É mais fácil verificar a solução do que encontrá-la (It is easier to verify the solution than to find it) – III

Valdir Monteiro dos Santos Godoi

Master Student at IMECC – UNICAMP
CEP 13083-859 Campinas, SP, Brasil

E-mail address: valdir.msgodoi@gmail.com

ABSTRACT

Introducing firstly the concepts of variable languages and languages with semantic and then without using them we presented original proofs of the famous P versus NP problem of Computer Science. These proofs don't implies that the NP-complete problems not belongs to P, so the P versus NP-complete problem remains open, as well as it is possible (perhaps) to solve the SAT problem in polynomial time.

***Keywords*:**
checking relation, deterministic algorithm, dynamical class, halting problem, language, non-deterministic algorithm, polynomial time, P versus NP, semantic, Turing machine, undecidability, universal Turing machine, unsolvable problem, variable language.

1. INTRODUÇÃO

Neste artigo responde-se afirmativamente à pergunta que dá título ao capítulo IV da parte B (Complexidade de Algoritmos) do livro "Aspectos teóricos da computação" [1], livro da coleção Projeto Euclides, do IMPA: É mais fácil verificar a solução do que encontrá-la? Resposta: Sim.

O mencionado capítulo traz no fim do seu primeiro parágrafo as igualdades

$$2^{67} - 1 = 147.573.952.589.676.412.927 = 193.707.721 \times 761.838.257.287,$$

o que contraria uma proposição (conjectura) de Mersenne (1588 – 1648). Ela foi apresentada por Frank Cole no encontro da Sociedade Americana de Matemática de 1903. Conforme citado em [1], esta era uma conjectura em aberto há mais de 250 anos, mas não levou mais do que alguns minutos para que Cole convencesse a sua audiência de matemáticos de que a conjectura em questão era falsa. Encontrar os fatores de um número composto pode em geral ser muito difícil, mas uma vez encontrados tais fatores torna-se um problema relativamente trivial verificar que o número em questão é realmente um número composto [1].

De novo citando [1], existe uma variedade de problemas conhecidos que parecem exibir o mesmo fenômeno, no sentido de que conhecemos algoritmos que verificam rapidamente se um candidato proposto como uma solução é ou não de fato uma solução, mas não conhecemos nenhum algoritmo rápido que encontra esta solução. Os melhores exemplos destes problemas são os problemas chamados de $NP-completos$: SAT (*satisfiability*),

problema do caixeiro-viajante, programação linear inteira, problema da mochila, soma de subconjunto, etc. [2, 3]

A pergunta que dá título ao capítulo, na realidade, é uma maneira mais simples de perguntar sobre a solução da questão *P versus NP* [1], o mais famoso problema em aberto da teoria da computação [4], um dos maiores problemas não resolvidos em ciência da computação teórica e matemática contemporânea [5], um dos mais profundos e mais importantes quebra-cabeças não resolvidos da matemática [6] com que se defrontam os matemáticos e cientistas da computação de hoje [7].

Esta questão foi levantada pela primeira vez por Cook [8], em 1971, quando mostrou que qualquer problema de reconhecimento resolvido por uma máquina de Turing não determinística polinomialmente limitada pode ser reduzido ao problema de determinar se uma dada fórmula proposicional é uma tautologia e discutiu sobre a forte evidência de que não é fácil determinar se uma dada fórmula proposicional é uma tautologia, mesmo se a fórmula está na forma normal disjuntiva, sugerindo que o conjunto $\{tautologies\}$ é um bom candidato para um interessante conjunto não em $\mathcal{L}_*$ (na notação de Cook [8], $\mathcal{L}_* = P$ e $\mathcal{L}^+ = NP$). Cook sentiu que vale a pena gastar um esforço considerável tentando provar essa conjectura e disse que tal prova seria um grande avanço na teoria da complexidade.

O propósito deste artigo é resolver o problema P *vs* NP, primeiramente através do uso de dois novos conceitos que chamamos de linguagem variável no tempo e linguagem com semântica, e a seguir sem usá-los. Na seção 2 descreveremos as definições básicas sobre as classes P e NP necessárias à conclusão principal deste artigo. Na seção 3 será usada uma propriedade fundamental que distingue um algoritmo (ou máquina) não determinístico de um determinístico, que é a capacidade ou habilidade (teórica) de criar cópias de si mesmo quando confrontado com uma escolha entre duas (ou mais) alternativas, uma cópia para cada alternativa, e prosseguir o processamento para cada uma delas, independentemente e sem interferir no processamento das demais, em paralelo [2, 5, 9, 10].

Horowitz e Sahni [11] dizem que esta é uma interpretação determinística de um algoritmo não determinístico, mas que uma máquina não determinística não faz qualquer cópia de um algoritmo a cada vez que uma escolha deve ser feita. Ao invés disso, ela tem a habilidade de selecionar um elemento correto do conjunto de escolhas (se tal elemento existe) a cada vez que uma escolha está sendo feita. Um elemento correto é definido relativo à menor sequência de escolhas que levam a um término bem sucedido. De maneira bastante pragmática, dizem que, como a máquina que estão definindo é fictícia, não é necessário para nós nos preocuparmos com o como a máquina pode fazer uma escolha correta a cada passo (step). Essa parece ser apenas uma opinião pessoal, uma definição alternativa, embora equivalente a outras, pois Michael Sipser, em [5], descreve de maneira mais detalhada como se processa o não determinismo usando o conceito de cópias e processamentos paralelos, sem referir-se a escolhas corretas bem sucedidas, com uma única tentativa, a cada vez em que é necessário escolher. Na seção 4 usaremos estas duas formas de compreender o não determinismo.

A seção 5 provará $P \neq NP$ sem usar máquinas de Turing não determinísticas, mas usando o conceito de relação de verificação, ou checagem (*checking relation*), que é baseado apenas em máquinas determinísticas, e a seção 6 provará $P \neq NP$ através de uma variação de um problema indecidível, o conhecido Problema da Parada (*Halting Problem*), e o uso de uma máquina de Turing universal, sem usar nenhum dos nossos dois novos conceitos. A seção 7 concluirá o presente trabalho.

2. AS CLASSES P E NP

Conforme definido em [8], P é a classe dos conjuntos reconhecíveis em tempo polinomial. Aqui está implícito o uso de máquinas de Turing. Mais explicitamente: Seja P a família de todos os conjuntos reconhecíveis em tempo polinomial por uma máquina de Turing determinística, e seja NP a família de todos os conjuntos aceitáveis em tempo polinomial por uma máquina de Turing não determinística [1]. Parece razoável identificar a classe de problemas cuja solução pode ser encontrada rapidamente por um algoritmo com P e a classe de problemas cuja solução pode ser verificada rapidamente por um algoritmo com NP [1].

Uma máquina de Turing M reconhece um conjunto $A \subseteq \Sigma_{e/s}^*$ se para toda entrada $x \in \Sigma_{e/s}^*$ M para, e aceita x se e somente se $x \in A$ [1]. Um problema de reconhecimento é uma questão que pode ser resolvida por *sim* ou *não* [12] (*yes* ou *no*, *aceita* ou *rejeita*, ACCEPT ou REJECT, SUCCESS ou FAILURE, 1 ou 0, etc.), ou seja, é um problema de decisão.

P é uma classe robusta e tem definições equivalentes sobre uma larga classe de modelos de computadores [13]. Podemos dispensar o uso das máquinas de Turing e definir P informalmente como a classe dos problemas de reconhecimento que podem ser resolvidos por um algoritmo de tempo polinomial [12], i.e., a classe dos problemas de decisão solúveis por algum algoritmo com um número de passos limitado por algum polinômio fixo no tamanho da entrada [13]. A classe P pode ser definida muito precisamente em termos de qualquer formalismo matemático para algoritmos, e todos estes modelos razoáveis de computação têm uma propriedade notável: se um problema pode ser resolvido em tempo polinomial por um deles, ele pode ser resolvido em tempo polinomial por todos os outros modelos. Esta classe P, portanto, é extremamente estável sobre variações nos detalhes de nossas suposições (a respeito de um modelo específico adotado). Em outras palavras, P é a classe dos problemas de reconhecimento relativamente simples, para os quais existem algoritmos eficientes [12], i.e., algoritmos que rodam em tempo polinomial em relação ao tamanho da entrada do problema (de reconhecimento, ou decisão) para fornecerem a resposta correta (*sim* ou *não*).

Formalmente, os elementos da classe P são linguagens [13]. A classe de todas as linguagens polinomialmente decidíveis (reconhecíveis) é denotada por P. Uma linguagem é dita polinomialmente decidível se houver alguma máquina (de Turing, etc.) polinomialmente limitada que a decida (responda *sim* ou *não*). Uma máquina (de Turing, etc.) é dita polinomialmente limitada se há um polinômio $p(n)$ tal que, para qualquer entrada x, a máquina para após, no máximo, $p(n)$ passos, onde n é o comprimento da cadeia de entrada. Seguimos [7], com parênteses nossos.

Damos o nome de linguagem a qualquer conjunto de cadeias (sequências finitas de símbolos) sobre um alfabeto Σ. Alfabeto é um conjunto finito de símbolos. O conjunto de todas as cadeias, incluindo a cadeia vazia (ε), sobre um alfabeto Σ, é denotado Σ^* (* é a estrela de Kleene) [7]. Cadeia e string (termo em inglês) são sinônimos.

Nós dizemos que um problema de reconhecimento A está na classe NP se existe um polinômio $p(n)$ e um algoritmo $\mathfrak{a}$ (o algoritmo de checagem do certificado) tal que é verdadeiro o seguinte [12]:

O string x é uma instância *sim* de A se e só se existe um string de símbolos em Σ, $c(x)$ (o certificado), sendo $|c(x)| \leq p(|x|)$, com a propriedade que $\mathfrak{a}$, se fornecida para ele a entrada $x\$c(x)$, obtém a resposta *sim* depois de no máximo $p(|x|)$ steps.

Nós não queremos que cada instância (entrada) possa ser respondida em tempo polinomial por algum algoritmo. Nós simplesmente requeremos que, se x é uma instância *sim* do problema, então existe um certificado conciso para x, i.e., com comprimento limitado por um polinômio no tamanho de x, que pode ser checado em tempo polinomial para validade [12].

Em termos de linguagem, podemos definir NP como a classe das linguagens decididas por máquinas de Turing não determinísticas em tempo polinomial [14]. Esta definição parece contrastar com a anterior, pois aqui parece que tanto as instâncias *sim*, quanto as instâncias *não*, devem ser decididas em tempo polinomial. Não obstante, exige-se apenas das instâncias *sim* que elas sejam reconhecidas em tempo polinomial: Se A é um algoritmo não determinístico de reconhecimento de strings então nós dizemos que A opera em tempo polinomial se há um polinômio $p(\cdot)$ tal que, sempre que A aceita x, há uma computação de aceitação para x de comprimento menor ou igual a $p(|x|)$ [2]. E assim vemos ser apropriado definir NP como feito no início desta seção: NP é a família (ou classe) de todos os conjuntos (ou linguagens) aceitáveis em tempo polinomial por uma máquina de Turing não determinística [1]. Não nos interessa o número de passos, ou steps, gastos para decidir as instâncias *não*, de rejeição.

No clássico livro de Garey e Johnson [3], fixa-se por convenção como igual a 1 o número de steps (ou complexidade do tempo) no caso de não aceitação de um string de comprimento n por um programa M em uma máquina de Turing não determinística (MTND). Se L_M é a linguagem reconhecida por M,

$$L_M = \{x \in \Sigma^* : M \text{ aceita } x\},$$

e $T_M: \mathbb{Z}^+ \to \mathbb{Z}^+$ é a função de complexidade de tempo para M, então

$$T_M(n) = \max(\{1\} \cup \{m: \text{há um } x \in L_M \text{ com } |x| = n \text{ tal que o tempo para aceitar } x \text{ por } M \text{ é } m\}).$$

O tempo requerido por um programa M em uma MTND para aceitar o string $x \in L_M$ é definido como o número mínimo de steps que ocorrem até o estado de parada q_Y acontecer.

O programa M em uma MTND é um programa em MTND de tempo polinomial se existe um polinômio p tal que $T_M(n) \leq p(n)$ para todo $n \geq 1$. Formalmente agora, a classe NP é definida como

$$NP = \{L: \text{há um programa } M \text{ de tempo polinomial em uma MTND tal que } L_M = L\}.$$

Outra maneira alternativa de definir NP existe [14], envolvendo o conceito de relação. Seja $R \subseteq \Sigma^* \times \Sigma^*$ uma relação binária sobre strings. R é chamado polinomialmente decidível se há uma máquina de Turing decidindo a linguagem $\{x; y : (x, y) \in R\}$ em tempo polinomial. Nós dizemos que R é polinomialmente balanceada se $(x, y) \in R$ implica que $|y| \leq |x|^k$ para algum $k \geq 1$. Ou seja, o comprimento do segundo componente é sempre limitado por um polinômio no comprimento do primeiro. Temos assim a seguinte proposição:

Seja $L \subseteq \Sigma^*$ uma linguagem. $L \in NP$ se e somente se há uma polinomialmente decidível e polinomialmente balanceada relação R tal que $L = \{x: (x, y) \in R \text{ para algum } y\}$.

Aqui consideramos x como o string de entrada e y como um certificado conciso para sua verificação e codificando uma computação de aceitação sobre a entrada x.

Cook em [13] também usou a relação de verificação ou checagem para definir a classe NP. Cook comenta que NP remete a tempo polinomial não determinístico, pois originalmente NP foi definido em termos de máquinas não determinísticas, i.e., máquinas que têm mais de um possível movimento para uma dada configuração, mas que agora é mais costumeiro dar uma definição equivalente usando a noção de relação de checagem, que é uma relação binária $R \subseteq \Sigma^* \times \Sigma_1^*$ para alfabetos finitos Σ e Σ_1. Nós associamos com cada relação R uma linguagem L_R sobre $\Sigma \cup \Sigma_1 \cup \{\#\}$ definida por

$$L_R = \{w\#y \mid R(w, y)\},$$

onde o símbolo # não está em Σ. Nós dizemos que R é de tempo polinomial se e somente se $L_R \in P$. Por sua vez, Cook [13] define a classe P de linguagens por

$$P = \{L \mid L = L(M) \text{ para alguma máquina de Turing } M \text{ que roda em tempo polinomial}\},$$

onde

$$L(M) = \{w \in \Sigma^* \mid M \text{ aceita } w\}$$

é a linguagem aceita por M.

A classe NP de linguagens é então definida pela condição de que uma linguagem L sobre Σ está em NP se e somente se há $k \in \mathbb{N}$ e uma relação R de checagem em tempo polinomial tal que, para todo $w \in \Sigma^*$,

$$w \in L \Leftrightarrow \exists y \left(|y| \leq |w|^k \text{ e } R(w, y)\right),$$

onde $|w|$ e $|y|$ denotam os comprimentos de w e y, respectivamente.

3. LINGUAGENS VARIÁVEIS NO TEMPO

Conforme sabemos, não é possível acertar 100% das vezes nossas previsões sobre o resultado de um jogo de par ou ímpar, cara ou coroa, lançamento de dados, roleta, cartas, loteria, mercado de ações, etc., enfim, todos os eventos onde o caráter probabilista domina, tem forte influência.

Não é possível acertar sempre nossas previsões no sentido determinista, admitindo-se que vivemos num mundo onde existe o livre arbítrio. Embora existam as rígidas leis da Física, elas não são capazes de predizer o futuro em toda a sua extensão, e com toda a precisão.

Sendo assim, está condenada ao fracasso qualquer tentativa de construir um algoritmo ou programa de computador determinísticos, destinados a acertar inequivocamente, sem erro algum, e em todas as vezes, estes resultados probabilísticos ou aleatórios (desde que não se trate de dados viciados, cartas marcadas, times e juízes comprados, etc.).

O mesmo já não pode ser dito de algoritmos não determinísticos. Um algoritmo não determinístico pode ser considerado como um processo que, quando confrontado com uma escolha entre duas (ou mais) alternativas, pode criar cópias de si mesmo para cada alternativa e prosseguir o processamento para cada uma delas, independentemente das demais, em paralelo. Se existir um conjunto de possibilidades que levem a uma resposta positiva então esse conjunto é sempre escolhido e o algoritmo terminará com sucesso [2, 5, 9, 10].

Num exemplo simples de lançamento de dados, onde os resultados possíveis são os elementos do conjunto DADOS = {1, 2, 3, 4, 5, 6}, nossa máquina (de Turing) determinística (ou computador moderno) poderá lançar seu palpite entre os elementos de DADOS, mas terá apenas 1/6 de probabilidade de acerto.

Como nossa idealizada máquina de Turing não determinística (MTND) poderá escolher cada uma das seis alternativas possíveis de DADOS (produzindo, digamos, seis cópias de si mesma), uma das alternativas deverá evidentemente coincidir com o resultado correto do lançamento do dado, e o processamento terminará no estado de sucesso ou aceitação.

Vimos assim que uma máquina ou algoritmo não determinísticos são capazes de algo que a correspondente versão determinística não é capaz: acertar sempre.

Para cada lançamento a linguagem relacionada ao resultado correto variará em função deste resultado, ou seja, se nosso dado mostrou a face 1 para cima então a linguagem $L = \{1\}$ é a linguagem que produzirá o resultado correspondente ao sucesso/aceitação naquele momento, enquanto os demais valores possíveis, 2, 3, 4, 5 e 6, não pertencerão a L durante aquela jogada específica, com aquele jogador específico.

Num momento seguinte poderemos ter $L = \{3\}$, no próximo lançamento e a seguir $L = \{6\}$, depois $L = \{2\}$, $L = \{5\}$ e novamente $L = \{1\}$, etc., evidenciando que temos um exemplo de linguagem não constante, e variável no tempo, dependente de cada nova situação. Se não há próxima jogada então podemos definir $L = \{0\}$, por convenção, ou L deixa de existir, i.e., $L = \emptyset$, conjunto vazio, a partir daquele momento.

Para que um algoritmo não determinístico, ou sua correspondente MTND, construído para reconhecer e aceitar estas linguagens seja capaz de decidir entre aceitar ou rejeitar um dado de entrada, entre informar um SIM ou NÃO, SUCESSO ou INSUCESSO, é necessário que venham dados do ambiente externo, a fim de se poder decidir, pela comparação, sobre a aceitação ou não de seu palpite, previsão, estimativa, etc.

Se definirmos uma linguagem da forma

(1) $L = \{w \mid w = (x\&y\&z)$, x é a chave do problema, y é o palpite da máquina, calculado e registrado antes de se saber o valor de z, z é o resultado correto do problema$\}$

a máquina aceitará a entrada sempre que $y = z$, e rejeitará caso contrário, ou seja, os elementos de L neste problema são os strings w tais que $y = z$. O conteúdo de x servirá como um identificador, garantindo a unicidade do problema. Estamos pressupondo que x, y, z sejam dados consistentes, caso contrário a entrada será rejeitada. O caractere & separará x, y e z, e vamos supor que o caractere branco (␣) não esteja em w.

Este é um procedimento em tempo polinomial: dado x gera-se y (de maneira não determinística para as MTND ou determinística para as MTD), espera-se a realização completa do evento indicado em x, obtém-se z (do ambiente externo, que informará o resultado que efetivamente ocorreu) e se $y = z$ aceita-se a entrada w, supondo valores consistentes. Um exemplo para x: PETR4-2021-02-22-CLOSE, onde se pergunta qual o valor que a ação PETR4 da Petrobrás terá no fechamento de 22/fevereiro/2021 (suponhamos valores possíveis na faixa de R$ 0,00 a R$ 10.000,00 apenas. Valor R$ 0,00 significa que não houve negócios de PETR4 naquele dia, p.ex., devido a um feriado).

Uma MTND, corretamente projetada, tenderá por aceitar (em tempo polinomial) um de seus palpites y, pois gerará todos os valores possíveis para z, um para cada string de entrada, então esta é uma classe de problemas que pertence a NP.

A versão determinística não terá a "habilidade", propriedade, de produzir cópias de si mesma, como se fossem processamentos em universos paralelos e simultâneos, e poderá apenas fornecer um único palpite para a chave x, que seja baseado em algum tipo de procedimento matemático e/ou estatístico mais adequado para a solução da questão. Dado o caráter incerto destes problemas (dados, roletas, cartas, bolsa de valores, adivinhações, etc.) não se poderá dizer que se construiu um algoritmo, nem máquina determinística, para se acertar/resolver o problema, com certeza absoluta, sendo assim estes tipos de problemas não pertencem a P, mas pertencem a NP, então $P \neq NP$.

Vemos que esta é uma demonstração que utiliza uma das mais fundamentais propriedades do não determinismo, uma característica que as máquinas determinísticas são incapazes de realizar, que é a produção de "cópias" de si mesma e o processamento simultâneo com as outras "versões" da máquina (ou programa), até mesmo uma quantidade exponencial de cópias de si mesma (em relação ao tamanho da entrada).

Se preferirmos não adotar esta propriedade de criação e paralelismo, e ao invés disso admitirmos que as MTND têm uma sorte absoluta, que são abençoadas com uma sorte inacreditável, de modo que sempre fazem a melhor escolha [15], na primeira tentativa, com o poder de adivinhar corretamente a cada passo, como escrevem Papadimitriou *et al* em [6] e Horowitz e Sahni em [11], nossa demonstração não mudaria em essência: o algoritmo não determinístico acertaria sempre, desta vez por sorte extrema, mesmo sem criar novas versões da máquina, uma para cada alternativa que é necessária ao algoritmo. O algoritmo determinístico, por sua vez, não teria nenhuma sorte absoluta, faculdade premonitória, nem poder "sobrenatural" de multiplicação instantânea, e só seria capaz de acertar, em geral, em termos probabilísticos.

Percebam que não é um eventual tempo de execução de ordem exponencial para se gerar um palpite a causa principal que faz com que estes problemas (DADOS, BOVESPA, etc.) não pertençam a P, mas sim a impossibilidade de resolver sempre e corretamente uma entrada lida pela máquina determinística. É como um navio de passageiros que encalha ou afunda 95% das vezes e só completa uma viagem com probabilidade de 5%. É um navio que obviamente não serve para se viajar, e ninguém deveria usá-lo. Idem uma máquina de calcular que troca aleatoriamente as funções de seus botões e não nos avisa. Ou seja, provar que $P \neq NP$ não implica que $SAT \notin P$ ou de forma genérica $NP-completo \notin P$. É possível ter $P \neq NP$ e também $NP-completo \in P$. Isso contraria o corolário ($P = NP$ se e somente se $SAT \in P$) [2] do Teorema de Cook (SAT é $NP-completo$) [8], mas está claro que Cook e Karp não se utilizaram do conceito de linguagens variáveis para provarem este teorema e corolário, portanto este corolário refere-se apenas às linguagens constantes no tempo.

Também vale a pena mencionar que em nosso exemplo de PETR4 perguntamos sobre um valor que será conhecido somente daqui a dois anos aproximadamente. Claro que foi um exemplo exagerado, pois podemos entrar com dados muito mais próximos de acontecer, e mais simples de verificar (como o lançamento de dados, o nosso exemplo inicial).

De qualquer forma, para que possamos processar o resultado correto, o computador (ou máquina de Turing, MT) deve ser informado através de algum mecanismo de entrada sobre este correto valor. Enquanto o evento não terminar, por exemplo, o pregão de ações do dia ou o lançamento do dado, o computador (ou MT) ficará aguardando a entrada do valor correto,

mas já fez sua previsão. Claro, uma MTND, que se multiplicou em n cópias ou versões, devido às n alternativas de possibilidades, deverá receber a entrada contendo o valor correto em todas estas suas n versões. Isso é diferente do que se faz, mas afinal nossa prova é original. Onde está definido ou prescrito que as MT e os computadores modernos só podem começar a processar seus dados após lerem todos os caracteres de entrada até o fim, sequencialmente, e só após a leitura completa e ininterrupta desta entrada resolver sua "questão", sem mais nenhuma possibilidade de ler outros caracteres de entrada durante o processamento? Tal restrição não está descrita formalmente em lugar algum, por exemplo, em [13]. Portanto, admitimos uma espera até que o evento probabilístico tenha ocorrido por completo e a informação do respectivo resultado tenha sido fornecida às MT.

Certamente que tem de ser $P \neq NP$.

4. LINGUAGENS COM SEMÂNTICA

Não parece difícil concordar que é mais fácil verificar o resultado de um jogo de loteria do que ganhar nesse mesmo jogo de loteria. Conferir um jogo é fácil, bem mais difícil é acertar este jogo. Não fosse isso teríamos muitos e muitos milionários em decorrência dos jogos de azar, o que não ocorre. Acertar em um jogo ou aposta, por que não, também é um tipo de problema, inclusive matemático, e usamos a característica probabilística dos jogos e da bolsa de valores para provar $P \neq NP$.

Pode-se criticar que, quando desprovida de significado e da necessidade de se operar como descrito na definição de L (gerar y de acordo com x e aguardar a obtenção do valor z correto), então $L \in P$, pois qualquer string de entrada no formato $(x\ y\ z)$ que tenha $y = z$ pode ser lido e imediatamente aceito pela máquina, rejeitando-se a entrada se $y \neq z$. Isso é verdade, mas também podemos ter linguagens que aceitam os strings com $y = z$ independentemente do tempo e do significado de x, sem nenhum vínculo com um acontecimento verdadeiro. Ou seja, não estão resolvendo o problema que queríamos: a previsão de um resultado específico. Por definição, não queremos que estes outros strings pertençam a L. Para que nosso exemplo de linguagem L em (1) faça algum sentido, possa registrar alguma ocorrência variável no tempo e se distinga de qualquer outro conjunto de strings é que necessitamos de um significado para x, além de significados para y e z relacionados com x, e operar como descrito na definição de L. O programa, ou máquina M, que lê w precisa efetivamente calcular y, sem conhecer previamente o resultado correto z. Temos, assim, linguagens com semântica (significado).

Como existem linguagens variáveis no tempo, então as classes P e NP também devem variar no tempo, quando contém estas linguagens, constituindo assim classes dinâmicas. E como também vimos que há linguagens com significado, podemos dividir estas classes em dois tipos: sem semântica, ou pura, e com semântica, ou seja,

$$P = P(t) = P_{pura}(t) \cup P_{sem\hat{a}ntica}(t)$$

e

$$NP = NP(t) = NP_{pura}(t) \cup NP_{sem\hat{a}ntica}(t),$$

onde podemos representar o tempo t, por convenção, como a data e horário de Greenwich ou senão por alguma outra maneira razoável para nossos fins teóricos.

Esses novos conceitos podem se estender também às outras classes de linguagens e tornam, sem dúvida, ainda mais rico e interessante o estudo da Ciência da Computação.

Perguntando se há algum t tal que $P(t) = NP(t)$, somos levados de volta ao nosso problema original: *P versus NP*. Este é um problema que também tem alcance na filosofia. As máquinas de Turing não determinísticas são fictícias, são apenas construções mentais para fins teóricos em ciência da computação, portanto não perde o sentido pensar como se comportam P e NP quando t é anterior ao nascimento dos modelos computacionais, até mesmo anterior à existência do ser humano. Para um tempo t passado devemos pensar na construção de P e NP como se estivéssemos em um tempo t' anterior a t, i.e., $t' < t$, de modo que se possa continuar a raciocinar em termos de previsões futuras, ainda que no passado. Podemos pensar nos mais diversos eventos probabilísticos ou previsão de fenômenos naturais, como chuvas, terremotos, relâmpagos, trovões, ventos, avalanches, incêndios, inundações, ou qual animal pré-histórico vencerá (ou melhor, venceu) determinada disputa, etc. Há uma quantidade inimaginável de ocorrências possíveis em qualquer período do tempo, do passado ou futuro, e cujo resultado não pode ser considerado completamente previsível. Prever com absoluta certeza e acertar sempre, em 100% das vezes, o resultado correto destes eventos é assim impossível para uma MTD, e portanto estes problemas não pertencem a P. Já uma MTND, como vimos, "chutará" rapidamente cada uma das alternativas possíveis de resultado, e uma delas terá necessariamente que ser o resultado correto (supondo-se um espectro discreto e finito de valores possíveis), fazendo estes problemas pertencerem a NP. Assim, $P(t) \neq NP(t)$, para qualquer valor de t, presente, passado ou futuro (pelo menos considerando a existência do Universo e seus múltiplos elementos neste instante t). Não precisamos nos preocupar sobre quem informará o resultado correto para a MTND. De seu já mencionado poder de acertar sempre na primeira tentativa, usando sua sorte inacreditável, sua habilidade de selecionar um elemento correto de cada conjunto de escolhas, então a própria MTND reconhecerá qual o resultado correto, sem interferência humana, mesmo que não saibamos como isso pode ser feito efetivamente. Sendo assim, ao contrário do que fizemos inicialmente na seção 3, ela só precisará dar um único palpite, que será certeiro, ao invés de $n, 2^n$, palpites simultâneos e em paralelo.

Neste caso podemos omitir o valor z na linguagem (1), ficando apenas com

(2) $L = \{w \mid w = (x\&y),$ x é a chave do problema, y é o palpite correto da máquina, calculado e registrado antes de se saber o resultado do problema$\}$.

Para uma MTND, usando a propriedade de sorte perfeita e permanente, o palpite y sempre estará correto, se existir um resultado correto, sendo desnecessária a posterior comparação com z. Isto pode tornar o problema mais interessante, menos óbvio, e a entrada dos dados mais simples. Lembremos que nesta forma (2) também está sendo usada a noção de linguagem com semântica, portanto y não poderá ser gerado após a ocorrência do evento descrito por x, nem aceito qualquer par de valores $(x\&y)$.

E quem informará a chave x para uma MT quando t refere-se ao passado ou futuro sem a existência do ser humano? Aliás, quem projetará uma MTND para esse fim e disponibilizará energia para ela funcionar? De fato, na realidade, ninguém. Já que as MTND são fictícias, irreais, tanto os problemas a serem resolvidos por elas quanto suas respectivas instâncias (entradas) são, em princípio, abstrações, idealizações, elementos de modelos matemáticos que parecem simular uma realidade mesmo na ausência de atores físicos reais. Façamos então que uma espécie de entidade inteligente superior, ainda que imaginária, possa, por exemplo,

ocupar o lugar do ser humano nesses momentos. Essa entidade também poderá fornecer o valor z no caso da linguagem (1).

Quanto ao valor y, o palpite (sempre correto) da máquina, ele pode ser gravado pela própria MTND em sua fita de entrada/saída. Para informar o término da realização do evento indicado por x e a subsequente decisão da máquina (sempre *sim* para a MTND, supondo entrada consistente e havendo alguma solução), pode-se informar um caractere final de controle na fita, como) ou # ou $, também digitado por este ser abstrato inteligente. A mencionada espera (pausa) entre a geração do palpite e o término do evento é conseguida ou através de vários movimentos para a direita e para a esquerda na fita, do tipo vai e volta, até ser digitado e lido nesta fita o caractere indicador de fim, ou através de uma implementação mais sofisticada das funções *pause/continue* nesta máquina.

5. RELAÇÃO DE CHECAGEM

Conforme vimos nas seções anteriores, algoritmos não determinísticos são fictícios [11], bastante irreais [6], portanto nesta seção vamos provar $P \neq NP$ sem usar o não determinismo das formas como ele é entendido. Não usaremos nem cópias, bifurcações, multiplicações instantâneas e processamentos paralelos simultâneos [2, 5, 9, 10], nem sorte absoluta, inacreditável, nenhum poder ou habilidade de adivinhar corretamente a cada vez que uma escolha precisa ser feita [6, 11, 15]. Precisaremos ainda, entretanto, das linguagens variáveis no tempo e com semântica.

Vamos usar uma linguagem parecida com a descrita em (2),

(3) $L = \{w \mid w = x\&y\&t,\ x$ é a chave do problema, y é o palpite da máquina, calculado e registrado antes de se saber o resultado correto do problema, t é o instante final da ocorrência do evento indicado por $x\}$,

mas omitindo-se os parênteses e incluindo o tempo t na linguagem. O caractere & será de novo o separador dos conteúdos das variáveis, neste caso, x, y e t. A realização completa do evento descrito por x será indicada por t (contendo data, hora e referência local ou Greenwich), e o primeiro caractere branco (espaço) após t indicará o fim do string w, a partir do qual a máquina decidirá (informará *sim* ou *não*).

Sejam Σ_0 e Σ_1 alfabetos finitos não vazios, tais que os caracteres & e # não pertençam a Σ_0 e Σ_1 e branco (espaço) não pertença a Σ_1, i.e., $\{\&,\ \#\} \cap \Sigma_0 = \emptyset$ e $\{\&,\ \#,\ \sqcup\} \cap \Sigma_1 = \emptyset$, com $x \in \Sigma_0^*$, $y, z, t \in \Sigma_1^*$ e $x, y, z, t \neq \varepsilon$ (cadeia vazia). O valor z, como na linguagem (1), é o valor da resposta correta do problema indicado por x, naquele instante t. As instâncias *sim* de L correspondem aos casos em que se tem $y = z$, sendo z considerado um certificado sucinto para o problema, portanto lendo-se apenas w, desacompanhado do respectivo certificado, não se pode garantir a sua pertinência à linguagem L. Isto torna $L \notin P$, como já sabemos das duas seções anteriores: não é possível construir uma MTD que acerte sempre, em todas as vezes, as suas previsões sobre determinado evento probabilístico. Trata-se, portanto, de um problema indecidível (também chamado de insolúvel). Veja, por exemplo, [7].

Para provar $L \in NP$ vamos usar uma relação binária de verificação (*checking relation*), $R(w, z)$, com $w = x\&y\&t$, $w \in \Sigma^*$, $\Sigma = \Sigma_0 \cup \{\&\}$, $\{\#\} \cap \Sigma = \emptyset$, supondo que o certificado z é sucinto, i.e., tem comprimento limitado por um polinômio p na variável $|w|$, o comprimento

do string de entrada, tal que $|z| \leq p(|w|)$. A relação R será verdadeira se e somente se $y = z$, i.e.,

$$R(x\&y\&t, z) \Leftrightarrow y = z.$$

Associemos à relação $R \subseteq \Sigma^* \times \Sigma_1^*$ a linguagem L_R sobre $\Sigma \cup \Sigma_1 \cup \{\#\}$ definida por

$$L_R = \{w\#z \mid R(w, z)\},$$

com $w = x\&y\&t$ e onde, conforme nossa definição, o símbolo # não está em Σ, nem em Σ_1. Este caractere # será o separador entre o string w e o certificado z, e definamos $w_R = w\#z$.

Embora não exista uma MTD (nem real, nem abstrata) que decida L no caso geral, para qualquer instante, é possível construir (ao menos teoricamente) uma MTD M capaz de decidir (verificar) L_R em tempo polinomial, comparando os valores de y e z. Claro que a leitura de w_R e subsequente comparação de y com z podem ser feitas em tempo polinomial, bem como a respectiva informação da resposta (*sim* ou *não*). Quando $y = z$ então temos que w_R é uma instância *sim* do problema que M verifica, e por isso $w_R \in L_R$ e $w \in L$, caso contrário w_R é uma instância *não* deste problema, e temos $w_R \notin L_R$ e $w \notin L$. Sendo assim, ao contrário de L, que é indecidível, a linguagem L_R é decidível (verificável) em tempo polinomial.

Seguindo [13], conforme já vimos na seção 2, a linguagem L sobre Σ está em NP se e somente se há $k \in \mathbb{N}$ e uma relação R de checagem em tempo polinomial tal que, para todo $w \in \Sigma^*$,

$$w \in L \Leftrightarrow \exists z \left(|z| \leq |w|^k \text{ e } R(w, z) \right),$$

onde $|w|$ e $|z|$ denotam os comprimentos de w e z, respectivamente. Estas condições são verificadas para todos os certificados z cujo comprimento é igual ao comprimento do palpite y da máquina, quando este palpite é correto, pois se a relação R é verdadeira então $y = z$ e assim $|z| = |y| \leq |w|$, donde podemos escolher $k = 1$. A mencionada relação $R(w, z)$ é a relação $R(x\&y\&t, z)$ que definimos anteriormente, para $w = x\&y\&t$. Portanto, obedecidas estas condições, nós temos $L \in NP$.

Como $L \notin P$, mas $L \in NP$, então $P \neq NP$.

6. O PROBLEMA DA PARADA

Pode-se criticar que as linguagens com semântica e as linguagens variáveis no tempo não fazem parte do "verdadeiro" problema P *vs* NP, que este se relaciona apenas com as linguagens constantes e puras, ou seja,

$$P = P_{pura}$$

e

$$NP = NP_{pura},$$

sem variação alguma no tempo, e que mesmo as linguagens (1), (2) e (3) aqui definidas podem ser consideradas como reconhecíveis em tempo polinomial por uma MTD.

Seria compreensível pensar assim nos eventos de uma única ocorrência, a exemplo de um único lançamento de dado nos valores possíveis de 1 a 6. Com 6 MTD, cada uma gerando um palpite diferente, necessariamente uma delas acertaria (responderia *sim* à entrada). Mas se fossem dois lançamentos em sequência, precisaríamos de 36 MTD para acertar ambos, cada uma gerando um dos pares possíveis de lançamentos, e para uma quantidade n genérica precisaríamos de 6^n MTD para garantirmos o acerto de uma delas em todas as jogadas. Para um número arbitrariamente grande de lançamentos ($n \to \infty$) deveríamos ter um correspondente número arbitrariamente grande de MTD, e o tamanho dessas máquinas aumentaria com o aumento de n. Isso, claro, porque não podemos mudar constantemente de máquina, já que, para analisarmos uma pertinência a P, a cada linguagem devemos associar suas respectivas MTD, definitivas, ao menos uma por linguagem. *A priori*, num tempo finito, com um número finito de MTD, sempre precisaríamos de novas MTD que codificassem todo o histórico de lançamentos já realizados e que também estivessem preparadas para os próximos lançamentos ainda a serem feitos. A capacidade de acertar o passado quando nos baseamos em um conjunto finito conhecido é um problema trivial, enquanto é impossível prever o futuro ilimitadamente. Nota-se que este é um exemplo de linguagem variável, que pode mudar com o tempo, por isso não existem apenas linguagens constantes. Podemos criar um problema onde seja natural este conceito de variabilidade, que foi, aliás, o que fizemos aqui. Alternativamente, ao invés de uma linguagem variável no tempo com um único elemento por ocorrência (tempo t), podemos ter uma linguagem que aumenta seu tamanho (cardinalidade) com o tempo, com o aumento do número de ocorrências (os elementos da linguagem), por exemplo,

$$L = \{r_1, r_2, r_3, \dots, r_n\},$$

onde r_k é o k-ésimo resultado do lançamento de um dado, mas mesmo assim esta linguagem também pode ser considerada como variável no tempo, pois ela descreve uma sequência de valores obtidos ao longo do tempo e com possibilidade de crescer ainda mais, indefinidamente. Precisaremos manter, para todo t, as mesmas MTD e MTND. Por brevidade, omitimos os valores x, y, t que definimos anteriormente, deixando apenas os valores correspondentes a z, o resultado correto, tal qual também fizemos na seção 3 com os elementos de DADOS.

Quanto às linguagens com semântica, são em quantidade bem maior do que poderíamos pensar inicialmente. Não há dificuldade em considerar todas as linguagens mais conhecidas, como SAT e todos os problemas $NP-completos$ [3], bem como todos os famosos problemas indecidíveis, como sendo linguagens com semântica. O uso da semântica, ou significado, dá uma forma de definir com mais detalhes e precisão determinada linguagem, numa gramática que nos façamos entender. Uma linguagem pura, por outro lado, pode ser considerada como sendo descrita de forma mais concisa, mesmo que também tenha uma clara regra matemática de formação, como se faz no estudo introdutório dos autômatos finitos. Tal separação conceitual, entretanto, pode não trazer maiores contribuições à Teoria da Computação, e poderemos evitá-la quando possível.

A fim de evitarmos as críticas com relação aos dois novos conceitos vamos a seguir abandonar o uso tanto das linguagens variáveis no tempo quanto das linguagens com semântica, e adotar outra linha de raciocínio. Utilizaremos uma linguagem baseada em um problema indecidível, mas que com a introdução de duas condições transforma-se em decidível. Embora um problema decidível, não pode ser resolvido em tempo polinomial por uma MTD, apenas por uma MTND, conforme mostraremos.

O Problema da Parada (PP) pergunta se, dado um programa (ou máquina de Turing) M e uma cadeia de entrada w, M para em resposta à entrada w, i.e., não entra em loop infinito. Este é o mais famoso e fundamental problema indecidível [7]. Em [7] se prova que a linguagem

$$H = \{ "M" "w" \mid \text{a máquina de Turing } M \text{ para em resposta à cadeia de entrada } w\}$$

não é recursiva, i.e, não há uma máquina de Turing que a decida, e assim ela não é solúvel por algoritmos (de acordo com a tese de Church-Turing [5, 7]), portanto H refere-se a um problema insolúvel ou indecidível. A prova se faz por redução ao absurdo, chegando-se a uma contradição. Isto não significa, como se observa, que não possa haver algumas circunstâncias em que seja possível prever se uma máquina de Turing irá parar em resposta a uma cadeia de entrada. Mesmo que um problema seja indecidível, uma subclasse menos geral do problema pode ser decidível [1]. A indecidibilidade do PP foi provada primeiramente por Alan Turing (1912 – 1954), em 1936 [16].

Vejamos. Se H for uma linguagem decidível então deve haver uma máquina de Turing M_H que a decida, i.e., que possa sempre fornecer uma resposta correta *sim* ou *não* para uma dada entrada. Essa máquina deve ser capaz de ler qualquer máquina de Turing M seguida por um string w que seja entrada para M e decidir se M para ou não em resposta ao string w.

Suponhamos então que exista essa máquina M_H, que leia alguma outra máquina M e em sequência um string w, que serve como entrada para M. A máquina M_H, ela própria, deve ser capaz de ler M, w, e daí decidir (informando corretamente *sim* ou *não*) se M pararia ou não em decorrência do processamento do string w.

Suponhamos agora que a codificação da máquina M seja equivalente ao seguinte algoritmo, que tem o string w como parâmetro de entrada:

```
procedure M (string: w)
{
    A:
        se M_H(M,w) então    // M_H decide que M para com a entrada w
                vá para A
        senão                // M_H decide que M não para com a entrada w
                {rejeite;    // informa não
                 pare;
                }
    B:
        aceite;              // informa sim
        pare;
}
```

Nossa máquina M_H deveria ser capaz de decidir sobre a parada de qualquer máquina de Turing, mas no caso de ler M, e para qualquer que seja inicialmente o parâmetro w, a que conclusão ela chegaria? Pela codificação acima do algoritmo equivalente a M se vê que quando $M_H(M, w)$ for verdadeiro, i.e., M_H decidir que M para com a entrada w, então M entra em loop infinito, sendo permanentemente levada ao *label* A, ou seja, M não para com a entrada w. Pelo contrário, se $M_H(M, w)$ for falso, i.e., M_H decidir que M não para com a entrada w, então M decide pelo *não* e para com a entrada w. Chegamos assim a um absurdo, uma contradição, ou seja,

$$(M \text{ para}) \Leftrightarrow \sim(M \text{ para}),$$

portanto nossa premissa inicial é inválida, i.e., M_H não decide a linguagem H. Assim, a linguagem H é indecidível, insolúvel.

A demonstração anterior continua válida se M e w mantém uma dependência entre si ou se $|w|$ é limitado por algum valor ou se w é vazio (ε), pois ela é válida para qualquer w. Por outro lado, a linguagem $L_{M,k}$ a seguir, para $k \in \mathbb{N}$, não é indecidível,

(4) $L_{M,k} = \{\langle M\rangle \mid$ há ao menos um string $w \in \Sigma_1^*$, com $\langle M\rangle \in \Sigma^*$ e $|w| \leq |\langle M\rangle|^k$, tal que a MTD M para em resposta à cadeia de entrada w em até $|w|^k + k$ passos$\}$,

mas temos $L_{M,k} \in NP$ e $L_{M,k} \notin P$, o que nos leva a $P \neq NP$. Chamaremos $L_{M,k}$ de Problema da Parada em Tempo Polinomial de ordem k (PPTP$_k$). Nossa notação para a codificação de um objeto O como uma cadeia é $\langle O\rangle$. Se tivermos vários objetos $O_1, O_2, \ldots, O_k$, denotamos sua codificação em uma única cadeia como $\langle O_1, O_2, \ldots, O_k\rangle$ [5]. Se k é um número inteiro positivo escrito na base 10 então o comprimento da cadeia $\langle k\rangle$, $|\langle k\rangle|$, é igual a $1 + \lfloor \log_{10} k \rfloor$.

Seguindo [13], nós definiremos que uma MTD M roda em tempo polinomial se existe k tal que, para todos os valores de $|w|$,

(5) $T_M(|w|) \leq |w|^k + k,$

onde $T_M(|w|) = \max\{t_M(w) \mid w \in \Sigma^n\}$, $t_M(w)$ é o número de passos na computação de M sobre a entrada $w \in \Sigma^*$ e Σ^n é o conjunto de todos os strings sobre Σ de comprimento n, $n \in \mathbb{N}$, com $n = |w|$. Ou seja, para todas as (infinitas) entradas finitas possíveis o número de passos de M é limitado pelo valor dado em (5), para algum $k \in \mathbb{N}$ fixo, característico de M.

Analogamente, definiremos que uma MTD M para em tempo polinomial de ordem k em resposta à cadeia de entrada w se M para em até $|w|^k + k$ passos sobre a entrada w, i.e., se nós temos um valor limitante $t_M max$ tal que

(6) $t_M(w) \leq |w|^k + k = t_M max,$

justificando assim o termo "tempo polinomial de ordem k" que usamos no nome dado à linguagem $L_{M,k}$.

Se são dados M e k então podemos calcular o limitante superior n para o comprimento de w: $n = |\langle M\rangle|^k$. Assim, podemos construir não deterministicamente e testar cada possível entrada $w \in \Sigma_1^*$, com $|w| \leq n$, para verificar (em tempo polinomial) se a parada de M se dá em até $t_M max = |w|^k + k$ passos. Isso faz com que $L_{M,k} \in NP$.

O algoritmo não determinístico principal para a decisão do PPTP$_k$ pode ser assim (baseado na linguagem C), onde $1..n$ representa o conjunto dos números naturais de 1 até n e Σ_1 é o alfabeto de entrada para M (consideramos $\langle M\rangle \in \Sigma^*$ e $w \in \Sigma_1^*$):

```
procedure L_{M,k} (string: M; integer: k)
{
      w = ´´; // w[0] = ´\0´
      calcula(m, n); // m = |<M>|, n = |<M>|^k = m^k
      tam = ESCOLHE(1..n); // |w|
      for  (i = 1; i ≤ tam; i++)
            {m = ESCOLHE(Σ_1);
             concatena(w, m);
            }
      calcula(t_Mmax); // n + k
      calcula(t_Umax); // (m + 1 + tam)^k + k; a MTD U simula a MTD M
      if (para(M, w, t_Mmax, t_Umax)) then
         aceita();
      else
         rejeita();
}
```

Para escrevermos o algoritmo determinístico que retorna se M para ou não em resposta a w em até $t_M max$ passos será preciso antes esclarecer com mais detalhes sobre como descrever e simular uma MT.

Considerando que uma MT é uma 7-upla $(Q, \Sigma, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$ [5], onde Q, Σ, Γ são todos conjuntos finitos e

1. Q é o conjunto de estados,

2. Σ é o alfabeto de entrada, que não contém o símbolo em branco ␣,

3. Γ é o alfabeto de fita, onde ␣ ∈ Γ e Σ ⊆ Γ,

4. $\delta: (Q - \{q_{aceita}, q_{rejeita}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{E, D\}$ é a função de transição, onde E representa o movimento à esquerda do cabeçote da fita e D o movimento à direita,

5. $q_0 \in Q$ é o estado inicial,

6. $q_{aceita} \in Q$ é o estado de aceitação, e

7. $q_{rejeita} \in Q$ é o estado de rejeição, com $q_{aceita} \neq q_{rejeita}$,

é possível simular a execução de uma MT M qualquer sobre uma entrada w qualquer. Para cada estado em que M se encontra, símbolo que lê, símbolo que grava, se a cabeça da fita de leitura e gravação vai para a direita ou a esquerda e o novo estado para o qual M vai, atualizamos o estado q da máquina, bem como o conteúdo da fita e a posição da cabeça de leitura e gravação, gerando assim uma nova configuração da máquina a cada passo, e somamos 1 ao contador de passos, contador este que deve ser somado até ser possível verificar com certeza a relação (6) sobre o tempo polinomial da máquina com relação à entrada w. Uma MT para ao entrar nos estados de aceitação ou rejeição, seus estados finais. Se nenhum desses ocorre, M continua para sempre [5].

Cada MT pode reconhecer uma única linguagem [5], i.e., é especializada em resolver um único problema [14], mas há um tipo especial de MT, chamada de máquina de Turing universal (MTU), capaz de simular qualquer outra MT a partir da descrição da mesma [5, 7,

14]. É este tipo especial de MT que precisamos para decidir $L_{M,k}$, pois ela deve simular a MT M sobre a entrada w e ser capaz de controlar seu próprio momento de parada. Mesmo quando M sobre a entrada w não para, esta máquina universal (U) deve parar, comparando o contador de passos de M, $t_M(w)$, com o valor máximo possível, $t_M max$, dado em (6). Se chegarmos a $t_M(w) > |w|^k + k$, nossa máquina U deverá cancelar a simulação do processamento de M e parar, independentemente da parada de M sobre a entrada w, e decidir pelo *não*, rejeitando a entrada. A MTD M se reduzirá a um conjunto de *arrays* descrevendo a função de transição δ.

Um algoritmo para U, que terá associado a ele seu próprio contador de passos, t_U, além do contador de passos de M, t_M, com $0 \leq t_M < t_U$, está descrito mais adiante (baseado na linguagem C). A fim de não prejudicar a sua legibilidade e entendimento, vamos omitir as sucessivas atualizações de t_U e t_M, como t_U++ e t_M++, mas entendamos que elas se inserem antes de cada instrução ou bloco de instruções relacionadas a U e a M, respectivamente, e logo a seguir (para as que fazem parte dos *loops* de *while*) devemos testar a parada da máquina correspondente pelo motivo de excesso de tempo. Em destaque com fundo amarelo estão todas as instruções que devem ser precedidas por uma sub-rotina cuja função é somar 1 a t_M e rejeitar a entrada caso $t_M > t_M max$, por exemplo, somar_testar_t_M(). Uma outra sub-rotina, por exemplo, somar_testar_t_U(), deve ter a mesma função com respeito ao controle de t_U, rejeitando a entrada caso $t_U > t_U max$, mas omitimos sua escrita porque ficará subentendido que ela deve ser inserida em praticamente todas as linhas do algoritmo. Consideramos $t_U\, max = (|\langle M, w\rangle|)^k + k$, mas também é possível permitir um limite de tempo maior, por exemplo, multiplicando k por uma constante: $t_U\, max = (|\langle M, w\rangle|)^{7k} + 7k$. O necessário é que U tenha um algoritmo em tempo polinomial.

Chamaremos as componentes da função de transição δ pelos seguintes nomes:

q_atual[pointer_δ],
w_lido[pointer_δ],
novo_q[pointer_δ],
gravar_w[pointer_δ],
sentido[pointer_δ],

onde 0 ≤ pointer_δ < qtd_δ e qtd_δ é a quantidade de elementos, ou número de ocorrências de cada componente, da função de transição δ.

Para t_M, vamos convencionar que o processamento em sequência de ao menos uma das cinco componentes de δ implicará em um único passo de M.

Deve ficar claro que uma instrução escrita para algoritmo ou linguagem de alto nível (C, Java, Pascal, Fortran, Matlab, Mathematica, Maple, R, etc.) geralmente corresponde a mais de uma instrução para MT. A propriedade do algoritmo de ser polinomial ou não se mantém entre as diversas formas, embora o grau do polinômio possa aumentar bastante numa MT. Segundo [10], obedecendo-se a determinadas hipóteses, n etapas de um computador podem ser simuladas por uma MT de uma fita em no máximo $O(n^6)$ etapas da MT. O algoritmo a seguir, portanto, dá apenas uma primeira ideia sobre o que deve ser codificado em uma MT para simular outra MT.

```
function integer para (string: M, w; integer: tMmax, tUmax)
{
// aqui simulamos uma máquina de Turing determinística
// w obedece necessariamente |w| ≤ |<M>|^k
tU = 1; // atualiza t antes de computar a respectiva instrução
tM = 0;
qtd_δ = 0;
tabelar_δ(qtd_δ); // lê e salva M do início ao fim
q = q0; // inicializar_estado();
pointer_w = 0;
pointer_δ = 0;
fim = 0;
while (fim == 0) // esta função é de U
        {σ = w[pointer_w];
         achou_δ = 0;
         while (achou_δ == 0) // busca função de transição δ
                {if (pointer_δ ≥ qtd_δ)
                    pointer_δ = 0;
                 if (q_atual[pointer_δ] == q) // estado atual
                 if (w_lido[pointer_δ] == σ) // símbolo sob a cabeça
                  {q = novo_q[pointer_δ];
                   w[pointer_w] = gravar_w[pointer_δ];
                   if (sentido[pointer_δ] == ´E´)
                      {if (pointer_w > 0)
                           pointer_w−−; // move o cabeçote para a esquerda
                      }
                   else
                       if (sentido[pointer_δ] == ´D´)
                          pointer_w++; // move o cabeçote para a direita
                   achou_δ = 1;
                   if (q == qaceita || q == qrejeita) // 1 ou 2 passos, M decide sobre sua parada
                      {aceita(); // p = 1
                       fim = 1;
                       break; // while (achou_δ == 0)
                      }
                  } // if (q_atual[pointer_δ] == q && w_lido[pointer_δ] == σ)
                if (tM > tMmax) // esta função é de U
                   {rejeita(); // p = 0
                    fim = 1;
                    break; // while (achou_δ == 0)
                   }
                pointer_δ++;
               } // while (achou_δ == 0)
        } // while (fim == 0)
return p;
} // function integer para (string: M, w; integer: tMmax, tUmax)
```

Conforme [1], um passo da máquina (de Turing) consiste das seguintes operações:

a) ler o símbolo σ sob a cabeça;

b) escrever o símbolo σ' no lugar de σ;

c) reposicionar a cabeça, que pode ser movida uma célula à direita ou à esquerda, ou deixada no mesmo lugar (também conforme definição alternativa em [5], a cabeça pode não se mover, ficando parada, entre um estado e outro);

d) mudar o controle central para um novo estado q';

mas nós aqui percebemos a necessidade de também contar como passo a busca da função de transição (estado atual e símbolo sob a cabeça), pois se a MT não encontrar quais as componentes da função δ que devem ser utilizadas num determinado momento, dentre todas as opções disponíveis (lidas), ela entrará em *loop* infinito. Usar corretamente tal controle pode fazer com que escapemos de *loops* indesejáveis e que obviamente consomem tempo.

Dos algoritmos anteriores e definição de $L_{M,k}$ vemos que $L_{M,k} \in NP$, pois tanto o comprimento das cadeias de entrada w quanto o tempo de execução de M sobre w e de U sobre M e w estão rigidamente limitados por funções polinomiais, e assim o processamento do algoritmo não determinístico que descrevemos para decidir $L_{M,k}$ é de ordem polinomial.

Se definirmos uma nova linguagem derivada de $L_{M,k}$ e que contenha, por definição, um certificado sucinto para $L_{M,k}$, verificável em tempo polinomial por uma MTD, provaremos mais uma vez que $L_{M,k} \in NP$, desta vez utilizando uma relação de checagem ou verificação.

Vamos a seguir dar uma prova mais rigorosa do que afirmamos no último parágrafo.

Seja a linguagem $R_{M,k}$ a seguir, com $k \in \mathbb{N}$, tal que

(7) $R_{M,k} = \{x\#y \mid x = \langle M \rangle$ é uma instância *sim* para o PPTP$_k$ $L_{M,k}$ e $y = w$ é um certificado sucinto para a respectiva instância de $L_{M,k}$, sendo $x \in \Sigma^*$, $y \in \Sigma_1^*$, $\# \notin \Sigma$ e $|y| \leq |x|^k\}$.

Uma MTD que decida $R_{M,k}$ deve verificar $L_{M,k}$ em tempo polinomial em relação ao tamanho da entrada para $R_{M,k}$, $|x\#y|$, ou seja, devemos ter $R_{M,k} \in P$. Pode-se também dizer que essa MTD deve verificar $L_{M,k}$ em tempo polinomial em relação ao tamanho da instância $x = \langle M \rangle$ [5].

Seguindo Cook [13], tal como fizemos na seção 5, uma linguagem L sobre Σ está em NP se e somente se há $k \in \mathbb{N}$ e uma relação R de checagem em tempo polinomial tal que, para todo $w \in \Sigma^*$,

$$w \in L \Leftrightarrow \exists y \left(|y| \leq |w|^k \text{ e } R(w, y) \right).$$

À relação binária $R \subseteq \Sigma^* \times \Sigma_1^*$ para algum alfabeto finito Σ e Σ_1 deve estar associada uma linguagem L_R sobre $\Sigma \cup \Sigma_1 \cup \{\#\}$ definida por

$$L_R = \{w\#y \mid R(w, y)\},$$

onde o símbolo # não está em Σ. A relação R é de checagem de tempo polinomial se e somente se $L_R \in P$.

Traduzindo a nomenclatura geral de Cook para a que usamos especificamente nas linguagens $L_{M,k}$ (4) e $R_{M,k}$ (7), Cook chamou de w o que nós chamamos de x e para nós é a representação em string da MTD M, i.e., $x = \langle M \rangle$, e Cook e nós chamamos de y o que para

nós é a cadeia de entrada w que M lerá, i.e., $y = w$, e é o nosso certificado sucinto. Os alfabetos Σ e Σ_1 e o uso do caractere separador #, com $\# \notin \Sigma$, estão conforme se usa em Cook.

O significado e valor de k definidos por Cook também valem para nós, e usamos explicitamente os possíveis valores de $k \in \mathbb{N}$ para nomear cada linguagem: $L_{M,1}$, $R_{M,1}$, $L_{M,2}$, $R_{M,2}$, $L_{M,3}$, $R_{M,3}$, etc., o que facilita estabelecer tanto o comprimento máximo para w quanto o tempo máximo de execução de M, ambos de ordem polinomial.

A linguagem L_R definida por Cook é equivalente à nossa linguagem $R_{M,k}$ e a relação binária de checagem $R(w, y)$ de Cook é a propriedade utilizada na definição de $R_{M,k}$, que necessariamente implica na verificação de $L_{M,k}$ em tempo polinomial, caso contrário $x\#y$ não será elemento da linguagem $R_{M,k}$:

(8) $R(x, y) \Leftrightarrow x = \langle M \rangle$ é uma instância *sim* para o PPTP$_k$ $L_{M,k}$ e $y = w$ é um certificado sucinto para a respectiva instância de $L_{M,k}$, sendo $x \in \Sigma^*$, $y \in \Sigma_1^*$, $\# \notin \Sigma$ e $|y| \leq |x|^k$.

Conforme já vimos anteriormente, o uso dos contadores t_U e t_M garantirá o processamento de U e simulação de M em tempo polinomial, e deve-se rejeitar a entrada e parar a simulação caso ultrapassem o limite de tempo $t_U max$ e $t_M max$, respectivamente. Quando a decisão sobre a parada de M é obtida, a parada de U ocorre poucos passos depois, com $t_M < t_U$.

Substituindo por $L_{M,k}$ a linguagem L que define uma pertinência a NP em Cook [13], temos então que $L_{M,k}$ sobre Σ está em NP se e somente se há $k \in \mathbb{N}$ e uma relação de checagem em tempo polinomial $R(x, y)$ tal que, para todo $x \in \Sigma^*$,

$$x \in L_{M,k} \Leftrightarrow \exists y \left(|y| \leq |x|^k \text{ e } R(x, y) \right),$$

onde $R(x, y)$ é a propriedade presente na linguagem $R_{M,k} \in P$,

$$R_{M,k} = \{x\#y \mid R(x, y)\},$$

e dada pela relação binária de equivalência (8).

Todas as condições necessárias para a pertinência a NP de $L_{M,k}$ sobre Σ, linguagem (4), estão satisfeitas.

Sendo

$L_{M,k} = \{\langle M \rangle \mid$ há ao menos um string $w \in \Sigma_1^*$, com $\langle M \rangle \in \Sigma^*$ e $|w| \leq |\langle M \rangle|^k$, tal que a MTD M para em resposta à cadeia de entrada w em até $|w|^k + k$ passos$\}$:

a) existem infinitos $k \in \mathbb{N}$ distintos, um por linguagem, e cada um deles é referido explicitamente na definição de $L_{M,k}$;

b) existe a relação de checagem $R(x, y)$, dada em (8), e ela é em tempo polinomial porque a linguagem $R_{M,k}$, dada em (7), que a tem por propriedade, pertence a P. $R_{M,k} \in P$ porque através de uma MTU, de $x = \langle M \rangle$ e do certificado $y = w$, com $|y| \leq |x|^k$, pode-se simular em tempo polinomial a MTD M sobre a entrada w e verificar se ela para em até $t_M max = |w|^k + k$ passos, como visto na descrição do algoritmo *para* anterior, o que em caso positivo implica em $x \in L_{M,k}$;

c) se $x \in L_{M,k}$ então necessariamente existe uma entrada $y = w$ que quando lida pela MTD M, com $x = \langle M \rangle$ e $|w| \leq |\langle M \rangle|^k$, i.e., $|y| \leq |x|^k$, faz M parar em resposta a w em até $t_M max = |w|^k + k$ passos, satisfazendo a relação $R(x, y)$;

d) reciprocamente, se $R(x, y)$ é verdadeira e $|y| \leq |x|^k$ então existe a cadeia de entrada $y = w$, assim como a MTD M, com $x = \langle M \rangle$, tal que x é uma instância *sim* para o PPTP$_k$ $L_{M,k}$, o que implica em $x \in L_{M,k}$;

e) de c) e d) vemos que, para todo $x \in \Sigma^*$,

$$x \in L_{M,k} \Leftrightarrow \exists y \left(|y| \leq |x|^k \text{ e } R(x, y) \right).$$

De a) a e) temos então satisfeitas todas as condições necessárias para a pertinência de $L_{M,k}$ a NP, donde $L_{M,k} \in NP$.

Vamos agora provar que $L_{M,k} \notin P$.

A linguagem

(9) $L_M = \{\langle M \rangle \mid$ há ao menos um string $w \in \Sigma_1^*$, com $\langle M \rangle \in \Sigma^*$, tal que a MTD M para em resposta à cadeia de entrada $w\}$,

é indecidível [7], o que pode ser provado, dentre outras maneiras, usando o teorema de Rice [5, 7, 10, 14, 17].

O que difere fundamentalmente a linguagem indecidível L_M (9) da linguagem decidível $L_{M,k}$ (4), além do número natural k, é a introdução de duas condições limitadoras presentes em $L_{M,k}$:

c_1: $|w| \leq |\langle M \rangle|^k$

e

c_2: a MTD M para em até $|w|^k + k$ passos,

para $k \in \mathbb{N}$.

Estas condições são as responsáveis pela transição indecidível – decidível do problema, pois com elas podemos, pelo método da força bruta (busca exaustiva ou tentativa e erro), construir possíveis soluções para w, com seu comprimento máximo obedecendo c_1, e verificar se M sobre a entrada w para em até $t_M max$ passos, conforme c_2.

Sem c_1 e c_2 a linguagem $L_{M,k}$ se transforma em indecidível, e portanto não há nenhum algoritmo que decida $L_{M,k}$, seja em tempo polinomial, exponencial ou qualquer outro, e com c_1 e c_2 a linguagem L_M se transforma em decidível.

O método de busca de soluções por tentativa e subsequente teste (verificação) é eminentemente um método combinatório, uma busca exaustiva para encontrar alguma solução, o que não é feito em tempo polinomial por um algoritmo determinístico. Se $w \in \Sigma_1^n \subset \Sigma_1^*$ então nós temos $(\#(\Sigma_1))^n$ combinações possíveis para o valor de w, onde $\#(\Sigma_1)$ representa o número de elementos do conjunto Σ_1, portanto um algoritmo determinístico que possa ser capaz de gerar e testar todas estas possibilidades é de ordem exponencial, não polinomial, em relação a $n = |w|$.

Como, por c_1, $|w| \leq |\langle M \rangle|^k$, no pior caso nós temos que este algoritmo também é de ordem exponencial em relação a $|\langle M \rangle|$, o tamanho da instância de $L_{M,k}$, e não de ordem polinomial, então, se não houver algum outro algoritmo determinístico mais eficiente para decidir $L_{M,k}$, nós temos $L_{M,k} \notin P$.

Haverá algum outro algoritmo determinístico mais eficiente para decidir $L_{M,k}$, de modo que tenhamos $L_{M,k} \in P$? Se sim, provavelmente é um algoritmo que não precisará "chutar", "tentar" ou "adivinhar" uma solução para em seguida testar (verificar) se ela (a cadeia de entrada w) corresponde de fato a uma solução que faz M parar em até $t_M max$ passos, com $|w| \leq |\langle M \rangle|^k$.

Suponhamos que $L_{M,k} \in P$ e que $M_{H,k}$ seja uma MTD que decide a linguagem $L_{M,k}$ em tempo polinomial. Lido um elemento $x = \langle M \rangle$, $M_{H,k}$ precisa decidir acertadamente se $x \in L_{M,k}$, i.e., se existe $w \in \Sigma_1^*$, com $|w| \leq |\langle M \rangle|^k$, tal que M sobre a entrada w para em até $|w|^k + k$ passos. Além disso, esta decisão precisa ser feita em tempo polinomial, por exemplo, em até $|\langle M \rangle|^k + k$ passos (seguindo a convenção (5) de Cook [13]).

Se é dado w previamente nosso problema pode ser resolvido rapidamente, e por isso $L_{M,k} \in NP$, como já vimos, mas se não é dado w temos um problema muito mais difícil.

Suponhamos que M corresponda a um algoritmo que leve a um número exponencial de diferentes processamentos, variando em função da cadeia w de entrada e de seu comprimento $|w|$: parte destes processamentos não para, parte para em tempo exponencial, parte para em tempo polinomial de ordem k e ainda uma parte para em tempo polinomial de ordem maior que k, em relação ao comprimento $|w|$. Não há nenhum critério bem estabelecido para localizar no algoritmo alguma ordenação e classificação destes diferentes tipos de processamentos, como número de passos e comentários.

Em virtude do caráter imprevisível da construção deste algoritmo e da quantidade exponencial dos vários processamentos possíveis, será preciso, no caso geral, que $M_{H,k}$ construa deterministicamente os diferentes valores válidos para w e teste o respectivo tempo de parada (ou número de passos) de M em função do comprimento $|w|$, simulando a execução de M sobre w, a fim de encontrar algum w que garanta a pertinência de $\langle M \rangle$ a $L_{M,k}$. Mas construir deterministicamente todas as possíveis entradas $w \in \Sigma_1^*$ para M se faz em tempo exponencial em relação a $|w|$, e não em tempo polinomial, e assim, no pior caso, como $|w| \leq |\langle M \rangle|^k$, esta construção se faz em tempo exponencial em relação ao comprimento de $\langle M \rangle$, donde a MTD $M_{H,k}$ não decide $L_{M,k}$ em tempo polinomial em relação ao tamanho da instância $\langle M \rangle$ e portanto $L_{M,k} \notin P$.

Como $L_{M,k} \in NP$ e $L_{M,k} \notin P$ então $P \neq NP$.

Vale mencionar que a insolubilidade do PP e de suas conhecidas variações nos garante que não há uma maneira sistemática para que uma MTD leia uma MTD M e decida se M para ou não ao ler uma dada cadeia de entrada w, ou se há alguma cadeia de entrada w que faça M parar quando w é lida. O teorema de Rice [5, 7, 10, 14, 17] é ainda mais forte, pois implica que apenas propriedades triviais presentes nos algoritmos podem ser algoritmicamente decidíveis (uma propriedade é dita trivial quando todas as linguagens a possuem ou nenhuma linguagem a possui). Conforme [5], testar qualquer propriedade das linguagens reconhecidas por MT é indecidível. Se P é qualquer propriedade não trivial da linguagem de uma MT, o problema de determinar se a linguagem de uma dada MT tem a propriedade P é indecidível.

Conforme [7, 14], se C é um subconjunto próprio, não vazio, do conjunto de todas as linguagens recursivamente enumeráveis, o seguinte problema é indecidível: dada uma MT M, temos $L(M) \in C$? Conforme [10], toda propriedade não trivial das linguagens recursivamente enumeráveis é indecidível, no sentido de que é impossível reconhecer por uma MT os strings (binários) que são códigos para uma MT cuja linguagem tem a propriedade. Note que a referência [10] identifica uma MT como códigos binários, mas muitos outros autores não fazem isso. Uma linguagem é dita recursivamente enumerável quando pode ser aceita por uma MT [10, 14]. De acordo com [7], uma linguagem L é recursivamente enumerável se e somente se existir uma MT M que semidecide L. Dizemos que M semidecide L se, para qualquer cadeia de entrada $w \in \Sigma_0^*$, $w \in L$ se e somente se M para em resposta à entrada w, onde $\Sigma_0 \subseteq \Sigma - \{\sqcup, \rhd\}$, Σ_0 é um alfabeto, Σ é o alfabeto de entrada, $\sqcup$ é o símbolo em branco e $\rhd$ é um símbolo especial fixo marcado na posição mais à esquerda da fita.

7. CONCLUSÃO

Resolvemos o problema P *versus* NP provando $P \neq NP$. Não utilizamos nenhum problema $NP-completo$ neste artigo, o que provavelmente despertará menor atenção ao fato, até mesmo um completo desinteresse, infelizmente. A questão original de Cook [8], se $\{tautologies\}$ pertence a P ou não, assim como SAT e demais problemas difíceis, continua sem solução. Mas isso não invalida nossa resposta ao assunto: é mais fácil verificar uma solução do que encontrá-la, no sentido de ser $P \neq NP$.

Vimos algumas maneiras equivalentes de definir P, NP e entender o não determinismo. A maneira mais realística de definir a classe NP, paradoxalmente, não envolve o uso do não determinismo, mas sim a de relação de verificação ou checagem em máquinas determinísticas, como está dado em Cook [13] e Papadimitriou *et al* [7, 12, 14], e usamos esta forma de relação na seção 5, de maneira simples e bastante plausível, para provar $P \neq NP$. Agora que esta solução mais simples foi encontrada, vê-se que chegamos a uma conclusão verdadeiramente óbvia, em acordo com nossa intuição. Outras demonstrações foram dadas nas seções 3 e 4, usando ideias envolvendo as máquinas não determinísticas, e na seção 6, com o uso de uma máquina de Turing universal decidindo uma linguagem baseadas no Problema da Parada.

A maior e mais difícil questão, entretanto, é saber se $SAT \in P$ ou não. O teorema de Cook [8] e seu corolário [2] aparentam ser bastante abrangentes, mas em princípio não levam em consideração nem as linguagens variáveis no tempo, nem as linguagens deriváveis de linguagens indecidíveis, o que nos tira algo da certeza de suas respectivas validades. Espero responder a este problema na próxima versão (IV) deste artigo.

18 de agosto de 2019.
03 de setembro de 2019.

References

[1] C.I. Lucchesi, I. Simon, I. Simon, J. Simon and T. Kowaltowski, *Aspectos teóricos da computação*. Rio de Janeiro: IMPA (1979), pp. 48, 59, 67, 106, 110.

[2] R.M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, eds. R.E. Miller and J.W. Thatcher. New York: Plenum Press (1972), pp. 85–103.

[3] M.R. Garey and D.V. Johnson, *Computers and Intractability – A Guide to the Theory of NP-completeness.* New York: W.H. Freeman and Company (1979), pp. 31, 187–284.

[4] F.S.C. Silva, M. Finger and A.C.V. Melo, *Lógica para Computação*. São Paulo: Thomson Learning Edições Ltda. (2006), p. 29.

[5] M. Sipser, *Introdução à Teoria da Computação*. São Paulo: Cengage Learning (2017), pp. 48–49, 146–148, 154, 162–164, 183, 187, 202, 224, 281, 286.

[6] S. Dasgupta, C. Papadimitriou and U. Vazirani, *Algoritmos*. São Paulo: McGraw-Hill Interamericana do Brasil Ltda. (2009), p. 244.

[7] H.R. Lewis and C.H. Papadimitriou, *Elementos de Teoria da Computação.* Porto Alegre: Bookman Companhia Editora (2004), pp. XV, 54–55, 57, 190, 193, 238–248, 262 –263, 267, 288–289.

[8] S. Cook, *The complexity of theorem-proving procedures*, in Conference Record of Third Annual ACM Symposium on Theory of Computing, ACM, New York (1971), pp. 151–158.

[9] N. Ziviani, *Projeto de Algoritmos com Implementações em Java e C++*. São Paulo: Thomson Learning (2007), p. 381.

[10] J.E. Hopcraft, J.D. Ullman and R. Motwani, *Introdução à Teoria dos Autômatos, Linguagens e Computação*. Rio de Janeiro: Elsevier e Campus (2003), pp. 350, 388–391, 417–420, 443, 452.

[11] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville: Computer Science Press, Inc. (1984), p. 503.

[12] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization – Algorithms and Complexity.* Mineola (New York): Dover Publications, Inc. (1998), pp. 347–349, 398–399.

[13] S. Cook, *The P versus NP Problem*, available at http://www.claymath.org/sites/default/files/pvsnp.pdf, accessed in 02/02/2019.

[14] C.H. Papadimitriou, *Computational Complexity.* New York: Addison-Wesley Publishing Company, Inc. (1994), pp. 24, 62–63, 57–58, 181.

[15] K. Devlin, *Os Problemas do Milênio – sete grandes enigmas matemáticos do nosso tempo*, chap. 3. Rio de Janeiro: editora Record (2004), pp. 168–169.

[16] A.M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series **2**, **42** (1936), pp. 230–265.

[17] H.G. Rice, *Classes of recursively enumerable sets and their decision problems*, Transactions of the AMS **89** (1953), pp. 25–59.