7. A Complexidade

7.1 Introdução

Caso um problema pertence à classe R (existe uma TM det que decide a linguagem que codifica tal problema), nada garante que existe sempre um algoritmo eficiente (?) que resolve tal problema.

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).

Mas será que é possível definir um tipo de limite sobre os recursos independentemente da tecnologia existente ?

Como formalizar tal conceito ?

A medição dos recursos utilizados à boa realização de um cálculo será dada através da noção de função de complexidade.

Tal função vai depender do tamanho n dos dados de entrada do problema (comprimento da palavra w no caso de uma TM).

Uma fronteira deverá então ser estabelecida entre uma função de complexidade aceitável e inaceitável .

Tal fronteira se encontrará entre funções de tipo polinomial (n^cst) e funções de tipo exponencial (cst^n).

Cuidado que em certos casos algoritmos de tipo polinomiais (que tenham uma função de complexidade polinomial) serão 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 necessita pelo menos uma execução (1 unidade de tempo) de uma instrução de programação.

Significa que resolvendo a complexidade em tempo, a gente resolve também a complexidade em memória (inclusive com os custos atuais baratos da memória).

Pelo contrário, resolvendo o problema da complexidade em memória, a gente não resolve necessariamente a complexidade em tempo.

Em certos casos, será interessante considerar assim mesmo classes de complexidade em espaço.

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

Geralmente, considerando dados de tamanho n, o tempo de calculo quando se trata de complexidade de algoritmo corresponde ao tempo de calculo máximo para dados de tamanho n: análise no pior dos casos (worst case analysis)

A complexidade em tempo é geralmente expressa usando a notação O (ordem de grandeza)

Def: Uma função g(n) é O(f(n)) se existe valores constantes c e n0 tais que para todo n>n0

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

Tal notação não considera o comportamento das funções por valores pequenos de n. Para dados de tamanho pequeno, a complexidade depende também de operações de inicialização que se tornam desprezíveis para n grande.

Ex: a função C.n^2 é O(n^2) → Provar

a função C1.n^2 + C2.n é O(n^2)

Prova: para n ≥ 1,

$$C1.n^2 + C2.n \leq (C1 + C2).n^2$$

Um algoritmo cujo tempo de calculo é dado por C1.n^2 + C2.n tem uma função de complexidade O(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 quasi sempre excessiva

Por exemplo, 2^100 ≈ 10^30 que representa um número excessivo.

Mas n=100 não é um tamanho excessivo : 100 casas que um carteiro deverá percorrer por exemplo.

Se cada instrução elementar representa 1 nanossegundo por exemplo, no pior dos casos, uma resposta será fornecida em mais ou menos 3.10^11 séculos !

Geralmente uma complexidade polinomial O(n^k) para k constante representa uma complexidade aceitável.

Um algoritmo eficiente será então sinônimo de algoritmo de complexidade polinomial.

É claro que um algoritmo em O(n^100) não é eficiente mas geralmente o grau do polinômio é $\leq$ 5

É então razoável aceitar que um problema para o qual não foi encontrado um algoritmo de tipo polinomial não tem um procedimento efetivo eficiente associado.

7.3 Problemas de complexidade polinomial

7.3.1 Complexidade de uma TM determinístico

Def: Seja uma TM det M que sempre para. A complexidade em tempo de M é a função :

$$T_M(m) = MAX\{ m \mid \exists x \in \Sigma^*, |x| = m \text{ , A EXECUÇÃO DE M SOBRE } x \text{ TEM } m \text{ ETAPAS} \}$$

Tal função fornece o máximo de etapas necessárias para a M decidir uma palavra de comprimento n (pior caso)

Uma TM det M será de complexidade polinomial se a função de complexidade tem como borda superior um polinômio em n.

Def: Uma TM det M é polinomial em tempo se existe um polinômio p(n) tal que a função de complexidade respeita:

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

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 det polinomial que decide a linguagem que codifica as instâncias positivas do problema.

A análise das transformações de modelos computacionais diferentes das TM det em uma TM det tradicional mostra que todas as transformações deste tipo são de complexidade polinomial.

Por exemplo, se existe uma máquina de memória a acesso direto (RAM) de complexidade T(n) que reconhece uma linguagem, então existe também uma TM det de complexidade (T(n))^n que aceita a mesma linguagem.

Existe uma diferença importante entre T(n) e (T(n))^3 mas se T(n) é um polinômio, (T(n))^3 é também um polinômio.

Na pratica isso mostra que a computação paralela pode diminuir a complexidade de algoritmos polinomiais complexas , mas não de algoritmos exponenciais !

7.3.2 A classe P

Def: A classe P (polinomial) é a classe das linguagens decididas por uma TM det polinomial.

Def: Uma função é calculável em tempo polinomial se existe uma TM polinomial que a calcula.

7.4 As transformações polinomiais

Como mostrar que certos problemas não devem ser em P ❓

Usando a noção de transformação polinomial.

7.4.1 Problema do viajante (TS:Travelling Salesman)

Corresponde a um tipo de problema que não deve ser em P.

Considere um conjunto V de n cidades e para cada par de cidade o comprimento d(vi,vj) da cidade vi até a cidade vj.

Considere também um valor constante b.

O problema é de determinar se existe um percurso fechado (ciclo) cujo comprimento total é menor ou igual a b.

Trata-se então de encontrar uma permutação vp1, vp2, vp3, ..., vpn tal que :

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

Existe um procedimento efetivo para resolver tal problema: calcular o comprimento global do ciclo (circuito) para todas as permutações e existentes.

O número de permutações existentes é igual a n!

Então o algoritmo não é polinomial.

O problema foi definido no contexto de um problema de decisão. É geralmente apresentado como um problema de otimização: encontrar o percurso mínimo .

Provar que se não existe um algoritmo para o problema de decisão, então também não existe um algoritmo para o problema de otimização !

Prova por contradição.

7.4.2 Problema do circuito Hamiltoniano

Considere um grafo G=(V,E) com V o conjunto de vértices e E o conjunto de arcos. O problema consiste em encontrar um percurso fechado (circuito) que contém cada vértice uma única vez.

Procedimento efetivo: encontrar uma permutação vp1, vp2, vp3, ..., vpn tal que (vpi, vpi+1) $\in$ E para todo $1 \leq i \leq n$ e (vpn, vp1) $\in$ E.

Verificar todas as permutações não constitua um algoritmo polinomial.

7.4.2 Problema do circuito Hamiltoniano

Considere um grafo G = (V,E) com V o conjunto de vértices e E o conjunto de arcos. O problema consiste em encontrar um percurso fechado (circuito) que contém cada vértices de V uma única vez.

Procedimento efetivo: Encontrar uma permutação vp1, vp2, vp3, ..., vpn tal que (vpi, vpi+1) $\in$ E e (vpn, vp1) $\in$ E.

Verificar todas as permutações não constitua uma algoritmo polinomial.

7.4.3 Definição das transformações polinomiais

Def: Seja uma linguagem $L1 \subseteq \Sigma^*1$ e uma linguagem $L2 \subseteq \Sigma^*2$. Uma transformação polinomial de L1 em L2 anotada $L1 \propto L2$ é uma função $f: \Sigma^*1 \rightarrow \Sigma^*2$ que verifica as seguintes condições :

(1) A transformação é calculável em tempo polinomial

(2) $f(x) \in L2$ se é somente se $x \in L1$

Ex: Provar que HC $\propto$ TS

Encontrar um algoritmo polinomial que transforma uma instância qualquer de HC numa instância de TS positiva se e somente se a instância de HC é positiva.

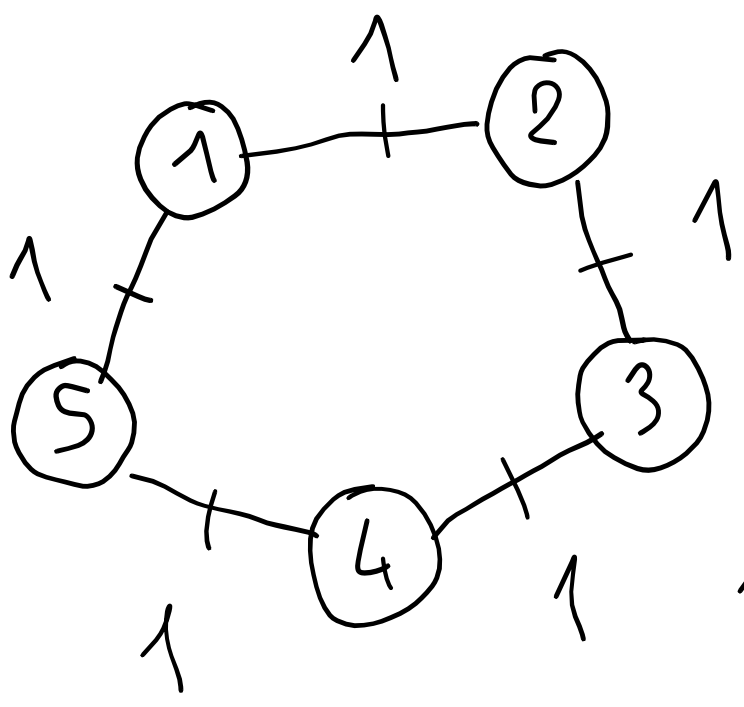
A transformação de uma instância G = (V,E) de HC numa instância (C, d(vi, vj), b) de TC é dada a seguir:

- o conjunto das cidades C é idêntico ao conjunto dos vértices do grafo V (C=V)

- os comprimentos entre pares de cidades d(vi, vj) são dados por:

* 1 se (vi, vj) ∈ E
* 2 se (vi, vj) ∉ E

- o valor constante b é igual ao número de cidades i.e. b = |V| = card (V)



$b = 5$

$$\sum d(v_{p_i}, v_{p_{i+1}})$$

$$1 \leq i < m + d(v_{p_m}, v_{p_1}) \leq 5$$

Qual o motivo de tal transformação ?

Se a transformação HC $\propto$ TS é polinomial, e se existe um algoritmo polinomial para resolver TS, então significa que 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.

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 (a prova é menos trivial que para a transformação $HC \propto TS$):

$TS \propto HC$

7.4.4 Propriedades das transformações polinomiais

Lema: Se $L1 \propto L2$ então:

- se $L2 \in P$ então $L1 \in P$
- se $L1 \notin P$ então $L2 \notin P$

Prova: se existe um algoritmo polinomial que decide L2 então a redução $L1 \propto L2$ produz um algoritmo polinomial que decide L1. A segunda conclusão pode ser provada por contradição : suponha que $L1 \notin P$ e $L2 \in P$; então temos uma contradição já que se $L2 \in P$ então $L1 \in P$ também.

As transformações $HC \propto TS$ e $TS \propto HC$ permitem deduzir que $HC \propto P$ se e somente se $TS \in P$.

Lema: as transformações polinomiais são transitivas:

Se L1 $\propto$ L2 e L2 $\propto$ L3 então L1 $\propto$ L3

7.4.5 Problemas polinomialmente equivalentes

Def: duas linguagens L1 e L2 são polinomialmente equivalentes se e somente se $L1 \propto L2$ e $L2 \propto L1$

A relação de equivalência entre as linguagens define então classes de equivalências sobre as linguagens. Isso significa que todas as linguagens de uma mesma classe tem uma solução polinomial ou nenhuma delas tem.

Por exemplo, HC e TC pertencem à mesma classe.

Para mostrar que um outro problema X pertence à mesma classe, então precisa mostrar:

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

e

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

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

O que se pode deduzir de tal fato ?

7.5 A classe NP

Os algoritmos conhecidos para resolver HC não são eficientes porque o princípio é de aplicar todas as permutações possíveis de vértices. Por outro lado, para uma permutação dada, a verificação a realizar (a permutação é Hamiltoniana ❓) é rápida.

Os problemas desta classe têm de verificar numerosos casos mas cada caso a verificar é simples.

Como dar uma definição abstrata desta classe ❓

7.5.1 Complexidade das TM não deterministas

Def: o tempo de calculo de uma TM não det numa palavra w é dado por:

- o comprimento mais curto da execução que aceita a palavra (se a TM não det aceita a palavra de entrada)

- o comprimento 1 se a palavra não é aceita pela TM não de

O escolha pelo valor 1 é para não considerar as execuções que não aceitam w (1 vai ser o menor valor de qualquer jeito)

Def: seja M uma TM não det. A complexidade em tempo de M é dada pela função :

$$T_M(n) =$$

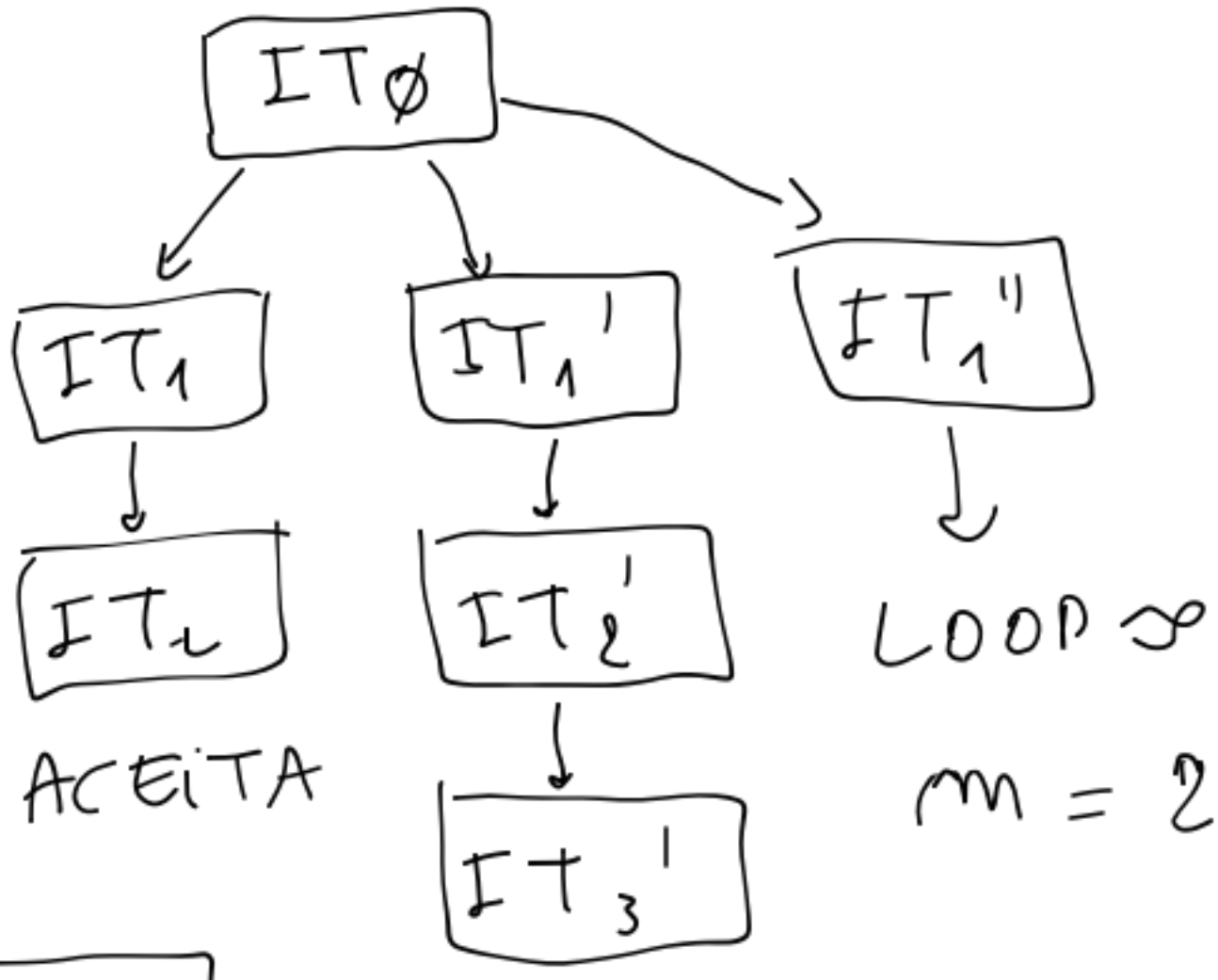
$$MAX\{m \mid \exists x \in \Sigma^*, |x| = n$$

$$\text{e o tempo de calculo de M sobre } x \text{ é } m\}$$

Significa que para as palavras de comprimento n, escolha-se a palavra cujo tempo de calculo é o maior, sabendo que para tal palavra o tempo de calculo é dado pela menor execução da TM não det que aceita a palavra (max dos min)

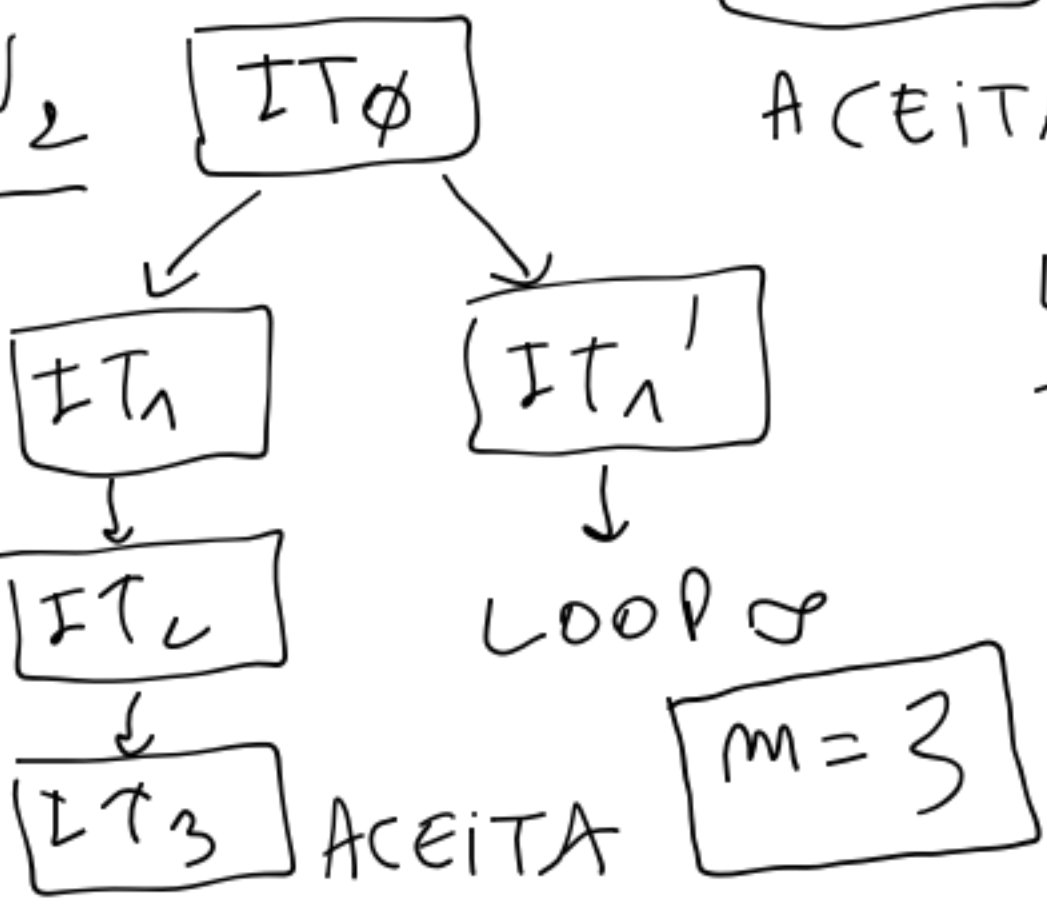
Ex:

W_1



m = 2

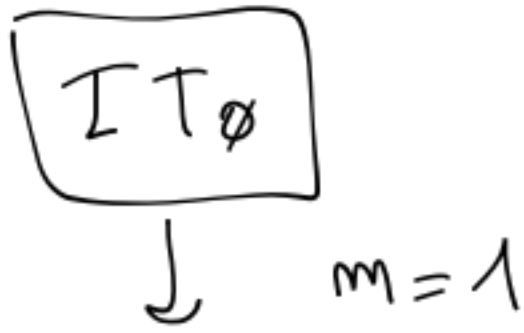
W_2



ACEITA

m = 3

W_3



7.5.2 Definição da classe NP

Def: a classe NP (Não Determinístico Polinomial) é a classe das linguagens aceitas por uma máquina de Turing não determinístico de complexidade polinomial .

Significa que a função :

$$T_M(n) \leq P(n)$$

Os problemas HC e TS pertencem à classe NP

Provar: encontrar um algoritmo não det polinomial que resolve HC e TS **!**

Prova HC: o algoritmo não det seguinte resolve em tempo polinomial HC:

1) o algoritmo gera de modo não determinista uma permutação dos vértices dos gratos (equivalente a ter uma infinidade de TM det que gera cada uma uma das permutação possível em HC)

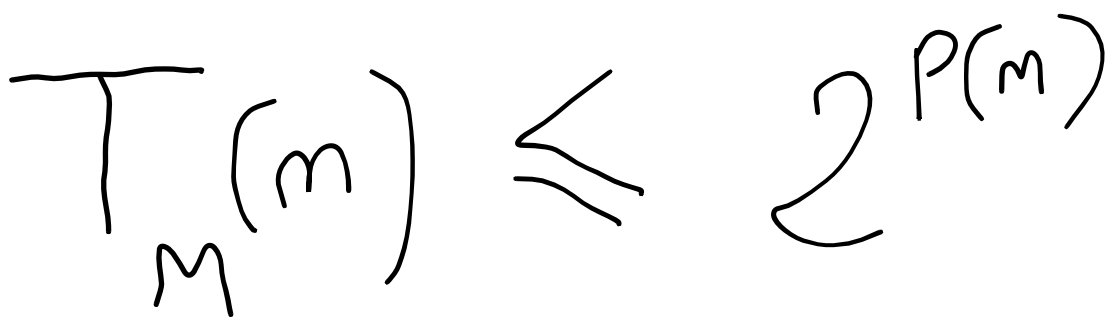
2) o algoritmo verifica que a permutação gerada corresponde a um circuito Hamiltoniano. (Tal etapa tem um número de iterações proporcional ao número de vértices do grafo e é então polinomial)

A enumeração sistemática de todas as permutações para encontrar uma solução pela TM det corresponde a uma enumeração única na TM não det, o que torna o algoritmo polinomial no caso não det.

A classe NP é definida em termo de linguagens aceitas por TM não det.

Tais linguagens são também decididas por TM det de complexidade exponencial !

Teorema: Se L é uma linguagem que pertence à NP então existe uma TM det M e um polinômio p(n) tal que M decide L e M é de complexidade em tempo:



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 não obteve resposta até hoje !

7.5.3 Estrutura da classe NP

Def: Uma classe de equivalência polinomial C1 é inferior a uma classe de equivalência polinomial C2 ($C1 \leq C2$) se existe uma transformação polinomial de toda linguagem de C1 para toda linguagem de C2

Se $C1 \leq C2$ então existe $L1 \in C1$ e $L2 \in C2$ tais que

$$L1 \propto L2$$

Isso NAO SIGNIFICA que se existe uma solução polinomial para os problemas de C1 então existe uma solução para os problemas de C2 **!**

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

Prova: uma TM det polinomial é um caso particular de uma TM não det polinomial

Lema: A classe P é uma classe de equivalência polinomial

Significa que para todo L1, L2 $\in$ P então L1$\propto$ L2

Prova: 1) determinar se uma palavra w $\in$ L1.
Existe um algoritmo polinomial para verificar isso já que L1 $\in$ P
2) se w $\in$ L1 a transformação polinomial produz uma palavra w' $\in$ L2 (sempre a mesma, qualquer que seja a palavra w)

se w $\notin$ L1 a transformação polinomial produz uma palavra w' $\notin$ L2 (sempre a mesma palavra)

Lema: Para todo L1 $\in$ P e para todo L2 $\in$ NP temos:

$$L1 \propto L2$$

Provar !

7.5.4 Problemas NP - Completos (classe NPC)

A classe a mais difícil de NP é a classe NPC

Def: Uma linguagem L é NPC se:

1) $L \in NP$ (L é aceita por uma TM não det polinomial)

2) Para toda linguagem $L' \in NP$ temos $L' \propto L$

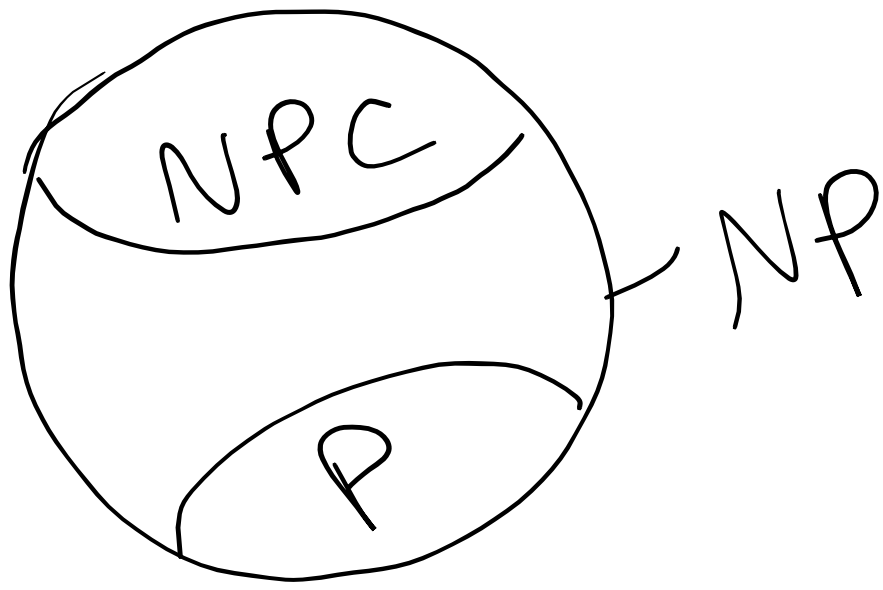
Tal definição implica que a classe NPC é uma classe de equivalência polinomial e que para qualquer outra classe de equivalência polinomial C de NP que não pertence à NPC temos $C \leq NPC$

Teorema : Se existe uma linguagem NPC L decidida por um algoritmo polinomial então toda linguagem de NP é decidida em tempo polinomial : $P = NP$

Não foi provado até hoje :

$P = NP$ ou $P \neq NP$

Já foi provado que se $P = NP$ então existem problemas em NP que não são nem em P e nem em NPC



Na pratica, considerando a hipótese P = NP, para provar que não existe uma solução polinomial para um problema devemos mostrar que a linguagem correspondente pertence à classe NPC.

HC e TS são exemplos de problemas NPC

7.5.5 Provar a NP-Completude

Para provar que uma linguagem L é NP-Completa, deve-se estabelecer o seguinte:

1) que a linguagem pertence à classe NP

2) que para toda linguagem L' de NP, $L' \propto L$

Um algoritmo não det polinomial mostra que a linguagem é em NP.

Estabelecer a segunda propriedade é mais difícil, ao menos de se conhecer um outro problema NPC L' e mostrar que $L' \propto L$

Na pratica, a transformação polinomial a ser encontrada é sempre do problema NPC conhecido para o novo problema NPC !

7.5.6 A classe NP-Hard (difícil)

Considerando a hipótese $P \neq NP$, na pratica, para mostrar que uma linguagem L não é reconhecida por um algoritmo polinomial, é só mostrar que para toda linguagem L' de NP temos $L' \propto L$ sem mostrar necessariamente que $L \in NP$

Neste caso, fala-se de uma linguagem que pertence à classe NP-Hard ❗

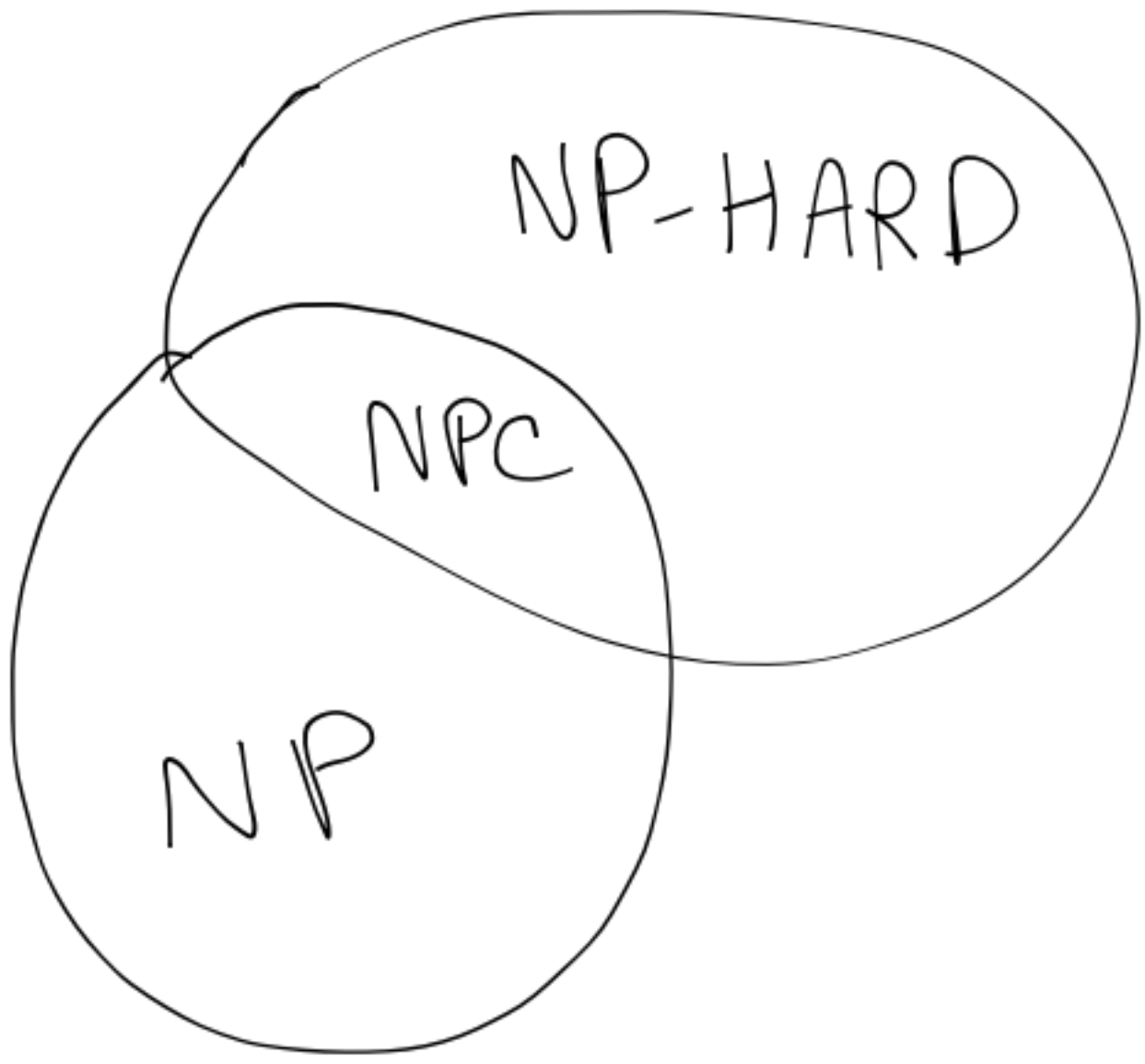
Qual é então a diferença entre as classes NPC e NP-Hard ❓

Parece que nenhuma linguagem que pertence à uma classe ou 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 det polinomial ❗

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



7.5.7 Problemas NPC conhecidos

Um primeiro problema NP-completo:

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: E1^E2^...^Ei^...^Ek onde Ei é uma cláusula

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

x1∨...∨xj∨...∨xk onde xj é uma expressão da forma p ou ¬p

Ex: (p∨¬q)∧p∧(q∨¬r∨s∨¬t)∧(s∨t)

Expressão válida: verdadeira qualquer que seja a interpretação das variáveis.

Expressão satisfatível: existe pelo menos uma função de interpretação que a torna verdadeira.

Prova:

A) Mostrar que SAT $\in$ NP

1) gerar de modo não determinístico uma função de interpretação

2) verificar que 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 : para este primeiro problema não se pode mostrar que SAT é NP-Hard a partir de um problema NPC conhecido.

Sabendo que HC é NPC, provar que TS é NPC !

Para provar que HC é NPC, pôde-se provar que VC $\propto$ HC com VC=Vértice Cover (1 Problema dos grafos)

VC é provado como NP-Hard a partir do problema 3-SAT que é provado a partir de SAT.

O problema do número chromático (coloração de um grafo) de um grafo é NPC.

A programação em números inteiros é NPC.

A programação em números racionais é P !
Programação Linear: algoritmo polinomial descoberto em 1979 - mas o método do simplex usado anteriormente é de complexidade exponencial

O problema da equivalência de autômatos finitos não deterministos é NP-Hard ! Não foi encontrado um algoritmo não det polinomial para resolver tal problema. É um problema completo (classe de equivalência polinomial) numa classe chamada PSPACE (classe de complexidade em espaço)

7.5.8 Interpretação da NP-Completude

Na pratica, ATUALMENTE, mostrar que um problema é NPC ou NP-Hard é a mesma coisa que mostrar que não foi encontrado um algoritmo polinomial de resolução.

Mesmo assim existem ferramentas computacionais para resolver problemas NPC.

Exemplo do problema SAT: uma fórmula que tem muitas variáveis e poucas cláusulas é quasi sempre satisfatível; por outro lado uma fórmula que tem muitas cláusulas e poucas variáveis é quasi sempre insatisfatível - 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).

7.5.9 NPC e tecnologia

Em relação ao paralelismo, o uso de n processadores pode reduzir no melhor dos casos o tempo de calculo de um fator n.

Para um algoritmo de complexidade exponencial, o número de processadores tem de ser exponencial em função do tamanho das instâncias a serem testadas.

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.

Problema 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 deterministicas) baseados na noção de bit quântico

- rever as técnicas de criptografia que na maioria dos casos não resistam a Máquinas de Turing não deterministicas.

7.6 Complexidade em espaço

O espaço de 1 cálculo de uma TM det M sobre 1 palavra w corresponde ao número maximo de células usadas simultaneamente na fita durante o cálculo.

Definir a função complexidade em espaço !

Teorema de Savitch: seja s uma função de n em R+ tal que $s(n) \geq n$ para todo $n \geq 0$. Uma TM não det que funciona em espaço O(s(n)) é equivalente a 1 TM det que funciona em espaço O(s(n)^2)

Prova baseada num algoritmo det recursivo que simula o algoritmo não det.

Consequências: na complexidade em tempo, a simulação de 1 TM não det numa TM det tem um custo exponencial; mas na complexidade em espaço, a simulação de 1 TM não det numa TM det tem um custo polinomial.

Relação entre complexidade em tempo e espaço:

O espaço usado por uma TM det que funciona em tempo $t(n) \geq n$ é no pior dos casos igual a $t(n)$
Por que ?

Temos então:

$P \subseteq PSPACE$

$NP \subseteq NPSPACE = PSPACE$ (Savitch)

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE$

EXPTIME ?

Existem problemas NP-Hard que não pertencem à EXPTIME e o contrário também (não existe relação de inclusão entre as duas classes então)

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 !

A redução polinomial não é em espaço mas em tempo para as classe completas de dimensão espacial.