexidade de tempo de M para executá-las corresponde ao máximo "tempo de cálculo" dentre todos os que correspondem a tais palavras (ver definição de "tempo de cálculo" no slide anterior).

> O próximo slide mostra um exemplo desse cálculo em uma situação em que são mostradas todas as execuções possíveis das 3 únicas palavras que têm tamanho "n" (w1, w2 e w3). No caso:

$$T_M(n) = 3$$

Ex :

W_1



W_2

IT_\emptyset

ACEITA

IT_1

IT_1'

W_3

IT_\emptyset

$m=1$

LOOP ∞

$T_m(n) = 3$

IT_2

LOOP ∞

IT_3

ACEITA

$m = 3$

7.5.2 Definição da classe NP

Definição: a classe NP (Não Determinístico Polinomial) é a classe das linguagens aceitas por uma MT-ND de complexidade polinomial .

Significa que a função :

$$\overline{T}_M(n) \leq P(n)$$

> Os problemas HC e TS pertencem à classe NP (mais especificamente, à classe de equivalência polinomial NPC contida em NP), conforme demonstrado por meio do algoritmo não determinístico polinomial que os resolve apresentado no próximo slide.

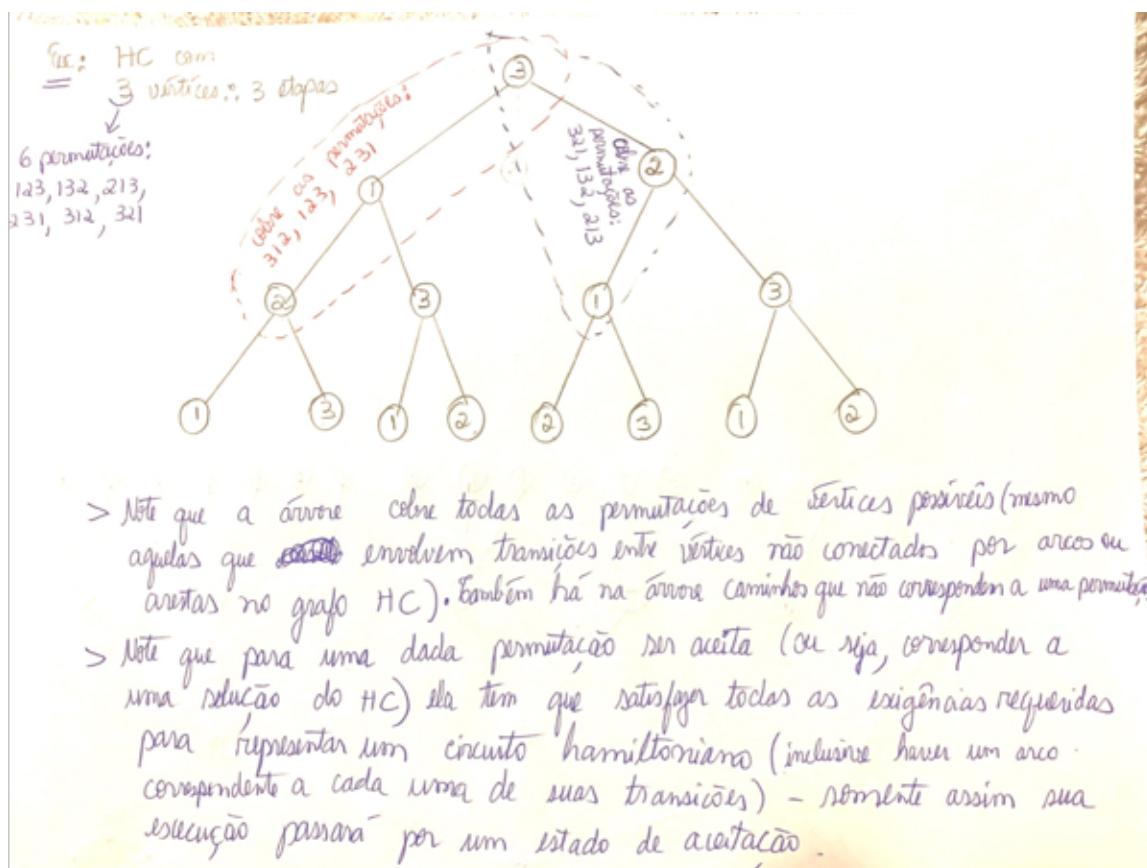
As duas fases seguintes de um algoritmo não determinístico resolvem HC em tempo polinomial:

- 1) O algoritmo gera de modo não determinístico uma permutação dos V vértices dos grafos. Intuitivamente, isso corresponde a gerar, em expansão em largura e paralelamente, uma árvore com raiz em um dos vértices V - sendo que a expansão de cada vértice produz os vértices remanescentes. Tal fase é polinomial, pois comporta um número de etapas proporcional ao número de vértices da instância. A permutação é aceita se a MT-ND passa por um estado de aceitação ao processá-la;
- 2) O algoritmo verifica de modo não determinístico (checando em paralelo cada nível) se a permutação gerada corresponde a um circuito Hamiltoniano, o que é também um procedimento polinomial (proporcional ao número de vértices);

> Note que a enumeração individual e sequencial de cada permutação em uma TM determinística impede uma solução polinomial, ao passo que a enumeração única feita por uma TM-ND torna HC polinomial.

> Lembremos que os problemas NP são decidíveis!

Exemplo de expansão para HC com 3 vértices:



- > A classe NP é definida em termo de linguagens aceitas por TMs-NDs ;
- > Tais linguagens são também decididas por TMs determinísticas de complexidade exponencial !
- > Teorema: Se L é uma linguagem que pertence à NP então existe uma TM determinística M e um polinômio $p(n)$ tal que M decide L e M tem a seguinte complexidade de tempo:

$$\overline{T}_M(n) \leq 2^{p(n)}$$

- > Tal teorema mostra que qualquer problema da classe NP pode ser decidido por um algoritmo exponencial.
- > Será que existe um algoritmo polinomial que decide qualquer linguagem de NP ($P=NP ?$) ?
Tal questão permanece sem resposta até hoje !

7.5.3 Estrutura da classe NP

Definição : Uma classe de equivalência polinomial C_1 é inferior a uma classe de equivalência polinomial C_2 (" $C_1 \leq C_2$ ") se existe uma transformação polinomial de toda linguagem de C_1 em direção a toda linguagem de C_2 ;

- > Pela transitividade das transformações polinomiais, $C_1 \leq C_2$ se, e somente se, existem $L_1 \in C_1$ e $L_2 \in C_2$ tais que:
 $L_1 \propto L_2$;
- > Em termos de complexidade, isso significa que a classe C_1 é "menos difícil" do que a classe C_2 . De fato, se existir uma solução polinomial para as linguagens de C_2 , também existirá para as linguagens de C_1 .

Provando que $C_1 \leq C_2$ se, e somente se, existem $L_i^{C_1} \in C_1$ e $L_j^{C_2} \in C_2$ tais que $L_i^{C_1} \alpha L_j^{C_2}$ (nó de anterior):

1º Passo: Se $\left(\exists L_i^{C_1} \in C_1 \text{ e } L_j^{C_2} \in C_2 \mid L_i^{C_1} \alpha L_j^{C_2} \right) \rightarrow C_1 \leq C_2$

H: hipótese

C: consequente

Como C_1 e C_2 são classes de equivalência polinomial.:

- de C_1 : $\forall L_i^{C_1}, \forall L_j^{C_1}, L_i^{C_1} \alpha L_j^{C_1} \text{ e } L_j^{C_1} \alpha L_i^{C_1}$ (1)

- de C_2 : $\forall L_i^{C_2}, \forall L_j^{C_2}, L_i^{C_2} \alpha L_j^{C_2} \text{ e } L_j^{C_2} \alpha L_i^{C_2}$ (2)

- de H: $L_i^{C_1} \alpha L_j^{C_2}$ (3)

- de (2): $\forall j, L_j^{C_2} \alpha L_j^{C_2}$ (4)

- de (3) e (4) (pela transitividade): $\forall j, L_i^{C_1} \alpha L_j^{C_2}$ (5)

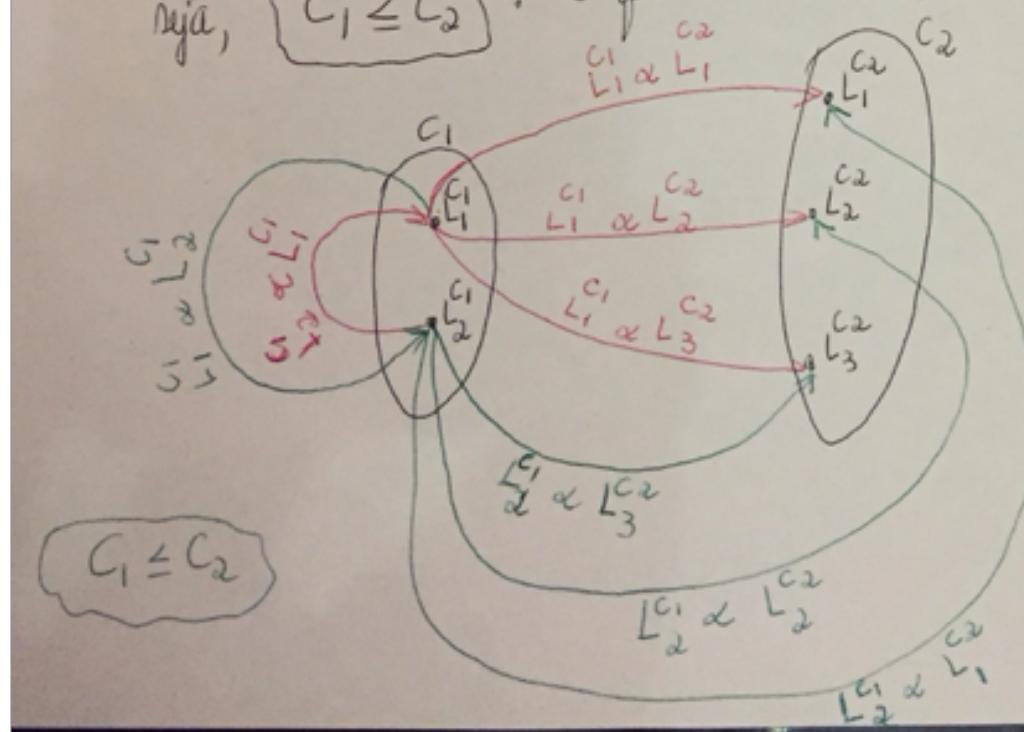
- de (1): ($\forall i$), $L_i^{C_1} \alpha L_i^{C_1}$ (6)

- de (6) e (5) (pela transitividade): $\forall i, \forall j, L_i^{C_1} \alpha L_j^{C_2}$, ou

- de (6) e (5) (exemplo):

Reja, $C_1 \leq C_2$.

Graficamente (exemplo):



EFETUE A
PROVA DO

2º PASSO

DA

Bi-IMPLICAÇÃO

> Dentre as classes de equivalência polinomial de NP, particularmente duas nos interessarão: P, a menor (mais fácil), e NPC, a maior (mais difícil).

7.5.3.1 Classe P (Mais fácil de NP):

> Para provar que P é a classe mais fácil de NP, é preciso mostrar que: $P \subseteq NP$; P é uma classe de equivalência polinomial; que P é a mais fácil.

Lema 1: A classe NP contém a classe P ($P \subseteq NP$).

Prova: uma TM determinística polinomial é um caso particular de uma TM-ND polinomial.

Lema 2: A classe P é uma classe de equivalência polinomial.

Significa que, para todo $L_1, L_2 \in P$, $L_1 \propto L_2$.

Prova: a transformação requerida corresponde ao seguinte algoritmo aplicado a uma palavra w:

1) Checar se $w \in L_1$:

Existe um algoritmo polinomial para verificar isso, já que $L_1 \in P$

2) Caso $w \in L_1$, a transformação polinomial produz uma palavra $w' \in L_2$ (tal palavra pode ser a mesma para toda palavra $w \in L_1$);

Caso $w \notin L_1$, a transformação polinomial produz ~~uma~~ palavra $w' \notin L_2$ (tal palavra pode ser a mesma para toda palavra $w \notin L_1$)

> OBS: note que tal transformação não poderia ser aplicada para provar que $HC \propto TS$, pois não se conhece algoritmo polinomial para resolver o seu primeiro passo !!

Lema 3: Para toda $L_1 \in P$ e para toda $L_2 \in NP$:
 $L_1 \propto L_2$.

- > Prova: análoga à do Lema 2;
- > O lema 3 implica diretamente que, para toda classe de equivalência polinomial $C_1 \subseteq NP$, $P \leq C_1$, ou seja, P é a classe "mais fácil" de NP ;
- > Estudaremos agora a classe "mais difícil" de NP : a **NP-Completa**.

7.5.3.2 Classe NP-Completa (NPC): Classe de equivalência polinomial mais difícil de NP:

> A classe a mais difícil de NP é a classe NPC

Definição: uma linguagem L é NPC se, e somente se:

1) $L \in \text{NP}$ (L é aceita por uma TM-ND polinomial);

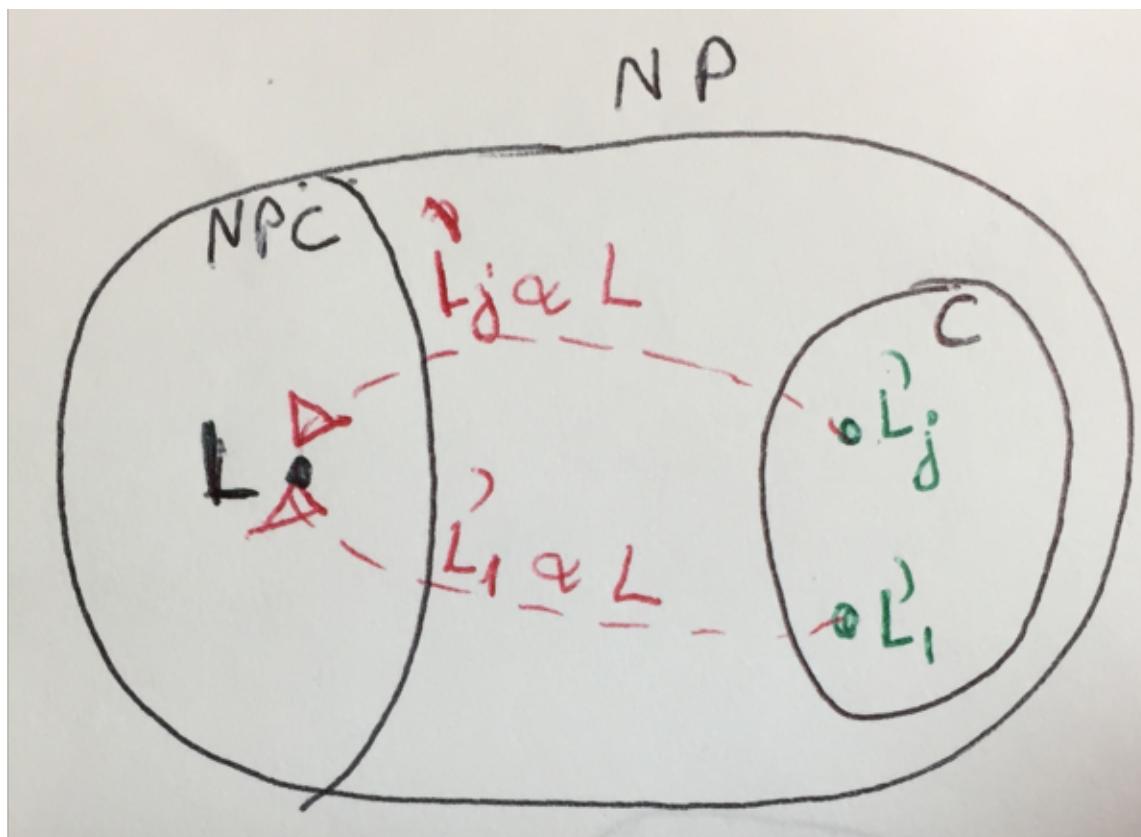
2) Para toda linguagem $L' \in \text{NP}$, $L' \propto L$.

> Logo, NPC é uma classe de equivalência polinomial. De fato, pela definição, qualquer que seja o par L_i, L_j de linguagens NPC, temos que: a) Para toda linguagem $L' \in \text{NP}$ (inclusive L_j), $L' \propto L_i$, ou seja, temos também $L_j \propto L_i$; b) Para toda linguagem $L' \in \text{NP}$ (inclusive L_i), $L' \propto L_j$, ou seja, temos também $L_i \propto L_j$. De a) e b) temos que L_i e L_j são polinomialmente equivalentes, o que faz a classe NPC ser de equivalência polinomial.

> Essa definição de que uma linguagem L de NP é NPC se, para toda $L' \in \text{NP}$, $L' \leq L$, também implica que:

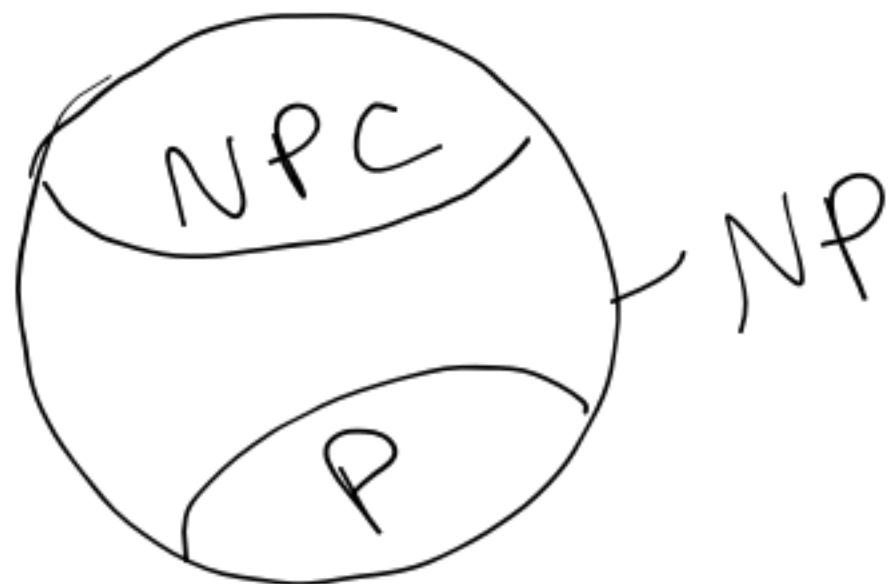
> Para qualquer outra classe de equivalência polinomial C de NP que não pertença à NPC, temos $C \leq \text{NPC}$.

De fato, sendo L pertencente a NPC, como C está contida em NP, pela definição de NPC, para qualquer linguagem L' de C , $L' \leq L$. Isso significa que $C \leq \text{NPC}$:



- > Teorema : Se existe uma linguagem NPC L decidida por um algoritmo polinomial, então toda linguagem de NP é decidida em tempo polinomial, ou seja, $P = NP$ (decorrente da definição de NPC).
- > Pela implicação lógica do teorema, então, " $P = NP$ " é condição necessária para que exista uma linguagem de NPC que seja decidida por um algoritmo polinomial (nesse caso, como NPC é uma classe de equivalência polinomial, todas suas linguagens o seriam);
- > Consequência do teorema: problemas NPC somente terão solução polinomial se $P = NP$. Assim sendo, para demonstrar que problemas NPC não têm solução polinomial, basta provar que $P \neq NP$!!

- > Não foi provado até hoje se: $P = NP$ ou $P \neq NP$;
- > Contudo, todas as evidências apontam que $P \neq NP$;
- > Já foi provado que se $P \neq NP$ então existem problemas em NP que não são nem em P e nem em NPC .

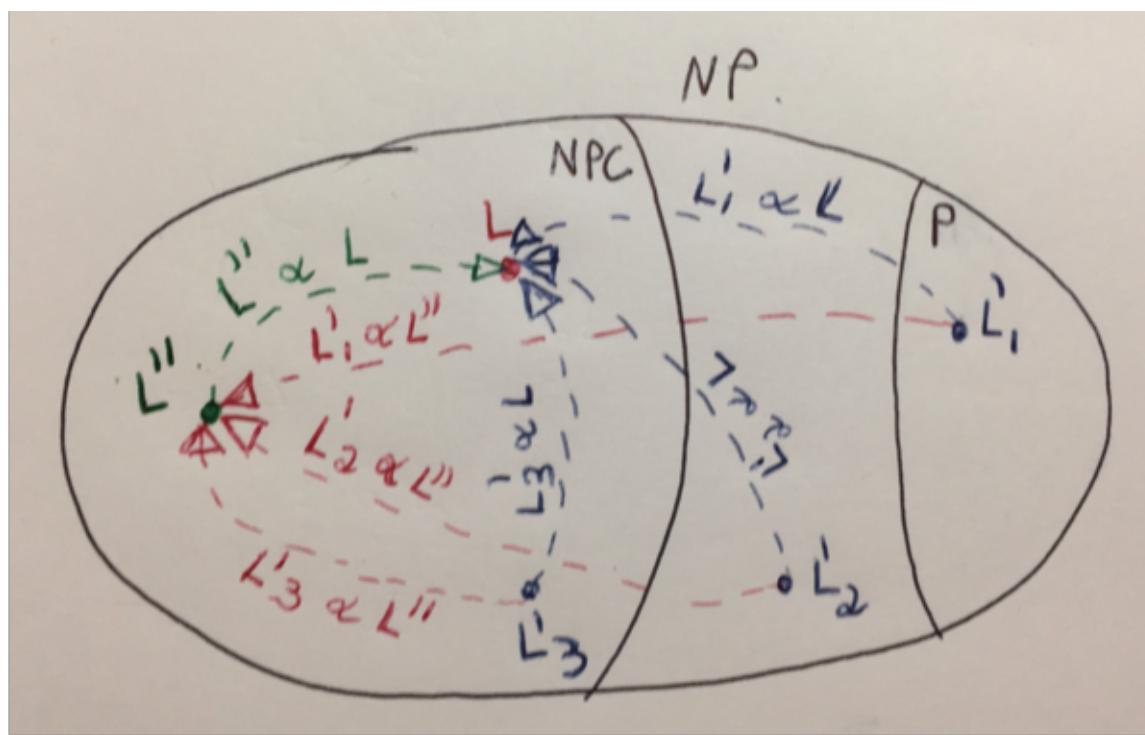


- > Na prática, considerando a hipótese $P \neq NP$, demonstrar que um problema é NPC permite concluir que ele não tem solução polinomial;
- > Tal argumento será usado para estabelecer que problemas tais como HC e TS, de fato, parecem não ter solução polinomial (particularmente, HC e TS são exemplos de NPC).

7.5.5 Provar a NP-Completude

- > Para provar que uma linguagem L é NP-Completa, deve-se mostrar o seguinte:
 - 1) que a linguagem L pertence à classe NP;
 - 2) que para toda linguagem L' de NP, $L' \leq L$.
- > Um algoritmo não determinístico polinomial para aceitar L basta para mostrar a primeira propriedade, ou seja, que uma linguagem pertence à NP;
- > Provar a segunda propriedade é mais difícil, a menos que se conheça um outro problema L'' pertencente a NPC e que se mostre que $L'' \leq L$ (tendo-se provado, antes, que L pertence a NP, o que corresponde à primeira propriedade)....
- > Na prática, a transformação polinomial a ser encontrada leva do problema NPC L'' conhecido para o novo problema NPC L , conforme mostrado a seguir !

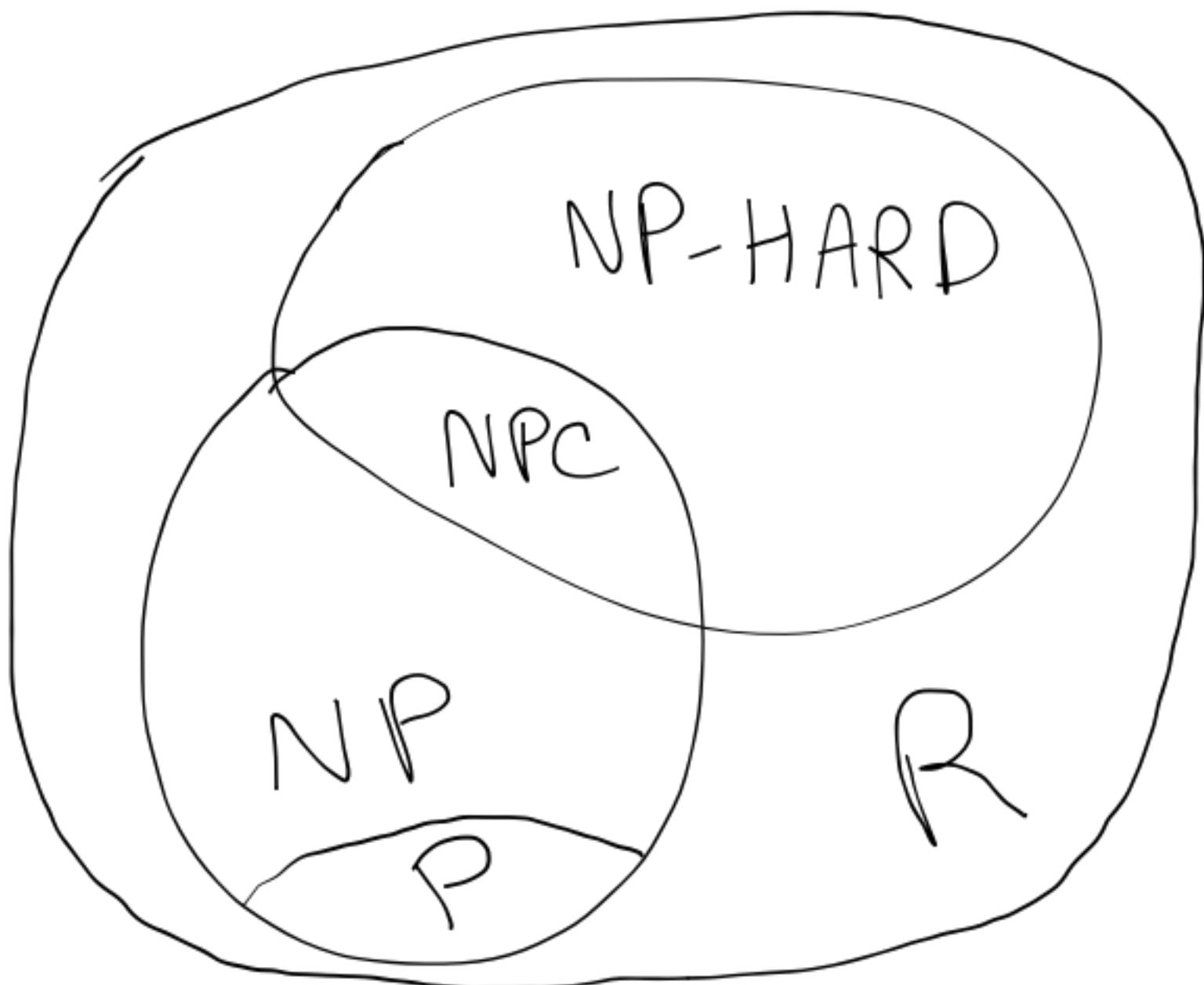
- > Provando a segunda propriedade a partir de uma linguagem L'' comprovadamente pertencente a NPC:
- > Segunda propriedade: sendo L uma linguagem pertencente a NP, caso, para toda $L' \in NP$, $L' \propto L$, então $L \in NPC$.
- > De fato, por exemplo, se L_1', L_2' e $L_3' \in NP$, como $L'' \in NPC$, então: $L_1' \propto L''$, $L_2' \propto L''$ e $L_3' \propto L''$. Caso se prove que $L'' \propto L$, então, pela transitividade de " \propto ", tem-se: $L_1' \propto L$, $L_2' \propto L$ e $L_3' \propto L$, conforme figura abaixo:



7.5.4 A classe NP-Hard (difícil): uma Classe com intersecção com a Classe NP

- > Considerando a hipótese $P \neq NP$, na prática, para mostrar que uma linguagem L não é reconhecida por um algoritmo polinomial, é só mostrar que para toda linguagem L' de NP (inclusive pertencente a NPC), temos $L' \in L$, sem que, necessariamente, $L \in NP$;
- > Neste caso, fala-se de uma linguagem que pertence à classe NP-Hard !
- > Qual é então a diferença entre as classes NPC e NP-Hard ?
- > Parece que nenhuma linguagem pertencente a uma classe ou à outra tem solução polinomial;
- > Mas um problema da classe NP-Hard pode ser ainda mais difícil a resolver que um problema da classe NPC;
- > Significa que existem problemas decidíveis NP-Hard que não são tratáveis em tempo polinomial por nenhuma TM não determinística polinomial !

Definição alternativa da classe NPC: a classe NPC é a intersecção da classe NP com a classe NP-Hard



7.5.5 Problemas NPC conhecidos

Um primeiro problema NP-completo (SAT):

Teorema de Cook: a satisfabilidade das fórmulas do cálculo das proposições na forma normal conjuntiva é NPC

A forma normal conjuntiva é uma expressão da forma: $E_1 \wedge E_2 \wedge \dots \wedge E_i \wedge \dots \wedge E_k$ onde E_i é uma cláusula

Uma cláusula é uma disjunção de variáveis proposicionais ou de negação de variáveis proposicionais da forma:

$x_1 \vee \dots \vee x_j \vee \dots \vee x_k$ onde x_j é uma expressão da forma p ou $\neg p$

Ex: $(p \vee \neg q) \wedge p \wedge (q \vee \neg r \vee s \vee \neg t) \wedge (s \vee t)$

- > Expressão válida: verdadeira qualquer que seja a interpretação das variáveis.
- > Expressão satisfável: existe pelo menos uma função de interpretação que a torna verdadeira.

> Prova de que SAT é NPC:

A) Mostrar que $SAT \in NP$

1) gerar de modo não determinístico uma função de interpretação

2) verificar se tal função torna a fórmula verdadeira

B) Mostrar que existe uma transformação polinomial de qualquer linguagem de NP para a linguagem das instâncias positivas de SAT : com isso se prova que SAT é NP-HARD. Logo, como em "A" se provou que SAT é NP, conclui-se que SAT é NPC.

> Note que, como SAT foi o primeiro problema demonstrado como sendo NPC, tal prova não tinha como ter sido feita a partir de um outro problema NPC conhecido. Tal demonstração, então, abriu frentes de prova para outros NPCs.

> HC é NPC (provado a partir da prova de que TS é NPC !), conforme resumo abaixo:

- Provou-se que HC é NPC a partir da prova de que VC \propto HC , onde VC é o problema "Vértice Cover", previamente demonstrado como pertencente à NPC:

- Problema VC: determinar se existe um subconjunto de vértices de um grafo, de tamanho maximo j, que cubra todos os arcos do grafo (cada arco tenha pelo menos uma extremidade no subconjunto);

- Provou-se que VC é NPC a partir da demonstração de que o problema 3-SAT (caso particular do SAT em que cada cláusula contém, exatamente, 3 literais) é NPC . Por sua vez, a prova de que 3-SAT é NPC foi efetuada a partir de SAT.

> O Problema da programação inteira é NPC

- Dados do problema:

- (1) Um conjunto de m pares (v_i, d_i) , onde: cada v_i é um vetor de inteiros de tamanho n e cada d_i é um número inteiro;
- (2) um vetor d de tamanho n;
- (3) uma constante b.

- Questão: Determinar se existe um vetor de inteiros y de tamanho n, tal que " $y \leq v_i \leq d_i$, para $1 \leq i \leq m$, e tal que " $\sum y \geq b$ ".

> Saliente-se que o mesmo problema da programação inteira em que se procura por um vetor y de racionais (denominado "programação linear") é P !

- Somente em 1979 foi encontrado um algoritmo polinomial para o problema da programação linear. Até então, o algoritmo usado para resolvê-lo era o método simplex, de complexidade exponencial (que, na prática, provê bons resultados).

> O problema do número cromático de um grafo (determinar se para um grafo G é uma constante k é possível colorir os vértices do grafo usando k cores distintas de modo que cada 2 vértices ligados por um arco tenham cores distintas) é NPC.

> O problema da equivalência de autômatos finitos não determinísticos é NP-Hard ! Não foi encontrado um algoritmo não determinístico de complexidade polinomial para resolver tal problema. É um problema completo (conforme seção 7.5.3.2) em uma classe chamada PSPACE (classe de complexidade em espaço) a ser vista.

7.5.6 Interpretação da NP-Completude

- > Na prática, ATUALMENTE, mostrar que um problema é NPC ou NP-Hard é a mesma coisa que mostrar que, sob a hipótese de que $P \neq NP$, ele não tem solução polinomial.
- > Isso não quer dizer que devemos desistir de uma solução algorítmica para o problema? NÃO, pois há muitas ferramentas computacionais eficazes para resolver problemas NPC. Além disso, a complexidade levantada se refere ao pior caso, que não é, necessariamente, o mais frequente.
- > A análise do problema SAT no próximo slide ilustra isso.

> Atenuantes da complexidade do problema SAT: uma fórmula que tem muitas variáveis e poucas cláusulas é quase sempre satisfável. Por outro lado, uma fórmula que tem muitas cláusulas e poucas variáveis é quase sempre insatisfável - Nestes duas situações, é geralmente rápido de decidir a satisfatibilidade das fórmulas. As instâncias de SAT difíceis de ser tratadas são aquelas que ficam nas bordas do problema (nos limites da satisfatibilidade).

7.5.7 NPC e tecnologia

- > Em relação ao paralelismo, o uso de n processadores, no melhor dos casos, pode reduzir o tempo de cálculo de um fator n.
- > Para um algoritmo de complexidade exponencial, o número de processadores teria de ser também exponencial em função do tamanho das instâncias a serem testadas, o que não seria realista;
- > Mesmo para baixas complexidades, nem sempre o paralelismo torna a resolução mais rápida, conforme analisado a seguir por meio da definição da classe de complexidade NC:
- > Definição: A classe NC é a classe das linguagens que podem ser reconhecidas em tempo $O(\log^c(n))$, ou seja, têm complexidade em $\log(n)$, sobre uma máquina comportando $O(n^k)$ processadores (i.e., máquinas com número de processadores polinomial em n).

- > Note que o modelo para definir a classe NC não pode ser a MT, pois, para baixas complexidades, não é indiferente usar uma MT ou uma máquina de memória de acesso direto que nos servirá de referência aqui (uma vez que o tempo de inicialização desta última, para baixas complexidades, é significativo);
- > Os problemas mais difíceis da classe P (denominados P-completos) não pertencem à classe NC, não sendo, assim, passíveis de serem paralelizados;
- > Os problemas P-completos são definidos de forma análoga aos NP-completos, contudo, a partir da classe P e de uma noção de transformação mais restritiva que as transformadas polinomiais (transformações LOGSPACE), conforme será visto na próxima subseção.

> E a computação Quântica ?

- > A computação atual considera que os modelos computacionais trabalham a partir de entidade elementares chamadas de BIT (0/1);
- > Um bit quântico representa simultaneamente mais de 1 valor (superposição de valores) e um registro de n bits quânticos permite codificar um número exponencial de estados.

Desafios da computação quântica:

- Construir computadores baseados nas leis da mecânica quântica (na verdade já é o caso considerando a miniaturização dos dispositivos atuais e os efeitos da física quântica em micro-circuitos);
- Descobrir novos algoritmos e modelos computacionais (Máquinas não determinísticas) baseados na noção de bit quântico ;
- Produzir novos algoritmos de criptografia com complexidade superior à dos problemas NPC (ou seja, complexidade NP-hard mais difícil do que NP) capazes de resistir ao poder de processamento de máquinas quânticas que operam como Máquinas de Turing não determinísticas (uma vez que algoritmos de criptografia NPC seriam facilmente quebrados pelas máquinas quânticas, para quem eles teriam complexidade polinomial).

7.6 Outras Classes de Complexidade

Definição: A classe co-NP é a classe das linguagens L cujos complementos ($\Sigma^* - L$) estão em NP.

- > Se um problema é NP, seu complemento não é, necessariamente, NP. De fato, como NP é definida a partir de MT não determinísticas, inverter as respostas dessa máquina não é trivial. Isso porque tais MTs aceitam se existe uma execução que aceita. Logo, elas recusam se todas as execuções recusam, condição que não é diretamente verificável em tais máquinas.
- > Contudo, para P, que é definida a partir de MTs determinísticas, $P = \text{co-}P$.

Definição: A classe EXPTIME é a classe decidida por uma MT determinística cuja complexidade de tempo é limitada por uma função exponencial ($2^{p(n)}$, onde $p(n)$ é um polinômio).

> Contrariamente à situação que concerne P e NP (em que não se provou se P é ou não igual à NP), provou-se que P está PROPRIAMENTE incluído em EXPTIME, ou seja, $P \subsetneq EXPTIME$. Logo, existem problemas em EXPTIME que não têm solução polinomial - tais problemas definem a classe de problemas COMPLETOS em EXPTIME (denominada EXPTIME-completa).

Definição: A classe PSPACE é a classe das linguagens decididas por uma MT determinística cuja complexidade em espaço (número de casas da fita utilizadas) é limitada por um polinômio.

- > Definição: A classe NPSPACE é a classe das linguagens aceitas por uma MT não determinística cuja complexidade em espaço é limitada por um polinômio.
- > Teorema de Savitch: seja s uma função de n em \mathbb{R}^+ tal que $s(n) \geq n$ para todo $n \geq 0$. Uma TM não determinística que funciona em espaço $O(s(n))$ é equivalente a uma TM determinística que funciona em espaço $O(s(n)^2)$ (prova baseada em um algoritmo determinístico recursivo que simula o algoritmo não determinístico).
- > Consequência do Teorema de Savitch: a simulação de uma MT não determinística de complexidade de espaço polinomial em uma MT determinística também tem custo polinomial (lembremos que, em termos de complexidade em tempo, a simulação de uma MT não determinística polinomial em uma MT determinística tem complexidade exponencial).

> Logo, pelo Teorema de Savitch, $\text{PSPACE} = \text{NPSPACE}$, ou seja, todo problema de complexidade PSPACE tem complexidade NPSPACE e vice-versa.

> Vimos que: 1) Como o espaço utilizado por uma MT é sempre igual ou inferior ao tempo que ela utiliza, a complexidade de espaço de uma MT não determinística que aceita um problema NP é polinomial (pois sua complexidade de tempo é polinomial); 2) A simulação de uma MT não determinística de complexidade de espaço polinomial em uma MT determinística também tem complexidade de espaço polinomial. Logo, de 1) e 2) temos que: $\text{NP} \subseteq \text{PSPACE}$.

> Assim como para as classes NP e EXPTIME, define-se também a classe de problemas completos em PSPACE. Um problema PSPACE-completo é certamente tão difícil, e prioritariamente mais difícil, que um problema NP-completo.

> Em síntese, a relação entre essas classes é:

$$P \subseteq \text{NP}^{\text{P}} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

> Conforme visto há pouco, a inclusão entre P e EXPTIME é PRÓPRIA, contudo, não se sabe em qual nível essa inclusão é própria (entre P e NP, entre NP e PSPACE, ou entre PSPACE e EXPTIME).

- > Existem problemas NP-Hard que não pertencem à EXPTIME e o contrário também (o que significa que não existe relação de inclusão entre as duas classes).
- > Definem-se também classes mais fáceis do que P. Por exemplo, a classe LOGSPACE composta por linguagens que podem ser decididas por MTs determinísticas utilizando um espaço logarítmico (sem contar o espaço necessário para guardar a palavra). Demonstra-se que $\text{LOGSPACE} \subseteq \text{P}$. É possível mostrar que alguns problemas de P não são LOGSPACE demonstrando que eles são completos em P (i.e., P-completos, ou não paralelizáveis, conforme definido na seção 7.5.7). Isso significa que, para qualquer problema de P, existe uma transformada dele para esses problemas P-completos. Contudo, obviamente, tal transformada não é polinomial, pois, como P é uma classe de equivalência, existe uma transformada polinomial (em termos de complexidade de tempo) entre qualquer par de problemas P, ou seja, se a transformada da definição de P-completo fosse a polinomial, todos os problemas de P seriam P-completos. Assim sendo, a definição de P-completo é feita em termos da transformação LOGSPACE (ou seja, de complexidade de espaço).

- > Saliente-se que existe também uma definição alternativa de NPC baseada em complexidade de espaço (transformações LOGSPACE).
 - > Se um problema é NPC com relação às transformações LOGSPACE, ele o é também com relação às transformadas polinomiais. Contudo, o contrário não é necessariamente verdadeiro. Isso significa que a classe NPC definida pela complexidade de espaço é ainda mais difícil que aquela definida pela complexidade de tempo.

Curiosidade sobre PSPACE - Completude

A classe mais difícil de PSPACE é a classe PSPACE-Completa.

Um problema A é PSPACE- Completo se:

1- o problema é em PSPACE

2 - todo problema PSPACE se reduz polinomialmente EM TEMPO à A !

> Logo, a classe PSPACE-completa é de transformada polinomial.

The end??? Never...

E todos serão felizes para sempre...