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

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 the concepts of variable languages and languages with semantic we presented an original proof of the famous P versus NP problem of Computer Science. This proof 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*:**
deterministic algorithm, dynamical class, language, non-deterministic algorithm, polynomial time, P versus NP, semantic, Turing machine, 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* através do uso de dois novos conceitos que chamei de linguagem variável no tempo e linguagem com semântica, bem como do uso de 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. Em nossa conclusão final usaremos estas duas formas de compreender o não determinismo.

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.

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. CONCLUSÃO

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,

$$P = P(t) = P_{pura}(t) \cup P_{semântica}(t)$$

e

$$NP = NP(t) = NP_{pura}(t) \cup NP_{semântica}(t),$$

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.

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, 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, 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, 267.

[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), p. 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.

[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), p. 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.