Teoria da Computação

Complexidade de Tempo

Aula 12

Prof. Felipe A. Louza

UFU

Roteiro

Introdução

Nas aulas anteriores, vimos que existem problemas **indecidíveis** (não pussuem solução computacional).

- Apresentamos a **tese de Church-Turing**.

  "Um problema tem solução algorítmica se, e somente se, ele tem solução em uma máquina de Turing"

Agora, vamos investigar problemas **decidíveis** mas que não possuem solução em *tempo aceitável*.

- Ou seja, o fato de um problema ser decidível não é suficiente.
- Esses problemas são chamados de **intratáveis**.

Introdução

Como medimos o tempo de execução ou complexidade de tempo de um algoritmo?

- O tempo está relacionado com a **quantidade de passos** realizados pelo algoritmo.
- Pode depender de diversos parâmetros. (ex. um grafo $G = (V, A)$)
- Por simplicidade, representaremos o tempo em função do tamanho da cadeia $\langle G \rangle$ que representa a entrada, $|\langle G \rangle| = n$.

Em geral, estamos interessados em analisar:

1. Pior caso: maior tempo (número de passos) considerando todas as entradas possíveis.
2. Caso médio: análise amortizada dos tempos para todas as entradas.

Introdução

Como medimos o tempo de execução ou complexidade de tempo de um algoritmo?

- O tempo está relacionado com a **quantidade de passos** realizados pelo algoritmo.
- Pode depender de diversos parâmetros. (ex. um grafo $G = (V, A)$)
- Por simplicidade, representaremos o tempo em função do tamanho da cadeia $\langle G \rangle$ que representa a entrada, $|\langle G \rangle| = n$.

Em geral, estamos interessados em analisar:

1. Pior caso: maior tempo (número de passos) considerando todas as entradas possíveis.
2. Caso médio: análise amortizada dos tempos para todas as entradas.

Introdução

Definição

Seja M uma **MT** determinística que pára sobre todas as entradas.
O tempo de execução ou complexidade de tempo de M é uma função

$$f : \mathbb{N} \to \mathbb{N}$$

em que $f(n)$ é o número **máximo de passos** que M realiza sobre entradas de **tamanho** n.

Podemos dizer que:

- M roda em tempo $f(n)$; ou
- M tem complexidade de tempo $f(n)$

Introdução

Calcular o número exato de passos pode ser complicado.

- Em geral, fazemos uma uma análise assintótica, em que buscamos entender o comportamento de $f(n)$ quando n é grande.

Por exemplo, considere:

$$f(n) = \underline{5n^3} + 2n^2 + 22n + 6$$

- O termo $\underline{5n^3}$ "domina" o crescimento de f.
- Dizemos que f é assintóticamente no máximo $\underline{n^3}$
- A notação assintótica ou notação O-grande para esse relacionamento é

$$f(n) = O(n^3)$$

A função $g(n) = n^8$ é um limitante superior da função f.

Introdução

Calcular o número exato de passos pode ser complicado.

- Em geral, fazemos uma uma análise assintótica, em que buscamos entender o comportamento de $f(n)$ quando n é grande.

Por exemplo, considere:

$$f(n) = \underline{5n^3} + 2n^2 + 22n + 6$$

- O termo $\underline{5n^3}$ "domina" o crescimento de f.
- Dizemos que f é assintóticamente no máximo $\underline{n^3}$
- A notação assintótica ou notação O-grande para esse relacionamento é
$$f(n) = O(n^3)$$

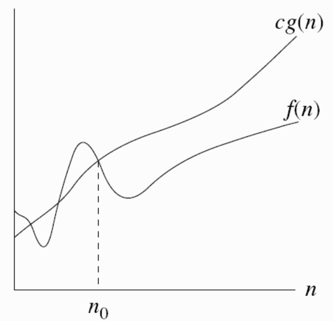
A função $g(n) = n^8$ é um limitante superior da função f.

Relembrando a notação assintótica (O-grande)

Definição

Sejam f e g funções, $f, g : \mathbb{N} \to \mathbb{R}^+$.
Dizemos que $f(n) = O(g(n))$ se existirem dois inteiros c e n_0, tais que, para todo $n \geq n_0$:

$$f(n) \leq c \cdot g(n)$$



$g(n)$ é um limitante superior assintótico:

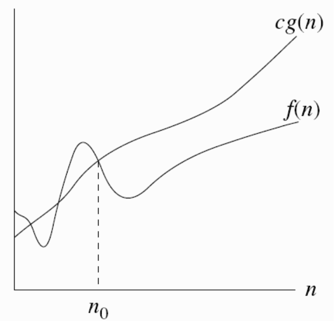
- $f(n) = \underline{5n^3} + n^2 + 22n + 6 = \underline{O(n^3)}$
  - para $c = 6$ e $n_0 = 10$, $f(n) \leq 6n^3$
- $f(n)$ também é $O(n^4)$?
  - Sim, mas estamos interessados em limites "apertados" para f.

Relembrando a notação assintótica (O-grande)

Definição

Sejam f e g funções, $f, g : \mathbb{N} \to \mathbb{R}^+$.
Dizemos que $\underline{f(n) = O(g(n))}$ se existirem dois inteiros c e n_0, tais que, para todo $n \geq n_0$:

$$f(n) \leq c \cdot g(n)$$



$g(n)$ é um limitante superior assintótico:

- $f(n) = \underline{5n^3} + n^2 + 22n + 6 = \underline{O(n^3)}$
  - para $c = 6$ e $n_0 = 10$, $f(n) \leq 6n^3$
- $f(n)$ também é $O(n^4)$?
  - Sim, mas estamos interessados em limites "apertados" para f.

Analisando algoritmos

Considere a linguagem $L_1 = \{0^k1^k \mid k \geq 0\}$, a seguinte **MT** decide L_1.

$$0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ \sqcup\ \ldots$$

- $M_1 =$ "Sobre a cadeia de entrada w:
  1. Faça uma *varredura* na fita e rejeite w ✗ se $w \neq 0^*1^*$.
  2. Repita o próximo passo **enquanto existir** ambos 0s e 1s na fita:
     3. Faça uma *varredura* na fita "marcando" um único 0 e um único 1.
  4. Se restarem 0s após sobreescrever todos os 1s (ou o contrário), rejeite w ✗, senão aceite w ✓"

Complexidade de tempo:

$$O(n) + \underbrace{O(n^2)}_{\text{passo 2}} + O(n) = O(n^2)$$

Número de passos: (i) $2n$, (ii) $n/2 \times O(n)$, (iv) $1 \times O(n)$.

Analisando algoritmos

Considere a linguagem $L_1 = \{0^k 1^k \mid k \geq 0\}$, a seguinte **MT** decide L_1.

$$\overset{\downarrow}{0}\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ \sqcup\ \ldots$$

- M_1 = "Sobre a cadeia de entrada w:
  1. Faça uma *varredura* na fita e rejeite w ✗ se $w \neq 0^*1^*$.
  2. Repita o próximo passo **enquanto existir** ambos 0s e 1s na fita:
     3. Faça uma *varredura* na fita "marcando" um único 0 e um único 1.
  4. Se restarem 0s após sobreescrever todos os 1s (ou o contrário), rejeite w ✗, senão aceite w ✓"

Complexidade de tempo:

$$O(n) + \underbrace{O(n^2)}_{\text{passo 2}} + O(n) = O(n^2)$$

Número de passos: (i) $2n$, (ii) $n/2 \times O(n)$, (iv) $1 \times O(n)$.

Analisando algoritmos

Vamos fixar uma notação para classificar linguagens (problemas) a partir de seus requisitos de tempo.

Definição

Seja $t : \mathbb{N} \rightarrow \mathbb{R}^+$ uma função. A classe de complexidade de tempo

$$TIME(t(n))$$

é a coleção de todas as linguagens decidíveis em tempo $t(n)$.

Então temos que $L_1 = \{0^k 1^k \mid k \geq 0\}$ é $TIME(n^2)$

- Será que existe outra **MT** (algoritmo) que decide L_1 em menos tempo?

Analisando algoritmos

A linguagem $L_1 = \{0^k 1^k \mid k \geq 0\}$ pode ser decida em tempo $O(n \log n)$

$$\overset{\downarrow}{0}\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ \sqcup\ \ldots$$

Nova ideia:

- $M_2 =$ "Sobre a cadeia de entrada w:
  1. Faça uma *varredura* na fita e rejeite w ✗ se $w \neq 0^*1^*$.
  2. Repita o próximo passo **enquanto existir** ambos 0s e 1s na fita:
     3. Faça uma *varredura* na fita contando se o **número** de 0s e 1s é **ímpar**, se for rejeite w ✗.
     4. Faça uma *varredura* na fita "marcando" alternadamente um 0 sim, outro não, faço o mesmo para os 1s.
  5. Se restarem 0s ou 1s, rejeite w ✗, senão aceite w ✓"

Complexidade de tempo:

$$O(n) + \underbrace{O(n \log n)}_{\text{passo 2}} + O(n) = O(n \log n)$$

Número de passos: (i) $2n$, (ii) $\lceil \log n \rceil \times O(n)$, (iv) $1 \times O(n)$.

Analisando algoritmos

A linguagem $L_1 = \{0^k1^k \mid k \geq 0\}$ pode ser decida em tempo $O(n \log n)$

$$\overset{\downarrow}{0}\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \sqcup \ldots$$

Nova ideia:

- M_2 = "Sobre a cadeia de entrada w:
  1. Faça uma *varredura* na fita e rejeite w ✗ se $w \neq 0^*1^*$.
  2. Repita o próximo passo **enquanto existir** ambos 0s e 1s na fita:
     3. Faça uma *varredura* na fita contando se o **número** de 0s e 1s é **ímpar**, se for rejeite w ✗.
     4. Faça uma *varredura* na fita "marcando" alternadamente um 0 sim, outro não, faço o mesmo para os 1s.
  5. Se restarem 0s ou 1s, rejeite w ✗, senão aceite w ✓"

Complexidade de tempo:

$$O(n) + \underbrace{O(n \log n)}_{\text{passo 2}} + O(n) = O(n \log n)$$

Número de passos: (i) $2n$, (ii) $\lceil \log n \rceil \times O(n)$, (iv) $1 \times O(n)$.

Analisando algoritmos

Então, na verdade, $L_1 = \{0^k 1^k \mid k \geq 0\} \in TIME(n \log n)$.

- Será que existe outra **MT** (algoritmo) ainda mais rápida para L_1?
  - Sim e não!
  - Na verdade, somente **Linguagens Regulares** podem ser decididas em menos de $O(n \log n)$ por uma **MT** de uma única fita.

Podemos decidir L_1 em $O(n)$, *tempo linear*, com uma **MT** de 2 fitas.

Ideia de M_3 (duas fitas):

1. Copie todos os 0s na segunda fita;
2. Compare os 0s e 1s.

Complexidade de tempo:

$$\underbrace{1 \cdot O(n)}_{\text{passos 1 e 2}} = O(n)$$

Essa complexidade de tempo é a melhor possível, já que para ler a entrada são necessário n passos.

Analisando algoritmos

Então, na verdade, $L_1 = \{0^k 1^k \mid k \geq 0\} \in TIME(n \log n)$.

- Será que existe outra **MT** (algoritmo) ainda mais rápida para L_1?
  - Sim e não!
  - Na verdade, somente **Linguagens Regulares** podem ser decididas em menos de $O(n \log n)$ por uma **MT** de uma única fita.

Podemos decidir L_1 em $O(n)$, *tempo linear*, com uma **MT** de 2 fitas.

Ideia de M_3 (duas fitas):

1. Copie todos os 0s na segunda fita;
2. Compare os 0s e 1s.

Complexidade de tempo:

$$\underbrace{1 \cdot O(n)}_{\text{passos 1 e 2}} = O(n)$$

Essa complexidade de tempo é a melhor possível, já que para ler a entrada são necessário n passos.

Analisando algoritmos

Então, na verdade, $L_1 = \{0^k 1^k \mid k \geq 0\} \in TIME(n \log n)$.

- Será que existe outra **MT** (algoritmo) ainda mais rápida para L_1?
  - Sim e não!
  - Na verdade, somente **Linguagens Regulares** podem ser decididas em menos de $O(n \log n)$ por uma **MT** de uma única fita.

Podemos decidir L_1 em $O(n)$, *tempo linear*, com uma **MT** de 2 fitas.

Ideia de M_3 (duas fitas):

1. Copie todos os 0s na segunda fita;
2. Compare os 0s e 1s.

\# \# \# \# \# \# 1 1 1 1 1 1 ⊔ ...

0 0 0 0 0 0 ⊔ ...

Complexidade de tempo:

$$\underbrace{1 \cdot O(n)}_{\text{passos 1 e 2}} = O(n)$$

Essa complexidade de tempo é a melhor possível, já que para ler a entrada são necessário n passos.

Analisando algoritmos

Então, na verdade, $L_1 = \{0^k 1^k \mid k \geq 0\} \in TIME(n \log n)$.

- Será que existe outra **MT** (algoritmo) ainda mais rápida para L_1?
  - Sim e não!
  - Na verdade, somente **Linguagens Regulares** podem ser decididas em menos de $O(n \log n)$ por uma **MT** de uma única fita.

Podemos decidir L_1 em $O(n)$, *tempo linear*, com uma **MT** de 2 fitas.

Ideia de M_3 (duas fitas):

1. Copie todos os 0s na segunda fita;
2. Compare os 0s e 1s.

```
# # # # # # 1 1 1 1 1 1 ␣ ...
0 0 0 0 0 0 ␣ ...
```

Complexidade de tempo:

$$\underbrace{1 \cdot O(n)}_{\text{passos 1 e 2}} = O(n)$$

Essa complexidade de tempo é a melhor possível, já que para ler a entrada são necessário n passos.

Relacionamento de Complexidade entre Modelos

Com isso, temos que:

- A classe de complexidade depende do modelo computacional escolhido:

$$\underbrace{L_1 \in TIME(n \log n)}_{\textbf{MT} \text{ de 1 fita}} \quad \text{e} \quad \underbrace{L_1 \in TIME(n)}_{\textbf{MT} \text{ multifita}}$$

Aqui temos uma diferença importante entre:

1. **Teoria da Computabilidade**: o modelo não importa para ser decidível/reconhecível ou não[1].
2. **Teoria da Complexidade**: a escolha do modelo pode afetar a complexidade de tempo.

[1] A tese de Church-Turing implica que todos os modelos razoáveis de computação são equivalentes.

Relacionamento de Complexidade entre Modelos

Com isso, temos que:

- A classe de complexidade depende do modelo computacional escolhido:

$$\underbrace{L_1 \in TIME(n \log n)}_{\textbf{MT}\text{ de 1 fita}} \quad \text{e} \quad \underbrace{L_1 \in TIME(n)}_{\textbf{MT}\text{ multifita}}$$

Aqui temos uma diferença importante entre:

1. **Teoria da Computabilidade**: o modelo não importa para ser decidível/reconhecível ou não[1].
2. **Teoria da Complexidade**: a escolha do modelo pode afetar a complexidade de tempo.

[1] A **tese de Church-Turing** implica que todos os modelos razoáveis de computação são equivalentes.

Relacionamento de Complexidade entre Modelos

Com isso, temos que:

- A classe de complexidade depende do modelo computacional escolhido:

$$\underbrace{L_1 \in TIME(n \log n)}_{\textbf{MT} \text{ de 1 fita}} \quad \text{e} \quad \underbrace{L_1 \in TIME(n)}_{\textbf{MT} \text{ multifita}}$$

Aqui temos uma diferença importante entre:

1. **Teoria da Computabilidade**: o modelo não importa para ser decidível/reconhecível ou não[1].
2. **Teoria da Complexidade**: a escolha do modelo pode afetar a complexidade de tempo.

[1]A **tese de Church-Turing** implica que todos os modelos razoáveis de computação são equivalentes.

Relacionamento de Complexidade entre Modelos

Qual modelo devemos escolher para classificar linguagens/problemas em $TIME(t(n))$?

- Vamos considerar o relacionamento de **tempo de execução** entre 3 modelos:
  1. **MT** (fita única);
  2. **MT** multifita;
  3. **MT** não-determinística (fita única).

$TIME(t(n))$ é a coleção de todas as linguagens decidíveis em tempo $t(n)$.

Relacionamento de Complexidade entre Modelos

Teorema 7.8 (Sipser)

Toda **MT multifita** que roda em tempo $t(n)$ possuí uma **MT (fita única)** equivalente que roda em tempo

$$O(t^2(n))$$

No pior dos casos, elevamos ao quadrado $t(n)$.

- Pode ser ruim na prática: $t^2(n^3) = O(n^6)$
- Mas $TIME(t(n))$ continua polinomial ← *ou seja, "tratável".*

Ideia da prova é analisar a simulação de uma **MT multifita** por uma **MT** (fita única) ← aula 10.

Relacionamento de Complexidade entre Modelos

Teorema 7.8 (Sipser)

Toda **MT multifita** que roda em tempo $t(n)$ possuí uma **MT (fita única)** equivalente que roda em tempo

$$O(t^2(n))$$

No pior dos casos, elevamos ao quadrado $t(n)$.

- Pode ser ruim na prática: $t^2(n^3) = O(n^6)$
- Mas $TIME(t(n))$ continua polinomial $\leftarrow$ *ou seja, "tratável".*

Ideia da prova é analisar a simulação de uma **MT multifita** por uma **MT** (fita única) $\leftarrow$ aula 10.

Relacionamento de Complexidade entre Modelos

Teorema 7.11 (Sipser)

Toda **MT não-determinística** que roda em tempo $t(n)$ possuí uma **MT (fita única)** equivalente que roda em tempo

$$2^{O(t(n))}$$

Nesse caso, a diferença de tempo pode ser muito maior, no máximo exponêncial $\leftarrow$ *considerada "intratável".*

Ideia da prova é analisar a simulação de uma **MT não-determinística** por uma **MT** (multifita), depois para uma **MT** (fita única) $\leftarrow$ aula 10.

Relacionamento de Complexidade entre Modelos

Para os nossos propósitos (em **Teoria da Complexidade**):

- Diferenças polinomiais serão consideradas pequenas;
- Diferenças exponênciais serão consideradas grandes (**enormes**).

Isso porque existe uma diferença dramática entre as **taxas de crescimento** de polinomios e exponênciais.

- Considere

$$n^3 \text{ e } 2^n, \text{ com } n = 1000$$

  - $n^3 = 1$ bilhão, não é muito em termos computacionais.
  - 2^n é **maior do que** o número de átomos no universo.

2^{1000} tem aproximadamente [illegible]

Relacionamento de Complexidade entre Modelos

Para os nossos propósitos (em **Teoria da Complexidade**):

- Diferenças polinomiais serão consideradas pequenas;
- Diferenças exponênciais serão consideradas grandes (**enormes**).

Isso porque existe uma diferença dramática entre as **taxas de crescimento** de polinomios e exponênciais.

- Considere

$$n^3 \text{ e } 2^n, \text{ com } n = 1000$$

  - $n^3 = 1$ bilhão, não é muito em termos computacionais.
  - 2^n é **maior do que** o número de átomos no universo.

2^{1000} tem aproximadamente 300 dígitos.

Relacionamento de Complexidade entre Modelos

Voltando à pergunta de qual modelo de **MT** escolher para classificar uma linguagem em $TIME(t(n))$?

- Todos os modelos computacionais determinísticos são polinomialmente equivalentes.
  1. **MT** (fita única);
  2. **MT** multifita;
  3. Computador real;
  4. Outros modelos "razoáveis" (**deterministicos**).

Para os *nossos propósitos* (que será classificar em polinomial e não-polinomial) a escolha do **modelo deterministico** não terá impacto.

- No exemplo, L_1 é resolvível em tempo polinomial;
- Isso nos permite desenvolver uma teoria independente do modelo.

Isto é, um simula o outro com uma diferença de tempo polinomial.

Relacionamento de Complexidade entre Modelos

Voltando à pergunta de qual modelo de **MT** escolher para classificar uma linguagem em $TIME(t(n))$?

- Todos os modelos computacionais determinísticos são polinomialmente equivalentes.
  1. **MT** (fita única);
  2. **MT** multifita;
  3. Computador real;
  4. Outros modelos "razoáveis" (**deterministicos**).

Para os *nossos propósitos* (que será classificar em polinomial e não-polinomial) a escolha do **modelo deterministico** não terá impacto.

- No exemplo, L_1 é resolvível em tempo polinomial;
- Isso nos permite desenvolver uma teoria independente do modelo.

Relacionamento de Complexidade entre Modelos

Voltando à pergunta de qual modelo de **MT** escolher para classificar uma linguagem em $\mathit{TIME}(t(n))$?

- Todos os modelos computacionais determinísticos são polinomialmente equivalentes.
  1. **MT** (fita única);
  2. **MT** multifita;
  3. Computador real;
  4. Outros modelos "razoáveis" (**deterministicos**).

Para os *nossos propósitos* (que será classificar em polinomial e não-polinomial) a escolha do **modelo deterministico** não terá impacto.

- No exemplo, L_1 é resolvível em tempo polinomial;
- Isso nos permite desenvolver uma teoria independente do modelo.

O nosso objetivo é estudar propriedades fundamentais da Computação.

Roteiro

A classe P

Em geral, dizemos que problemas que podem ser resolvidos em tempo polinomial, podem ser resolvidos em um *tempo aceitável*;

- Enquanto que problemas que levam tempo exponencial (ou mais) não podem ser resolvidos na prática (**intratáveis**).
- Esses algoritmos são "úteis" apenas para entradas pequenas.

A classe P

Em geral, dizemos que problemas que podem ser resolvidos em tempo polinomial, podem ser resolvidos em um *tempo aceitável*;

- Enquanto que problemas que levam tempo exponencial (ou mais) não podem ser resolvidos na prática (**intratáveis**).
- Esses algoritmos são "úteis" apenas para entradas pequenas.

A classe P

Agora, vamos definir uma importante classe de linguagens (problemas) em **Teoria de Complexidade**.

Definição

P é a classe de linguagens que são decidíveis em tempo polinomial sobre uma **MT** determinística.
Em outras palavras,

$$\mathrm{P} = \bigcup_k \mathit{TIME}(n^k)$$

1. P é invariante para os modelos de computação polinomialmente equivalentes à **MT** (modelos determinísticos); e
2. P corresponde $\approx$ à classe de problemas que são **realisticamente** solúveis por um computador.

A classe P

Então, quando um problema está em P, temos um algoritmo para resolvê-lo em tempo $O(n^k)$, **polinomial**, para alguma constante k.

- Esse tempo é prático? depende de k e da aplicação.
  - É improvável que $O(n^{100})$ seja útil na prática.
  - Entretanto, sempre que uma **solução polinomial** é encontrada para um problema (que não sabiamos estar em P), novas ideias são obtidas, o que pode permirtir futuras reduções em $t(n)$.

Com o passar do tempo, dizer que P é o limite da solubilidade tem se provado útil.

> *Essa vai ser a linha divisória entre problemas tratáveis e problemas intratáveis*

A classe P

Então, quando um problema está em P, temos um algoritmo para resolvê-lo em tempo $O(n^k)$, **polinomial**, para alguma constante k.

- Esse tempo é prático? depende de k e da aplicação.
  - É improvável que $O(n^{100})$ seja útil na prática.
  - Entretanto, sempre que uma **solução polinomial** é encontrada para um problema (que não sabiamos estar em P), novas ideias são obtidas, o que pode permirtir futuras reduções em $t(n)$.

Com o passar do tempo, dizer que P é o limite da solubilidade tem se provado útil.

> *Essa vai ser a linha divisória entre problemas tratáveis e problemas intratáveis*

A classe P

Então, quando um problema está em P, temos um algoritmo para resolvê-lo em tempo $O(n^k)$, **polinomial**, para alguma constante k.

- Esse tempo é prático? depende de k e da aplicação.
  - É improvável que $O(n^{100})$ seja útil na prática.
  - Entretanto, sempre que uma **solução polinomial** é encontrada para um problema (que não sabiamos estar em P), novas ideias são obtidas, o que pode permirtir futuras reduções em $t(n)$.

Com o passar do tempo, dizer que P é o limite da solubilidade tem se provado útil.

> *Essa vai ser a linha divisória entre problemas tratáveis e problemas intratáveis*

A classe P

Então, quando um problema está em P, temos um algoritmo para resolvê-lo em tempo $O(n^k)$, **polinomial**, para alguma constante k.

- Esse tempo é prático? depende de k e da aplicação.
  - É improvável que $O(n^{100})$ seja útil na prática.
  - Entretanto, sempre que uma **solução polinomial** é encontrada para um problema (que não sabiamos estar em P), novas ideias são obtidas, o que pode permirtir futuras reduções em $t(n)$.

Com o passar do tempo, dizer que P é o limite da solubilidade tem se provado útil.

> *Essa vai ser a linha divisória entre problemas tratáveis e problemas intratáveis*

A classe P

Então, quando um problema está em P, temos um algoritmo para resolvê-lo em tempo $O(n^k)$, **polinomial**, para alguma constante k.

- Esse tempo é prático? depende de k e da aplicação.
  - É improvável que $O(n^{100})$ seja útil na prática.
  - Entretanto, sempre que uma **solução polinomial** é encontrada para um problema (que não sabiamos estar em P), novas ideias são obtidas, o que pode permirtir futuras reduções em $t(n)$.

Com o passar do tempo, dizer que P é o limite da solubilidade tem se provado útil.

> *Essa vai ser a linha divisória entre problemas tratáveis e problemas intratáveis*

A classe P

Daqui por diante, descreveremos algoritmos por pseudo-códigos

- Sem considerar um modelo computacional específico (ou detalhes como fitas/cursores).
- Podemos fazer isso já que qualquer algoritmo pode rodar em uma **MT** em tempo polinomialmente equivalente.

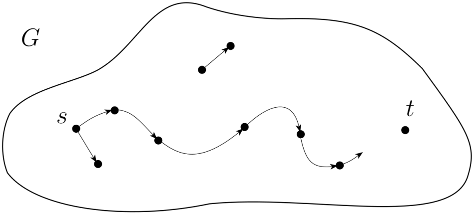
Um algoritmo é **polinomial** quando:

1. Cada estágio roda em tempo polinomial;
2. Cada estágio é executado um número polinomial de vezes;
3. A codificação/decodificação da entrada ocorrem em tempo polinomial.

CAM está em P?

Vamos considerar o problema de determinar se existe **um caminho** em um grafo direcionado G do nó $s \rightsquigarrow t$.

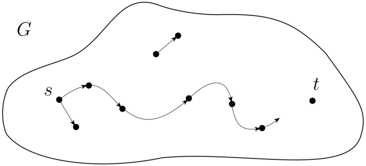
$$CAM = \{\langle G, s, t\rangle \mid G \text{ é um grafo que tem um caminho de } s \text{ para } t\}$$



- $CAM \in \mathrm{P}$?

A codificação de $\langle G, s, t\rangle$ pode ser feita (em tempo polinomial) como uma lista de nós e arestas $\leftarrow$ vimos um exemplo na aula 10.

CAM está em P?



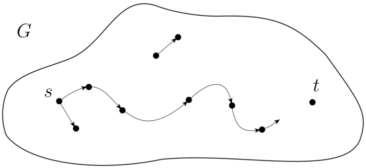
Uma solução (força bruta) decide *CAM* em tempo exponencial:

1. Lista todos os caminhos de tamanho máximo m (número de nós).
2. Verifica se um caminho $s \rightsquigarrow t$, nesse caso, aceite $\langle G, s, t \rangle$ ✓

O problema é que o número de caminho no pior caso (grafo completo):

$$\approx m^m \text{ (exponencial)}$$

CAM está em P?



Outra possibilidade é fazer uma busca em largura no grafo:

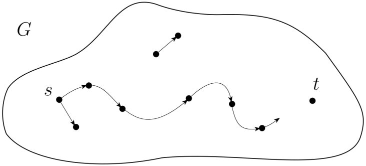
- Algoritmo: "Sobre a cadeia de entrada $\langle G, s, t \rangle$:
  1. "Marque" o nó s
  2. Repita o próximo passo **até que** nenhum novo nó seja marcado:
     3. Faça uma *varredura* na lista de arestas. Se (a, b) for encontrada, com a marcado e b não marcado, marque b.
  4. Se t estiver marcado aceite w ✓, senão rejeite w ✗"

Complexidade de Tempo:

$$\approx 1 + m + 1 = O(n) \Rightarrow CAM \in \mathbf{P} \text{ (polinomial)}$$

Número de passos: (i) $O(1)$, (ii) no pior caso (distância m): $O(m)$, (iv) $\times O(1)$.

CAM está em P?



Outra possibilidade é fazer uma busca em largura no grafo:

- Algoritmo: "Sobre a cadeia de entrada $\langle G, s, t \rangle$:
  1. "Marque" o nó s
  2. Repita o próximo passo **até que** nenhum novo nó seja marcado:
     3. Faça uma *varredura* na lista de arestas. Se (a, b) for encontrada, com a marcado e b não marcado, marque b.
  4. Se t estiver marcado aceite w ✓, senão rejeite w ✗"

Complexidade de Tempo:

$$\approx 1 + m + 1 = O(n) \Rightarrow CAM \in \mathbf{P} \text{ (polinomial)}$$

Número de passos: (i) $O(1)$, (ii) no pior caso (distância m): $O(m)$, (iv) $\times O(1)$.

Outros problemas em P?

$$PRIM\text{-}ES = \{\langle x, y \rangle \mid x \text{ e } y \text{ são inteiros primos entre si}\}$$

$$PRIM\text{-}ES \in \mathrm{P} \checkmark$$

$$A_{GLC} = \{\langle G, w \rangle \mid G \text{ é uma GLC que gera } w\}$$

$$A_{GLC} \in \mathrm{P} \checkmark$$

- E **muitos** outros ...

Teoremas 7.15 e 7.16 (Spiser).

A_{GLC} está relacionado ao problema de compilar linguagens de programação.

Outros problemas em P?

$$\mathit{PRIM\text{-}ES} = \{\langle x, y \rangle \mid x \text{ e } y \text{ são inteiros primos entre si}\}$$

$$\mathit{PRIM\text{-}ES} \in \mathrm{P} \checkmark$$

$$A_{GLC} = \{\langle G, w \rangle \mid G \text{ é uma GLC que gera } w\}$$

$$A_{GLC} \in \mathrm{P} \checkmark$$

- E **muitos** outros ...

Teoremas 7.15 e 7.16 (Spiser).
A_{GLC} está relacionado ao problema de compilar linguagens de programação.

Roteiro

A classe NP

Existem muitos (MUITOS) problemas interessantes, práticos e úteis, para os quais não conhecemos nenhuma solução **polinomial**.[2]

1. Pode ser que exista um **algoritmo polinomial**, ainda desconhecido.
2. Ou o problema é intrinsicamente difícil (simplesmente não existe solução em tempo polinomial).

No momento, não sabemos distinguir essas duas situações.

[2] O que existem são algoritmos exponenciais que fazem uma **busca exaustiva** em todo espaço de soluções.

A classe NP

Existem muitos (MUITOS) problemas interessantes, práticos e úteis, para os quais não conhecemos nenhuma solução **polinomial**.[2]

1. Pode ser que exista um **algoritmo polinomial**, ainda desconhecido.
2. Ou o problema é intrinsicamente difícil (simplesmente não existe solução em tempo polinomial).

No momento, não sabemos distinguir essas duas situações.

[2] O que existem são algoritmos exponenciais que fazem uma **busca exaustiva** em todo espaço de soluções.

A classe NP

Existem muitos (MUITOS) problemas interessantes, práticos e úteis, para os quais não conhecemos nenhuma solução **polinomial**.[2]

1. Pode ser que exista um **algoritmo polinomial**, ainda desconhecido.
2. Ou o problema é intrinsicamente difícil (simplesmente não existe solução em tempo polinomial).

No momento, não sabemos distinguir essas duas situações.

---

[2] O que existem são algoritmos exponenciais que fazem uma **busca exaustiva** em todo espaço de soluções.

A classe NP

Para certos problemas, embora para resolver (decidir) conhecemos apenas algoritmos exponenciais:

- Verificar se uma *solução candidata* é mesmo uma resposta para o problema pode ser feito em **tempo polinomial**.
- Essa classe de problemas é chamada de NP.

é fácil ver que $\mathrm{P} \subseteq \mathrm{NP}$

Vamos ver que esse termo NP vem de **decidível em tempo polinomial** sobre uma **MT** não-determinística.

A classe NP

Para certos problemas, embora para resolver (decidir) conhecemos apenas algoritmos exponenciais:

- Verificar se uma *solução candidata* é mesmo uma resposta para o problema pode ser feito em **tempo polinomial**.
- Essa classe de problemas é chamada de NP.

é fácil ver que $\mathrm{P} \subseteq \mathrm{NP}$

Vamos ver que esse termo NP vem de **decidível em tempo polinomial** sobre uma **MT** não-determinística.

A classe NP

Para certos problemas, embora para resolver (decidir) conhecemos apenas algoritmos exponenciais:

- Verificar se uma *solução candidata* é mesmo uma resposta para o problema pode ser feito em **tempo polinomial**.
- Essa classe de problemas é chamada de NP.

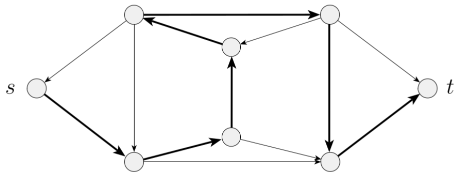
é fácil ver que $\mathrm{P} \subseteq \mathrm{NP}$

Vamos ver que esse termo NP vem de **decidível em tempo polinomial** sobre uma **MT** não-determinística.

CAMHAM está em NP?

Vamos considerar o problema de determinar se existe **um caminho Hamiltoniano** em um grafo direcionado G do nó $s \rightsquigarrow t$.

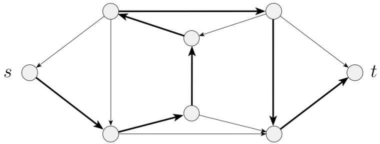
$$CAMHAM = \{\langle G, s, t\rangle \mid G \text{ é um grafo com um } \textbf{caminho Hamiltoniano} \text{ de } s \text{ para } t\}$$



- $CAMHAM \in \mathrm{NP}$?

Um **caminho Hamiltoniano** de $s \rightsquigarrow t$ é um caminho que passa por cada nó exatamente 1 vez.

CAMHAM está em NP?



Qual a complexidade de tempo de *CAMHAM*?

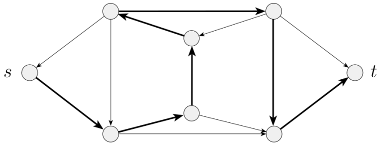
1. Não conhecemos nenhuma solução **polinomial**.
2. Mas! Dado um caminho $s \rightsquigarrow t$ no grafo, é "fácil" verificar se ele é ou não **Hamiltoniano** em tempo polinomial.

- $V =$ "Sobre a cadeia de entrada $\langle\langle G, s, t\rangle, c\rangle$:
  1. Verifique se o caminho c:
     - Começa em s e termina em t; e passa por todos os m nós 1 vez.
  2. Caso positivo aceite✓, senão rejeite ✗"

Portanto, *CAMHAM* $\in \mathrm{NP}$

Não sabemos se tal solução pode existir.

CAMHAM está em NP?



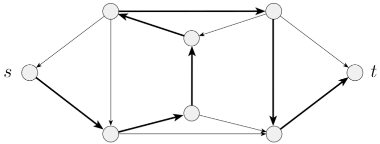
Qual a complexidade de tempo de *CAMHAM*?

1. Não conhecemos nenhuma solução **polinomial**.
2. Mas! Dado um caminho $s \rightsquigarrow t$ no grafo, é "fácil" verificar se ele é ou não **Hamiltoniano** em tempo polinomial.

- $V =$ "Sobre a cadeia de entrada $\langle\langle G, s, t\rangle, c\rangle$:
  1. Verifique se o caminho c:
     - Começa em s e termina em t; e passa por todos os m nós 1 vez.
  2. Caso positivo aceite✓, senão rejeite ✗"

Portanto, $CAMHAM \in \mathrm{NP}$

CAMHAM está em NP?



Qual a complexidade de tempo de *CAMHAM*?

1. Não conhecemos nenhuma solução **polinomial**.
2. Mas! Dado um caminho $s \rightsquigarrow t$ no grafo, é "fácil" verificar se ele é ou não **Hamiltoniano** em tempo polinomial.

- $V =$ "Sobre a cadeia de entrada $\langle\langle G, s, t\rangle, c\rangle$:
  1. Verifique se o caminho c:
     - Começa em s e termina em t; e passa por todos os m nós 1 vez.
  2. Caso positivo aceite✓, senão rejeite ✗"

Portanto, $CAMHAM \in \mathrm{NP}$

Número de passos: (i) no pior caso (distância m): $O(m)$, (ii) $\times O(1)$.

A classe NP

Definição

NP é a classe de linguagens que são verificáveis em **tempo polinomial** sobre uma **MT** determinística.

Obviamente,

$$\mathrm{P} \subseteq \mathrm{NP}$$

- Mas será que $\mathrm{P} = \mathrm{NP}$?
- Falaremos mais sobre isso (depois).

A classe NP

Na verdade, o termo NP vem de **tempo polinomial não-determinístico** (*Nondeterministic Polynomial Time*).

Definição (alternativa)

Uma linguagem está em NP se e somente se ela é decidida por uma **MT** não-determinística em **tempo polinomial**.

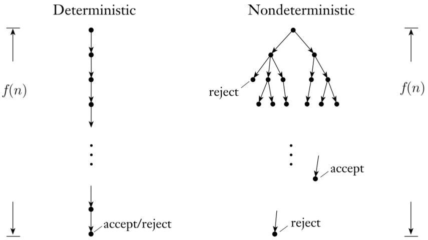
Mais uma vez,

$$\mathrm{P} \subseteq \mathrm{NP}$$

- Toda **MT** é uma **MT** não-determinística (sem opções de movimento).
- A **MT** não-determinística "adivinha" (em **tempo polinomial**) um dos caminhos de computação para aceitar a linguagem.

A classe NP

A intuição é que a **MT** não-determinística tem a habilidade de avaliar *"em paralelo"* todo o **espaço de possíveis soluções** (que pode ser exponencial) e verificar cada ramo em **tempo polinomial**



- Com isso, falar que um problema é verificável em tempo polinomial é o mesmo que falar que existe uma **MTN** que decide o problema em **tempo polinomial**.

Se existe um "caminho de computação" para a solução a **MTN** o encontra (mesmo caminho para a verificação).

A classe NP

Definição

Seja $t : \mathbb{N} \to \mathbb{R}^+$ uma função. A classe de complexidade de tempo não-deterministico

$$NTIME(t(n))$$

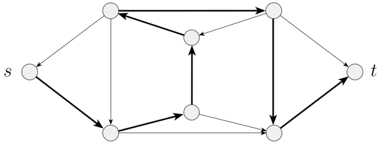
é a coleção de todas as linguagens decidíveis por alguma **MT** não-determinística em tempo $t(n)$.

Com isso,

$$\mathrm{NP} = \bigcup_k NTIME(n^k)$$

Tempo polinomial **não-deterministico**.

A classe co-NP



Para alguns problemas, nem para a verificação conhecemos um **algoritmo polinomial**.

$\overline{CAMHAM} = \{\langle G, s, t\rangle \mid G$ é um grafo que não tem um **caminho Hamiltoniano** de s para $t\}$

- Não sabemos como verificar se um grafo não tem *CAMHAM* em **tempo polinomial**
- Solução exponencial: (i) liste todos os m^m caminhos e (ii) verifique-os.

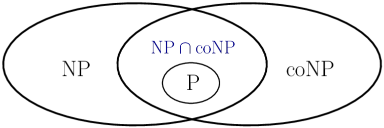
A classe CO-NP

Aparentemente, verificar se algo não está presente "parece" mais difícil[3].

- Definimos uma classe de complexidade separada para esses problemas, CO-NP:

  a classe das linguagens que são complemento de $L \in \mathrm{NP}$

- Não se sabe se $\text{CO-NP} \neq \mathrm{NP}$.



[3] Conhecemos apenas **soluções exponenciais**.

Roteiro

A questão P versus NP

Resumindo:

1. P: é a classe de problemas que podem ser resolvidos em **tempo polinomial** (*"rapidamente"*).
2. NP: é a classe de problemas que podem ser verificados em **tempo polinomial**.

Um exemplo de problemas em NP, mas não sabemos se pertence à P é o *CAMHAM*.

- O poder/capacidade de verificar "parece" ser muito maior do que o de decididir (em **tempo polinomial**).
- No entanto, por mais difícil que seja imaginar, pode ser que

$$\mathrm{P} = \mathrm{NP}$$

Somos incapazes de provar um único problema que esteja em NP mas que **não esteja** em P.

A questão P versus NP

Resumindo:

1. P: é a classe de problemas que podem ser resolvidos em **tempo polinomial** (*"rapidamente"*).
2. NP: é a classe de problemas que podem ser verificados em **tempo polinomial**.

Um exemplo de problemas em NP, mas não sabemos se pertence à P é o *CAMHAM*.

- O poder/capacidade de verificar "parece" ser muito maior do que o de decididir (em **tempo polinomial**).
- No entanto, por mais difícil que seja imaginar, pode ser que

$$\mathrm{P} = \mathrm{NP}$$

Somos incapazes de provar um único problema que esteja em NP mas que **não esteja** em P.

A questão P versus NP

Resumindo:

1. P: é a classe de problemas que podem ser resolvidos em **tempo polinomial** (*“rapidamente”*).
2. NP: é a classe de problemas que podem ser verificados em **tempo polinomial**.

Um exemplo de problemas em NP, mas não sabemos se pertence à P é o *CAMHAM*.

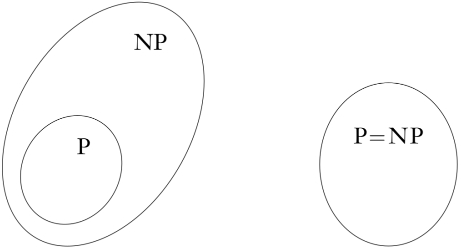
- O poder/capacidade de verificar “parece” ser muito maior do que o de decididir (em **tempo polinomial**).
- No entanto, por mais difícil que seja imaginar, pode que

$$\mathrm{P} = \mathrm{NP}$$

Somos incapazes de provar um único problema que esteja em NP mas que **não esteja** em P.

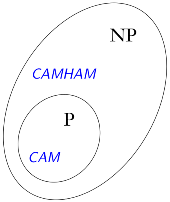
A questão $P = NP$? é um dos **maiores** problemas em aberto da Computação teórica e Matemática[4].

- Dois cenários possíveis:



[4] https://en.wikipedia.org/wiki/Millennium_Prize_Problems

A questão P versus NP



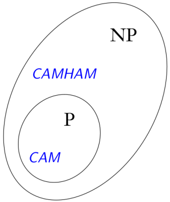
Muitos (a maioria) acreditam que $\mathrm{P} \neq \mathrm{NP}$.

- Isso por que muitas tentativas foram feitas para encontrar **soluções polinomiais** para *certos problemas* em NP (todas sem sucesso).
  - Esses problemas são chamados de **NP-completos** (uma subclasse especial).
  - O que provaria que $\mathrm{P} = \mathrm{NP}$.

Mas, provar (de fato) que $\mathrm{P} \neq \mathrm{NP}$ também é difícil.

- Envolve provar que um algoritmo não existe.
- Ou, mostrar que o método conhecido para simular uma **MTN** em uma **MT** determinística (tempo exponencial) é o **melhor possível**.

A questão P versus NP



Muitos (a maioria) acreditam que $\mathrm{P} \neq \mathrm{NP}$.

- Isso por que muitas tentativas foram feitas para encontrar **soluções polinomiais** para *certos problemas* em NP (todas sem sucesso).
  - Esses problemas são chamados de **NP-completos** (uma subclasse especial).
  - O que provaria que $\mathrm{P} = \mathrm{NP}$.

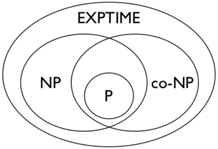
Mas, provar (de fato) que $\mathrm{P} \neq \mathrm{NP}$ também é difícil.

- Envolve provar que um algoritmo não existe.
- Ou, mostrar que o método conhecido para simular uma **MTN** em uma **MT** determinística (tempo exponencial) é o **melhor possível**.

A questão P versus NP

No momento, sabemos que NP é um subconjunto da classe dos problema exponenciais[5].

$$\mathrm{NP} \subseteq EXPTIME = \bigcup_k TIME(2^k)$$



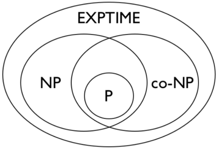
- Mas, não sabemos se NP está contida em outra classe de problemas de tempo **determinístico** menor.

[5]A **MTN** que resolve o problema em tempo **polinomial não-determinístico** sempre pode ser convertida em tempo exponencial: $2^{O(t(n))}$.

A questão P versus NP

No momento, sabemos que NP é um subconjunto da classe dos problema exponenciais[5].

$$\mathrm{NP} \subseteq \mathit{EXPTIME} = \bigcup_k \mathit{TIME}(2^k)$$



- Mas, não sabemos se NP está contida em outra classe de problemas de tempo **determinístico** menor.

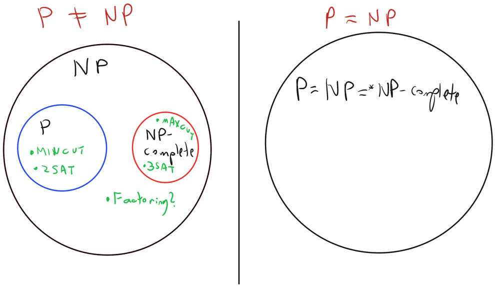
[5] A MTN que resolve o problema em tempo **polinomial não-determinístico** sempre pode ser convertida em tempo exponencial: $2^{O(t(n))}$.

Roteiro

NP-completude

Existe uma classe especial de problemas em NP que "*capturam*" a dificuldade de resolver qualquer outro problema em NP (em **tempo polinomial**).

- Um problema é NP-completo se **todo problema** em NP pode ser **reduzido** à ele em **tempo polinomial**.



Qual a relação com P versus NP?

Voltando à questão P versus NP

1. Se houver uma única solução **polinomial** para algum problema NP-completo, então todos os problemas em NP estão em P:

$$\mathrm{P} = \mathrm{NP}$$

2. Por outro lado, se alguém provar que **não existe** solução **polinomial** para algum problema NP-completo:

$$\mathrm{P} \neq \mathrm{NP}$$

Mas, **até hoje** ninguém resolveu nenhum desses problemas.

Voltando à questão P versus NP

1. Se houver uma única solução **polinomial** para algum problema NP-completo, então todos os problemas em NP estão em P:

$$\mathrm{P} = \mathrm{NP}$$

2. Por outro lado, se alguém provar que **não existe** solução **polinomial** para algum problema NP-completo:

$$\mathrm{P} \neq \mathrm{NP}$$

Mas, **até hoje** ninguém resolveu nenhum desses problemas.

Voltando à questão P versus NP

1. Se houver uma única solução **polinomial** para algum problema NP-completo, então todos os problemas em NP estão em P:

$$\mathrm{P} = \mathrm{NP}$$

2. Por outro lado, se alguém provar que **não existe** solução **polinomial** para algum problema NP-completo:

$$\mathrm{P} \neq \mathrm{NP}$$

Mas, **até hoje** ninguém resolveu nenhum desses problemas.

Redutibilidade por mapeamento

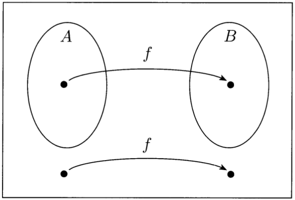
O conceito de redução de um problema A **em tempo polinomial** a um outro problema B, é o mesmo que vimos anteriormente, agora denotado por

$$A \leq_P B$$

- A **função computável** $f : \Sigma^* \to \Sigma^*$, deve rodar em **tempo polinomial**.

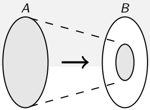
- Podemos *converter* questões do tipo

  "$w \in A$" em "$f(w) \in B$"



Podemos relacionar a "dificuldade" de resolução dos problemas A e B.

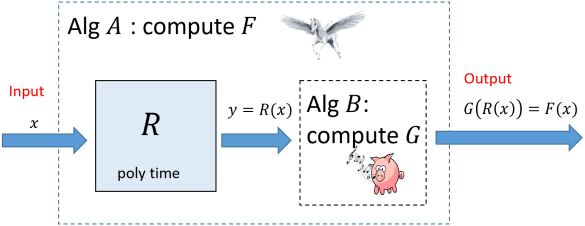
Redutibilidade por mapeamento



Primeiro resultado interessante:

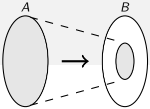
Teorema 7.31 (Sipser)

Se $A \leq_P B$ e $B \in \mathrm{P}$, **então** $A \in \mathrm{P}$



- Podemos "reutilizar" a solução de B para $f(w)$ em **tempo polinomial**.

NP-completude

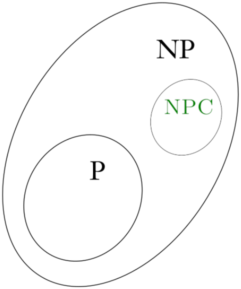


Teorema 7.34 (Sipser)

Uma linguagem B é NP-completa se satisfaz:

1. B está em NP; e
2. toda linguagem $A \in \mathrm{NP}$ é **redutível em tempo polinomial** a B.

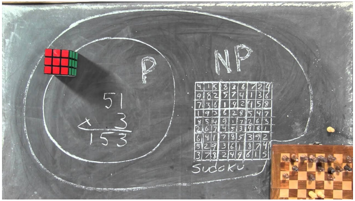
- Podemos falar que esses são os mais difícies problemas em NP.



NP-completude

Não é imediatamente óbvio que problemas NPC existam, mas acontece que existem vários deles (WIKIPEDIA lista $\approx$ 3.000).

- Aparentemente, problemas NPC são mais comuns na prática, do que problemas que estão apenas em NP.
- Existe uma **grande chance** de você se deparar com um problema desses: *SAT*, *CAMHAM*, *CLIQUE*, *SOMA-SUB*, ...



https://youtu.be/YX40hbAHx3s

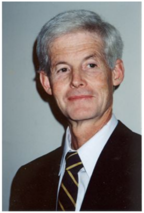
https://en.wikipedia.org/wiki/List_of_NP-complete_problems

NP-completude

Historicamente, o primeiro problema provado NPC foi o problema da **satisfabilidade** (*SAT*).

Teorema 7.27 (Sipser)

SAT é NP-completo



- Esse resultado foi provado por Stephen Cook e Leonid Levin (independentemente) no início dos anos 1970.

Teorema de Cook-Levin.

SAT

O problema *SAT* consiste em verificar se uma fórmula booleana é **satisfazível** ou não, isto é, se existe alguma atribuição de 0s e 1s às variáveis que faz a fórmula ter valor 1.

- Por exemplo:

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

é satisfazível,
pois $x = 0$, $y = 1$ e $z = 0$ fazem com que $\phi = 1$.

Variáveis boolenas: valores falso ou verdadeiro (0 ou 1).
Dizemos que a atribuição $x = 0$, $y = 1$ e $z = 0$ **satisfaz** ϕ.

SAT é NP-completo

Ideia da prova:

1. *SAT* $\in \mathrm{NP}$: é fácil ver que podemos verificar uma solução **em tempo polinomial**
2. Todo problema em NP é **redutível** (**em tempo polinomial**) ao *SAT*.
   - Mais difícil!!
   - A ideia é mostrar que uma **MTN** (que decide um problema em NP) pode ser representada por uma expressão booleana.

$$\text{A } \textbf{MTN} \text{ aceita } w \iff \phi \text{ é satisfazível}$$

Parece "razoável" já que circuitos eletrônicos
são baseados em operações booleanas.

SAT é NP-completo

Ideia da prova:

1. *SAT* $\in \mathrm{NP}$: é fácil ver que podemos verificar uma solução **em tempo polinomial**
2. Todo problema em NP é **redutível (em tempo polinomial)** ao *SAT*.
   - Mais difícil!!
   - A ideia é mostrar que uma **MTN** (que decide um problema em NP) pode ser representada por uma expressão booleana.

A **MTN** aceita $w \iff \phi$ é satisfazível

Parece "razoável" já que circuitos eletrônicos
são baseados em operações booleanas.

Ver prova no livro, Spiser (pág. 293).

SAT é NP-completo

Ideia da prova:

1. *SAT* $\in \mathrm{NP}$: é fácil ver que podemos verificar uma solução **em tempo polinomial**
2. Todo problema em NP é **redutível (em tempo polinomial)** ao *SAT*.
   - Mais difícil!!
   - A ideia é mostrar que uma **MTN** (que decide um problema em NP) pode ser representada por uma expressão booleana.

A **MTN** aceita $w \iff \phi$ é satisfazível

Parece "razoável" já que circuitos eletrônicos são baseados em operações booleanas.

Ver prova no livro, Spiser (pág. 293).

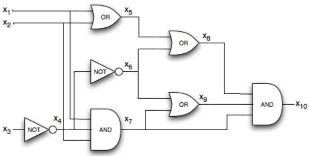
SAT é NP-completo

Ideia da prova:

1. $SAT \in \mathrm{NP}$: é fácil ver que podemos verificar uma solução **em tempo polinomial**
2. Todo problema em NP é **redutível (em tempo polinomial)** ao SAT.
   - Mais difícil!!
   - A ideia é mostrar que uma **MTN** (que decide um problema em NP) pode ser representada por uma expressão booleana.

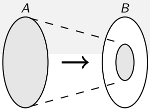
$$\text{A } \textbf{MTN} \text{ aceita } w \iff \phi \text{ é satisfazível}$$

Parece "razoável" já que circuitos eletrônicos são baseados em operações booleanas.



Ver prova no livro, Spiser (pág. 293).

SAT é NP-completo

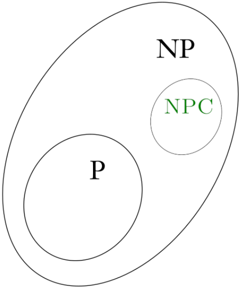


Conhecer um problema NPC é **extremamente poderoso**

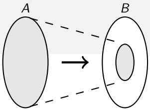
Teorema 7.36 (Sipser)

Se $B \in \mathrm{NPC}$ e $B \leq_P C$, para C em NP, então $C \in \mathrm{NPC}$.

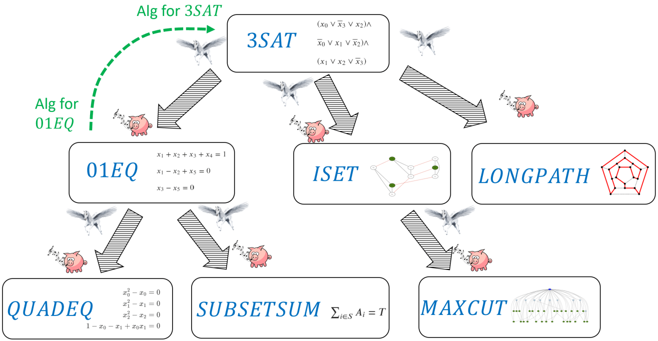
- Para provar que um problema C é NPC, basta reduzir o *SAT* (ou outro) para ele.
- Não é preciso uma prova direta, apenas uma redução (**tempo polinomial**).



SAT é NP-completo



O problema SAT serve como uma "semente" para encontrar outros problemas NP-completos



Por que é importante?

Problemas NP-completos formam a base para os **melhores argumentos** de que um problema é **intratável**.

- Não é uma prova formal (**questão em aberta** P versus NP)
- Para efeitos práticos, caso você encontre um problema NPC[6], não vale a pena tentar uma **solução polinomial**.
- O melhor a fazer é buscar alternativas: aproximações, heurísticas, ...

[6] Você deve provar que ele é NPC (por redução, por exemplo)

Por que é importante?

Problemas NP-completos formam a base para os **melhores argumentos** de que um problema é **intratável**.

- Não é uma prova formal (**questão em aberta** P versus NP)
- Para efeitos práticos, caso você encontre um problema NPC[6], não vale a pena tentar uma **solução polinomial**.
- O melhor a fazer é buscar alternativas: aproximações, heurísticas, ...

[6] Você deve provar que ele é NPC (por redução, por exemplo)

Por que é importante?

Problemas NP-completos formam a base para os **melhores argumentos** de que um problema é **intratável**.

- Não é uma prova formal (**questão em aberta** P versus NP)
- Para efeitos práticos, caso você encontre um problema NPC[6], não vale a pena tentar uma **solução polinomial**.
- O melhor a fazer é buscar alternativas: aproximações, heurísticas, ...

[6] Você deve provar que ele é NPC (por redução, por exemplo)

NP-completude

Vamos finalizar (o curso!) com um exemplo:

- Suponha que você encontrou um problema A.
  1. Você sabe como verificar a resposta (em tempo polinomial),
  2. Mas acha que o problema é **intratável**.

- O seu chefe não acredita em você e quer que você encontre uma solução eficiente para o problema A.

O que você pode fazer nessa situação?

$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

NP-completude

Vamos finalizar (o curso!) com um exemplo:

- Suponha que você encontrou um problema A.
  1. Você sabe como verificar a resposta (em tempo polinomial),
  2. Mas acha que o problema é **intratável**.

- O seu chefe não acredita em você e quer que você encontre uma solução eficiente para o problema A.

O que você pode fazer nessa situação?

$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

NP-completude

Vamos finalizar (o curso!) com um exemplo:

- Suponha que você encontrou um problema A.
  1. Você sabe como verificar a resposta (em tempo polinomial),
  2. Mas acha que o problema é **intratável**.

- O seu chefe não acredita em você e quer que você encontre uma solução eficiente para o problema A.

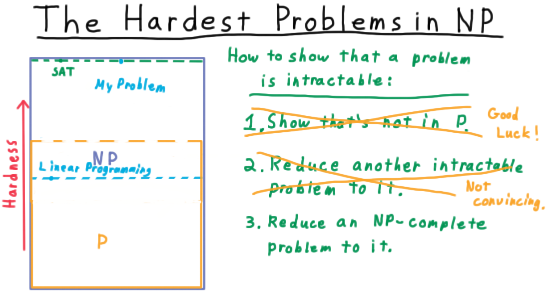
O que você pode fazer nessa situação?

$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

NP-completude

Algumas possibilidade:

1. Provar que $A \notin \mathrm{P}$
   - Isso de fato resolveria o problema, mas você faria muito mais! Você estaria provando que $\mathrm{P} \neq \mathrm{NP}$

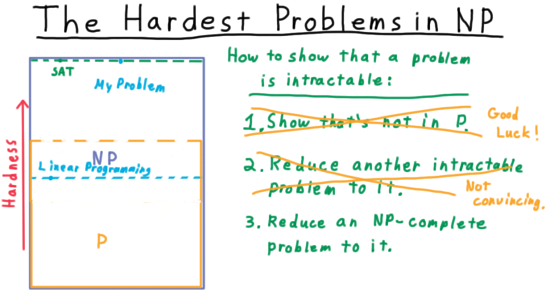


$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

NP-completude

Algumas possibilidade:

2. Reduzir um problema em NP B para A, isto é, mostrar $B \leq_P A$
   - Isso não significa muita coisa, apenas que B é **tão dificil** quanto A, só que B pode ser mostrado mais tarde em P.

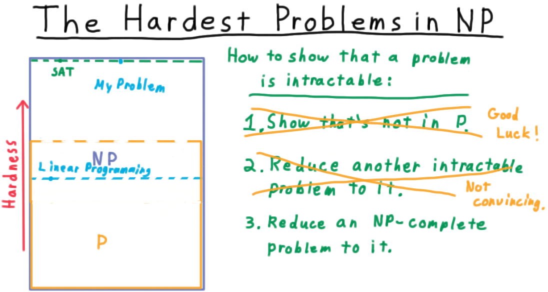


$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

NP-completude

Algumas possibilidade:

3. Reduzir um problema NPC C para A, isto é, mostrar $C \leq_P A$
   - Isso mostra que o seu problema é **tão dificil** quanto qualquer outro problema NPC, e que ele é NPC.



A menos que $\mathrm{P} = \mathrm{NP}$, você pode dizer que A é intratável.

$A \in \mathrm{NP}$ mas não sabemos se $A \in \mathrm{P}$.

Fim

Dúvidas?

Roteiro

Referências

Referências:

1. *"Introdução à Teoria da Computação"* de M. Sipser, 2007.
2. *"Introdução à Teoria de Autômatos, Linguagens e Computação"* de J. E. Hopcroft, R. Motwani, e J. D. Ullman, 2003.