Unidade II

5 DECIDIBILIDADE

Na teoria da computação, a decidibilidade é um conceito fundamental que se refere à capacidade de resolver ou decidir um problema de forma algorítmica. Um problema é considerado decidível se existir um algoritmo que, para qualquer entrada possível, sempre produz uma resposta correta em um tempo finito.

A decisão está relacionada à capacidade de um algoritmo ou MT de tomar uma decisão definitiva em relação a uma pergunta ou um problema. Acentuaremos alguns pontos-chave sobre a decidibilidade:

- **Problemas decidíveis**: aqueles que podem ser resolvidos por um algoritmo em tempo finito. Assim, um algoritmo sempre terminará e produzirá uma resposta, seja ela "sim", seja ela "não".

- **Problemas indecidíveis**: existem problemas na teoria da computação que não podem ser decididos por nenhum algoritmo. Isso significa que não existe um procedimento geral que possa determinar uma resposta correta para todas as instâncias do problema.

- **Problema da parada**: pergunta se um programa (MT) com uma entrada específica eventualmente para ou entra em um loop infinito. Alan Turing declarou que o problema da parada é indecidível. Estudaremos melhor esse assunto adiante.

- **Lema do bombeamento**: para linguagens regulares e linguagens livres de contexto, ele é usado para mostrar a indecidibilidade de certas propriedades. É uma ferramenta importante para estabelecer limites sobre a decidibilidade de problemas relacionados à linguagem formal.

Em suma, a decidibilidade, na teoria da computação, é uma propriedade fundamental que separa problemas que podem ser resolvidos algoritmicamente daqueles que não podem. É uma área de estudo importante para entender os limites da computação.

5.1 Linguagens decidíveis

Estudaremos alguns exemplos de linguagens cuja solução pode ser determinada através do uso de algoritmos, ou seja, linguagens que são decidíveis. Como ilustração, pode-se apresentar um algoritmo que verifica se uma sequência de caracteres pertence a uma linguagem regular ou a uma linguagem livre de contexto. Essa questão está relacionada ao reconhecimento e à compilação de programas em uma linguagem de programação. A demonstração de algoritmos resolvendo diversos problemas relacionados

a autômatos é esclarecedora, pois adiante você se deparará com outras questões relacionadas a autômatos que não podem ser resolvidas por meio de algoritmos.

Iniciamos abordando questões computacionais relacionadas a autômatos finitos, e fornecemos algoritmos para verificar se um autômato finito aceita uma cadeia, se a linguagem associada a um autômato é vazia e se dois autômatos finitos são equivalentes. Para simplificar, utilizamos linguagens para representar diversos problemas computacionais. Por exemplo, o problema de verificar se um autômato finito específico aceita uma determinada cadeia pode ser formulado como uma linguagem chamada A_{AFD}, que inclui as representações de todos os autômatos finitos e com as cadeias que esses autômatos aceitam.

Considere que a linguagem $A_{AFD} = \{\langle A, \omega\rangle / A$ é um AFD que aceita $\omega\}$, ou seja, a linguagem que contém o par autômato e cadeia, denotado por $\langle A, \omega\rangle$, sendo que o autômato aceita a cadeia.

Assim, temos a relação entre duas questões: a primeira é saber se um AFD denominado A aceita uma entrada ω, e a segunda é determinar se o par $\langle A, \omega\rangle$ faz parte da linguagem A_{AFD}. Provar a decidibilidade de uma linguagem equivale a mostrar que o problema computacional associado é solucionável.

A seguir, por meio do teorema e prova, veremos que A_{AFD} é uma linguagem cuja solucionabilidade está estabelecida. Portanto, esse teorema confirma que a tarefa de verificar se um autômato finito específico aceita uma cadeia dada é, de fato, uma tarefa solucionável.

Teorema 1: a linguagem A_{AFD} é decidível.

Para realizarmos a prova, basta uma MT que decida $A_{AFD.}$ Assim:

M = sobre a entrada $\langle A, \omega\rangle$, onde A é um AFD e ω é uma cadeia:

1. Simule A sobre a entrada ω.

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

Prova

Consideramos a entrada $\langle A, \omega\rangle$, que consiste na representação de um AFD denominado A e de uma cadeia ω. Para representar A de forma adequada, listam-se seus cinco componentes essenciais: conjunto de estados Q, alfabeto Σ, função de transição δ, estado inicial q_0 e conjunto de estados finais F.

Quando uma MT M recebe essa entrada, sua primeira verificação é garantir que ela representa corretamente um AFD A com uma cadeia w. Caso contrário, M rejeita a entrada. 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 entrada w, registrando essas informações em sua fita. Inicialmente, o estado atual de A é q_0, 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 w, 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.

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

$A_{AFN} = \{\langle A, \omega\rangle / A \text{ é um AFN que aceita } \omega\}$

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

Prova

Considere uma MT M_1 que tem uma capacidade de decidir A_{AFN}. Em vez de projetar M_1 para operar como M, que simula um 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 M_1 envolve inicialmente a conversão do AFN que recebe como entrada em um AFD equivalente antes de aplicar M.

O procedimento de M1 pode ser resumido da seguinte forma:

M_1 = Sobre a entrada $\langle A, \omega\rangle$, 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 M na entrada $\langle B, \omega\rangle$.

3. Se M aceitar, aceite a entrada; caso contrário, rejeite-a.

Sabemos que as linguagens regulares podem ser expressas através de expressões regulares. Assim, $A_{ER} = \{\langle R, \omega\rangle / R \text{ é ER que gera } \omega\}$.

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

Prova

Apresentamos agora uma MT M_2 que resolve o problema A_{ER} (aceitação ER). M_2 opera da seguinte forma: ao receber a entrada (R, ω), onde R representa uma ER e ω é uma cadeia:

1. Inicie convertendo a expressão R em um AFN equivalente, chamado de A.

2. Execute a máquina M_1 na entrada $\langle A, \omega\rangle$.

3. Caso M_1 aceitar a entrada, também a aceite; no entanto, se M_1 rejeitar a entrada, rejeite-a também.

Nos três teoremas vistos, simulamos, em uma MT, se um autômato finito ou uma ER decide uma cadeia específica. Façamos agora a prova se um autômato finito aceita alguma cadeia. Para responder a esse problema, testamos a vacuidade de uma linguagem para um AFD. Portanto, nosso problema é o seguinte:

$V_{AFD} = \{\langle A \rangle \mid A \text{ é um AFD e } L(A) = \emptyset\}$

Teorema 4: V_{AFD} é uma linguagem decidível.

Prova

Um AFD aceita uma cadeia **somente** se ele alcançar um estado final a partir de seu estado inicial seguindo as transições definidas no AFD. Para verificar essa condição, podemos criar uma MT chamada T que utiliza um algoritmo de marcação.

A MT T funciona da seguinte maneira, considerando a entrada (A), onde A é um AFD:

1. Inicialmente, marque o estado inicial de A.

2. Marque o estado que possua uma transição chegando até ele a partir do estado inicial.

3. Seguindo a função de transição, repita o processo de marcação de estados até que não seja possível marcar mais nenhum estado.

4. Após a conclusão do processo de marcação, se nenhum estado de final estiver marcado, aceite a entrada; caso contrário, rejeite-a.

O próximo problema a ser solucionado é saber se dois AFDs são equivalentes, ou seja, se reconhecem a mesma linguagem. Assim:

$E_{AFD} = \{\langle A,B \rangle \mid A \text{ e } B \text{ são AFD's e } L(A) = L(B)\}$

Teorema 5: a linguagem E_{AFD} é decidível.

Prova

Podemos estabelecer a decidibilidade do E_{AFD} através da construção de um novo AFD chamado C a partir de A e B, conforme o teorema 4. O AFD C foi projetado para aceitar apenas aquelas cadeias que são reconhecidas por A ou B, mas não por ambos. Portanto, se A e B aceitarem a mesma linguagem, C não aceitará nada.

A linguagem de C pode ser representada como:

$L(C) = (L(A) \cap \neg L(B)) \cup (\neg L(A) \cap L(B))$

Ela é denominada **diferença simétrica** entre L(A) e L(B), onde ¬L(A) representa o complemento de L(A) e ¬L(B) é o complemento de L(B).

A diferença simétrica é uma ferramenta útil aqui, pois L(C) será vazia, mas somente se L(A) for igual a L(B). Além disso, podemos construir o AFD C a partir de A e B utilizando as construções que provam que a classe das linguagens regulares é fechada sob complementação, união e interseção. Essas construções são algoritmos que podem ser executados por MT.

Após a construção de C, podemos testar se a linguagem L(C) é vazia com o teorema 4. Se for vazia, isso implica que L(A) e L(B) são iguais.

Portanto, apresente uma MT F que funcione da seguinte maneira, considerando a entrada ⟨A, B⟩, onde A e B são AFDs:

1. Construa o AFD C conforme descrito anteriormente.

2. Execute um MT T na entrada (C).

3. Se T aceitar, aceite a entrada; caso contrário, rejeite-a.

Observação

Uma linguagem só é decidível se ela for aceita ou rejeitada, a máquina que a lê jamais entrará em loop.

5.2 Problema da parada

É um dos problemas fundamentais e mais conhecidos na teoria da computação. Sua história remonta ao trabalho de Alan Turing, que é frequentemente considerado o pai da ciência da computação. Em 1936, ele apresentou sua MT, uma abstração matemática de um modelo computacional universal. O problema da parada surgiu naturalmente como uma questão crucial em relação ao funcionamento dessas máquinas.

O problema da parada desempenha um papel central na teoria da computação, pois aborda a questão de decidir se um programa de computador irá parar (terminar sua execução) ou entrar em um loop infinito quando aplicado a uma entrada específica. É muito relevante para a computabilidade e para a demonstração de limites na resolução de problemas algorítmicos. A prova da indecidibilidade do problema da parada teve implicações profundas, revelando que existem limites teóricos para a resolução automática de certos problemas.

Ele é formulado da seguinte maneira: considerando um programa de computador (representado por uma descrição matemática) e uma entrada para esse programa, é possível determinar se o programa irá parar (terminar a execução) ou se continuará indefinidamente em um loop infinito?

Teorema 6: o problema da parada é indecidível.

Prova

Suponha, por contradição, que existe um algoritmo H (uma MT hipotética) que resolva o problema da parada. Isso significa que, considerando uma descrição de um programa P e uma entrada I, H pode determinar se P irá parar quando aplicado a I.

Agora, consideramos a seguinte situação:

Construímos uma MT especial Q e esta recebe uma descrição de um programa P como entrada.

Quando Q é executado com P como entrada, ela faz o seguinte:

1. Se H determina que P para quando aplicado a si mesmo como entrada (P, P), então Q entra em um loop infinito.

2. Se H determina que P entra em loop infinito quando aplicado a si mesmo (P, P), então Q para sua execução.

Agora, vejamos o que acontece quando executamos Q com sua própria descrição como entrada:

- Se H determina que Q para quando aplicado a si mesmo (Q, Q), então, pela definição de Q, esta deveria entrar em loop infinito. No entanto, isso contradiz a decisão de H.

- Se H determina que Q entra em loop infinito quando aplicado a si mesmo (Q, Q), pela definição de Q, esta deveria parar sua execução. Isso também contradiz a decisão de H.

Portanto, vemos uma contradição em ambos os casos. Isso implica que H não pode existir, o que significa que o problema da parada não é decidível, provando assim o teorema da indecidibilidade do problema da parada. Isso é uma demonstração vital dos limites da computação, acentuando que não existe um algoritmo geral que possa determinar se qualquer programa irá parar ou entrar em loop infinito.

6 REDUTIBILIDADE

A ideia de redutibilidade na teoria da computação é como o processo de encontrar semelhanças entre problemas. Isso nos ajuda a entender quais problemas são mais difíceis ou mais simples de resolver. Situação similar ocorre na geometria, quando usamos a semelhança de triângulos para simplificar problemas complexos.

Na teoria da computação, 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, como o teorema de Cook-Levin, que localizou a ideia de problemas NP-completos. Isso nos ajuda a entender os limites do que pode ou não ser resolvido por um computador.

Portanto, a redutibilidade nos auxilia a encontrar semelhanças entre problemas e entender quais são os mais difíceis. É como um guia que nos ajuda a navegar pelo terreno complexo da teoria da computação.

Sabemos que uma MT e algoritmos são sinônimos. Portanto, neste livro-texto, usaremos a redução como uma técnica de projeto de MT (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 que esta tem uma certa relação com a linguagem A, recriaremos uma máquina que resolva a linguagem A.

Informalmente, veja como funciona a redução através de algoritmos.

Seja o problema P, I_P é sua instância e S_P é sua solução. Devemos reduzir um problema A para um problema B, isto é, usar um algoritmo B para resolver A:

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

Figura 256

A instância do problema deve ser interpretada como "entrada" e a solução, como "aceita ou rejeita". 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.

Dessa forma, em 2 transformamos uma entrada válida do problema A em uma entrada válida do problema B através da relação f. Da mesma maneira, independentemente da solução apresentada pelo algoritmo B, devemos transformar, através da relação g, S_B em S_A.

Assim, o conteúdo retangular em 3 se transforma em um algoritmo de A.

Observe no exemplo dado que a redução serviu para, indiretamente, criar um algoritmo.

Vejamos outra aplicação: transformar um problema de ordenação em um problema de seleção.

Problema de seleção

Recebemos como entrada um vetor $\langle V, n, k\rangle$ 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, deve-se 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. Assim, deve-se descobrir o terceiro menor elemento de V. Logo, a resposta é 4.

Problema de ordenação

Recebemos como entrada um vetor $\langle A, m\rangle$ 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 que, conhecendo algum algoritmo de ordenação, o k-ésimo menor elemento vai ocupar exatamente a posição k, resolvendo o problema da seleção.

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, com ilustrado no esquema.

Dessa forma, é necessário transformar a entrada, através da relação f, no caso,.

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

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. Assim, em código, temos:

```
1 ALG_selecao(V. n, k) {

2 V = ALG ordenacao(V. n);

3 return V[k];

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.

Observação

O quicksort é um algoritmo de ordenação eficiente que divide uma lista em partições menores, ordenando-as recursivamente usando um pivô para comparações e trocas.

Introdução à redutibilidade

Lembre-se de que usaremos os conceitos de redutibilidade como uma técnica de projeto de MT (algoritmo), em que se usa uma máquina feita para uma linguagem B (problema B) para criar uma máquina para a linguagem A (problema A). Usaremos, a partir de agora, o termo reduzir de A para B.

Considerando as máquinas M_A e M_B, a redução de A para B é:

M_A = sobre a entrada x:

1. Seja y = f(x).

2. Processe M_B sobre y.

3. Aceite se M_B aceitar; se M_B rejeitar, rejeite.

Criamos uma máquina M_A que recebe uma entrada x, válida para A. Criamos também uma função y que representa uma entrada válida para o problema B e processamos a entrada y na máquina M_B.

Na realidade, quando estudamos decidibilidade, aplicamos o conceito de redutibilidade através de MT. Recordemos:

A_{AFD} = {⟨A, ω⟩/ A é um AFD que aceita ω}

M = sobre a entrada ⟨A, ω⟩, onde A é um AFD e ω é uma cadeia:

1. Simule A sobre a entrada ω.

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

A_{AFN} = {⟨A, ω⟩/ A é um AFN que aceita ω}

M_1 = 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 uma MT M na entrada ⟨B, ω⟩.

3. Se M aceitar, aceite a entrada; caso contrário, rejeite-a.

A_{ER}= {⟨R, ω⟩/ R é ER que gera ω}

M_2 = sobre a entrada ⟨R, ω⟩, onde R é ER e ω é uma cadeia:

1. Inicie convertendo a expressão R em um AFN equivalente, chamado de A.

2. Execute a máquina M_1 na entrada ⟨A, ω⟩.

3. Caso M_1 aceitar a entrada, também a aceite; no entanto, se M_1 rejeitar a entrada, rejeite-a também.

Concluindo, o que podemos extrair de importante sobre redutibilidade?

> 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. Essa última versão é a chave para se provar que vários problemas são indecidíveis (Sipser, 2007, p. 151).

Lembrete

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

6.1 Problemas indecidíveis da teoria das linguagens

Já estabelecemos a indecidibilidade do problema parada, estudado anteriormente, cuja descrição é $P_{MT} = \{\langle M, \omega\rangle / M \text{ é MT que para sobre } \omega\}$. Outra descrição do problema da parada constante nas bibliografias é $A_{MT} = \{\langle M, \omega\rangle / M \text{ é MT que aceita } \omega\}$. 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 P_{MT} para decidir A_{MT}? Em outras palavras, com reduzir A_{MT} para P_{MT}?

Agora, vamos abordar um problema relacionado chamado P_{MT}, que envolve a determinação de uma MT para (aceitar ou rejeitar) uma entrada de dados.

Para provar a indecidibilidade de $PARA_{MT}$, usamos a indecidibilidade de A_{MT} para reduzir A_{MT} a $PARA_{MT}$. Definimos P_{MT} como o conjunto de pares $\langle M, \omega\rangle$, onde M é uma MT e M para sobre a entrada ω.

Teorema 7: P_{MT} é indecidível.

Ideia da prova

Essa prova é por contradição. Suponhamos que P_{MT} seja decidível e usemos essa suposição para mostrar que A_{MT} também é decidível, o que contradiz o problema da parada. A ideia central é demonstrar que A_{MT} é redutível a P_{MT}.

Assumimos a existência de uma MT R que decide P_{MT} e, com base nisso, construímos S, uma MT que decide A_{MT}. 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 P_{MT}. Com R, podemos testar se M para sobre ω. Se R indica que M não para sobre ω, rejeitamos porque $\langle M, \omega\rangle$ não pertence a A_{MT}. Por outro lado, se R indica que M para sobre ω, podemos simular M sem riscos de looping. Isso leva a uma contradição, pois A_{MT} é conhecida por ser indecidível. Portanto, concluímos que R não pode existir, o que implica que P_{MT} é indecidível.

Prova

Supondo que o MT R decida $PARA_{MT}$, construímos o MT S para decidir A_{MT} da seguinte forma:

1. Execute um MT R na entrada $\langle M, \omega\rangle$.

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.

6.2 Redutibilidade por mapeamento

Uma função computável pode ser definida matematicamente de várias maneiras, mas uma das definições mais comuns e precisas é em termos de MT. Uma função f é considerada computável se existir uma MT 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: $\Sigma^* \rightarrow \Sigma^*$ que mapeia cadeias de caracteres (símbolos em um alfabeto Σ) em cadeias de caracteres. A função f é computável **somente** se existir uma máquina MT tal que, a cada entrada $\omega \in \Sigma^*$, a MT para após um número finito de passos e, ao parar, a fita contiver f(ω).

Nessa definição:

- $\Sigma^* \rightarrow \Sigma^*$ é o conjunto de todas as cadeias de caracteres possíveis.
- MT é uma MT que computa a função f.
- ω é uma entrada para a função f.
- f(ω) é a produção de MT quando processa ω.

Essa definição formaliza a ideia de que uma função é computável se houver um procedimento algorítmico (representado pela MT) que pode calcular o valor da função para qualquer entrada possível. Se tal MT existe, então a função é considerada computável na teoria da computação.

São exemplos de funções computáveis:

f(ω) = \$ ω, com ω qualquer

Essa função recebe uma cadeia ω e, ao recebê-la, transfere uma célula à direita inserindo, à esquerda de ω, o símbolo de \$ à esquerda da cadeia, a fim de detectar a borda esquerda da fita.

f($\langle x, y \rangle$) = $\langle x + y \rangle$, com x e y inteiros

Essa função recebe dois números inteiros (x e y) e devolve o resultado da soma destes.

f($\langle M, \omega \rangle$) = $\langle M', \omega \rangle$, com M uma MT e ω uma cadeia

Essa função recebe uma máquina M e uma cadeia ω. Devolve uma máquina M' que é igual a M acrescida de estados iniciais para rejeitar qualquer cadeia diferente de ω.

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 existir uma função computável f: $\Sigma^* \rightarrow \Sigma^*$ 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 seja 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:

- 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, $f(x) \in B$.

- 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, $f(x) \notin B$.

Em outras palavras, a função f mapeia instâncias de A para instâncias correspondentes de B de tal forma que a resposta para o problema A pode ser deduzido da resposta para o problema B. Isso implica que, se você tiver um algoritmo eficaz para resolver o problema B, você também poderá resolver o problema A de maneira eficaz usando f.

Em suma, podemos demonstrar a importância da redutibilidade na teoria da computação da seguinte maneira:

- **Teorema**: se A reduz para B e
  - B é decidível, então A é decidível.
  - B é Turing reconhecível, então A é Turing reconhecível.
- **Demonstração**: M_A = sobre a entrada x:
  - Seja y = f(x).
  - Rode MB sobre y.
  - Dê como saída o que MB der como saída.
- **Corolário**: se A se reduz para B e
  - A é indecidível, então B é indecidível.
  - A é Turing irreconhecível, então B é Turing irreconhecível.

7 COMPLEXIDADE I

Em teoria da computação, complexidade envolve a análise do desempenho dos algoritmos e o estudo de quão difícil é resolver problemas computacionais. Isso inclui os seguintes elementos:

- **Comportamento assintótico de funções**: uma complexidade muitas vezes se concentra no comportamento de funções à medida que o tamanho da entrada cresce indefinidamente. A notação O-grande (Big O) é usada para descrever esse comportamento e fornece uma estimativa superior da complexidade de tempo ou espaço de um algoritmo.

- **Problemas em classe P**: podem ser resolvidos de forma eficiente em tempo polinomial, ou seja, uma complexidade de tempo é limitada por uma função polinomial do tamanho da entrada. Exemplos incluem soma de números inteiros e ordenação de uma lista.

- **Problemas em classe NP**: são problemas cujas soluções podem ser verificadas eficientemente em tempo polinomial. Embora não se saiba se eles podem ser resolvidos de forma eficiente, a verificação da solução é rápida. Um exemplo é o problema do caixeiro-viajante.

- **Problemas NP-completos**: são problemas em NP que são tão difíceis quanto o mais difícil dos problemas em NP. Se um problema NP-completo puder ser resolvido de forma eficiente (P = NP), todos aqueles em NP terão soluções eficientes. Um exemplo de NP-completo é o problema da satisfatibilidade booleana (SAT).

O estudo da complexidade ajuda a determinar quais problemas são práticos de resolver em situações do mundo real (P), quais são difíceis de resolver. Contudo, suas soluções podem ser rapidamente verificadas (NP), e quais são os problemas mais desafiadores e potencialmente intratáveis (NP-completos). A conjectura P *versus* NP continua sendo uma questão em aberto na teoria da computação e é fundamental para a compreensão dos limites da computação.

7.1 Comportamento assinóptico de funções

Inicialmente, devemos acentuar o tempo de execução.

Após vermos vários problemas decidíveis e não decidíveis, podemos pensar que os limites da computação se resumem apenas ao que é ou não decidível. Entretanto, esse pensamento é equivocado. Em problemas decidíveis, existem aqueles que são tratáveis e os que não são tratáveis. Isso quer dizer que qualquer problema decidível exige o gasto de recursos como tempo e memória.

Assim, tomaremos alguns conceitos de análise de algoritmos para verificar a corretude, prever desempenho e comparar soluções sem que seja necessário implementá-las. Tudo isso é importante pois, de modo geral, não existe apenas um único algoritmo que resolva um problema.

Em se tratando de tempo de execução de um algoritmo, são vários os fatores que o afetam. Como exemplo, uma MT com mais de uma fita pode reduzir o tempo de execução. Outros fatores podem ser o

tamanho da entrada e a forma de expressá-la (binária ou unária), assim como a forma de representação do algoritmo.

Portanto, precisamos definir o que é tempo de execução. Podemos fazê-lo de maneira suficientemente simples e geral, de acordo com o tamanho da entrada de dados. Assim, se tivermos como entrada uma cadeia, o comprimento desta será o tamanho da entrada; se for um vetor, será o tamanho do vetor; se for um número, será a quantidade de bits para armazená-lo; se for um grafo, o tamanho da entrada será a quantidade de vértices e arestas.

Dessa forma, assim que conhecermos o tamanho da entrada, a próxima etapa é contar o número de passos executados pelo algoritmo no pior dos casos. Em uma cadeia, o número de passos será a quantidade de movimentos e gravações que o cabeçote da MT executará no pior dos casos.

Vejamos através de um exemplo o que significa o pior dos casos.

Exemplo de aplicação

Considere a linguagem $L = \{0^n1^n / n \geq 0\}$ e uma máquina M decisora. Dadas as cadeias de entrada $\omega = 0101$ e $\omega_1 = 0011$, determine o tempo de execução.

Resolução

Primeiro, 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.

Façamos agora o processamento da cadeia $\omega = 0101$

0	0	1	1	β	β	...

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.

Façamos agora o processamento da cadeia $\omega_1 = 0011$

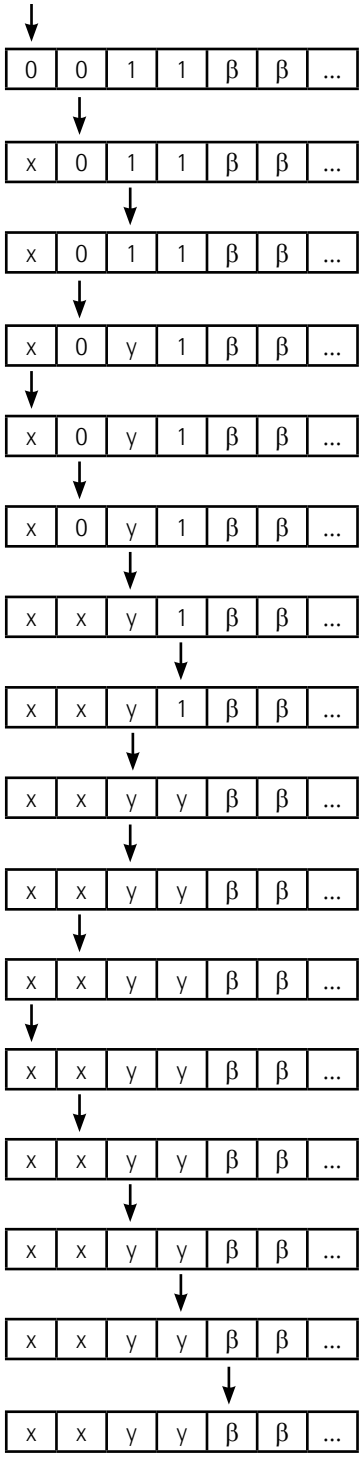

Figura 257

Nesse caso, houve 16 movimentos do cabeçote de leitura e gravação, portanto esse é o pior dos casos, logo o tempo de execução é 16.

Assim, para uma MT 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 MT 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.

Comportamento assintótico de funções

Desempenha um papel vital na análise da complexidade de algoritmos e na teoria da computação. Ele se concentra no estudo do comportamento de funções à medida que seus argumentos se aproximam de valores extremamente grandes (ou extremamente pequenos). A ideia é entender como as funções se comportam "no infinito" e como seu crescimento se compara a outras funções. Isso quer dizer que queremos medir um algoritmo em termos de tempo de execução ou o espaço (memória) usado. Como vimos, conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de n. Essas operações podem ser comparações, operações aritméticas, movimento de dados etc.

Estudaremos a seguir alguns conceitos importantes relacionados ao comportamento assintótico de funções.

Notação assintótica

É uma abstração matemática que auxilia na análise do tempo de execução. Considere uma expressão matemática, por exemplo, $5n^4 + 3n^2 - 10n + 100$. Observe que, para valores muito grandes de n, o que determinará o crescimento da função é o termo n^4, tornando os demais termos seus respectivos coeficientes e a constante sem relevância.

Portanto, em notação assintótica, essa é uma expressão da ordem de n^4. Graficamente:

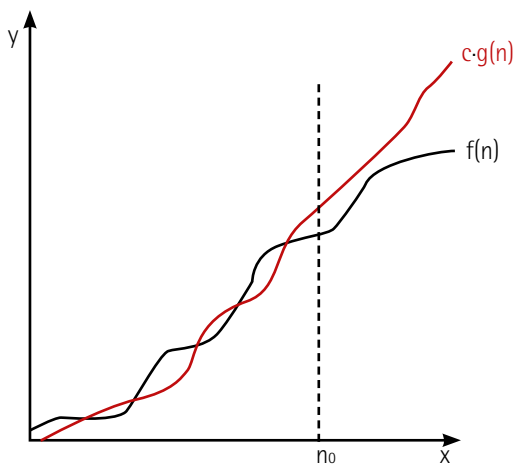

Figura 258

Na realidade, estamos determinando um limite superior para f(n).

Lembrete

O comportamento assintótico refere-se à tendência de uma função ou processo matemático se aproximar de um limite ou padrão à medida que se estende ao infinito.

Notação O-grande (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 n_0 após o qual $f(n) \leq c \cdot g(n)$, onde c é uma constante inteira e positiva para todo $n \geq n_0$.

Considere 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 a f(n), por exemplo:

$$4n^3 + 5n^2 - 3n + 15 \leq 4n^3 + 5n^2 + 15 \leq \frac{4n^3 + 5n^3 + 15n^3}{24n^3}$$

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

Observe com atenção o exemplo a seguir.

Exemplo de aplicação

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

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:

- $f(n) \leq c \cdot n^2 + c_1 \cdot \log n$
- $c \cdot n^2 + c_1 \cdot \log n \leq c \cdot n^2 + c_1 \cdot n^2$

Então $f(n) = (c + c_1)n^2$

Portanto, $f(n) = O(n^2)$

Classes de complexidade

Na teoria da complexidade computacional, a notação Big O é usada para descrever a complexidade de tempo ou espaço de algoritmos em termos de seus tamanhos de entrada.

Por exemplo, um algoritmo com complexidade de tempo $O(n^2)$ significa que seu tempo de execução cresce quadraticamente com o tamanho da entrada.

Tipos comuns de funções assintóticas

Destacaremos a seguir alguns deles:

- **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.
- **Complexidade O(log n) (logaritmo)**: 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 O(n) (linear)**: o crescimento no número de operações é diretamente proporcional ao crescimento do número de itens. Exemplo: o algoritmo de busca em uma lista/vetor.
- **Complexidade O(n log n) (linearítmica ou quasilinear)**: é resultado das operações (log n) executada n vezes. Exemplo: o caso médio do algoritmo de ordenação quicksort.
- **Complexidade $O(n^2)$ (quadrático)**: 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.
- **Complexidade $O(2^n)$ (exponencial)**: nela, 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 O(n!) (fatorial)**: 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.

O gráfico de algumas funções ilustra seu crescimento:

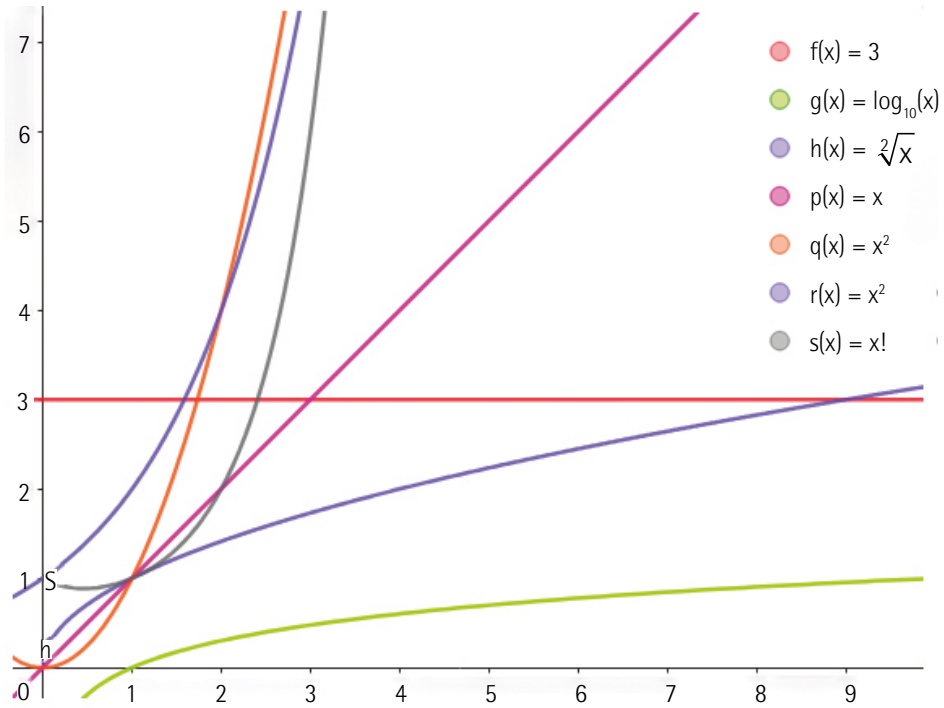

Figura 259

Observação

Análise de algoritmos envolve o estudo e a avaliação do desempenho de algoritmos computacionais, examinando seu tempo e espaço de execução para determinar eficiência e escalabilidade.

7.2 Problemas classe P e NP

Esse tema certamente é o maior problema em aberto na ciência da computação e um dos maiores da matemática. É conhecido como P *versus* NP e sua resolução trará impactos profundos para a 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?

Aparentemente, essa questão não parece tão difícil de ser respondida, mas durante a leitura será mostrado o que realmente ela 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, através da teoria dos jogos, uma estratégia perfeita para o jogo de xadrez, só que ela não é conhecida por ninguém, assim como não se sabe 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.

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

Vejamos uma adaptação do problema do caixeiro-viajante.

Exemplo de aplicação

Suponha que um vendedor more na cidade A e precisa visitar, de carro, outras três cidades (B, C, D) e depois retornar para casa, conforme a figura a seguir. Ele quer economizar tempo e gasolina. Dessa forma, qual é o menor percurso para que ele não deixe de visitar nenhuma cidade? Ao lado da imagem, 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

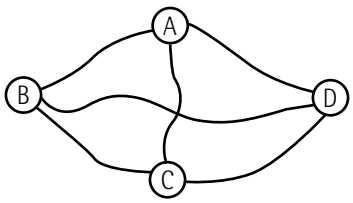

Figura 260

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

Tabela 2

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

Contudo, e se aumentarmos a quantidade de cidades para dez?

Listar as rotas manualmente não seria mais viável, pois existem 3.620.800 (10 fatorial) rotas possíveis, mas um computador ainda conseguiria executar essa tarefa com certa facilidade. Entretanto, se aumentarmos para apenas 30 cidades (30 fatorial de rotas possível), o melhor computador demoraria mais que a idade do universo para completar a tarefa.

Observe que esse problema é vital para a indústria e para o comércio. Imagine uma empresa de logística que precise planejar rotas para algumas dezenas de lugares.

Hoje já existem algoritmos melhores, conhecidos como algoritmos de força bruta, assim, não é preciso listar rotas e somar suas distâncias. Todavia, mesmo eles 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 da linguagem $L = \{0^n1^n / n \geq 0\}$, vimos que ele tem tempo de execução 16 para uma entrada de tamanho 4(ω = 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 $O(2^n)$, 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.

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

Todavia, existem problemas cuja complexidade é $O(2^n)$ ou O(n!). Esses algoritmos são denominados 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, mas descobri-la é inviável. A pergunta que fica é: como resolver esse tipo de problema?

Problemas de otimização *versus* problemas de decisão

O problema de otimização é aquele no qual 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).

Por sua vez, o problema de decisão envolve 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. Saber a menor distância é inviável, mas saber se determinada rota não ultrapassa

a distância de 260 é bem mais fácil, basta somar as distâncias entre os lugares. A vantagem é que agora o problema pode ser resolvido em tempo polinomial, pois pertence à classe NP (não determinísticos polinomiais). Os problemas em NP são aqueles que, 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 MT 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.

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

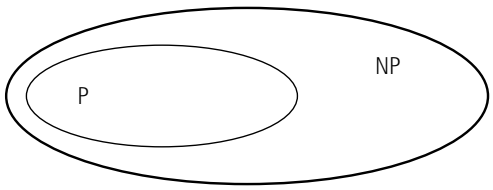

Figura 261

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, mas isso não quer dizer que ela não exista.

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 será fácil resolver o problema (também em tempo polinomial)?

No estudo do próximo tópico será apresentado por que essa questão é tão importante.

7.3 Problemas NP-completos e 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.

A ideia-chave que Cook apresentou será acentuada a seguir.

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 no modelo de uma fórmula booleana (conjunto de cláusulas). A pergunta em questão 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 ele é pelo menos tão difícil quanto qualquer outro problema em NP e que qualquer problema em NP pode ser limitado a ele de forma eficiente.

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.

Contudo, foi o cientista Richard Karp que apresentou novos problemas NP- completos com interesses muito mais práticos.

Como estudado anteriormente, reduzir um problema A em um problema B significa uma forma de interpretar o problema A dentro do problema B. Assim, 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). 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.

Também existe outra classe de problemas denominadas NP-difíceis, e a diferença é que não se exige que essa classe esteja dentro da classe NP. Assim:

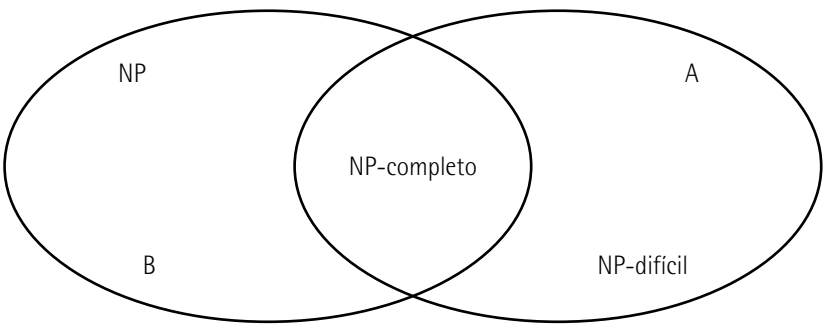

Figura 262

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 descobrir se P = NP.

Saiba mais

Para conhecer outros problemas NP-completos, consulte o item 7.5 do livro indicado a seguir:

SIPSER, M. *Introdução à teoria da computação*. São Paulo: Thomson Pioneira, 2007. p. 202.

8 COMPLEXIDADE II

8.1 Grafos eulerianos

Um grafo qualquer é classificado como euleriano quando contém um ciclo que percorre todas as suas arestas. Esse ciclo é denominado ciclo euleriano. O grafo ilustrado a seguir é um exemplo, uma vez que inclui o ciclo: 1 – 0 – 3 – 4 – 0 – 2 – 1, que é euleriano.

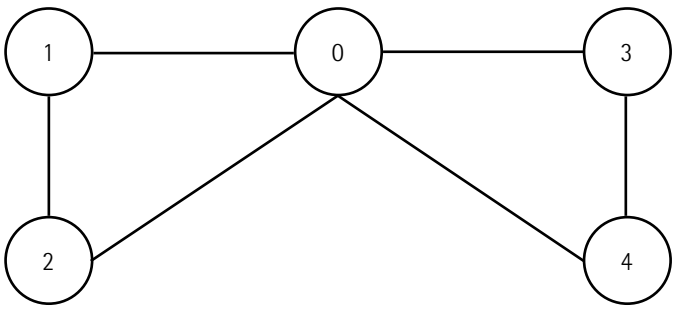

Figura 263

Entretanto, o grafo a seguir não pode ser classificado como euleriano.

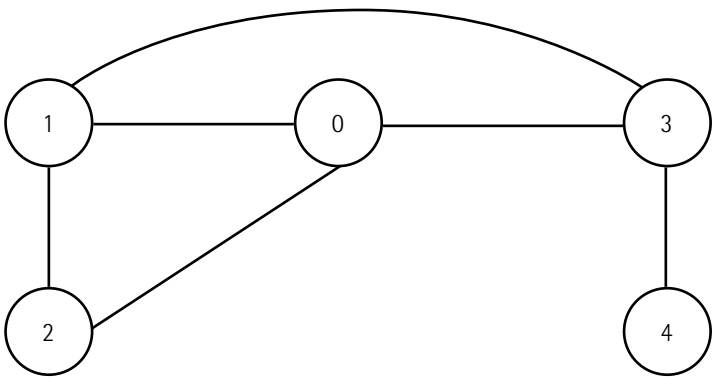

Figura 264

Tente percorrer as arestas e constate que, independentemente do vértice de partida, sempre terá, pelo menos, uma aresta não percorrida.

O seguinte teorema oferece uma maneira simples de determinar se um grafo é euleriano:

• **Teorema**: um multigrafo M é euleriano **somente** se M for conexo e cada vértice de M possuir grau par.

Agora, suponha que um multigrafo qualquer tenha uma trilha (não necessariamente um ciclo) que inclua todas suas arestas. Nesse caso, ele é denominado grafo atravessável, e a trilha é chamada de trilha euleriana. No grafo apresentado a seguir, a trilha 1 – 2 – 3 – 4 – 1 – 3 – 5 é atravessável.

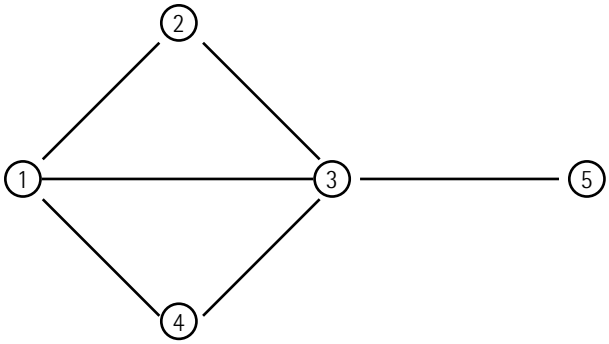

Figura 265

- **Teorema**: um multigrafo M é atravessável **somente** se M for conexo e possuir exatamente dois vértices de grau ímpar. Portanto, qualquer trilha euleriana em M inicia em um dos vértices de grau ímpar e termina no outro vértice de grau ímpar.

Esses teoremas são úteis para identificar grafos eulerianos e grafos atravessáveis e para determinar as propriedades dos vértices nesses contextos.

Observação

Um multigrafo é um tipo de grafo que permite múltiplas arestas entre os mesmos pares de vértices, incluindo laços (arestas que conectam um vértice a si mesmo).

O termo grafo euleriano é uma homenagem a Leonhard Euler, um matemático suíço do século XVIII. Em 1735, ele resolveu o famoso problema das sete pontes de Königsberg, que envolvia um grafo. Ele declarou que o grafo não tinha um caminho que atravessasse cada ponte uma única vez, mas mostrou que o grafo tinha um ciclo euleriano, que é um ciclo que passa por todas as arestas do grafo uma única vez.

Saiba mais

Para saber mais sobre o problema das sete pontes de Königsberg, assista ao vídeo da professora Sônia Regina Leite Garcia:

PONTES de Königsberg. 2021. 1 vídeo (9 min.). Publicado por Matemateca IME-USP. Disponível em: https://shre.ink/ntTp. Acesso em: 4 out. 2023.

8.2 Grafos hamiltonianos

Um grafo conexo qualquer é hamiltoniano quando contém um ciclo que visita cada vértice do grafo exatamente uma vez. Esse ciclo é denominado ciclo hamiltoniano.

De maneira geral, em todos os sólidos platônicos é possível estabelecer um ciclo hamiltoniano (cubo, tetraedro, octaedro, dodecaedro, icosaedro).

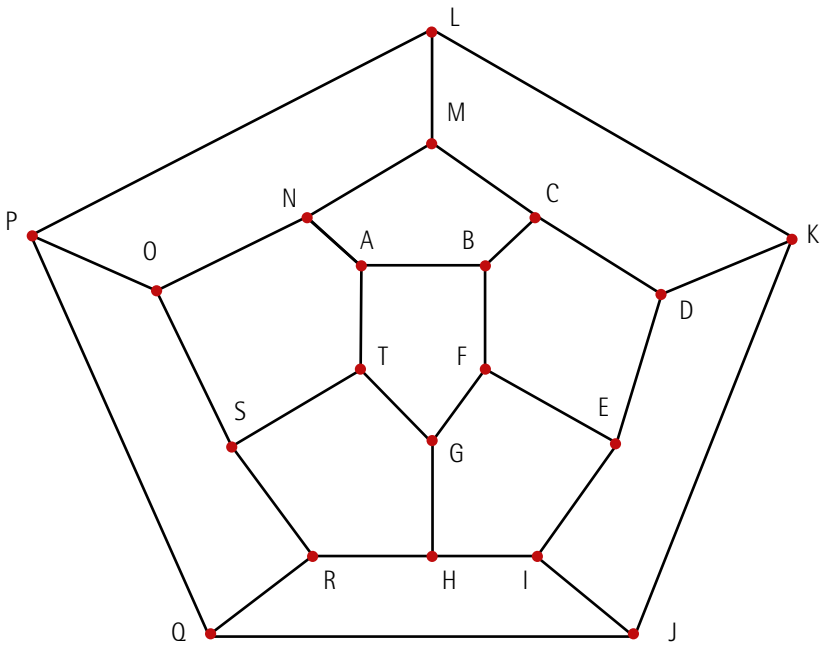

Figura 266 – Grafo hamiltoniano

É importante frisar que, no clico hamiltoniano, o único vértice com possibilidade de repetição é o primeiro.

O termo grafo hamiltoniano é uma homenagem a Sir. William Rowan Hamilton, um matemático e físico irlandês do século XIX. Ele desenvolveu a teoria dos quaternários e fez contribuições significativas para a matemática e para a física. Os grafos hamiltonianos foram nomeados após ele desenvolver seus estudos sobre circuitos em grafos. Um grafo hamiltoniano tem um ciclo hamiltoniano, que é um ciclo que passa por todos os vértices do grafo uma única vez. Essa propriedade é uma generalização do problema do caixeiro-viajante, que busca o caminho mais curto para visitar todos os locais uma vez.

8.3 Relação entre grafos e problemas P e NP

Os problemas P e NP são conceitos da teoria da complexidade computacional, enquanto os grafos eulerianos e hamiltonianos são conceitos da teoria dos grafos. Embora sejam áreas diferentes da matemática, há algumas relações indiretas entre eles.

Problema do circuito hamiltoniano e problema do circuito euleriano

Ambos estão diretamente relacionados aos conceitos de grafos hamiltonianos e eulerianos. O primeiro pergunta se um dado grafo contém um ciclo hamiltoniano, ou seja, um ciclo que passa por todos os vértices exatamente uma vez. Já o segundo pergunta se um grafo contém um ciclo euleriano, ou seja, um ciclo que passa por todas as arestas exatamente uma vez. O primeiro é um problema da classe NP, o que significa que não se sabe se existe um algoritmo eficiente para resolvê-lo em tempo polinomial. Já o segundo é um problema da classe P, ou seja, pode ser resolvido em tempo polinomial.

A teoria dos grafos pode ser usada para demonstrar a NP-completude de alguns problemas relacionados à teoria da complexidade computacional. Por exemplo, muitos problemas NP-completos podem ser formulados em termos de grafos e, em seguida, mostrados como redutíveis a outros problemas em teoria dos grafos, como o problema do caixeiro-viajante (circuito hamiltoniano).

A resolução de problemas relacionados a grafos, como encontrar ciclos hamiltonianos ou ciclos eulerianos em grafos específicos, pode ser intrinsecamente complexa e dependente das características do grafo em questão. Isso significa que a complexidade computacional de problemas de grafos pode variar bastante, dependendo das propriedades do grafo em questão.

Resumo

Aprendemos, inicialmente, o que é decidibilidade. Vimos que linguagens decidíveis são conjuntos de strings (cadeias) para os quais existe um algoritmo que, dada uma entrada, decide se essa entrada pertence à linguagem ou não.

Nesta unidade, estudamos diversos tipos de problema. Nesse contexto, acentuamos que o problema da parada é um exemplo clássico de problema indecidível. Pergunta-se se é possível construir um programa que, dada uma entrada e um código de programa, determinando se o programa irá parar ou entrar em loop infinito.

Em seguida, elencamos os problemas indecidíveis. Neles, não existe um algoritmo que possa resolver todas as instâncias do problema. O problema da parada é um exemplo de problema indecidível.

Também ganharam destaque os problemas classe P, classe NP, fazendo-se a comparação entre eles.

Finalmente, fizemos a relação entre grafos eulerianos e hamiltonianos com problemas classe P e NP: encontrar um ciclo euleriano em um grafo é um problema em P, enquanto encontrar um ciclo hamiltoniano é um problema NP-completo. Isso ilustra a diferença entre problemas em P e NP e a complexidade associada a cada um.

Exercícios

Questão 1. (FCM/2018, adaptada) Sobre uma importante classe de complexidade, a classe dos problemas NP-completos, avalie as afirmativas a seguir.

I – O circuito hamiltoniano não é um problema NP-completo.

II – O problema SAT é NP-completo.

III – A ordenação de uma lista é um problema NP-completo.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa B.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: o problema do circuito hamiltoniano é, de fato, um problema NP-completo. Isso significa que ele está na classe NP (cujas soluções propostas podem ser verificadas eficientemente em tempo polinomial) e é tão difícil quanto o mais difícil dos problemas em NP.

II – Afirmativa correta.

Justificativa: o problema da satisfatibilidade booleana (SAT) é um dos primeiros problemas NP-completos descobertos. Ele serve como base para a teoria da NP-completude.

III – Afirmativa incorreta.

Justificativa: a ordenação de uma lista não é um problema NP-completo, mas sim de classe P. A ordenação pode ser realizada de forma eficiente em tempo polinomial, como são os casos do algoritmo de ordenação rápida (quicksort) e do algoritmo de ordenação por fusão (mergesort).

Questão 2. (Fumarc/2018, adaptada) O comportamento assintótico de funções desempenha papel fundamental na análise da complexidade de algoritmos e na teoria da computação. A notação Big O é utilizada para descrever o comportamento assintótico superior de uma função em relação a outra.

A respeito dessa notação, que designa a complexidade de algoritmos, avalie as afirmativas a seguir.

I – Algoritmos de complexidade O(log n), ou complexidade logarítmica, são aqueles em que o número de instruções executadas cresce muito rapidamente para um pequeno número de dados.

II – Algoritmos de complexidade O(n), ou complexidade linear, são aqueles em que um pequeno trabalho é realizado sobre cada elemento de entrada.

III – Algoritmos de complexidade O(1), ou complexidade constante, são aqueles em que as instruções do algoritmo são executadas um número fixo de vezes.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa D.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: a complexidade O(log n) significa que o número de instruções executadas pelo algoritmo cresce de forma muito eficiente em relação ao crescimento do tamanho dos dados (n). Dessa forma, à medida que o tamanho dos dados aumenta, o número de instruções executadas cresce muito lentamente. A afirmativa descreve algoritmos de complexidade O(n!), ou fatorial, em que o número de instruções executadas cresce muito rapidamente em relação ao número de entradas.

II – Afirmativa correta.

Justificativa: a complexidade O(n) significa que o número de instruções executadas é proporcional ao tamanho dos dados de entrada. Dessa forma, para cada elemento de entrada, um trabalho constante é realizado.

III – Afirmativa correta.

Justificativa: a complexidade O(1) significa que o número de instruções executadas pelo algoritmo é constante, independentemente do tamanho dos dados. Desse modo, não importa quantos elementos de entrada existam, pois o número de instruções permanece fixo.

REFERÊNCIAS

Audiovisuais

O JOGO da imitação. Direção: Morten Tyldum. EUA: Netflix, 2014. 114 min.

PONTES de Königsberg. 2021. 1 vídeo (9 min.). Publicado pelo canal Matemateca IME-USP. Disponível em: https://shre.ink/ntTp. Acesso em: 4 out. 2023.

Textuais

BRAINERD, W. S.; LANDWEBER, L. H. *Theory of computation*. New York: John Wiley & Sons, 1974.

COPELAND, B. J. The Church-Turing thesis. *Stanford Encyclopedia of Philosophy Archive*, 2020. Disponível em: https://shre.ink/ntnP. Acesso em: 4 out. 2023.

DIVERIO, T. A.; MENEZES, P. B. *Teoria da computação*. Porto Alegre: Bookman, 2008.

DIVERIO, T. A.; MENEZES, P. B. *Teoria da computação*: máquinas universais e computabilidade. Rio Grande do Sul: Sagra Luzzatto, 1999. (Série Livros Didáticos Número 5).

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Rio de Janeiro: LTC, 2001.

HOPCROFT, J.; ULLMAN, J.; MOTWANI, R. *Introdução à teoria dos autômatos, linguagens e computação*. Rio de Janeiro: Campus, 2002.

JOSÉ NETO, J. *Introdução à compilação*. Rio de Janeiro, LTC, 1987.

KOZEN, D. C. *Automata and computability*. New York: Springer Verlag, 1997.

LEISERSON, C. E. *et al. Algoritmos*: teoria e prática. 2. ed. Rio de Janeiro: Campus, 2002.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos de teoria da computação*. 2. ed. Porto Alegre: Bookman, 2000.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos de teoria da computação*. 3. ed. Porto Alegre: Bookman, 2004.

MENEZES, P. B. *Linguagens formais e autômatos*. Porto Alegre: Bookman, 2010.

MENEZES, P. B. *Linguagens formais e autômatos*. 3. ed. Porto Alegre: Sagra Luzzatto, 2000.

MORAES, M. *Linguagens formais e autômatos*. São Paulo: Sol, 2013.

RAMOS, M. V. M.; JOSÉ NETO, J.; VEGA, I. S. *Linguagens formais*. Porto Alegre: Bookman, 2009.

ROSA, J. L. G. *Linguagens formais e autômatos*. Rio de Janeiro: LTC, 2010.

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

YAN, S. Y. *An introduction to formal languages and machine computation*. New Jersey: World Scientific, 1998.

Informações:
www.sepi.unip.br ou 0800 010 9000