EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE II

Aspectos Teóricos da Computação

Prof. Me. Hugo Insua

Decidibilidade na Teoria da Computação

- Nesta apresentação, vamos explorar como computadores resolvem problemas e quando tais problemas são decidíveis ou não.

Decidibilidade na Teoria da Computação

- Problemas decidíveis são aqueles que podem ser resolvidos por um algoritmo (máquina de Turing) em tempo finito.

- Sempre terminam e produzem uma resposta (aceita ou rejeita).

- São exemplos de problemas decidíveis a verificação de que um número é primo ou encontrar o caminho mais curto entre dois pontos em um grafo.

Decidibilidade na Teoria da Computação

- Problemas indecidíveis são aqueles que não podem ser decididos por nenhum algoritmo.
- Não há um procedimento geral para determinar uma resposta correta.
- São exemplos de problemas indecidíveis o problema da parada e o jogo da vida de John Conway.
- Vejamos, a seguir, a decidibilidade sobre as linguagens por meio de teorema e prova.

Decidibilidade – Linguagens Decídiveis

- Verificar se um autômato finito específico aceita uma determinada cadeia é algo que pode ser formulado como uma linguagem chamada "A_{AFD}", que inclui as representações de todos os autômatos finitos determinísticos, juntamente com as cadeias que esses autômatos aceitam.

- Considere a linguagem A_{AFD} = {⟨A,ω⟩/ A é um AFD que aceita "ω" }, ou seja, a linguagem que contém o par autômato e cadeia, denotado por ⟨A,ω⟩, sendo que autômato aceita a cadeia.

    - Aqui, temos a relação entre duas questões: a primeira é saber se o autômato A aceita a cadeia de entrada ω e a segunda é determinar se o par ⟨A,ω⟩ faz parte da linguagem A_{AFD}. Assim, provar a decidibilidade de uma linguagem equivale a mostrar que o problema computacional associado é solucionável.

Decidibilidade – Linguagens Decídiveis

- Teorema 1: A linguagem A_{AFD} é decidível.

Vejamos uma ideia de prova:

Tomamos uma máquina de Turing M que decida A_{AFD}. Assim:

M = “Sobre a entrada ⟨A,ω⟩, em que A é um AFD e ω é uma cadeia:

1. Simule A sobre a cadeia de entrada ω.

2. Se a simulação termina em um estado de aceitação, aceite. Se ela termina em um estado de não aceitação, rejeite.”

Decidibilidade – Linguagens Decídiveis

Prova:

- Consideramos a entrada $\langle A, \omega \rangle$. Para representar A de forma adequada, lista-se seus cinco componentes essenciais: conjunto de estados Q, alfabeto Σ, função de transição δ, estado inicial q0 e conjunto de estados finais F.

- Quando uma máquina de Turing (M) recebe essa entrada, sua primeira verificação é garantir que ela representa corretamente o AFD A juntamente com uma cadeia ω. Caso contrário, M rejeita a entrada.

Decidibilidade – Linguagens Decídiveis

- Em seguida, M prossegue com a simulação de uma maneira direta. Ela acompanha o estado atual de A e a posição atual de A na cadeia entrada ω, registrando essas informações em sua fita. Inicialmente, o estado atual de A é q0, e a posição atual de A na entrada é o símbolo mais à esquerda de ω. Os estados e a posição são atualizados de acordo com a função de transição especificada por δ.

- Quando M termina de processar o último símbolo de ω, ela aceita a entrada se A estiver em um estado de aceitação, caso contrário, M rejeita a entrada se A estiver em um estado de não aceitação.

  - C.q.d.

Decidibilidade – Linguagens Decídiveis

Da mesma maneira, pode-se proceder para autômatos finitos não determinísticos. Assim:

- Teorema 2: A_{AFN} é uma linguagem decidível.

Prova:

- Considere uma máquina de Turing M1 que tem uma capacidade de decidir A_{AFN}. Em vez de projetar M1 para operar como M, ou seja, simulando um autômato finito não determinístico (AFN) em vez de um AFD, façamos uma abordagem sob uma nova perspectiva: M1 utiliza M como uma sub-rotina. Isso ocorre porque M é especificamente projetado para lidar com AFDs. Portanto, o processo de M1 envolve inicialmente a conversão do AFN que recebe como entrada um AFD equivalente antes de aplicar M.

Decidibilidade – Linguagens Decídiveis

O procedimento de M1 pode ser resumido da seguinte forma:

M1 = “Sobre a entrada ⟨A,ω⟩, onde A representa um AFN e ω é uma cadeia:

1. Realize a conversão do AFN A em um AFD equivalente B.
2. Execute a máquina de Turing M (Teorema 1) na entrada ⟨B,ω⟩.
3. Se M aceitar, M1 aceita a entrada; caso contrário, rejeita.”

- C.q.d.

Decidibilidade – Linguagens Decídiveis

- Sabemos que as linguagens regulares podem ser expressas por meio de expressões regulares. Assim, A_{ER}= {⟨R,ω⟩/ R é expressão regular que gera "ω" }.

- Teorema 3: A_{ER} é uma linguagem decidível.

Prova:

- Considere uma máquina de Turing M2 que resolve o problema A_{ER} (Aceitação de Expressão Regular).

Decidibilidade – Linguagens Decídiveis

M2 opera da seguinte forma: ao receber a entrada ⟨R,ω⟩:

1. Inicia convertendo a expressão regular R em um autômato finito não determinístico (AFN) equivalente, chamado de A.
2. Executar a máquina de Turing M1 (Teorema 2) na entrada ⟨A,ω⟩.
3. Caso M1 aceite a entrada, M2 também a aceita; no entanto, se M1 rejeitar a entrada, então M2 a rejeita também.

- C.q.d.

Decidibilidade – Linguagens Decídiveis

- Nos três teoremas vistos, simulamos, em uma MT, se um autômato finito ou uma expressão regular decide uma cadeia específica.

- Também é possível provar se um AFD aceita alguma cadeia ou se dois AFD's são equivalentes.

- Para conhecer outros problemas decidíveis, consulte o livro-texto e a bibliografia recomendada.

Decidibilidade – O Problema da Parada

- Um dos problemas ainda sem solução na matemática é a Conjectura de Goldbach, que estabelece que todo número par maior que 2 pode ser representado pela soma de dois números primos.

- Se executarmos em uma máquina de Turing um algoritmo que verifique a veracidade da conjectura, podem ocorrer duas situações:

1. A máquina parar em estado de rejeição, pois verificou um determinado número par que não pode ser escrito como a soma de dois primos;

2. Executar indefinidamente a verificação, pois o conjunto dos números pares é infinito.

Decidibilidade – O Problema da Parada

- Como se trata de um problema ainda sem solução, significa que não sabemos se a máquina de Turing para ou entra em loop infinito.

- Portanto, resolver a Conjectura de Goldbach, significa resolver o problema da parada. E mais que isso, resolve a questão da decidibilidade de todo e qualquer algoritmo.

Decidibilidade – O Problema da Parada

O problema da parada levanta a seguinte questão: dado um programa e uma entrada, é possível determinar se esse programa eventualmente terminará ou continuará sendo executado indefinidamente?

Alan Turing conseguiu dar uma resposta definitiva da seguinte maneira:

- Ele imaginou o que aconteceria se o problema da parada tivesse uma solução positiva, ou seja, se existisse uma máquina de Turing capaz de verificar em tempo finito se um programa com uma certa entrada iria parar ou não.

Decidibilidade – O Problema da Parada

À essa máquina que resolve o problema da parada deu-se o nome de T e para testar se outra máquina de Turing M, com uma certa entrada, vai parar ou não, toma-se a descrição de M mais sua entrada e a insere-se em T. Assim:

- Se M parar ⇒ T diz que M para.
- Se M não para ⇒ T diz que M não para.

Decidibilidade – O Problema da Parada

Fazendo um modificação em T, ou seja, acoplando outro programa que faz o oposto de T, criando assim, uma máquina maior que chamaremos de T1, que funciona da seguinte maneira:

Toma-se novamente a descrição de M e sua entrada e insere em T1. Assim:

- Se M parar ⇒ T para e T1 não para.
- Se M não para ⇒ T não para e T1 para.

Decidibilidade – O Problema da Parada

O segredo é fazer T1 executar sua própria descrição. Assim, como a descrição de T1 passou primeiro por T, e se T conclui que:

- T1 para ⇒ T1 não para.
- T1 não para ⇒ T1 para.

- Isso quer dizer que em qualquer situação ocorre uma contradição. Logo, T não pode existir, pois o problema da parada tem solução negativa, ou seja, não tem como afirmar se uma máquina de Turing vai parar ou não para uma certa entrada.

Interatividade

Qual das seguintes afirmações sobre a decidibilidade está correta?

a) Todos os problemas são decidíveis, desde que haja recursos computacionais suficientes.
b) Um problema é decidível se existir um algoritmo que sempre produzirá uma resposta correta em tempo finito.
c) Um problema é decidível apenas se for possível resolvê-lo em tempo linear.
d) A decidibilidade de um problema depende apenas de sua complexidade.
e) A decisão se aplica apenas a problemas estritamente matemáticos.

Resposta

Qual das seguintes afirmações sobre a decidibilidade está correta?

a) Todos os problemas são decidíveis, desde que haja recursos computacionais suficientes.
b) Um problema é decidível se existir um algoritmo que sempre produzirá uma resposta correta em tempo finito.
c) Um problema é decidível apenas se for possível resolvê-lo em tempo linear.
d) A decidibilidade de um problema depende apenas de sua complexidade.
e) A decisão se aplica apenas a problemas estritamente matemáticos.

Redutibilidade

- A ideia de redutibilidade na teoria da computação é como o processo de encontrar semelhanças entre problemas.

- A redutibilidade é uma ferramenta poderosa que nos ajuda a comparar problemas e classificá-los em grupos, como problemas simples e difíceis. Também é fundamental para provar resultados importantes e para entender os limites do que pode ou não ser resolvido por um computador.

Redutibilidade

- Sabemos que uma máquina de Turing e algoritmos são sinônimos. Portanto, em nossa disciplina, usaremos a redução como uma técnica de projeto de máquinas de Turing (algoritmo) em que se usa uma máquina feita para uma linguagem B (problema B) para criar uma máquina para a linguagem A (problema A). Isso quer dizer que, conhecendo a solução para a linguagem B e sabendo que esta tem uma certa relação com a linguagem A, recriaremos uma máquina que resolva a linguagem A.

Redutibilidade

- Informalmente, veja como funciona a redução por meio de algoritmos.
- Seja o problema P, denota-se por IP sua instância e por SP sua solução. Devemos reduzir um problema A para um problema B, isto é, usar um algoritmo B para resolver A:
- A instância do problema deve ser interpretada como uma “entrada” e a solução interpretada como “aceita ou rejeita”.

1. $I_B \rightarrow$ [Algoritmo B] $\rightarrow S_B$
2. $I_A \xrightarrow{f} I_B \rightarrow$ [Algoritmo B] $\rightarrow S_B \xrightarrow{g} S_A$
3. $I_A \rightarrow$ [$\xrightarrow{f} I_B \rightarrow$ Algoritmo B $\rightarrow S_B \xrightarrow{g}$] $\rightarrow S_A$

Redutibilidade

- Em “1”, temos um algoritmo B qualquer, que recebe uma instância e devolve uma solução. Assim, devemos pensar em uma forma de usar o algoritmo B para resolver o problema A. Entretanto, sabemos que o algoritmo B só recebe entradas válidas para B.
- Desta forma, em “2”, transformamos uma entrada válida do problema A em uma entrada válida do problema B por meio da relação “f”. Da mesma maneira, independentemente da solução apresentada pelo algoritmo B, devemos transformar, por meio da relação “g”, SB em SA.
- Assim, o conteúdo retangulado em “3” transforma-se em um algoritmo de A.
- Observe no exemplo dado, que a redução serviu para, indiretamente, criar um algoritmo.

1. $I_B \rightarrow$ [Algoritmo B] $\rightarrow S_B$
2. $I_A \xrightarrow{f} I_B \rightarrow$ [Algoritmo B] $\rightarrow S_B \xrightarrow{g} S_A$
3. $I_A \rightarrow$ [$\xrightarrow{f} I_B \rightarrow$ Algoritmo B $\rightarrow S_B \xrightarrow{g}$] $\rightarrow S_A$

Redutibilidade

- Vejamos outra aplicação, ou seja, transformar um problema de ordenação em um problema de seleção.

Problema de seleção

- Recebemos como entrada um vetor ⟨V, n, k⟩ com n elementos, ou seja, V[1...n] e um determinado valor de k entre 1 e n, isto é $1 \leq k \leq n$. Como saída, descobrir o k-ésimo menor elemento armazenado em V.

    - Para entender o que esse problema nos pede, suponha que V = (7,5,9,3,4,1) e k = 3, o problema pede para descobrir o terceiro menor elemento de V. Logo, a resposta é 4.

Redutibilidade

Problema de ordenação

- Recebemos como entrada um vetor ⟨"A,m"⟩ com m elementos, isto é, A[1...m] e como saída, queremos um vetor B, com o mesmo conteúdo de A, de modo que esse conteúdo esteja ordenado em ordem crescente, ou seja, $B[1] \leq B[2] \leq ... \leq B[m]$.

- Portanto, o problema nos pede, por exemplo, que dado o vetor A = (7,5,9,3,4,1) como entrada, obtemos o vetor B = (1,3,4,5,7,9) como saída.

    - Perceba aqui que, conhecendo algum algoritmo de ordenação, o k-ésimo menor elemento vai ocupar exatamente a posição k, resolvendo, assim, o problema da seleção.

Redutibilidade

- Sabemos que há vários algoritmos de ordenação. Tomemos como exemplo o quicksort. Assim:

$$\langle V, n, k\rangle \xrightarrow{f} \langle A, m\rangle \longrightarrow \text{QUICKSORT} \longrightarrow A \xrightarrow{g} g(A)=V[\cdot]$$

- Esses algoritmos recebem apenas um único vetor como entrada e o devolvem rearranjado, como ilustrado no esquema

$$\langle A,m\rangle \longrightarrow \text{QUICKSORT} \longrightarrow A.$$

- Dessa forma, é necessário transformar a entrada, por meio da relação f, no caso:

$$\langle V, n, k\rangle \xrightarrow{f} \langle A, m\rangle.$$

Redutibilidade

$$\langle V, n, k\rangle \xrightarrow{f} \langle A, m\rangle \longrightarrow \text{QUICKSORT} \longrightarrow A \xrightarrow{g} g(A)=V[\cdot]$$

- Note que na entrada, $\langle V, n, k\rangle \xrightarrow{f} \langle A, m\rangle$, o que queremos resolver é o problema de seleção e que nele, há um terceiro valor, k. Na saída, V[·], queremos apenas uma posição específica, que é o k-ésimo menor elemento.

Redutibilidade

Assim, em código, temos:

1. ALG_selecao(V. n, k) {
2. V = ALG ordenacao(V. n);
3. return V[k];
4. 4 }

- Em “1” criamos um algoritmo de seleção que recebe uma entrada válida. Usamos um algoritmo de ordenação em “2”, que ordenará o vetor V e devolverá em “3” a posição k do vetor ordenado.

Redutibilidade

- Visto como funciona a redutibilidade, o que podemos extrair de importante?

- “Redutibilidade desempenha um papel importante na classificação de problemas por decidibilidade e mais tarde em teoria da complexidade também. Quando A é redutível a B, resolver A não pode ser mais difícil que resolver B porque uma solução para B dá uma solução para A. Em termos de teoria da computabilidade, se A é redutível a B e B é decidível, A também é decidível. Equivalentemente, se A é indecidível e redutível a B, B é indecidível.”(Sipser, 2007, p.151).

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

- Já estabelecemos, anteriormente, a indecidibilidade do problema da parada, cuja descrição é PMT = {⟨M,ω⟩/ M é MT que para sobre ω}. Outra descrição do problema da parada constante nas bibliografias é AMT = {⟨M,ω⟩/ M é MT que aceita ω}. Note que essa descrição não é a mais completa, visto que apenas aceitar ω é diferente de parar, aceitar ou rejeitar com ω.
  A pergunta aqui é: como usar uma máquina que decide PMT para decidir AMT? Em outras palavras com reduzir AMT para PMT?

- Agora, vamos abordar um problema relacionado chamado PMT, que envolve a determinação de uma máquina de Turing para aceitar ou rejeitar uma entrada de dados.

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

- Para provar a indecidibilidade de PMT, usamos a indecidibilidade de AMT para reduzir AMT a PMT. Definimos PMT como o conjunto de pares $\langle M,\omega \rangle$, onde M é uma máquina de Turing e M para sobre a entrada ω.

- Teorema: PMT é indecidível.

- Ideia da prova: Esta prova é por contradição. Suponhamos que PMT seja decidível e usemos essa suposição para mostrar que AMT também é decidível, o que contradiz o problema da parada. A ideia central é demonstrar que AMT é redutível a PMT.

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

- Assumimos a existência de uma máquina de Turing R que decide PMT e, com base nisso, construímos S, uma máquina de Turing que decide AMT. A ideia inicial de construir S é tentar simular M sobre ω e decidir com base nisso, mas isso não funciona porque S não pode entrar em loop. Em vez disso, utilizamos a suposição de que R decide PMT. Com R, podemos testar se M para sobre ω. Se R indica que M não para sobre ω, rejeitamos porque ⟨M,ω⟩ não pertence a AMT. Por outro lado, se R indica que M para sobre ω, podemos simular M sem riscos de looping. Isso leva a uma contradição, pois AMT é conhecida por ser indecidível. Portanto, concluímos que R não pode existir, o que implica que PMT é indecidível.

Prova: Supondo que o MT R decida PMT, construímos a MT S para decidir AMT da seguinte forma:

1. Execute uma MT R na entrada ⟨M,ω⟩.
2. Se R rejeitar, rejeite.
3. Se R aceitar, simule M sobre ω até que ela pare.
4. Se M aceitar, aceite; se M rejeitar, rejeite.

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

- Uma função computável pode ser definida em termos de máquinas de Turing. Uma função f é considerada computável se existir uma máquina de Turing que, para cada entrada x, pare após um número finito de passos e produza f(x) como saída.

Matematicamente, podemos expressar essa definição da seguinte forma:

- Seja uma função $f: \Sigma^* \longrightarrow \Sigma^*$ que mapeia cadeias de caracteres (símbolos em um alfabeto Σ). A função f é computável se, e somente se, existir uma máquina de Turing MT tal que a cada entrada $\omega \in \Sigma^*$, a máquina de Turing MT para após um número finito de passos e, ao parar, a fita contém $f(\omega)$.

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

São exemplos de funções computáveis:

- f(ω) = $ω, com ω qualquer: Essa função recebe uma cadeia ω e, ao recebê-la, a transfere uma célula à direita, inserindo, à esquerda de ω, o símbolo de $, a fim de detectar a borda esquerda da fita.

- f(⟨x,y⟩) = ⟨x+y⟩, com x e y inteiros: Essa função recebe dois números inteiros x e y e devolve o resultado da soma destes.

- f(⟨M,ω⟩) = ⟨M',ω⟩, com M uma máquina de Turing e ω uma cadeia: Essa função recebe uma máquina de Turing M e uma cadeia ω. Devolve uma máquina de Turing M que é igual a M acrescida de estados iniciais para rejeitar qualquer cadeia diferente de "ω".

Redutibilidade – Problemas Indecidíveis da Teoria das Linguagens

- Agora que sabemos o que é uma função computável, definiremos formalmente o que é redução, também denominada redução por mapeamento.
- A linguagem A é redutível à linguagem B se existe uma função computável f: $\Sigma^* \longrightarrow \Sigma^*$ que recebe uma cadeia qualquer ω pertencente a A e devolve outra cadeia f(ω) pertencente a B.

Essa definição pode ser explicada da seguinte forma:

- Seja A um problema de decisão e B outro problema de decisão. A redução por mapeamento de A para B é uma função computável f que mapeia instâncias de A para instâncias de B de forma que:

1. Se x é uma instância de A para a qual a resposta é "sim", ou seja, $x \in A$, então f(x) é uma instância de B para a qual a resposta também é "sim", ou seja, $f(x) \in B$.
2. Se x é uma instância de A para a qual a resposta é "não", ou seja, $x \notin A$, então f(x) é uma instância de B para a qual a resposta também é "não", ou seja, $f(x) \notin B$.

Interatividade

O conceito de redutibilidade na teoria da computação refere-se à:

a) Habilidade de reduzir a complexidade de um problema de NP para um problema de P.
b) Capacidade de tornar um problema indecidível em um problema decidível.
c) Transformação de um problema em outro de tal forma que a solução do segundo problema possa ser usada para resolver o primeiro.
d) Técnica de otimização de algoritmos para torná-los mais eficientes.
e) Ideia de transformar um problema em outro problema de maior complexidade.

Resposta

O conceito de redutibilidade na teoria da computação refere-se à:

a) Habilidade de reduzir a complexidade de um problema de NP para um problema de P.
b) Capacidade de tornar um problema indecidível em um problema decidível.
c) Transformação de um problema em outro de tal forma que a solução do segundo problema possa ser usada para resolver o primeiro.
d) Técnica de otimização de algoritmos para torná-los mais eficientes.
e) Ideia de transformar um problema em outro problema de maior complexidade.

Complexidade

Complexidade na teoria da computação envolve a análise do desempenho dos algoritmos e o estudo de quão difícil é resolver problemas computacionais. Isso inclui:

- Comportamento Assintótico de Funções;
- Problemas em Classe P;
- Problemas em Classe NP;
- Problemas NP-Completos.

Complexidade – Comportamento Assintótico de Funções

- Os limites da computação não se resumem apenas à decidibilidade, pois dentro dos problemas decidíveis existem os tratáveis e os não tratáveis, que requerem recursos como tempo e memória. Para avaliar a eficiência dos algoritmos, conceitos de Análise de Algoritmos são aplicados, permitindo verificar a correção, prever o desempenho e comparar soluções sem implementá-las. Múltiplos fatores afetam o tempo de execução, como máquinas de Turing com várias fitas, tamanho de entrada, representação de dados e algoritmo. O tempo de execução é definido de acordo com o tamanho da entrada, e a contagem dos passos executados no pior dos casos é usada para avaliar a eficiência.

Complexidade – Comportamento Assintótico de Funções

- Vejamos por meio de um exemplo o que significa o pior dos casos.

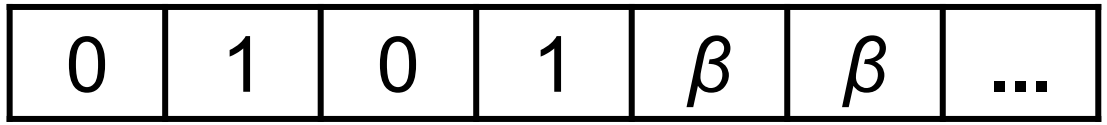
- Considere a linguagem L = $\{0^n1^n / n \geq 0\}$ e uma máquina de Turing M decisora. Dadas as cadeias de entrada ω = 0101 e ω1 = 0011, determine o tempo de execução.

Complexidade – Comportamento Assintótico de Funções

- Primeiramente determinamos como M decidirá as cadeias de entrada.
- Inspecione a fita, a partir da célula mais à esquerda e rejeite a cadeia de entrada se encontrar um símbolo 0 à direita de um símbolo 1.
- Continue a inspeção enquanto houver símbolos 0s e 1s na fita e marque-os um a um alternadamente.
    - Se encontrar símbolos 0s depois de marcados todos os 1s ou se encontrar símbolos 1s depois de marcados todos os 0s, rejeite a cadeia de entrada. Caso contrário, aceite-a.

Complexidade – Comportamento Assintótico de Funções

- Para a cadeia ω é fácil perceber que o cabeçote de leitura e gravação se moverá, apenas, até a terceira célula, momento em que encontrará um símbolo 0 à direita do símbolo 1. Nesse caso específico, o tempo de execução é 3.



Complexidade – Comportamento Assintótico de Funções

- Agora o processamento da cadeia ω1 = 0011

0	0	1	1	β	β	...

- Nesse caso, haverá 16 movimentos do cabeçote de leitura e gravação, pois como a cadeia ω1 pertence à linguagem L, o cabeçote terá que ler toda a cadeia e, só após leitura, parar. Portanto, esse é o pior dos casos. Logo, o tempo de execução é 16.

Complexidade – Comportamento Assintótico de Funções

Assim, definimos tempo de execução como:

- Para uma máquina de Turing determinística M que opera sobre uma entrada n, o tempo de execução é a função f : N ⟶ N, onde f(n) é definido como o número total de etapas (ou transições) que a máquina M leva para processar a entrada de comprimento n até que ela alcance um estado de aceitação ou rejeição.

    - Já para uma máquina de Turing não determinística N que opera sobre uma entrada de comprimento n, o tempo de execução não determinístico é função f:N⟶N, onde f(n) é definido como o número máximo de etapas (ou transições) entre todas as possíveis ramificações da computação de N sobre n. Isso inclui todas as “linhas do tempo” possíveis resultantes das escolhas não determinísticas feitas por N.

Complexidade – Comportamento Assintótico de Funções

O comportamento assintótico de funções é fundamental na análise da complexidade de algoritmos e na teoria da computação. Ele se concentra em como as funções se comportam à medida que seus argumentos se aproximam de valores extremamente grandes. Isso é crucial para medir o desempenho de algoritmos em termos de tempo de execução e espaço. Alguns conceitos importantes incluem:

- Notação assintótica: Abstração matemática que ajuda a analisar o tempo de execução. Para grandes valores de n, o termo dominante determina o crescimento da função, permitindo que outros termos se tornem irrelevantes. Representado como “ordem de n”.
    - Exemplo: $5n^4 + 3n^2 – 10n + 100$, em notação assintótica, é uma expressão da ordem de n^4.

Complexidade – Comportamento Assintótico de Funções

- Notação "O-grande" (Big O): A notação Big O é usada para descrever o comportamento assintótico superior de uma função em relação a outra.

- Ela é escrita como f(n) = O(g(n)) e significa que a função f(n) cresce não tão rápido quanto a função g(n) para grandes valores de n.

- Limites Assintóticos: Quando estudamos o comportamento assintótico, estamos especificamente nos limites quando n se aproxima do infinito.

    - Por exemplo, quando dizemos que f(n) = O(g(n)), estamos afirmando que existe um valor n0 após o qual $f(n) \leq c \cdot g(n)$, em que c é uma constante inteira e positiva, para todo $n \geq n0$.

Complexidade – Comportamento Assintótico de Funções

- Considerando a função $f(n) = 4n^3 + 5n^2 – 3n + 15$

- Sabemos que a parte relevante é o termo n^3. Portanto, dizemos que f(n) é da ordem n^3 ou que $f(n) = O(n^3)$. Sabemos também que $f(n) \leq c \cdot g(n)$.

Assim, podemos atribuir um limitante superior qualquer à f(n), por exemplo:

$$4n^3 + 5n^2 – 3n + 15 \leq 4n^3 + 5n^2 + 15 \leq \underbrace{4n^3+5n^3+15n^3}_{24n^3}$$

Se $f(n) \leq c \cdot g(n)$, então $4n^3 + 5n^2 – 3n + 15 \leq 24n^3$

Complexidade – Comportamento Assintótico de Funções

- Exemplo: $f(n) = O(n^2) + O(\log n)$

Resolução:

A expressão $O(n^2) + O(\log n)$ representa outras duas funções h(n) e j(n). Assim, pela definição temos que:

- $h(n) = O(n^2)$ e que $h(n) \leq c \cdot n^2$
- $j(n) = O(\log n)$ e que $j(n) \leq c_1 \cdot \log n$

Com base nisso, pode-se estabelecer que:

1. $f(n) \leq c \cdot n^2 + c_1 \cdot \log n$
2. $c \cdot n^2 + c_1 \cdot \log n \leq c \cdot n^2 + c_1 \cdot n^2$
   - Então $f(n) = (c + c_1)n^2$
   - Portanto $f(n) = O(n^2)$

Complexidade – Comportamento Assintótico de Funções

Alguns tipos comuns de funções assintóticas incluem:

- A complexidade O(1) (constante) é aquela em que não há crescimento do número de operações, pois não depende do volume de dados de entrada (n).
- Exemplo: o acesso direto a um elemento de uma matriz.
- A complexidade O(log n) (logaritmo) é aquela em que o crescimento do número de operações é menor do que o do número de itens.
    - Exemplo: caso médio do algoritmo de busca em árvores binárias ordenadas.

Complexidade – Comportamento Assintótico de Funções

- A complexidade O(n) (linear) é aquela em que o crescimento no número de operações é diretamente proporcional ao crescimento do número de itens.
- Exemplo: o algoritmo de busca em uma lista/vetor.
- A complexidade O(n log n) (linearítmica ou quasilinear) é aquela em que é resultado das operações (log n) executada n vezes.
- Exemplo: o caso médio do algoritmo de ordenação Quicksort.

Complexidade – Comportamento Assintótico de Funções

- A complexidade $O(n^2)$ (quadrático) é aquela que ocorre quando os itens de dados são processados aos pares, muitas vezes, com repetições dentro da outra. Com dados suficientemente grandes, tendem a se tornar muito ruins.

- Exemplo: o processamento de itens de uma matriz bidimensional.

- A complexidade $O(2^n)$ (exponencial) é aquela em que à medida que n aumenta, o fator analisado (tempo ou espaço) aumenta exponencialmente. Não é executável para valores muito grandes e não são úteis do ponto de vista prático.

    - Exemplo: busca em uma árvore binária não ordenada.

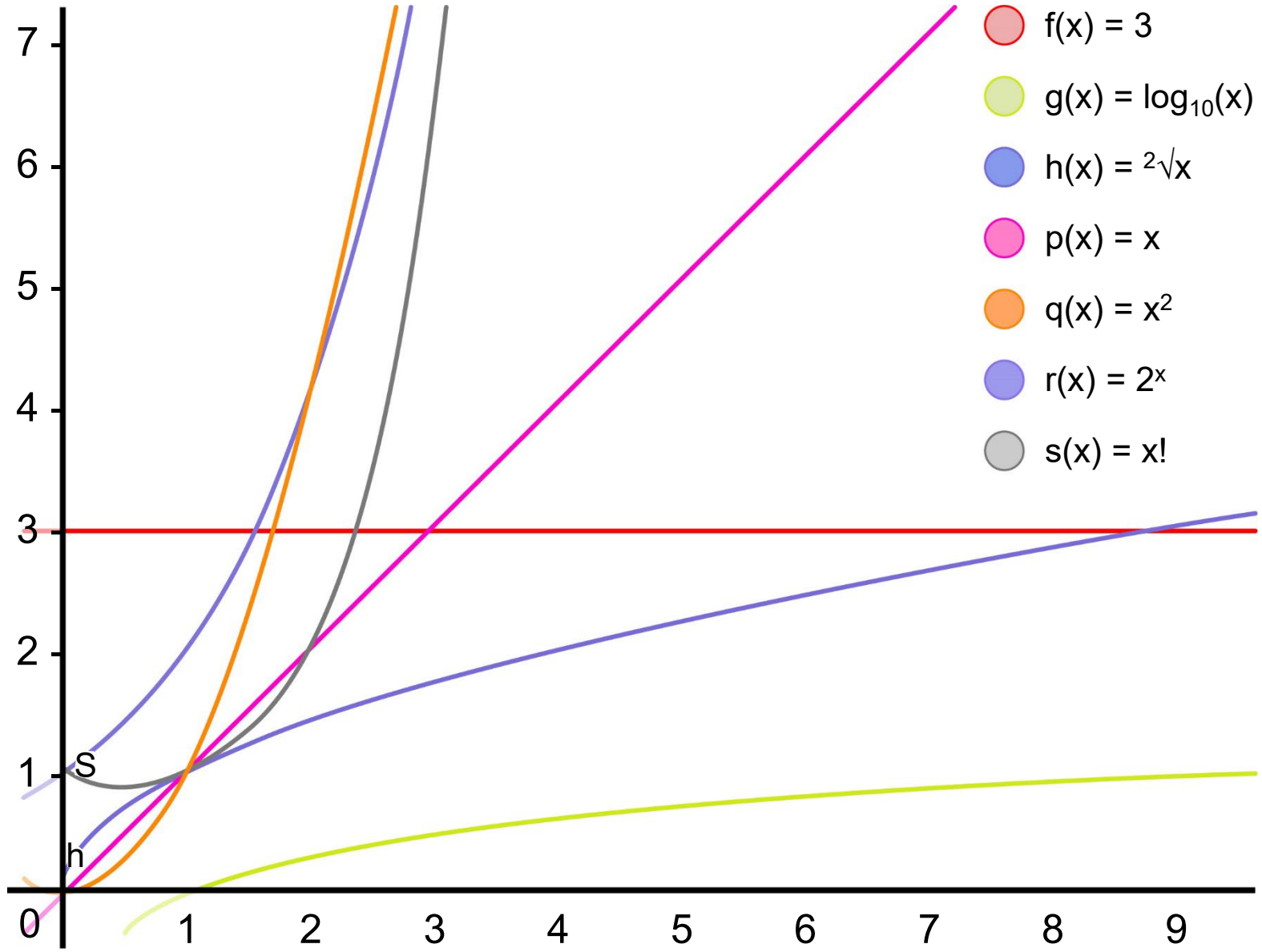
Complexidade – Comportamento Assintótico de Funções

- A complexidade O(n!) (fatorial) é aquela em que o número de instruções executadas cresce muito rapidamente para um pequeno número de dados.

- Exemplo: um algoritmo que gere todas as possíveis permutações de uma lista.

Complexidade – Comportamento Assintótico de Funções

Graficamente:



Fonte: autoria própria.

Interatividade

Qual é o objetivo da notação “Big O” (O-grande) no contexto do comportamento assintótico de funções?

a) Descrever o comportamento exato de uma função para todos os tamanhos de entrada.
b) Provar que uma função é irredutivelmente complexa.
c) Determinar a complexidade exata do tempo de execução de um algoritmo.
d) Fornecer uma estimativa superior do crescimento de uma função à medida que o tamanho da entrada cresce.
e) Descrever a complexidade espacial de um algoritmo.

Resposta

Qual é o objetivo da notação “Big O” (O-grande) no contexto do comportamento assintótico de funções?

a) Descrever o comportamento exato de uma função para todos os tamanhos de entrada.
b) Provar que uma função é irredutivelmente complexa.
c) Determinar a complexidade exata do tempo de execução de um algoritmo.
d) Fornecer uma estimativa superior do crescimento de uma função à medida que o tamanho da entrada cresce.
e) Descrever a complexidade espacial de um algoritmo.

Complexidade – Problemas Classe P e Classe NP

- O tema que agora vamos estudar é, com certeza, o maior problema em aberto na Ciência da Computação e um dos maiores da Matemática. Esse é o problema conhecido como P versus NP e sua resolução trará impactos profundos na sociedade.

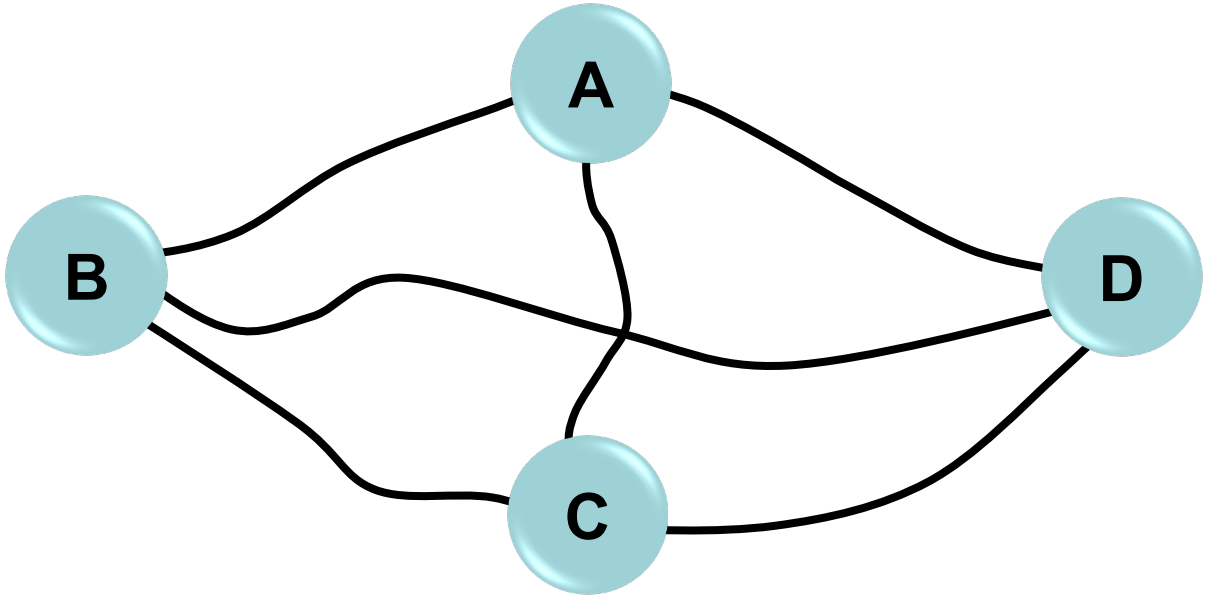
- Resumidamente P versus NP estabelece a seguinte questão: se é fácil verificar que uma solução para um problema é correta, então também será fácil resolver o problema?

Complexidade – Problemas Classe P e Classe NP

- Aparentemente essa questão não parece tão difícil de ser respondida, mas durante esta aula será mostrado o que realmente essa questão significa e sua importância para a Ciência e a sociedade.

- O interesse da Ciência, de modo geral, é provar se algo pode ou não ser feito. Por exemplo, já sabemos que existe, por meio da teoria dos jogos, uma estratégia perfeita para o jogo de xadrez, só que ela não é conhecida por ninguém, nem tão pouco se o jogo será vencido sempre pelas peças brancas ou pelas peças pretas. Também não se conhece se o resultado do jogo será empate se essa estratégia for usada por dois jogadores que a dominam.

Complexidade – Problemas Classe P e Classe NP



- Portanto, vemos que saber se um problema tem solução, não é tão útil como saber resolvê-lo.

- Vejamos uma adaptação do problema do caixeiro-viajante.

Suponha que um vendedor more na cidade A e precise visitar, de carro, outras três cidades, B, C, D e depois retornar para casa. Ele quer economizar tempo e gasolina. Dessa forma, qual o menor percurso de forma que ele não deixe de visitar nenhuma cidade?

Complexidade – Problemas Classe P e Classe NP

- A seguir, a tabela mostra a distância entre as cidades.

De/Para	A	B	C	D
A	0	54	17	79
B	54	0	49	104
C	17	49	0	91
D	79	109	91	0

- Perceba que há diferença na viajem de B até D (104) e de D até B (109). Suponha que essa diferença se dá pelo sistema de mão única da cidade.
- Para resolver esse problema, sem pensar muito, listamos todas as alternativas possíveis.
- Se há três cidades para visitar, significa que há seis possíveis rotas (3 fatorial).

Complexidade – Problemas Classe P e Classe NP

- Agora basta somar as distâncias em cada rota para descobrir que a menor é A-D-B-C-A com 254.

Rota	Distância
A-B-C-D-A	54+49+91+79=273
A-C-B-D-A	17+49+104+79=249
A-C-D-B-A	17+91+109+54=271
A-D-B-C-A	79+109+49+17=254
A-D-C-B-A	79+91+49+54=273
A-B-D-C-A	54+104+91+17=266

Complexidade – Problemas Classe P e Classe NP

- Mas e se aumentarmos a quantidade de cidades para dez?

- Listar as rotas manualmente não seria mais viável, pois existem 3.628.800 (10 fatorial) rotas possíveis, mas um computador ainda conseguiria executar essa tarefa com certa facilidade. Entretanto, se aumentarmos para apenas trinta cidades (30 fatorial de rotas possível), o melhor computador demoraria mais que a idade do universo para completar a tarefa.

- Veja que este problema é importantíssimo para a indústria e comércio. Imagine uma empresa de logística que precise planejar rotas para algumas dezenas de lugares.

Complexidade – Problemas Classe P e Classe NP

- Hoje já existem algoritmos melhores do que listar rotas e somar suas distâncias, conhecidos como algoritmos de força bruta. Mas mesmo estes se tornam inviáveis para algumas poucas dezenas de lugares.
- Assim, devemos saber o que é um algoritmo viável. A viabilidade de um algoritmo está intimamente ligada ao tempo de execução, ou seja, depende de sua complexidade.
- No exemplo dado, na aula anterior, da linguagem $L = \{0^n1^n / n \geq 0\}$, vimos que ele tem tempo de execução 16 para uma entrada de tamanho 4($\omega = 0011$). Isso significa que sua complexidade é $O(n^2)$.
    - Há problemas com complexidade menor ainda, como é o caso do algoritmo de soma de números de três dígitos. Nesse caso, na pior das hipóteses, sua complexidade é O(2n), ou seja, é linear.
    - Nos dois casos, o tempo de execução é polinomial. A palavra polinomial se inicia com a letra P, que é o P dos problemas P versus NP.

Complexidade – Problemas Classe P e Classe NP

- Essa classe consiste em problemas de decisão que podem ser resolvidos em tempo polinomial por uma máquina de Turing determinística, ou seja, por qualquer algoritmo de complexidade O que não seja O(2n) ou O(n!), ou ainda, segundo a definição, não maior que $c \cdot n^k$ passos, em que c é uma constante e n é o tamanho da entrada.

- Todavia, esses problemas de complexidade O(2n) ou O(n!) existem! Esses algoritmos são denominados de força bruta e O(n!) é a complexidade do problema do caixeiro-viajante, tornando o problema inviável para um número moderado de cidades.

    - Concluindo, sabemos que o problema do caixeiro-viajante tem solução, todavia, descobri-la é inviável. A pergunta que fica é como resolver esse tipo de problema?

Complexidade – Problemas Classe P e Classe NP

Problemas de otimização x Problemas de decisão

Problema de otimização:

- É um problema em que cada solução possível tem um valor associado e desejamos encontrar a melhor solução em relação a esse valor.

- Exemplo: Encontrar a melhor rota (mais curta) no caso do exemplo dado do vendedor (adaptação do problema do caixeiro-viajante).

Complexidade – Problemas Classe P e Classe NP

Problema de decisão:

- Problemas cuja resposta é simplesmente sim ou não.

Existe solução para um dado problema?

- Exemplo: Existe uma rota menor que 260 para o problema do vendedor.

- Note que o problema do vendedor não mudou de complexidade, entretanto, verificar a solução ficou bem mais fácil. Ora, saber a menor distância é inviável, mas saber se determinada rota não ultrapassa a distância de 260 é bem mais fácil, bastando somar as distâncias entre os lugares. A vantagem aqui é que agora o problema pode ser resolvido em tempo polinomial. Problemas desse tipo são da classe NP (não determinísticos polinomiais).

Complexidade – Problemas Classe P e Classe NP

- Assim os Problemas em NP são aqueles para os quais, se você tiver uma suposta solução, poderá verificar rapidamente se a solução está correta em tempo polinomial. No entanto, encontrar uma solução em si pode ser muito difícil ou demorado.

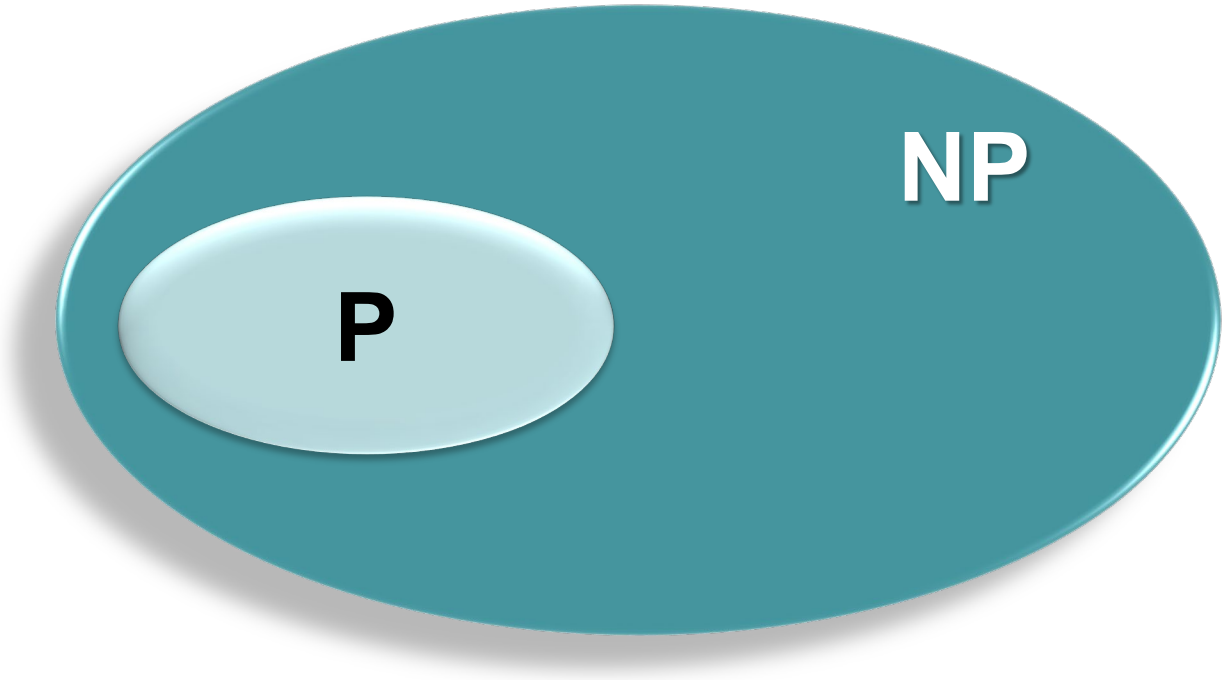
- O termo “não determinístico” refere-se ao fato de que, em uma máquina de Turing não determinística, você pode adivinhar a solução correta em tempo polinomial e, em seguida, verificar se está correta.

- Os problemas NP incluem problemas de otimização, como o problema do caixeiro-viajante.

Complexidade – Problemas Classe P e Classe NP

- Fica evidente que a classe P é subconjunto da classe NP, pois qualquer problema que pode ser resolvido em tempo polinomial pode, também, ser verificado em tempo polinomial.

- Como dito anteriormente, para a classe NP, encontrar uma solução em tempo polinomial pode ser muito difícil ou demorado, tanto é que ainda não se conhece tal solução para o problema do caixeiro-viajante, embora isso não queira dizer que ela não exista.



Complexidade – Problemas Classe P e Classe NP

Sendo assim, a grande questão ainda aberta na Ciência da Computação é: P = NP? Ou seja, se é fácil verificar que uma solução para um problema (execução em tempo polinomial) é correta, então também será fácil resolver o problema (também em tempo polinomial)?

- No estudo do próximo tópico, explicaremos porque essa questão é tão importante.

Complexidade – Problemas NP-Completos e Classe NP-Difícil

- O cientista que levantou a questão se P = NP foi Stephen Cook, da Universidade de Toronto, na década de 1970. No mesmo trabalho, apresentou o conceito dos problemas NP-Completos que, grosso modo, são problemas NP tais que qualquer outro problema NP pode ser reduzido a eles.

Complexidade – Problemas NP-Completos e Classe NP-Difícil

A ideia-chave que Cook apresentou é a seguinte:

- Ele definiu um problema específico chamado de “problema SAT” (Problema de Satisfatibilidade) que consiste em um circuito lógico (um conjunto de portas lógicas conectadas por fios) e uma expressão lógica na forma de uma fórmula booleana (conjunto de cláusulas). A pergunta é: Existe uma atribuição de valores verdadeiros ou falsos às variáveis de entrada do circuito, de modo que a expressão lógica seja avaliada como verdadeira?

    - Cook declarou que o problema SAT é NP-Completo. Isso significa que o problema SAT é, pelo menos, tão difícil quanto qualquer outro problema em NP e qualquer problema em NP pode ser limitado a ele de forma eficiente.

Complexidade – Problemas NP-Completos e Classe NP-Difícil

- Cook usou uma técnica chamada "redução polinomial" para mostrar que, se você pudesse resolver o problema SAT de maneira eficiente (em tempo polinomial), então você poderia resolver qualquer problema em NP de maneira eficiente também.

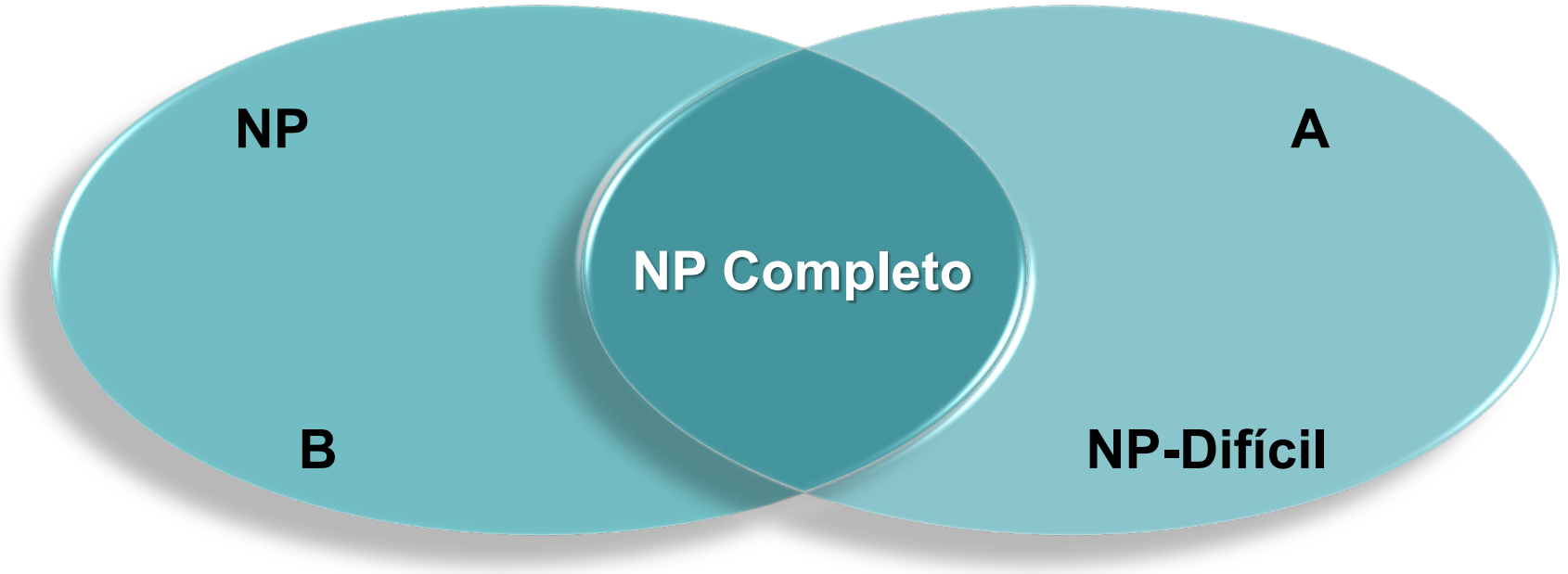
- Mas foi Richard Karp que apresentou novos problemas NP-Completos com interesses muito mais práticos. O do caixeiro-viajante, por exemplo.

Complexidade – Problemas NP-Completos e Classe NP-Difícil

- Como estudado anteriormente, reduzir um problema A em um problema B significa uma forma de interpretar o problema A dentro do problema B. Então, se sabemos resolver B, também sabemos resolver A. Por exemplo, achar um caminho que passe uma única vez por todos os vértices de um grafo e termine no ponto de partida. Esse é um problema NP que pode ser reduzido ao problema do caixeiro-viajante (NP-Completo). Assim, com N pontos, atribui-se distância 1 a cada uma das arestas e pergunta-se se existe uma rota fechada com tamanho de no máximo N.

Complexidade – Problemas NP-Completos e Classe NP-Difícil

Também existe outra classe de problemas denominadas NP-Difíceis, a diferença é que a essa classe não é exigido que esteja dentro da classe NP. Assim:



- Se A é um problema NP-Difícil, então todo problema B em NP pode ser reduzido a A, desde que A esteja em NP-Completo. Assim, a classe dos NP-Completos é chave para responder o problema de responder se P=NP.

Interatividade

Qual é a principal diferença entre problemas na classe P e problemas na classe NP?

a) Problemas em P são solucionáveis em tempo polinomial, enquanto problemas em NP não têm solução.
b) Problemas em P são mais difíceis de resolver do que problemas em NP.
c) Problemas em P têm complexidade exponencial, enquanto problemas em NP têm complexidade polinomial.
d) Problemas em P podem ser resolvidos em tempo polinomial, enquanto problemas em NP podem ser verificados em tempo polinomial.
e) Não há diferença significativa entre problemas em P e NP.

Resposta

Qual é a principal diferença entre problemas na classe P e problemas na classe NP?

a) Problemas em P são solucionáveis em tempo polinomial, enquanto problemas em NP não têm solução.
b) Problemas em P são mais difíceis de resolver do que problemas em NP.
c) Problemas em P têm complexidade exponencial, enquanto problemas em NP têm complexidade polinomial.
d) Problemas em P podem ser resolvidos em tempo polinomial, enquanto problemas em NP podem ser verificados em tempo polinomial.
e) Não há diferença significativa entre problemas em P e NP.

Referências

- SIPSER, Michael. *Introdução à teoria da computação*. São Paulo: Thomson Pioneira, 2007.

ATÉ A PRÓXIMA!