Provas por Diagonalização

Computabilidade e Complexidade

Márcio Nicolau

2025-08-18

Table of contents

List of Figures

Introdução: A Ferramenta para Provar o Impossível

Na aula anterior, estabelecemos uma ideia surpreendente: existem infinitos de tamanhos diferentes. Vimos que o conjunto dos números reais ($\mathbb{R}$) é "maior" que o conjunto dos números naturais ($\mathbb{N}$). A ferramenta que nos permitiu provar isso foi o **Argumento de Diagonalização de Cantor**.

Hoje, vamos mergulhar fundo nessa técnica. A diagonalização não é apenas um truque matemático; é uma das ferramentas mais poderosas da teoria da computação. É a chave que nos permitirá provar, mais adiante no curso, que existem problemas computacionais fundamentalmente impossíveis de serem resolvidos. Um exemplo famoso é o **Problema da Parada (Halting Problem)**.

O objetivo desta aula é:

> **Aplicar a técnica de diagonalização para provar a não enumerabilidade de diferentes conjuntos relevantes para a computação.**

Vamos dissecar a "receita" de uma prova por diagonalização e aplicá-la a um novo domínio: o conjunto de todas as linguagens computacionais.

A Receita da Prova por Diagonalização

Uma prova por diagonalização é, em sua essência, uma elegante **prova por contradição**. A estrutura geral, independentemente do conjunto que estamos analisando, segue os mesmos passos lógicos.

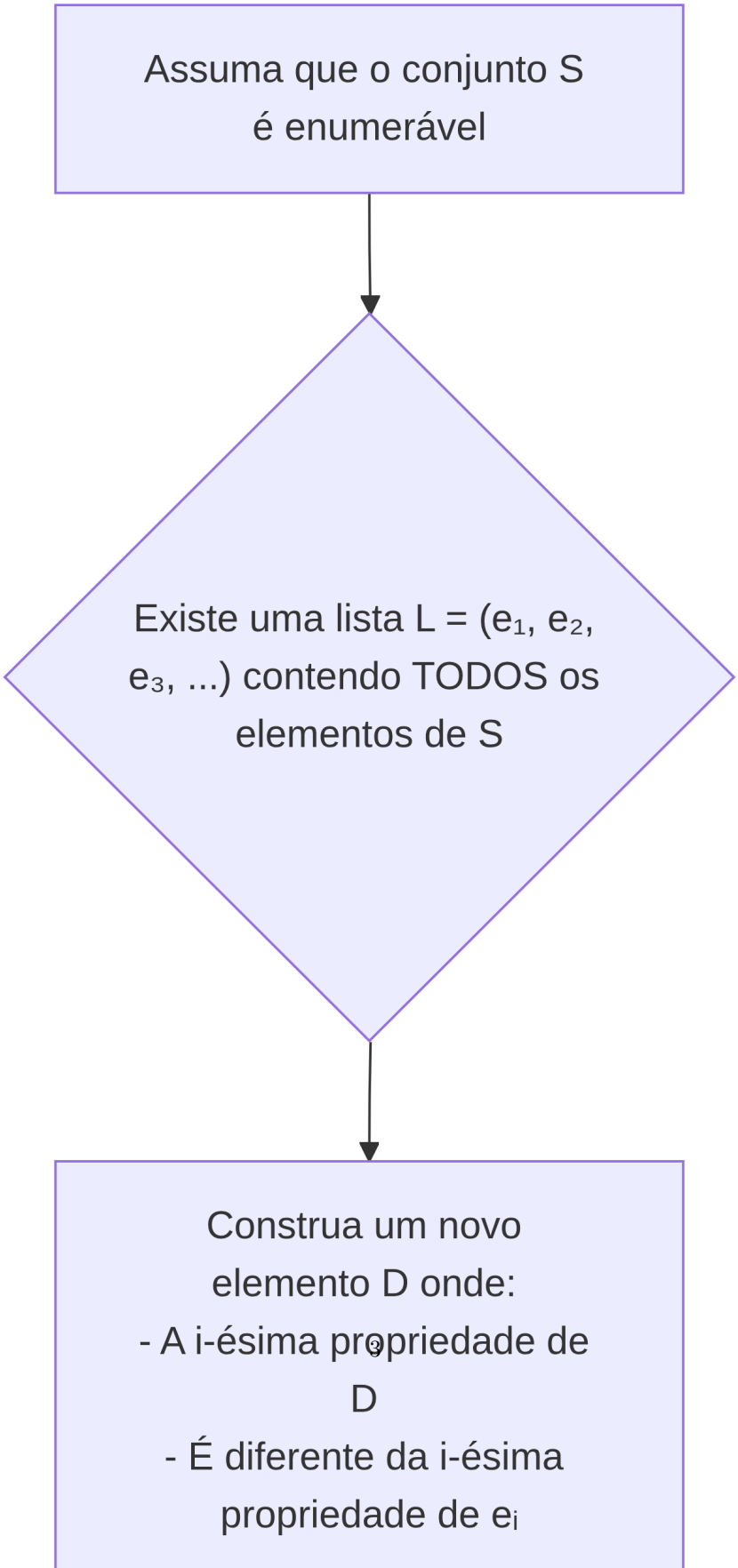
Vamos formalizar essa "receita":

1. **A Hipótese Contraditória:** Comece assumindo o oposto do que você quer provar. Se você quer provar que o conjunto `S` é não enumerável, sua hipótese inicial é: "**Suponha que `S` seja enumerável.**"
2. **A Consequência da Hipótese:** Se `S` é enumerável, isso significa que podemos criar uma lista (uma enumeração) que contém *todos* os elementos de `S`, sem exceção. Vamos chamar essa lista de $e_1, e_2, e_3, ...$
3. **A Construção do "Antagonista":** Este é o coração da prova. Construa um novo elemento, que chamaremos de D (de "diagonal"). Este elemento D é projetado especificamente para ser diferente de *todos* os elementos da lista. A construção funciona da seguinte forma:
   - Para garantir que $D \neq e_1$, fazemos com que a "primeira parte" de D seja diferente da "primeira parte" de e_1.
   - Para garantir que $D \neq e_2$, fazemos com que a "segunda parte" de D seja diferente da "segunda parte" de e_2.
   - ...
   - Para garantir que $D \neq e_i$, fazemos com que a "i-ésima parte" de D seja diferente da "i-ésima parte" de e_i.
4. **O Paradoxo:** Agora temos um problema.
   - Por sua construção, o elemento D pertence ao tipo de objetos do conjunto `S`.
   - No entanto, por sua construção, D é diferente de todos os elementos e_i da nossa lista.
   - Isso significa que D é um elemento de `S` que **não está na lista**.
5. **A Conclusão:** A existência de D contradiz nossa hipótese inicial de que a lista continha *todos* os elementos de `S`. A única saída lógica é que a hipótese estava errada. Portanto, o conjunto `S` **não é enumerável**.

Aplicação: O Conjunto de Todas as Linguagens é Não Enumerável

Vamos agora usar nossa receita para provar um resultado fundamental para a ciência da computação. Primeiro, algumas definições rápidas:

- **Alfabeto (Σ):** Um conjunto finito de símbolos. Para a computação, o mais comum é o alfabeto binário, $\Sigma = \{0, 1\}$.


Assuma que o conjunto S é enumerável
Existe uma lista L = (e_1, e_2, e_3, ...) contendo TODOS os elementos de S
Construa um novo elemento D onde:
- A i-ésima propriedade de D
- É diferente da i-ésima propriedade de e_i

- **String:** Uma sequência finita de símbolos de um alfabeto. Ex: “01101”.
- **Linguagem:** Um conjunto (possivelmente infinito) de strings.

Por exemplo, a linguagem “TODOS OS NÚMEROS PARES EM BINÁRIO” é o conjunto {“0”, “10”, “100”, “110”, ...}.

Teorema: O conjunto de todas as linguagens sobre o alfabeto $\Sigma = \{0, 1\}$ é **não enumerável**.

Vamos seguir a receita para provar isso.

Passo 1: A Hipótese Contraditória Suponha, para fins de contradição, que o conjunto de todas as linguagens sobre $\Sigma = \{0, 1\}$ **seja enumerável**.

Passo 2: A Consequência da Hipótese Isso significa que podemos criar uma lista $L_1, L_2, L_3, ...$ que contém *todas* as linguagens possíveis. Para trabalhar com isso, primeiro precisamos de uma lista de todas as *strings* possíveis. O conjunto de todas as strings, Σ^*, *é* enumerável. Podemos listá-las em ordem canônica (primeiro por tamanho, depois lexicograficamente): $s_1 = \epsilon$ (string vazia), $s_2 = 0$, $s_3 = 1$, $s_4 = 00$, $s_5 = 01, ...$

Agora, podemos representar nossa lista de linguagens em uma tabela infinita. As linhas são as linguagens (L_i) e as colunas são as strings (s_j). O valor da célula (i, j) é 1 se a string s_j pertence à linguagem L_i, e 0 caso contrário.

	$s_1(\epsilon)$	$s_2(0)$	$s_3(1)$	$s_4(00)$	...
L_1	1	0	1	0	...
L_2	0	1	1	1	...
L_3	1	0	0	1	...
L_4	0	1	0	0	...
...	...	...	...	...	...

Passo 3: A Construção da Linguagem “Antagonista” (L_D) Vamos construir uma nova linguagem, L_D, olhando para a diagonal da tabela (os valores em vermelho) e invertendo cada um deles. A regra de construção é: > A string s_i pertence a L_D **se e somente se** a string s_i **não** pertence à linguagem L_i.

No nosso exemplo: * $s_1 \notin L_D$ porque $s_1 \in L_1$. * $s_2 \notin L_D$ porque $s_2 \in L_2$. * $s_3 \in L_D$ porque $s_3 \notin L_3$. * $s_4 \in L_D$ porque $s_4 \notin L_4$. * ...e assim por diante.

A linguagem L_D é perfeitamente bem definida. É um conjunto de strings, portanto, é uma linguagem.

Passo 4: O Paradoxo Se nossa lista original estava completa, a linguagem L_D que acabamos de construir deve estar em algum lugar nela. Digamos que L_D seja a k-ésima linguagem da lista, ou seja, $L_D = L_k$.

Agora vamos nos fazer uma pergunta crucial: **a string s_k pertence a L_k?**

- Pela definição de L_D, a string s_k pertence a L_D se, e somente se, s_k **não** pertence a L_k.
- Mas nós assumimos que $L_D = L_k$.
- Substituindo, obtemos: s_k pertence a L_k se, e somente se, s_k **não** pertence a L_k.

Isso é uma contradição lógica da forma $P \iff \neg P$. É como dizer “esta afirmação é falsa”.

Passo 5: A Conclusão Nossa suposição de que poderíamos listar todas as linguagens nos levou a um paradoxo. Portanto, a suposição é falsa.

O conjunto de todas as linguagens sobre um alfabeto não é enumerável. (Sipser, 2012, p. 201)

Implicações: Por que isso é Importante para a Computação?

O resultado que acabamos de provar não é apenas um exercício teórico. Ele estabelece uma limitação fundamental da computação.

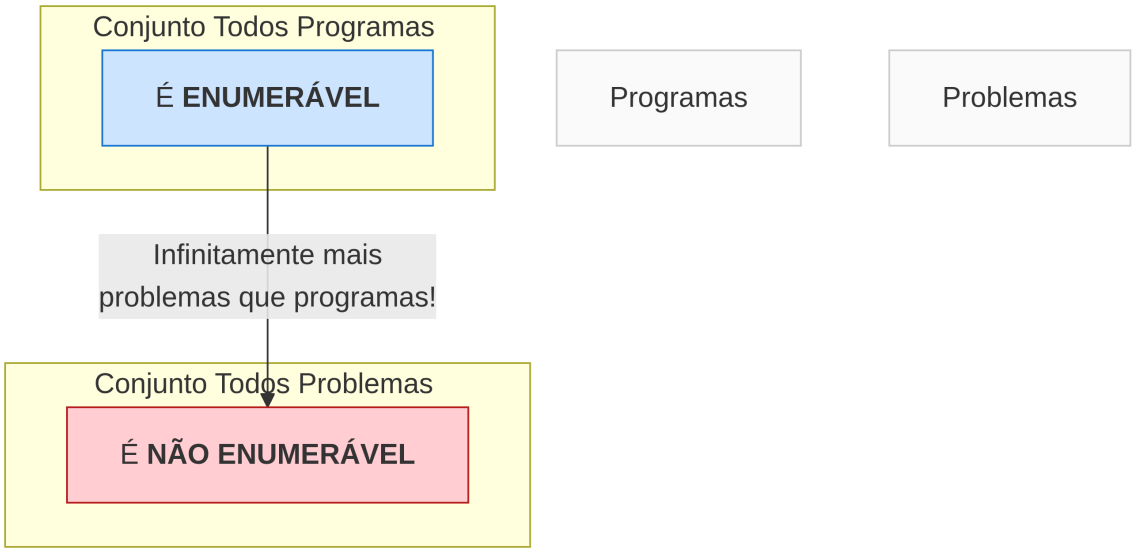


Figure 2: O descompasso entre a quantidade de programas e a quantidade de problemas.

1. **O Conjunto de todos os programas é enumerável:** Um programa de computador, em qualquer linguagem (Python, C++, Java), é apenas uma string finita de texto, escrita a partir de um alfabeto finito (caracteres ASCII ou Unicode). Assim como listamos todas as strings binárias, podemos, em teoria, listar todos os programas de computador possíveis.
2. **O Conjunto de todos os problemas (linguagens) é não enumerável:** Como acabamos de provar. Cada linguagem pode ser vista como um "problema de decisão": dada uma string, ela pertence à linguagem?

A Conclusão Inevitável: Se temos um conjunto enumerável de "soluções" (programas) e um conjunto não enumerável de "problemas" (linguagens), então deve haver uma quantidade infinita de problemas para os quais **não existe** um programa de computador correspondente que os resolva.

Esses problemas são chamados de **indecidíveis** ou **não computáveis**. A diagonalização nos deu a primeira prova matemática de que os computadores têm limites fundamentais.

Verificação de Aprendizagem

1: Conceitual

Explique com suas próprias palavras por que a construção do elemento "antagonista" D foca especificamente nos elementos da **diagonal** da tabela. O que aconteceria se tentássemos construir D usando, por exemplo, apenas a primeira coluna da tabela?

2: Prova por Diagonalização

Use o argumento de diagonalização para provar que o conjunto de todas as **sequências binárias infinitas** (ex: "010101…", "111111…", "101101…") é não enumerável. Siga os 5 passos da "receita" de forma explícita.

3: Desafio Python (Ilustrativo)

O código abaixo recebe uma *lista finita* de strings binárias de mesmo tamanho. Sua tarefa é completar a função `construir_antagonista` que, usando a ideia da diagonalização, gera uma nova string binária (do mesmo tamanho) que é garantidamente diferente de todas as strings da lista de entrada.

```python
def construir_antagonista(lista_de_strings: list[str]) -> str:
    """
    Dada uma lista de N strings binárias de tamanho N, constrói uma nova
    string binária de tamanho N que é diferente de todas as strings da lista.

    A nova string deve ser construída modificando a diagonal.
    O i-ésimo caractere da nova string deve ser o inverso do i-ésimo
    caractere da i-ésima string da lista.
    """
    antagonista = []
    # Itere pela lista, olhando para o caractere da diagonal
    for i, s in enumerate(lista_de_strings):
        # Pegue o i-ésimo caractere da i-ésima string (s)
        char_diagonal = s[i]

        # Inverta o caractere
        if char_diagonal == '0':
            antagonista.append('1')
        else:
            antagonista.append('0')

    return "".join(antagonista)

# Exemplo de uso:
minha_lista = [
    "1011",  # L1
    "0100",  # L2
    "1101",  # L3
```

```
    "0010"   # L4
]

# A diagonal é "1100"
# O antagonista deve ser "0011"

nova_string = construir_antagonista(minha_lista)
print(f"Lista de entrada: {minha_lista}")
print(f"String antagonista construída: {nova_string}")

# Verificação
if nova_string in minha_lista:
    print("Algo deu errado! A string foi encontrada na lista.")
else:
    print("Sucesso! A string construída não está na lista.")
```

```
Lista de entrada: ['1011', '0100', '1101', '0010']
String antagonista construída: 0011
Sucesso! A string construída não está na lista.
```

Referências Bibliográficas