TÉCNICO LISBOA

Análise e Síntese de Algoritmos

Complexidade Computacional

Prof. Pedro T. Monteiro

IST - Universidade de Lisboa

2024/2025

Contexto

- Revisão [CLRS, Cap.1-13]
  - Fundamentos; notação; exemplos
- Técnicas de Síntese de Algoritmos [CLRS, Cap.15-16]
  - Programação dinâmica [CLRS, Cap.15]
  - Algoritmos greedy [CLRS, Cap.16]
- Algoritmos em Grafos [CLRS, Cap.21-26]
  - Algoritmos elementares
  - Caminhos mais curtos [CLRS, Cap.22,24-25]
  - Árvores abrangentes [CLRS, Cap.23]
  - Fluxos máximos [CLRS, Cap.26]
- Programação Linear [CLRS, Cap.29]
  - Algoritmos e modelação de problemas com restrições lineares
- Tópicos Adicionais
  - Complexidade Computacional [CLRS, Cap.34]

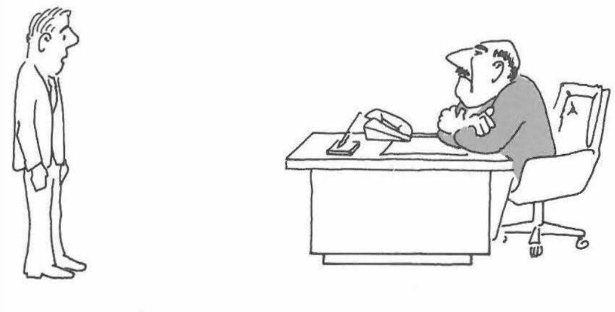
Resumo

Motivação

- A empresa onde trabalham vai entrar no competitivo mercado dos gambozinómetros
- O vosso chefe, chama-vos ao gabinete e pede-vos para elaborarem um algoritmo que permita determinar as especificações óptimas para o primeiro modelo do gambozinómetro
- Após consultarem o departamento responsável pelo desenho dos gambozinómetros, para se inteirarem dos contornos do problema, colocam mãos à obra ...
- Várias semanas depois, já com o vosso gabinete atolado de pilhas de papel, e muitas noites mal dormidas, o vosso entusiasmo diminuiu ...

Motivação

- Até agora ainda não descobriram um algoritmo melhor do que percorrer e avaliar todos os desenhos possíveis de gambozinómetros, para determinar qual o melhor
  Isso provavelmente envolverá vários anos de computação...
- Mas, não querem voltar ao gabinete do vosso chefe e dar-lhe a má notícia...



"I can't find an efficient algorithm, I guess I'm just too dumb."

Motivação

- Para evitar e mancharem a vossa reputação como engenheiros informáticos do IST, seria preferível se conseguissem provar que o problema dos gambozinómetros é *intrinsecamente intratável*, ou seja, que não existe um algoritmo eficiente para sua solução

"I can't find an efficient algorithm, because no such algorithm is possible!"

Motivação

- Infelizmente, provar que um problema é intrinsecamente intratável pode ser tão difícil quanto encontrar um algoritmo eficiente para o resolver

- A teoria da completude-NP (*NP-completeness*) fornece-nos um conjunto de técnicas para provar que um dado problema computacional é tão difícil quanto um conjunto de outros problemas amplamente reconhecidos como sendo difíceis de resolver

Motivação

- Utilizando estas técnicas poderão conseguir provar ao vosso chefe que o problema em questão é NP-completo, e que portanto é equivalente a um conjunto de outros problemas já estudados por cientistas ilustres, e para a solução dos quais não se conseguiu descobrir um algoritmo eficiente

Motivação

Perspectiva

- Problemas de decisão
  - Resposta sim(1)/não(0)
- Classe de complexidade P — **P**olynomial time
  - Problemas resolúveis em tempo polinomial
- Codificação de problemas
- Linguagens formais
- Algoritmos de verificação
- Classe de complexidade NP — **N**ondeterministic **P**olynomial time
  - Problemas verificáveis em tempo polinomial
- Redutibilidade entre problemas de decisão
- Problemas NP-completos

Motivação

Algoritmos Polinomiais

- Complexidade em $O(n^k)$
- Quase todos os algoritmos estudados em ASA (até agora)
  (Exemplo excepção: Simplex)
- Existem algoritmos polinomiais para qualquer problema ? Não !
  - Existem problemas não resolúveis
  - Existem problemas não resolúveis em tempo $O(n^k)$ para qualquer k
  - Problemas intratáveis: requerem tempo superpolinomial (ex: $O(2^n)$ ou $O(n!)$)

Problemas NP-completos (desde 1971)

- Não provado que são tratáveis ou que são intratáveis
- Conjectura: problemas NP-completos são intratáveis

Motivação

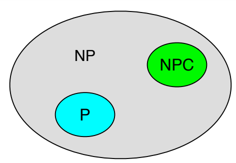
Problemas Resolúveis em Tempo Polinomial vs. NP-completos

- Caminhos mais curtos vs. caminhos mais longos
  - Mesmo com arcos com peso negativo é possível determinar caminhos mais curtos em tempo $O(VE)$
  - Determinar o caminho mais longo entre dois vértices é NP-completo
- 2-CNF SAT vs. 3-CNF SAT
  - Determinar valores de x_1, x_2, x_3, x_4 que satisfazem $(\overline{x}_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_4) = 1$ pode ser feito em tempo polinomial
  - Determinar valores de x_1, x_2, x_3, x_4 que satisfazem $(\overline{x}_1 \vee x_2 \vee \overline{x}_4) \wedge (x_2 \vee \overline{x}_3 \vee x_4) = 1$ é um problema NP-completo

Motivação

Classes de Problemas P, NP e NPC

- Classe **P**: problemas resolúveis em tempo polinomial
- Classe **NP**: problemas verificáveis em tempo polinominal
  - Dado um certificado de uma solução, é possível verificar que o certificado é correcto, em tempo polinomial na dimensão do problema
- Classe **NPC**: problemas NP-completos
  - Problemas tão difíceis como qualquer problema em NP
  - Se *algum* problema NP-completo puder ser resolvido em tempo polinomial, então *todos* os problemas NP-completos podem ser resolvidos em tempo polinomial



Motivação

Exemplos de Problemas NPC

- Satisfação de fórmulas proposicionais - SAT
- Coloração de grafos
- Instalação de pacotes de software
- Problemas em lógica
- Problemas em grafos / redes
- Problemas de autómatos
- Problemas em geração de código / compiladores
- Problemas em redes de computadores
- Problemas em bases de dados
- Problemas em investigação operacional
- . . . (largas centenas)

Motivação

Exemplo Prático – Gestão de Dependências de Software

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$

$$\begin{array}{ll} \mathit{depende}(p_1) = \{\{p_2\}, \{p_3, p_4\}\} & \mathit{conflitos}(p_1) = \{p_5\} \\ \mathit{depende}(p_2) = \{\{p_1, p_5\}, \{p_3, p_4\}\} & \mathit{conflitos}(p_2) = \emptyset \\ \mathit{depende}(p_3) = \emptyset & \mathit{conflitos}(p_3) = \{p_5\} \\ \mathit{depende}(p_4) = \{\{p_2, p_5\}\} & \mathit{conflitos}(p_4) = \emptyset \\ \mathit{depende}(p_5) = \emptyset & \mathit{conflitos}(p_5) = \{p_2\} \end{array}$$

- $p_1 \in P$ instalável? Não é apenas um grafo de dependências. Existem conflitos!

Problema é NP-Completo

Problemas Resolúveis em Tempo Polinomial

Porquê admitir problemas resolúveis em tempo polinomial como tratáveis?

- Algoritmos polinomiais são normalmente limitados em $O(n^k)$, com k "baixo"
- Para modelos de computação usuais, algoritmo polinomial num modelo é polinomial noutros modelos
  - *Serial random-access machine* (habitual), computador paralelo
- Propriedades de fecho dos algoritmos polinomiais (soma, multiplicação e composição)

Problemas Resolúveis em Tempo Polinomial

Problema Abstracto Q
Relação binária entre conjunto I de instâncias e conjunto S de soluções

Exemplo: Problema SHORTEST-PATH

- Instância: $i = \langle G, u, v\rangle$, grafo G, vértice origem u e destino v
- Solução: (uma) sequência de vértices do caminho mais curto

O problema é a relação que associa a cada instância uma ou mais soluções

Problemas de Decisão

- Problemas cuja resposta/solução é sim/não (1/0), $Q(i) \in \{0, 1\}$
- Problemas de optimização:
  - Reformulados como problemas de decisão
  - Se problema de optimização é tratável, então reformulação como problema de decisão também é tratável

Exemplo: Problema PATH

- Instância: $i = \langle G, u, v, k \rangle$, número máximo de arcos k
- Solução: 1/0, se um caminho mais curto entre u e v tem ou não no máximo k arcos

Codificação de Problemas

- Codificação:
  - Dado conjunto abstracto de objectos S, uma codificação e é uma função que mapeia os elementos de S em strings binárias
- Problema concreto:
  - Problema com conjunto de instâncias I representadas como strings binárias
- Uma codificação e permite mapear um problema abstracto, Q, num problema concreto, $e(Q)$

Exemplo

- Codificação dos números naturais $IN = \{0, 1, 2, 3, 4, \dots\}$ nas strings binárias $\{0, 1, 10, 11, 100, \dots\}$
- Utilizando esta codificação, $e(17) = 10001$

Codificação de Problemas

- Problema resolúvel em tempo polinomial
  - Solução para instância $i \in I$, $n = |i|$, pode ser encontrada em tempo $O(n^k)$, com k constante
- Classe de complexidade **P**
  - Conjunto de problemas de decisão concretos resolúveis em tempo polinomial

Codificação de Problemas

- A codificação utilizada tem impacto na eficiência com que é possível resolver um problema
- Para codificações "razoáveis" de problemas abstractos, a codificação utilizada não afecta se um dado problema pode ser resolúvel em tempo polinomial
- Função $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ é computável em tempo polinomial se existe um algoritmo de tempo polinomial A que, dado $x \in \{0, 1\}^*$, calcula $f(x)$
- Codificações e_1 e e_2 são relacionadas polinomialmente se existem duas funções polinomialmente computáveis, f_{12} e f_{21}, tal que para $i \in I$, $f_{12}(e_1(i)) = e_2(i)$ e $f_{21}(e_2(i)) = e_1(i)$

Problemas Resolúveis em Tempo Polinomial

Codificação de Problemas

Seja Q um problema de decisão abstracto com conjunto de instâncias I, e sejam e_1 e e_2 codificações relacionadas polinomialmente.

Então, $e_1(Q) \in P$ se e só se $e_2(Q) \in P$

- Admitir que $e_1(Q)$ é resolúvel em tempo $O(n^k)$ (k constante)
- $e_1(i)$ calculável a partir de $e_2(i)$ em tempo $O(n^c)$, com $n = |e_2(i)|$
- Para resolver o problema $e_2(Q)$ sobre a instância $e_2(i)$
  - Calcular $e_1(i)$ a partir de $e_2(i)$
  - Resolver o problema $e_1(Q)$ sobre a instância $e_1(i)$
- Complexidade polinomial $O(n^{ck})$
  - Conversão de codificações: $O(n^c)$ (c constante)
  - $|e_1(i)| = O(n^c)$, a saída é limitada pelo tempo de execução
  - Tempo para resolver $e_1(i)$: $O(|e_1(i)|^k) = O(n^{ck})$
    - Polinomial por c e k serem constantes

Utilização de Linguagens Formais

Utilização de Linguagens Formais

- Alfabeto Σ: conjunto finito de símbolos
- Linguagem L definida em Σ: conjunto de strings de símbolos de Σ
- Linguagem Σ^*: todas as strings de Σ
  - String vazia: ϵ
  - Linguagem vazia: $\emptyset$
- Qualquer linguagem L em Σ é um subconjunto de Σ^*
- Operações sobre linguagens: união, intersecção, complemento, concatenação, fecho

Exemplo

Se $\Sigma = \{0, 1\}$, então $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ é o conjunto de todas as strings binárias

Utilização de Linguagens Formais

Utilização de Linguagens Formais

- Para qualquer problema de decisão Q, o conjunto de instâncias é Σ^*, com $\Sigma = \{0, 1\}$
- Q é completamente caracterizado pelas instâncias que produzem solução 1 (sim)
- Q pode ser interpretado como linguagem L definida em $\Sigma = \{0, 1\}$

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

Exemplo

PATH= $\{\langle G, u, v, k\rangle :$ $G = (V, E)$ é um grafo não dirigido
$u, v \in V$,
$k \geq 0$ é um inteiro, e
existe um caminho de u para v em G
que consiste em, no máximo, k arcos

Utilização de Linguagens Formais

Utilização de Linguagens Formais

- Algoritmo A aceita $x \in \{0, 1\}^*$ se $A(x) = 1$
- Algoritmo A rejeita $x \in \{0, 1\}^*$ se $A(x) = 0$
- Linguagem aceite por A: $L = \{x \in \{0, 1\}^* : A(x) = 1\}$
- L é decidida por A se qualquer string $x \in \{0, 1\}^*$ é aceite ou rejeitada
- L aceite/decidida em tempo polinomial se A tem tempo de execução em $O(n^k)$, com $n = |x|$

Utilização de Linguagens Formais

Definições Alternativas para a Classe P

- $P = \{L \in \{0,1\}^* : \text{ existe um algoritmo } A \text{ que decide } L \text{ em tempo polinomial}\}$
- $P = \{L \in \{0,1\}^* : L \text{ é aceite por um algoritmo de tempo polinomial}\}$
  - Conjunto das linguagens decididas em tempo polinomial é subconjunto das linguagens aceites em tempo polinomial
  - Basta provar que se L é aceite por algoritmo polinomial, implica que L é decidida por algoritmo polinomial
  - A aceita L em $O(n^k)$, pelo que A aceita L em tempo não superior a $T = cn^k$
  - Utilizar A' que executa A e observa resultado após $T = cn^k$
    - Se A aceita, A' aceita; se A não aceita (ainda), A' rejeita

Problemas Verificáveis em Tempo Polinomial

Problemas Verificáveis em Tempo Polinomial

- Objectivo é verificar se uma instância pertence a uma dada linguagem utilizando um certificado (i.e. uma possível solução)

Problemas Verificáveis em Tempo Polinomial

Algoritmos de Verificação

Algoritmo de verificação A:

- Argumentos:
  - string x: entrada
  - string y: certificado
- O algoritmo A verifica, para uma entrada x e certificado y, se $A(x, y) = 1$
  - Certificado permite provar que $x \in L$
- A linguagem verificada por A é:
  - $L = \{x \in \{0,1\}^* : \text{existe } y \in \{0,1\}^* \text{ tal que } A(x, y) = 1\}$
- Exemplo: CNFSAT

Classe de Complexidade NP

Classe NP

- Classe de complexidade NP:
  - Linguagens que podem ser verificadas por um algoritmo de tempo polinomial A
    - $L = \{x \in \{0,1\}^* : \text{ existe um certificado } y \in \{0,1\}^*,$ com $|y| = O(|x|^c), \text{ tal que } A(x, y) = 1\}$
    - $L \in NP$
    - A verifica L em tempo polinomial
- Classe co-NP:
  - Linguagens L tal que $\bar{L} \in NP$
  - Exemplo: CNFUNSAT

Problemas Verificáveis em Tempo Polinomial

Relações entre classes de complexidade

- $P \subseteq NP$
  - Poder decidir implica poder verificar
- $P \subseteq NP \cap$ co-NP
  - P fechado quanto a complemento
- Questões em aberto:
  - $P = NP$??
  - $P = NP \cap$ co-NP ??
  - Existe L em $(NP \cap$ co-$NP) - P$??

Problemas Verificáveis em Tempo Polinomial

Relações entre classes de complexidade

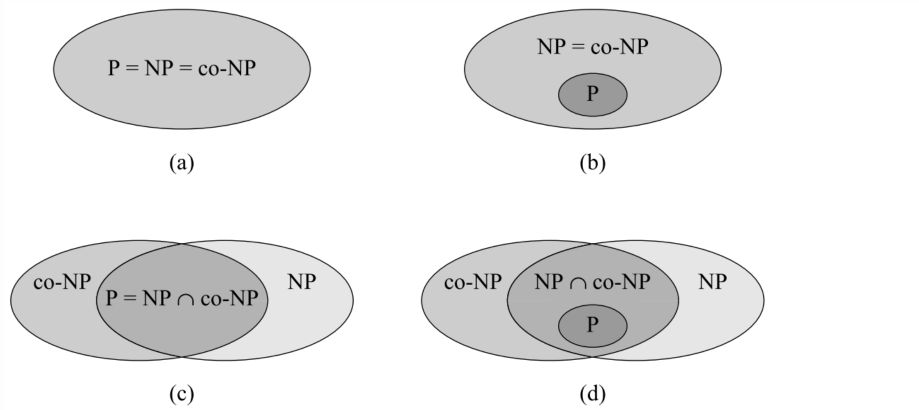


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P = NP = co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. **(c)** P = NP∩co-NP, but NP is not closed under complement. **(d)** NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

Complexity Theory's 50-Year Journey to the Limits of Knowledge

Recomendo a versão texto:
https://www.quantamagazine.org/complexity-theorys-50-year-journey-to-the-limits-of-knowledge-20230817/

Recomendo a versão vídeo:
https://www.youtube.com/watch?v=pQsdygaYcE4

Quantamagazine

Redutibilidade e Completude-NP

Relações entre classes de complexidade

- Noção de redução entre problemas
- Definição de problemas NP-Completos
- Um problema NP-completo
- Provar problemas NP-completos

Redutibilidade

- Z é redutível em tempo polinomial a X, $Z \leq_P X$, se existir uma função, $f : Z \rightarrow X$, calculável em tempo polinomial, tal que para qualquer $z \in Z$:
  - $Z(z) = 1$ se e só se $X(x) = X(f(z)) = 1$
- f é designada por função de redução, e o algoritmo F de tempo polinomial que calcula f é designado por algoritmo de redução
- Se Z, X são problemas de decisão, com $Z \leq_P X$, então $X \in P$ implica $Z \in P$

Completude-NP

- Um problema de decisão X diz-se NP-difícil se:
  - $Z \leq_P X$ para qualquer $Z \in NP$
- Um problema de decisão X diz-se NP-completo se:
  - $X \in NP$ (verificável em tempo polinomial)
  - X é NP-difícil
- NPC: classe de complexidade dos problemas de decisão NP-completos

Completude-NP

- Se existir problema NP-completo X, resolúvel em tempo polinomial, então $P = NP$
  - Todos os problemas em NP redutíveis a X (em tempo polinomial)
  - Logo, resolúveis em tempo polinomial
- Se existir problema X em NP não resolúvel em tempo polinomial, então todos os problemas NP-completos não são resolúveis em tempo polinomial
  - Se existisse Y em NPC resolúvel em tempo polinomial, dado que $X \leq_P Y$, então X seria resolúvel em tempo polinomial

Provar Problemas NP-Completos

Seja X um problema de decisão tal que $Y \leq_P X$, em que $Y \in NPC$.
Se $X \in NP$, então $X \in NPC$

- $Y \in NPC$
  - $\forall Z \in NP, Z \leq_P Y$
- Notando que $\leq_P$ é transitiva e que $Y \leq_P X$, obtemos:
  - $\forall Z \in NP, Z \leq_P X$
- Deste modo:
  - $X \in NP$
  - $\forall Z \in NP, Z \leq_P X$
- Pelo que $X \in NPC$!

Completude-NP

Provar Problemas NP-Completos

- Abordagem para provar $X \in NPC$:
  - Provar que $X \in NP$
  - Escolher $Y \in NPC$
  - Descrever um algoritmo que calcula função f, a qual converte qualquer instância de Y numa instância de X, $Y \leq_P X$
  - Provar que $x \in Y$ se e só se $f(x) \in X$, para qualquer instância x
  - Provar que algoritmo que calcula f tem tempo de execução polinomial
- Como definir $Y \in NPC$ inicial ?

Problema NP-Completo: SAT

Problema NP-Completo

Problema de decisão: SAT

- Fórmula proposicional ϕ:
  - variáveis proposicionais: $x_1, \ldots, x_n$
  - conectivas proposicionais: $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
  - parêntesis
- Atribuição de verdade: atribuir valores proposicionais (0 ou 1) às variáveis
- Atribuição de satisfação: valor da fórmula é 1
  - Se atribuição de satisfação existe, ϕ é satisfeita
- Problema SAT: determinar se uma instância ϕ é satisfeita
  - SAT $= \{\langle\phi\rangle : \phi$ é uma fórmula proposicional satisfeita$\}$

Problema NP-Completo: SAT

Problema NP-Completo

- $SAT \in NP$:
  - O certificado consiste numa atribuição de valores às variáveis
  - Substituir valores e analisar fórmula resultante
  - Tempo de execução é polinomial no tamanho da fórmula
- SAT é NP-difícil (1º problema a ser provado)
  [Teorema de Cook, 1971]
- $\therefore$ SAT é NP-completo

Problema NP-Completo: 2CNFSAT

Problema 2CNFSAT

3CNFSAT é NP-Completo, mas 2CNFSAT $\in P$

- Definição:
  - 2CNFSAT é uma restrição do problema CNFSAT em que cada cláusula contém exactamente 2 literais
- Teorema:
  - O problema 2CNFSAT $\in P$
- Prova:
  - Existe algoritmo para decidir 2CNFSAT com tempo de execução linear no tamanho de $|\varphi|, \varphi \in 2CNFSAT$
  - Cada cláusula binária corresponde a dois arcos (implicações) num grafo
  - Identificar SCCs no grafo
  - Se existe SCC com x e $\neg x$ então instância não é satisfeita

Problema NP-Completo: HornSAT

Problema HornSAT

- Definição:
  - HornSAT é uma restrição do problema CNFSAT em que cada cláusula contém não mais do que 1 literal não complementado
- Teorema:
  - O problema HornSAT $\in P$
- Prova:
  - Existe algoritmo para decidir HornSAT com tempo de execução linear no tamanho de $|\varphi|, \varphi \in$ HornSAT
  - Repetidamente satisfazer cláusulas com apenas 1 literal x_i não complementado (i.e. atribuir valor 1 (TRUE) a x_i)
  - Reduzir cláusulas com literal complementado
  - Terminar quando identificada cláusula vazia (UNSAT) ou todas as cláusulas com literais complementados
  - Atribuir valor 0 (FALSE) às restantes variáveis; cláusulas satisfeitas !

Problema NP-Completo: HornSAT

Problema HornSAT

HornSAT(φ)

```
while ∃ cláusulas com literal positivo x_i do
  x_i ← 1
  satisfazer clásulas com x_i
  reduzir clásulas com ¬x_i
  if ∃ cláusula vazia then
    eliminar atribuições
    return FALSE
  end if
end while
x_j ← 0 às variáveis ainda não atribuídas
return TRUE
```


Questões?

Dúvidas?