UM ALGORITMO HEURÍSTICO PARA A SOLUÇÃO DE SAT EM TEMPO POLINOMIAL
(A HEURISTIC ALGORITHM FOR THE SOLUTION OF SAT IN POLYNOMIAL TIME)

Valdir Monteiro dos Santos Godoi
valdir.msgodoi@gmail.com

IMECC – Instituto de Matemática, Estatística e Computação Científica
UNICAMP – Universidade de Campinas, Campinas-SP, Brasil

RESUMO

Usando um novo conceito de linguagem variável, já provei anteriormente que $P \neq NP$, mas tal prova não utilizou nenhum dos problemas clássicos conhecidos como NP-completos, a exemplo de SAT (satisfatibilidade, *satisfiability*), caixeiro-viajante (*travelling-salesman*), soma de subconjuntos (*subset-sum*), da mochila (*knapsack*), programação linear inteira (*integer linear programming*), etc. Tal prova não implica que sendo $P \neq NP$ então devemos ter NP-completo $\notin P$, ou seja, os mencionados famosos problemas difíceis podem ainda ser resolvidos em tempo polinomial, sem precisar encerrar a pesquisa nesta direção. Tal como ocorre com o método simplex, que pode resolver em tempo polinomial a grande maioria dos problemas de programação linear, também é possível resolver SAT em tempo polinomial na maioria das vezes, que é o que eu mostro neste trabalho.

PALAVRAS-CHAVE: SAT, satisfatibilidade, P x NP, NP-completo, solução em tempo polinomial.

ÁREA PRINCIPAL: Heurísticas.

1. Introdução

Pretendemos neste trabalho apresentar um algoritmo heurístico para a solução do problema SAT em tempo polinomial em relação ao tamanho da entrada. Definiremos heurística, os conceitos principais acerca do problema SAT e mostraremos sua relação com os problemas de Pesquisa Operacional.

Conforme definição dada pelo *Google*, heurística é a arte de inventar, de fazer descobertas, ciência que tem por objeto a descoberta dos fatos. Na *wikipedia* diz-se que são processos cognitivos empregados em decisões não racionais, sendo definidas como estratégias que ignoram parte da informação com o objetivo de tornar a escolha mais fácil e rápida. A heurística pode ser considerada um "atalho mental" usado no pensamento humano para se chegar aos resultados e questões mais complicadas de modo rápido e fácil, mesmo que estes sejam incertos ou incompletos (www.significados.com.br/heuristica).

Na Pesquisa Operacional em particular, quando existem problemas muito complicados para se encontrar uma solução ótima, de tal forma que pode não ser possível encontrá-la, ainda é importante encontrar uma boa solução viável, que seja pelo

menos razoavelmente próxima da solução ótima. Os métodos heurísticos são comumente usados para procurar essa solução.[1]

Um método heurístico é um procedimento que provavelmente encontrará uma excelente solução viável, mas não necessariamente ótima, para o problema específico em questão. Não se pode dar nenhuma garantia sobre a qualidade da solução obtida, porém um método heurístico bem elaborado em geral é capaz de fornecer uma solução que se encontra pelo menos próximo da ótima (ou concluir que tais soluções na realidade não existem). O procedimento também deve ser suficientemente capaz para lidar com problemas muito grandes. O procedimento normalmente é um algoritmo iterativo completo em que cada iteração envolve a condução da procura de uma nova solução que poderia ser melhor que a melhor solução encontrada previamente. Quando o algoritmo termina após um tempo razoável, a solução por ele fornecida é a melhor que foi encontrada durante qualquer iteração.[1]

Em linha com as definições dadas nos parágrafos anteriores, classificamos de heurístico o algoritmo que exporemos aqui porque descobrimos (após muitas tentativas com outros algoritmos, algo em torno de 20 outros métodos) que ele nos leva à solução de várias instâncias do problema SAT em um número de iterações bastante pequeno em geral. Uma exceção óbvia ocorre quando a instância é, de fato, insatisfazível, pois em caso contrário temos obtido apenas respostas em tempo polinomial em relação ao tamanho da entrada. Ainda não consegui provar matematicamente, rigorosamente, o porquê deste comportamento, e por isso o título de algoritmo heurístico dado aqui me parece conveniente.

Na seção 2 definiremos o significado de SAT, os conceitos principais a ele relacionados, o que são os problemas NP-completos e seu relacionamento com a Pesquisa Operacional: muitos problemas de Pesquisa Operacional são NP-completos, sendo SAT o paradigma destes problemas[2], [3].

Na seção 3 descreveremos o método, e respectivo algoritmo, que utilizamos para a solução de instâncias de SAT, bem como a sucessão de ideias que nos levou a ele, e na seção 4 exibiremos resultados do processamento deste algoritmo, codificado especificamente na linguagem C, no ambiente Dev C++.

O famoso problema P x NP da Ciência da Computação é apresentado na seção 5, e a seção 6 traz nossa Conclusão final, encerrando o presente trabalho. Complementando o que fizemos, introduzimos ainda dois anexos, o primeiro com meu mais recente artigo sobre o problema P x NP, onde crio o conceito de linguagem variável, e o segundo contendo o programa escrito em C que foi utilizado para processar todos os resultados aqui apresentados.

Estamos usando a expressão “tempo polinomial” com o mesmo significado de “complexidade temporal de ordem polinomial”.

2. SAT, Problemas NP-completos e Pesquisa Operacional

Seguindo [4], SATISFATIBILIDADE, ou SAT, é um problema de grande importância prática, com aplicações variando de teste de chip e de projeto de computadores à análise de imagens e engenharia de software. Ele é também um problema difícil canônico. A aparência de uma instância de SAT é, por exemplo:

$$(x \vee y \vee z\ )\ (x \vee \overline{y})\ (y \vee \overline{z})\ (z \vee \overline{x})\ (\overline{x} \vee \overline{y} \vee \overline{z}).$$

Isso é uma fórmula booleana na forma normal conjuntiva. Ela é uma coleção de cláusulas (os parênteses), cada uma consistindo em uma disjunção (*ou* lógico, denotado por $\vee$) de vários literais, onde um literal é uma variável booleana (tal como x) ou a negação de uma (tal como $\overline{x}$). Uma atribuição de valor verdade satisfatória é uma atribuição de falso ou verdadeiro a cada variável de modo que cada cláusula contenha um literal cujo valor seja verdadeiro. O problema SAT é o seguinte: dada uma fórmula booleana na forma normal conjuntiva, encontre uma atribuição satisfatória ou então declare que nenhuma existe.

Na instância mostrada anteriormente, tornar todas as variáveis verdadeiro satisfaz todas as cláusulas exceto a última. Mas será que existe alguma atribuição que satisfaça todas as cláusulas? Com um pouco de reflexão, não é difícil argumentar que nesse caso nenhuma atribuição desse tipo existe.

SAT é um típico problema de busca. É fornecida uma instância I (ou seja, algum dado de entrada especificando o problema em mãos, nesse caso uma fórmula booleana na forma normal conjuntiva), e nos pedem para encontrarmos uma solução S (um objeto que satisfaça uma especificação, neste caso uma atribuição que satisfaça cada cláusula). Se nenhuma solução desse tipo existe, então devemos afirmar tal fato.

Podemos também usar o símbolo $\wedge$, conectivo representando a conjunção "e". Segundo [5], uma fórmula booleana é uma expressão envolvendo variáveis booleanas e operações. Por exemplo,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

é uma fórmula booleana. Uma fórmula booleana é satisfazível se alguma atribuição de 0s (falsos) e 1s (verdadeiros) faz a fórmula ter valor 1 (verdadeiro). A fórmula precedente é satisfazível porque a atribuição $x = 0,\ y = 1$ e $z = 0$ faz ϕ ter valor 1. Dizemos que a atribuição satisfaz ϕ. O problema da satisfazibilidade é testar se uma fórmula booleana é satisfazível. Mais formalmente, usando a notação de Teoria de Conjuntos, temos ;.

$$SAT = \{\langle\phi\rangle |\ \phi \text{ é } uma\ fórmula\ booleana\ satisfazível\}.$$

Notamos que a maneira de representar as fórmulas booleanas é uma convenção que pode variar de autor para autor. Em [3] vemos o seguinte exemplo de fórmula booleana na forma normal conjuntiva,

$$F = \{(x_1 \vee \overline{x_2} \vee x_3), (\overline{x_1}), (x_2 \vee \overline{x_2})\},$$

enquanto em [6] temos outro exemplo,

$$x1 \wedge \neg(x10 \vee x11),$$

e a equivalência entre as diversas maneiras parece ser bastante intuitiva de entender. O símbolo ¬ representa uma negação.

Como problemas NP-completos entendemos todos os problemas que podem ser reduzidos a (ou transformados em) SAT em tempo polinomial em relação ao tamanho da entrada (dos dados). Estes problemas também são conhecidos como problemas difíceis[4]. Vejamos a seguir uma tabela extraída de [4] relacionando alguns problemas difíceis (NP-completos) e outros fáceis (em P). P refere-se ao conjunto dos problemas que podem ser resolvidos em tempo polinomial em relação ao tamanho da entrada, por uma máquina de Turing determinística (ou equivalentemente, também por um computador moderno)[6].

Problemas difíceis (NP-completos)	Problemas fáceis (em P)
3SAT	2SAT, HORN SAT
CAIXEIRO-VIAJANTE	ÁRVORE ESPALHADA MÍNIMA
CAMINHO MAIS LONGO	CAMINHO MÍNIMO
CASAMENTO 3D	EMPARELHAMENTO BIPARTIDO
MOCHILA	MOCHILA UNÁRIA
CONJUNTO INDEPENDENTE	CONJUNTO INDEPENDENTE EM ÁRVORE
PROGRAMAÇÃO LINEAR INTEIRA	PROGRAMAÇÃO LINEAR
CAMINHO RUDRATA	CAMINHO EULERIANO
CORTE BALANCEADO	CORTE MÍNIMO

Tabela 1 – Alguns problemas NP-completos e problemas em P

Na direita da tabela temos problemas que podem ser resolvidos com eficiência, i.e., em tempo polinomial em relação ao tamanho da entrada dos dados (e não, por exemplo, em tempo estritamente exponencial). Na esquerda, temos *um cacho de nozes duras* que escaparam de uma solução eficiente por muitas décadas ou séculos.

Os vários problemas da direita podem ser resolvidos por diversos e especializados algoritmos: programação dinâmica, fluxos em redes, busca em grafos, algoritmo guloso. Esses problemas são fáceis por várias razões diferentes.

Em um contraste forte, os problemas da esquerda são todos difíceis pela mesma razão (e não por razões diferentes!). Eles são todos equivalentes: cada um pode ser reduzido para qualquer um dos outros, e vice-versa. Não é a intenção deste trabalho dar uma explicação detalhada de cada um destes problemas, mas a referência [4] que seguimos pode esclarecer as particularidades de cada um. Explicaremos a seguir dos

problemas mencionados apenas aqueles que são subconjuntos de SAT: 3SAT, 2SAT e HORN SAT.

O problema 3SAT é uma variante de SAT onde todas as cláusulas possuem três literais (ou no máximo três literais). Semelhantemente, em 2SAT todas as cláusulas possuem dois literais apenas, e no problema HORN SAT todas as cláusulas possuem no máximo um literal positivo.

Podemos agora deixar claro o motivo de SAT estar relacionado com os problemas de Pesquisa Operacional. Muitos dos problemas de Pesquisa Operacional são também problemas NP-completos, por isso reduzem-se a SAT em tempo polinomial em relação ao tamanho da entrada dos dados. Sabendo-se resolver SAT em tempo polinomial, pode-se resolver também em tempo polinomial todos aqueles outros problemas difíceis da Pesquisa Operacional, como o famoso problema do Caixeiro-Viajante e os diversos problemas que usam Programação Linear Inteira, ainda que não seja nada trivial efetuar estas reduções; sabe-se, entretanto, que podem ser feitas em tempo polinomial.

As referências [1] e [7] dão muitos exemplos de problemas que são encontrados em Pesquisa Operacional, e parece verdadeiramente uma quantidade infinita de problemas e aplicações. Sua importância é assim enorme. Como grandes áreas, podemos citar aplicações em Estratégia, Finanças, Logística, Produção, Marketing e Vendas, isto mencionando apenas as áreas funcionais descritas em [7]. O clássico [1] traz também em destaque Teoria dos Jogos, Análise de Decisão, Cadeias de Markov, Teoria das Filas, Teoria dos Estoques e Simulação.

3. O método

Descreveremos com certa brevidade como é o método que desenvolvemos, mas de forma tão precisa quanto pudermos. Conforme mencionamos na Introdução, não temos uma demonstração matemática de que a maioria das instâncias do problema SAT podem ser resolvidas em tempo polinomial, mas com todos os exemplos que obtivemos randomicamente, vários milhões de exemplos, isto aconteceu. As únicas exceções referem-se a instâncias não satisfazíveis.

Representaremos a entrada de nosso problema através de uma matriz $m \times n$, i.e., m linhas e n colunas. Cada linha refere-se a uma cláusula e cada coluna a uma variável. Cada elemento da matriz admite somente três valores: $+1, -1$ ou 0 (zero). O valor $+1$ refere-se a uma variável afirmativa na cláusula, o valor -1 refere-se à negação de uma variável na cláusula, e o valor 0 à ausência da respectiva variável.

O primeiro exemplo dado na seção 2 na forma normal conjuntiva (FNC),

$$(x \vee y \vee z\ )\ (x \vee \overline{y})\ (y \vee \overline{z})\ (z \vee \overline{x})\ (\overline{x} \vee \overline{y} \vee \overline{z}),$$

fica assim representado na nossa convenção:

$$\begin{pmatrix} +1 & +1 & +1 \\ +1 & -1 & 0 \\ 0 & +1 & -1 \\ -1 & 0 & +1 \\ -1 & -1 & -1 \end{pmatrix}$$

No terceiro exemplo,

$$F = \{(x_1 \vee \overline{x_2} \vee x_3), (\overline{x_1}), (x_2 \vee \overline{x_2})\},$$

elimina-se a última cláusula, por ser sempre verdadeira:

$$\begin{pmatrix} +1 & -1 & 1 \\ -1 & 0 & 0 \end{pmatrix}$$

Obviamente, podemos omitir o sinal de positivo $+$ em $+1$. O segundo e último exemplos não estão na FNC e não serão representados em nosso modelo. Para nós interessarão apenas as expressões booleanas na FNC, que foi o primeiro conjunto provado ser NP-completo, por S. Cook [8]. Vale mencionar que é possível provar que o conjunto das expressões booleanas na forma normal disjuntiva (FND) pertence a P, i.e. $FND-SAT \in P$.

Dada uma matriz $m \times n$ representando uma fórmula booleana, com todos os seus elementos pertencentes ao conjunto $\{+1, -1, 0\}$, ela será satisfazível se existir ao menos um conjunto de valores que torne verdadeira cada linha da matriz. Deveremos selecionar um elemento não nulo em cada linha, e não poderá existir contradição em nenhuma coluna. Por exemplo, se na linha 1 selecionamos $+1$ na coluna 3, não poderemos depois selecionar -1 na coluna 3, para nenhuma linha. Vejamos a matriz a seguir, onde se aplica esta regra, e que é satisfazível:

$$\begin{pmatrix} -1 & 0 & \boxed{+1} & +1 \\ 0 & \boxed{-1} & -1 & -1 \\ 0 & \boxed{-1} & +1 & +1 \end{pmatrix}$$

A matriz acima representa uma solução onde a primeira e quarta variáveis não precisaram ser obtidas ou explicitadas (a princípio, poderiam ser verdadeiras ou falsas), enquanto a segunda variável é representada por -1 (falso) e a terceira por $+1$ (verdadeiro).

O primeiro fato que descobrimos é que quando não há variáveis faltantes (zeros) em nenhuma das cláusulas (linhas) pode-se saber de maneira muito imediata se a fórmula booleana (matriz) é satisfazível ou não. Supondo não haver linhas repetidas, se $m < 2^n$ então a fórmula pertence a SAT, caso contrário não. Isso é claramente feito em tempo polinomial. Se uma matriz contém todas as 2^n linhas correspondentes a cada combinação possível de valores para cada uma de suas variáveis, então esta matriz é não

satisfazível, por exemplo, $\begin{pmatrix} -1 & -1 \\ -1 & +1 \\ +1 & -1 \\ +1 & +1 \end{pmatrix}$ é não satisfazível, enquanto a eliminação de uma, duas ou três de quaisquer das linhas torna esta matriz satisfazível. As soluções possíveis são dadas pelo negativo da matriz complementar, i.e., o negativo das linhas que faltam para completar a matriz. O negativo de cada uma das linhas desta matriz complementar pode resolver a matriz principal. No exemplo da matriz 4 x 2 anterior, tirando-se a segunda linha $(-1 \quad +1)$ e multiplicando-a por -1 obtemos $(+1 \quad -1)$, que portanto resolve a matriz $\begin{pmatrix} -1 & -1 \\ +1 & -1 \\ +1 & +1 \end{pmatrix}$, cuja solução é representada por $\begin{pmatrix} -1 & \boxed{-1} \\ \boxed{+1} & -1 \\ \boxed{+1} & +1 \end{pmatrix}$.

Outro fato é que se há no máximo um número limitado de variáveis faltantes em cada linha, por exemplo, no máximo 1, 2 ou 3 variáveis faltantes por linha, também podemos decidir em tempo polinomial se esta matriz é satisfazível ou não.

Se há um único zero em uma linha, este zero corresponderá a uma duplicação da linha, uma linha com o valor -1 na coluna do zero, e outra com o valor $+1$ na coluna do zero, e os valores das demais colunas permanecem inalterados. De maneira similar, k zeros em uma linha corresponderão a 2^k novas linhas, contendo todas as combinações possíveis de -1 e $+1$ nas colunas onde estão os zeros na linha original, com os valores das demais colunas diferentes de zero permanecendo inalteradas.

Se z_i é o número de zeros na linha i, na matriz M $m \times n$ representando uma fórmula booleana F na FNC, com $0 \leq z_i \leq n-1$, se

$$\sum_{i=1}^{m} 2^{z_i} < 2^n$$

então $F \in SAT$.

Se M' é a matriz que foi gerada substituindo todos os zeros de M (que representa F) por todas as combinações possíveis de -1 e $+1$ em cada um desses zeros, mantendo-se inalteradas as outras colunas e eliminando-se todas as linhas duplicadas (o que se faz em tempo polinomial se o número de zeros é limitado ou de ordem logarítmica em relação a n), gerando-se assim uma nova matriz M' $m' \times n$, sem nenhum zero, se $m' < 2^n$ então $F \in SAT$.

Se k é o número máximo de zeros em uma linha, com $k = O(\log_2 P(m,n))$, onde P é um polinômio nas variáveis m e n, ou fixando o valor de k, independentemente dos valores de m e n, chamando $kZSAT$ o conjunto das fórmulas booleanas satisfazíveis representadas por uma matriz com no máximo k zeros por linha

(ou k variáveis faltantes (ausentes) por cláusula na respectiva fórmula booleana), então $kZSAT \in P$. Mais explicitamente, $0ZSAT \in P, 1ZSAT \in P,\ 2ZSAT \in P,\ 3ZSAT \in P$, etc.

Estas afirmações anteriores, teoremas, estão dadas aqui sem demonstração, e não é nossa intenção fazer isso neste momento. Em princípio, nenhum destes teoremas parece nos levar à conclusão de que $3SAT \in P$ ou que $SAT \in P,\ NP-completo \in P$, etc. Mas é verdade que qualquer matriz M de 0s, $+1$s e -1s representando uma fórmula booleana F na forma normal conjuntiva tem um número máximo bem determinado de zeros por linha, digamos, k. Então esta matriz (i.e., sua correspondente fórmula F) pertence a $kZSAT$, donde $F \in P$, e daí, por indução a todas as matrizes nestas condições, para todos os valores inteiros positivos de k, vem $SAT \in P$. É preciso formalizar este raciocínio com mais rigor, sem contradições. Vamos nos limitar, por ora, à apresentação de um algoritmo mais geral, que não contará zeros.

Vimos que é importante encontrar uma linha (não nula) que não pertence à matriz. O negativo dessa linha será uma solução da matriz (sem mudar os 0s ou considerando cada 0 como duas novas linhas, uma com -1 e outra com $+1$ nestas colunas 0).

Para encontrar uma linha que não pertence à matriz procedemos através de tentativas. Poderemos encontrar ou não, mas a cada tentativa testaremos se a valoração das variáveis que obtivemos satisfaz realmente a matriz. Se não satisfaz, buscamos nova tentativa, novos valores para as variáveis, procedimento repetido até encerrarem as linhas ou uma solução for encontrada.

Geramos randomicamente estas matrizes, consistindo para que não haja colunas nulas, nem linhas nulas ou repetidas. Ordenamos as linhas desta matriz, com $-1 < 0 < +1$ e comparando coluna por coluna da esquerda para a direita. Verificamos se existem linhas que só possuem um único elemento, e se sim escolhemos estes valores para comporem nossa solução y nas colunas em que se encontram. Guardamos estes valores num *array* de n ocorrências que chamamos *opcao_unica_col*.

Para as demais colunas, e para cada uma das linhas da matriz enquanto não encontrarmos uma solução, inicializamos nossa solução tentativa y com os valores das colunas correspondentes à i-ésima linha. Havendo zeros em alguma coluna de y substituímo-los pelo primeiro elemento não nulo desta coluna, buscando da primeira à m-ésima linha, até tornar y sem nenhum zero. Algoritmos alternativos podem substituí-los por $+1, -1$, o valor que mais ocorre na coluna, etc.

Completado y, podemos verificar se $-y$ resolve a matriz, o que se faz em tempo polinomial. Notemos que $-y$ não atinge as colunas que têm opção única.

Se não resolver, substituímos y pelo seu sucessor, somando -1 ao elemento mais a direita de y e considerando também os "vai -1" que podem ocorrer. Testamos $-y$ novamente. Usamos a convenção de que $-1 < 0 < +1$, o sucessor de -1 é $+1$, o

sucessor de $+1$ é -1 e "vai -1", etc. Notemos também que este cálculo do sucessor não atinge as colunas que têm opção única.

Podemos repetir este cálculo de sucessor e respectiva verificação de solução um número de vezes da ordem do tamanho da entrada, por exemplo, $m \times n$ vezes ou L vezes, ou alguma outra função polinomial em L de vezes, por exemplo, L^2, $3L^3$, etc., quantidade limitada a $2^n - 1$. Essa busca de solução também é feita em tempo polinomial. Chamamos de L o tamanho da fórmula booleana correspondente à matriz M gerada, somando-se 1 a cada parênteses, conectivos $\vee$, $\wedge$ e de negação ($\sim$ ou $\neg$), nome da variável (x) e somando-se também o tamanho em número de caracteres dos vários índices destas variáveis na base 10 $(1, 2, 3, \ldots, n)$, por exemplo, o tamanho de 30 é 2 e o tamanho de 150 é 3.

Se esse procedimento ainda não foi capaz de encontrar uma solução, passamos a somar linhas e testar cada soma parcial, ou seja, verificamos se $-(l_1 + l_2)$ é solução, se $-(l_1 + l_2 + l_3)$ é solução, e de modo mais geral, se $-(l_i + l_{i+1} + \cdots + l_j)$ é solução, para $1 \leq i, j \leq m$, inicializando y como descrito na etapa anterior. Indicamos por l_i a i-ésima linha da matriz M. Como das outras vezes, o cálculo destas somas não atinge as colunas que têm opção única.

Se ainda assim não tivermos sucesso, classificamos nossa matriz como não satisfazível (NSAT). Obtendo sucesso nesta etapa ou em etapas anteriores nossa matriz é satisfazível (SAT).

Se obtermos NSAT o resultado correto precisa ser dado através de outro algoritmo, conhecido por "força bruta", que testará as 2^n soluções possíveis da matriz até encontrar alguma solução (SAT) ou ter atingido o limite máximo de possibilidades, caso em que M será realmente não satisfazível.

Consideramos resultados compatíveis quando nosso algoritmo principal, que deve rodar em tempo polinomial em relação ao tamanho da entrada, e o algoritmo "força bruta" fornecem a mesma resposta (SAT ou NSAT). Foi também surpreendente descobrir que o algoritmo "força bruta", usando nossa matriz com os elementos ordenados, é capaz de fornecer uma resposta em um número de iterações bastante pequeno na maioria das vezes. Inicializamos o valor de y neste algoritmo "força bruta" primeiramente com as colunas de opção única, a seguir pelas colunas não nulas da primeira linha de M e por fim completando-se as colunas zero com o valor $+1$. Testa-se y como solução e em caso negativo calcula-se o sucessor de y, pelo método já exposto, repetindo-se o teste e se preciso o cálculo de novo sucessor até $2^n - 1$ vezes ou encontrarmos uma solução.

É fácil perceber que todas as etapas de nosso algoritmo principal rodam em tempo polinomial em relação ao tamanho da entrada. Vale lembrar que a entrada deste problema é uma fórmula booleana na forma normal conjuntiva, e não apenas o número n de variáveis ou colunas. Para um esclarecimento maior sobre complexidade

computacional de diversos algoritmos básicos sugerimos ao leitor o clássico CLRS[8] ou o livro de Ziviani [9].

4. Resultados

Conforme [4], cuja edição original americana *Algorithms* data de 2008, se um algoritmo $O(2^n)$ para satisfatibilidade booleana (SAT) roda por uma hora, ele resolveria instâncias com 25 variáveis em 1975, 31 variáveis nos computadores mais rápidos de 1985, 38 variáveis em 1995, e cerca de 45 variáveis com as máquinas de "hoje" (suponhamos, em 2008). Bastante progresso − exceto que cada variável extra requer um ano e meio de espera, enquanto o apetite de aplicações (muitas das quais estão, ironicamente, relacionadas com projetos de computadores) cresce muito mais rápido.[4] Seguindo esta estimativa, em uma hora este algoritmo resolveria instâncias de cerca de 52 variáveis neste ano de 2018.

Colin[7], na edição de 2018, fornece uma interessante tabela de tempo de execução computacional para diferentes tamanhos de problemas. Para problemas $O(2^n)$ com $n = 40$ o tempo de execução seria de 0,349 anos (ou algo próximo de 4 meses e 1 semana) e para $n = 60$ seriam inimagináveis 365.589 anos!

Kenneth H. Rosen[10] também não nos deixa otimista. Para $n = 10$ um algoritmo $O(2^n)$ roda em apenas 10^{-6} s, mas para $n = 10^2$ ele rodaria em 4×10^{13} anos e para $n = 10^3$ seriam mais de 10^{100} anos!

Por sorte, não precisamos esperar tanto tempo assim.

O valor máximo de n, número de colunas ou variáveis booleanas, que usamos é igual a 56. Valores maiores que esse perdem precisão na variável do tipo *double* usado em nosso programa escrito em C para calcular o valor de 2^n, o número máximo de iterações. Mas creio ser um bom limite, provavelmente ultrapassando o mencionado limite das fórmulas booleanas que conseguiríamos resolver em uma hora com os computadores mais rápidos dos dias de hoje, aproximadamente.

Calculando no máximo um único sucessor por linha para y em nosso algoritmo e usando 1 milhão de exemplos gerados randomicamente pelo computador, montamos a tabela 2 a seguir com os 16 últimos resultados do processamento. A tabela 3 foi montada usando no máximo $m \times n$ sucessores por linha e a tabela 4 com no máximo L sucessores por linha. Registramos a razão (q) entre o número de tentativas usadas para encontrar uma solução no algoritmo "força bruta" (FB) e o tamanho da entrada (considerando L e $m \times n$). O percentual de compatibilidade nas soluções entre nosso algoritmo principal e o "força bruta" é superior a 99,0 %. Com matrizes de dimensões até 250 x 56 e até L sucessores por linha foi possível processar 1 milhão de matrizes em $4.362\ s$, ou 1 hora, 12 minutos e 42 segundos, e o percentual de compatibilidade entre os dois algoritmos foi de exatamente 100 %. Substituindo L por $m \times n$ o tempo total de

processamento diminuiu para $3.098\ s$, ou 51 minutos e 38 segundos, também com 100 % de compatibilidade. Em geral, $L > m \times n$. Com no máximo 1 sucessor por linha, mais um novo conjunto de 1 milhão de matrizes de dimensões até 250 x 56 foi gerado e processadas estas matrizes em $2.446\ s$, ou 40 minutos e 46 segundos, com 99,996 % de compatibilidade.

m	n	Tentativas FB	L	$q(L)$	$m \times n$	$q(m \times n)$
157	49	1	22468	0,000045	7963	0,000130
207	28	1	16626	0,000060	5796	0,000173
16	30	1	1414	0,000707	480	0,002083
29	48	1	4024	0,000249	1392	0,000718
167	45	1	22073	0,000045	7515	0,000133
22	11	2	646	0,003096	242	0,008264
206	10	1025	5207	0,196850	2060	0,497573
250	56	1	41013	0,000024	14000	0,000071
73	35	1	7303	0,000137	2555	0,000391
124	55	1	20018	0,000050	6820	0,000147
241	10	1025	6189	0,165616	2410	0,425311
130	46	1	17542	0,000057	5980	0,000167
126	19	1	6639	0,000151	2394	0,000418
250	56	1	40734	0,000025	14000	0,000071
236	53	1	36732	0,000027	12508	0,000080
77	54	1	12244	0,000082	4158	0,000241

Tabela 2 – Número de tentativas no algoritmo força bruta, com até 1 sucessor por linha no algoritmo principal.

m	n	Tentativas FB	L	$q(L)$	$m \times n$	$q(m \times n)$
1	1	1	5	0,200000	1	1,000000
15	24	1	1036	0,000965	360	0,002778
112	19	1	5898	0,000170	2128	0,000470
241	52	1	36693	0,000027	12532	0,000080
248	53	1	38096	0,000026	13144	0,000076
250	56	1	41304	0,000024	14000	0,000071
74	42	1	9145	0,000109	3108	0,000322
228	31	1	20475	0,000049	7068	0,000141
66	52	1	9998	0,000100	3432	0,000291
2	1	3	12	0,250000	2	1,500000
6	27	1	480	0,002083	162	0,006173
53	46	1	7125	0,000140	2438	0,000410
144	30	1	12312	0,000081	4320	0,000231
147	50	1	21610	0,000046	7350	0,000136
71	47	1	9729	0,000103	3337	0,000300
21	36	1	2199	0,0000455	756	0,001323

Tabela 3 – Número de tentativas no algoritmo força bruta, com até $m \times n$ sucessores por linha no algoritmo principal.

m	n	Tentativas FB	L	$q(L)$	$m \times n$	$q(m \times n)$
202	13	5	7073	0,000707	2626	0,001904
101	44	1	13088	0,000076	4444	0,000225
33	37	1	3452	0,000290	1221	0,000819
6	5	2	92	0,021739	30	0,066667
144	31	1	12661	0,000079	4464	0,000224
191	41	1	22996	0,000043	7831	0,000128
146	37	1	15841	0,000063	5402	0,000185
50	14	1	1929	0,000518	700	0,001429
120	47	1	16347	0,000061	5640	0,000177
75	55	1	12267	0,000082	4125	0,000242
200	47	1	27560	0,000036	9400	0,000106
50	14	1	1764	0,000567	700	0,001429
130	28	1	10434	0,000096	3640	0,000275
18	41	1	2117	0,000472	738	0,001355
11	41	1	1265	0,000791	451	0,002217
124	43	1	15569	0,000064	5332	0,000188

Tabela 4 – Número de tentativas no algoritmo força bruta, com até L sucessores por linha no algoritmo principal.

Foi possível constatar que o número de tentativas "FB" é quase sempre igual a 1: em 42 dos 48 exemplos reunidos nas tabelas 2, 3 e 4, ou 87,5 % das vezes, em apenas 1 única tentativa já se obteve a solução. O número de tentativas não cresceu com o aumento de n.

A seguir listamos um comparativo entre o número de tentativas feito pelo algoritmo FB e o algoritmo principal (AP), com 1 milhão de novos exemplos de matrizes de dimensões até 250×56. Surpreendentemente, constatamos que o algoritmo FB foi mais eficiente que o AP, na maioria das vezes. Para o algoritmo AP utilizamos até $m \times n$ sucessores por linha.

Tentativas	FB	AP
1	880.019	55.468
2 – 10	84.109	892.263
11 – 100	29.827	40.132
101 – 1.000	6.031	6.904
1.001 – 10.000	14	2.100
mais de 10.000	0	3.133

Tabela 5 – Comparativo entre os algoritmos

Pela tabela 5 acima se calcula que cerca de 96,41 % dos exemplos tiveram uma solução encontrada em até 10 tentativas pelo algoritmo FB, e um percentual um pouco menor, 94,77 %, pelo algoritmo AP. Mas usando apenas uma única tentativa o algoritmo FB demonstrou ser bem mais eficiente. Além disso, em 886.051 destes exemplos o número de tentativas do AP supera o do FB, em 68.986 exemplos os

números são iguais, e nos restantes 44.963 exemplos o número de tentativas do FB supera o do AP.

Encerramos a exibição de resultados mostrando na tabela 6 todos os exemplos contabilizados na tabela 5 cujo número de tentativas no algoritmo FB foi superior a 1.000. Verificamos que em todos eles o valor L do tamanho da entrada supera o número de tentativas, seja para matrizes (ou fórmulas booleanas) satisfazíveis ou não. O resultado final, por sua vez, mostrou com estes 4 milhões de exemplos (gerados para montar as tabelas 2, 3, 4 e 5) que a razão entre o número de tentativas e o tamanho da entrada é menor que 4:

$$q_{máx}(L) = 1{,}816901$$
$$q_{máx}(m \times n) = 3{,}685714$$

SAT	m	n	Tentativas FB	L	$q(L)$	$m \times n$	$q(m \times n)$
0	249	10	1025	6355	0,161290	2490	0,411647
1	61	13	1058	2039	0,518882	793	1,334174
1	189	14	3287	6953	0,472746	2646	1,242252
1	249	13	1572	8745	0,179760	3237	0,485635
0	249	11	2049	7245	0,282816	2739	0,748083
0	243	10	1025	6050	0,169421	2430	0,421811
1	123	13	1012	4190	0,241527	1599	0,632896
0	206	10	1025	5207	0,196850	2060	0,497573

Tabela 6 – Número de tentativas no algoritmo força bruta, com até $m \times n$ sucessores por linha no algoritmo principal. Selecionados os exemplos com tentativas maiores que 1.000 e excluídos os resultados repetidos. A coluna SAT indica se a matriz gerada é satisfazível (SAT = 1) ou não (SAT = 0).

Como exemplo de uma grande matriz satisfazível cuja solução foi encontrada logo na primeira tentativa pelo algoritmo AP, finalizamos esta seção exibindo na figura 1 uma matriz 14×25 e a solução encontrada. A solução foi

$$y = (1, -1, -1, -1, 1, 0, 0, 0, 0, 0, -1, 0, \ldots, 0),$$

onde os zeros representam tanto -1 quanto $+1$.

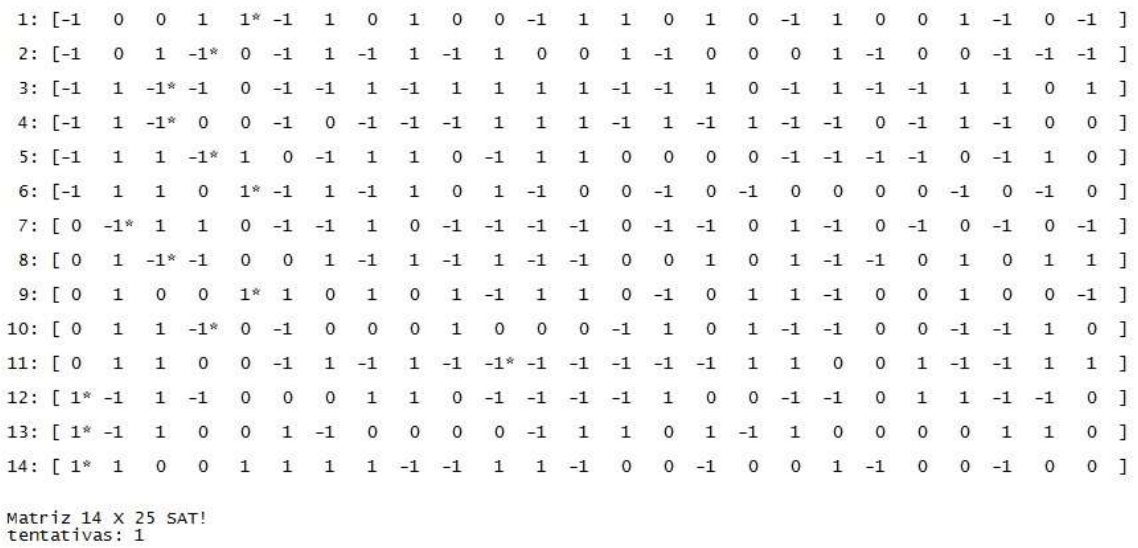

Fig. 1 – Exemplo de Matriz SAT e uma de suas soluções (indicada com *).

5. O problema P x NP

P x NP é o mais importante problema em aberto da Ciência da Computação, e foi formulado em 1971, com o trabalho de Cook[11], embora Karp[12], Edmonds[13,14,15], Hartmanis e Stearns[16], Cobham[17], von Neumann[18] e outros também tenham contribuído para o estabelecimento desta questão. Para uma história mais exata e completa deste problema pode-se consultar, por exemplo, [19], [20] e [21].

O problema P x NP pretende determinar quando uma linguagem L aceita em tempo polinomial por um algoritmo não determinístico $(L \in NP)$ é também aceita em tempo polinomial por algum algoritmo determinístico $(L \in P)$ [22].

Se P = NP então todo problema resolvido em tempo polinomial (em relação ao tamanho da entrada) por um algoritmo não determinístico também será resolvido em tempo polinomial (em relação ao tamanho da entrada) por um algoritmo determinístico.

Se P ≠ NP haverá ao menos um problema pertencente a NP que não terá para sua solução um algoritmo determinístico com complexidade polinomial em relação ao tamanho da entrada.

A classe NP contém uma grande quantidade de problemas importantes cuja solução em tempo polinomial só é possível (ao que se sabe) ou de maneira aproximada ou em casos particulares (mesmo que para infinitos casos particulares), por exemplo, os problemas da satisfatibilidade (SAT, *satisfiability*), caixeiro-viajante (*travelling-salesman*), soma de subconjuntos (*subset-sum*), da mochila (*knapsack*), programação linear inteira (*integer linear programming*), equações diofantinas quadráticas (*quadratic*

diophantine equations), congruência quadrática (*quadratic congruence*), etc. São os já mencionados problemas NP-completos, e vale NP-completo $\subset$ NP.

Como até hoje não se encontrou nenhum algoritmo "perfeito" e "rápido" (em tempo polinomial em relação ao tamanho da entrada) para se resolver estes problemas em NP, a opinião geral dos especialistas é que de fato P $\neq$ NP [4].

P x NP é um dos problemas do milênio[22], cuja solução vale 1 milhão de dólares, daí pode-se dizer que seu interesse passou a ser enorme. Este autor também produziu um artigo sobre a solução deste problema, utilizando um novo conceito de linguagem variável, e que se encontra em anexo. O que provei foi que P $\neq$ NP, mas tal prova não utilizou nenhum destes problemas clássicos NP-completos. Tal prova não implica que sendo P $\neq$ NP então devemos ter NP-completo $\notin$ P, ou seja, os famosos problemas difíceis podem ainda ser resolvidos em tempo polinomial, sem precisar encerrar a pesquisa nesta direção. Tal como ocorre com o método simplex, que pode resolver em tempo polinomial a grande maioria dos problemas de programação linear[4], também é possível resolver SAT em tempo polinomial na maioria das vezes, conforme exposto aqui neste trabalho, nas seções 3 e 4.

6. Conclusão

Este trabalho não é uma prova de P $=$ NP, nem de SAT $\in$ P. Não demos nenhuma demonstração matemática, mas descrevemos interessantes resultados sobre a solução de fórmulas booleanas, quando convertidas para matriz. As afirmações feitas aqui foram verificadas desde bastante tempo por este autor, de maneira livre e independente, e com vários testes computacionais. Prova-se os teoremas da seção 3 utilizando-se as técnicas do Problema de Post[23],[24] e os princípios multiplicativo e aditivo da Análise Combinatória[25].

Temos um antigo interesse pelo problema P $\times$ NP, e assim já escrevemos alguns artigos sobre o tema, sempre provando P $\neq$ NP. Os procedimentos heurísticos em tempo polinomial tantas vezes criados para a solução de várias fórmulas booleanas satisfazíveis[2],[26], entretanto, podem e devem ser ainda procurados. Nosso algoritmo da solução, "Sucessor e Soma Linhas", encontrou uma solução em tempo polinomial em mais de 99,0 % das vezes, quando calculamos no máximo um sucessor por linha, e em exatamente 100 % das vezes quando calculamos até $m \times n$ e L sucessores por linha, o que me parece um resultado animador. O algoritmo da força bruta, que executamos a cada geração de uma nova matriz pois ele é o nosso modelo para a produção de um teste perfeito sobre a decisão entre SAT e não SAT, também é capaz de encontrar alguma solução, em geral, em um tempo pequeno, muitas vezes em até 10 iterações (tentativas), para as fórmulas satisfazíveis.

Todos os nossos exemplos têm como característica que os elementos $-1, 0$ e $+1$ da matriz são equiprováveis, quando foram gerados randomicamente (com nova

semente aleatória a cada 10.000 exemplos), por isso não fizemos um estudo específico focado no conjunto 3SAT, onde só aparecem no máximo 3 elementos -1 e $+1$ por linha e os zeros são predominantes quando $n > 6$. Será nossa próxima etapa de estudo.

Não esperamos que a eficiência dos algoritmos aqui expostos se mantenha tão boa quando se trata exclusivamente do problema 3SAT, ao invés de SAT em geral, mas mesmo assim esperamos uma compatibilidade entre as soluções dos dois algoritmos (AP polinomial e FB exponencial) pelo menos maior que 94,4 % (valor computado em um teste preliminar nosso com 50 mil exemplos e matrizes até 80×18), caso contrário buscaremos outro método. Uma forma melhorada que deve ser implementada (não codificada no programa em anexo) é limitar o número máximo de sucessores por linha a $2^n - 1$. Se em uma linha já testamos todas as 2^n combinações possíveis de -1s e $+1$s, sem sucesso, então a matriz é não satisfazível.

O algoritmo FB ("força bruta") acabou tendo neste trabalho um destaque maior do que previámos inicialmente, quando começamos a escrevê-lo, e talvez devesse ser ele nosso "algoritmo principal" (AP). Por ora, é ainda apenas nosso algoritmo auxiliar, que fornece a correta decisão entre SAT e não SAT.

A Pesquisa Operacional é apaixonante, e traz ricas aplicações, como se pode verificar no clássico de Lieberman[1] e em Colin[7]. Conforme sabemos, muitas destas aplicações acabam por esbarrar nas dificuldades de tempo de processamento quando o tamanho dos dados de entrada é grande, tempo que cresce exponencialmente com este tamanho. A redução destes problemas para SAT parece interessante, e algo a ser verificado.

Além dos algoritmos vistos na seção 3, encontra-se no Anexo 2 o programa em C que rodamos (várias vezes) para obtermos os resultados descritos na seção 4. Um terceiro algoritmo, da decisão, que utiliza o princípio da inclusão-exclusão[25], fornece sempre a resposta correta, mas processa em tempo de ordem exponencial em relação ao número de linhas. Ainda trabalho para deixá-lo mais eficiente, e não será exposto aqui.

Note que existem outros algoritmos conhecidos na literatura para a solução de SAT, e uma exposição de alguns destes métodos pode ser encontrada em [26]. Um deles é o método DPLL, ou método Davis-Putnam, data de 1962 e vem sendo implementado com as mais diferentes heurísticas desde então. Já ganhou diversas competições de implementações na resolução de problemas como SAT, planejamento e outros problemas NP-completos que podem ser traduzidos no problema SAT. Outro método é o Chaff, uma extensão do DPLL que utiliza aprendizado, e outro é o método incompleto GSAT, que se baseia em processos probabilísticos sobre uma busca local.[26]

A Évariste Galois,
in memoriam.
Quando temos certeza, mesmo sem provar.

Referências

1. Hillier, F. S. and Lieberman, G. J., *Introdução à Pesquisa Operacional*. Porto Alegre: AMGH Editora Ltda. (2013).

2. Biere, A. *et al* (editors), *Handbook of satisfiability*. Amsterdam: IOS Press (2009).

3. Lewis, H. R. and Papadimitriou, C. H., *Elementos de Teoria da Computação*. Porto Alegre: Bookman (2000).

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

5. Sipser, M., *Introdução à Teoria da Computação.* São Paulo: Cengage Learning (2007).

6. Hopcraft, J. E., Ullman, J. D. and Motwani, R., *Introdução à Teoria dos Autômatos, Linguagens e Computação*. Rio de Janeiro: Elsevier e Campus (2003).

7. Colin, E. C., *Pesquisa Operacional – 170 Aplicações em Estratégia, Finanças, Logística, Produção, Marketing e Vendas*. São Paulo: Editora Atlas Ltda. (2018).

8. Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Algoritmos – Teoria e Prática.* Rio de Janeiro: Editora Campus Ltda. (2002).

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

10. Rosen, K. H., *Matemática Discreta e Suas Aplicações*. São Paulo: McGraw-Hill Interamericana do Brasil Ltda. (2007).

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

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

13. Edmonds, J., *Maximum matchings and a polyhedron with 0,1-vertices*, Journal of Research at the National Bureau of Standards, Section B, 69 (1965), 125-130.

14. Edmonds, J., *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards, Section B, 69 (1965), 67–72.

15. Edmonds, J., *Paths, trees and flowers*, Canadian Journal of Mathematics, 17 (1965), 449-467.

16. Hartmanis, J. and Stearns, R. E., *On the computational complexity of algorithms*, Transactions of the AMS 117 (1965), 285-306.

17. Cobham, A., *The intrinsic computational difficulty of functions*, in Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science, Y. Bar-Hille, ed., Elsevier/North-Holland, Amsterdam (1964), 24–30.

18. von Neumann, J., *A certain zero-sum two-person game equivalent to the optimal assignment problem*, in Contributions to the Theory of Games II, H.W. Kahn and A.W. Tucker, eds., Princeton Univ. Press, Princeton, NJ (1953), 5–12.

19. Fortnow, L. and Homer, S., *A Short History of Computational Complexity*, available at http://people.cs.uchicago.edu/~fortnow/beatcs/column80.pdf, accessed in 02/22/2011.

20. Herken, R., ed., *The Universal Turing Machine – A Half-Century Survey*, Verlag Kammerer & Unverzagt, Hamburg-Berlin (1988).

21. Sipser, M., *The History and Status of the P versus NP question*, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (1992), 603-619, available at http://www.win.tue.nl/~gwoegi/P-versus-NP/sipser.pdf, accessed in 02/22/2011.

22. Cook, S., *The P versus NP Problem*, available at http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf, accessed in 02/22/2011.

23. Castrucci, B., *Introdução à Lógica Matemática*. São Paulo: Livraria Nobel S.A. (1975).

24. Hegenberg, L., *Lógica: o cálculo sentencial*. São Paulo: EPU – Editora Pedagógica e Universitária (1977).

25. Santos, J. P. O., Mello, M. P. e Murari, I. T. C., *Introdução à Análise Combinatória*. Rio de Janeiro: Editora Ciência Moderna Ltda. (2007).

26. Silva, F. S. C., Finger, M. and Melo, A. C. V., *Lógica para Computação*. São Paulo: Thomson Learning Edições (2006).

ANEXO 1

Linguagens Variáveis no Tempo e o Problema P x NP

(Languages Varying in Time and the P x NP Problem)

Valdir Monteiro dos Santos Godoi
valdir.msgodoi@gmail.com

An original proof of P ≠ NP.

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 ([1], [2], [3]).

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 = {1}, L = {5}, etc., evidenciando que temos um exemplo de linguagem não constante, e variável no tempo, dependente de cada nova situação.

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

$L = \{w = (x\ y\ z); x$ 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 $y = z$.

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. Um exemplo para x: PETR4-2018-03-08-CLOSE, onde se pergunta qual o valor que a ação PETR4 da Petrobrás terá no fechamento de 08/março/2018 (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 a mais fundamental propriedade 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 [4], na primeira tentativa, como parece defender Papadimitriou *et al* em [5], 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 $NPCOMPLETO \notin P$. É possível ter $P \neq NP$ e também $NPCOMPLETO \in P$.

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”? Tal restrição não está descrita formalmente em lugar algum, por exemplo, em [6]. 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 $P \neq NP$.

A Alan Turing, in memoriam.

Referências

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

[2] Hopcraft, J. E., Ullman, J. D. and Motwani, R., *Introdução à Teoria dos Autômatos, Linguagens e Computação*. Rio de Janeiro: Elsevier e Campus (2003), 452.

[3] Ziviani, N., *Projeto de Algoritmos com Implementações em Java e C++*. São Paulo: Thomson Learning (2007), 381-383, 388, 551.

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

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

[6] Cook, S., *The P Versus NP Problem*, available at http://www.claymath.org/sites/default/files/pvsnp.pdf (2000).

ANEXO 2

(contém comentários, mensagens e algumas instruções inibidas, úteis para algum teste)

```
// P x NP
// 1: algoritmo da solução (Sucessor e Soma Linhas)
// 2: algoritmo da força bruta (gera as 2^n combinações possíveis)
// (decide acertadamente se SAT ou não)
//
// Valdir Monteiro dos Santos Godoi
// 17/06/2018 (PM015)

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#define dimLin 250
#define dimLin1 dimLin + 1

#define dimCol 56 // use dimCol máximo até 56
#define dimCol1 dimCol + 1

void gera_entrada(void);
void calcula_tamanho_entrada(void);
void soma_digitos_indice(void);
void ordena_matriz(void);
void permuta(int, int);
void inicializacoes(void);
void busca_opcoes_unicas(void);
void calcula_moda(void);

void programa1(void);
void zera(void);
void inicializa_y_solucao(void);
void substitui_zeros(void);
void negativa(void);
void testa_solucao(void);
void calcula_sucessor(void);

void inicializa_soma(void);
void soma(void);
void soma_vai_y(void);
void soma_y_lin2(void);

void exibe_resultado(void);
void exibe_resultado_PM015(void);

void programa2(void);
void inicializa_y(void);
void substitui_zeros_fb(void);
void calcula_sucessor_fb(void);
```

```
long int cont_sat, cont_sat1, cont_sat2,
       cont_compativeis, cont_incompativeis, total_testes,
       tentativas_AP_maior_FB, tentativas_AP_igual_FB, tentativas_AP_menor_FB,
       tentativas_AP, tentativas_FB, tab_tentativas_FB[6], tab_tentativas_AP[6];

double cont_tentativas, cont_tentativas_max;

int sat, sat1, sat2,
   m, n, lin, col, lin2, r, loop_sucessor, igual, exemplo, vai,
   x[dimLin1][dimCol1],
   y[dimCol1],
   col_lin[dimLin1], col_lin2[dimLin1],
   default_col[dimCol1],
   opcao_unica_col[dimCol1],
   usou_col[dimCol1], contradicao[dimCol1],
   preenchidos, valor, unico, ha_contradicao, somei,
   quantos_mais, quantos_menos, fim, exibe;

int soma_mais, soma_menos, L, mn;

float q_max, q_mn_max;

int main()
{
   cont_sat = 0;
   cont_sat1 = 0;
   cont_sat2 = 0;
   cont_compativeis = 0;
   cont_incompativeis = 0;
   exemplo = 0;
   q_max = 0;
   q_mn_max = 0;
   tentativas_AP_maior_FB = 0;
   tentativas_AP = tentativas_FB = 0;
   tab_tentativas_FB[0] = tab_tentativas_FB[1] = tab_tentativas_FB[2] =
   tab_tentativas_FB[3] = tab_tentativas_FB[4] = tab_tentativas_FB[5] = 0;
   tab_tentativas_AP[0] = tab_tentativas_AP[1] = tab_tentativas_AP[2] =
   tab_tentativas_AP[3] = tab_tentativas_AP[4] = tab_tentativas_AP[5] = 0;
   srand(time(NULL));
   system("CLS");
   printf("RAND_MAX: %d\n", RAND_MAX);

   while (1)
     {

      exemplo++;

      gera_entrada;
      calcula_tamanho_entrada;
      ordena_matriz;
      inicializacoes;
```

```
      busca_opcoes_unicas;
      calcula_moda;

      //printf("Matriz %d: %d X %d \n", exemplo, m, n);

      //if (sat1 == 0)
        {//printf("2 - Algoritmo da forca bruta (2^n combinacoes).\n");
         programa2;
         //printf("2 - Algoritmo da forca bruta (2^n combinacoes).\n\n");
        }
      sat2 = sat;

      /*
      printf("1 - Algoritmo da solucao.\n");
      */
      programa1;
      /*
      printf("1 - Algoritmo do sucessor e Soma Linhas.\n\n");
      */
      sat1 = sat;

      if (sat1 == 1 || sat2 == 1)
         cont_sat++;

      cont_sat1 += sat1;
      cont_sat2 += sat2;

      if (tentativas_AP > tentativas_FB)
         tentativas_AP_maior_FB++;
      else
      if (tentativas_AP == tentativas_FB)
         tentativas_AP_igual_FB++;
      else
         tentativas_AP_menor_FB++;

      if (sat1 == sat2)
        {cont_compativeis++;
         total_testes = cont_compativeis + cont_incompativeis;
         //if (exemplo % 1000 == 0)
         //    printf("Resultados compativeis! %d\n\n", cont_compativeis);
        }
      else
        {cont_incompativeis++;
         total_testes = cont_compativeis + cont_incompativeis;
         printf("==> Incompatibilidade...\n\n");
         printf("    Matrizes SAT: %7ld\n", cont_sat);
         printf("    Solucao: %d    %7ld\n", sat1, cont_sat1);
         printf("    FBruta : %d    %7ld\n", sat2, cont_sat2);
         printf("    Compativeis:  %7ld ( %7.3f %% )\n", cont_compativeis, (float)
cont_compativeis/total_testes * 100);
         printf("    Incompativeis: %6ld ( %7.3f %% )\n", cont_incompativeis, (float)
cont_incompativeis/total_testes * 100);
```

```
        if (sat2 == 1)
           if (sat1 == 0)
              printf("\n==> Matriz SAT %d x %d, solucao falhou!\n", m, n);
        if (sat1 == 1)
           if (sat2 == 0)
              printf("\n==> Matriz SAT %d x %d, forca bruta falhou!\n", m, n);
        printf("\n");
        printf("y solucao: ");
        for (col = 1; col <= n; col++)
            printf("%2d ", y[col]);
        printf("\n\n");
        //system("PAUSE");
        //srand(time(NULL));
        }

      if (exemplo >= 1000000)
        {printf("\n");
         printf("    Dimensao maxima: %d X %d\n", dimLin, dimCol);
         printf("    Exemplos....: %7d\n", exemplo);
         printf("    Matrizes SAT: %7ld\n", cont_sat);
         printf("    Solucao: %d    %7ld\n", sat1, cont_sat1);
         printf("    FBruta : %d    %7ld\n", sat2, cont_sat2);
         printf("    Decisao: %d    %7ld\n", sat3, cont_sat3);
         printf("    Compativeis:  %7ld ( %7.3f %% )\n", cont_compativeis, (float)
cont_compativeis/total_testes * 100);
         printf("    Incompativeis: %6ld ( %7.3f %% )\n", cont_incompativeis, (float)
cont_incompativeis/total_testes * 100);
         printf("    q L max: %f\n", q_max);
         printf("    q_mn max: %f\n\n", q_mn_max);
         printf("    Tentativas no Forca Bruta\n");
         printf("    1 tentativa           : %d\n", tab_tentativas_FB[0]);
         printf("    ate 10 tentativas     : %d\n", tab_tentativas_FB[1]);
         printf("    ate 100 tentativas    : %d\n", tab_tentativas_FB[2]);
         printf("    ate 1000 tentativas   : %d\n", tab_tentativas_FB[3]);
         printf("    ate 10000 tentativas  : %d\n", tab_tentativas_FB[4]);
         printf("    mais de 10000 tentativas: %d\n", tab_tentativas_FB[5]);
         printf("\n");
         printf("    Tentativas no Algoritmo Principal\n");
         printf("    1 tentativa           : %d\n", tab_tentativas_AP[0]);
         printf("    ate 10 tentativas     : %d\n", tab_tentativas_AP[1]);
         printf("    ate 100 tentativas    : %d\n", tab_tentativas_AP[2]);
         printf("    ate 1000 tentativas   : %d\n", tab_tentativas_AP[3]);
         printf("    ate 10000 tentativas  : %d\n", tab_tentativas_AP[4]);
         printf("    mais de 10000 tentativas: %d\n", tab_tentativas_AP[5]);
         printf("\n");
         printf("    tentativas_AP > tentativas_FB: %d\n", tentativas_AP_maior_FB);
         printf("    tentativas_AP = tentativas_FB: %d\n", tentativas_AP_igual_FB);
         printf("    tentativas_AP < tentativas_FB: %d\n", tentativas_AP_menor_FB);
         printf("\n");
         printf("    RAND_MAX: %d\n", RAND_MAX);
         printf("\n");
         break;
```

```
        }

     }
   system("pause");
   return 0;
}

void gera_entrada()
{
    int zerada, repetida, lin1, lin2;

    if (exemplo % 10000 == 0)
       srand(time(NULL));

    repetida = 1;
    while (repetida == 1)
      {n = rand()%dimCol + 1; // colunas
       m = pow(2, n); // linhas
       m = rand()%m + 1;
       if (m > dimLin)
          m = rand()%dimLin + 1; // linhas
       for (lin = 1; lin <= m; lin++)
          {zerada = 1;
           while (zerada == 1) // não permite linha totalmente zerada
             for (col = 1; col <= n; col++)
                {r = rand()%3 - 1; // -1, 0, +1
                 x[lin][col] = r;
                 if (r != 0)
                    zerada = 0;
                }
          }
       // não permite coluna totalmente zerada
       for (col = 1; col <= n ; col++)
          {zerada = 1;
           for (lin = 1; lin <= m; lin++)
               if (x[lin][col] != 0)
                 {zerada = 0;
                  break;
                 }
            if (zerada == 1)
               break;
          }
       if (zerada == 1)
          continue; // fica com repetida = 1
       //
       if (m == 1)
          repetida = 0;
       else
         {for (lin = 2; lin <= m; lin++)
             {lin1 = lin - 1;
              for (lin2 = 1; lin2 <= lin1; lin2++)
                 {repetida = 1;
```

```
                for (col = 1; col <= n; col++)
                   if (x[lin][col] != x[lin2][col])
                      {repetida = 0;
                       col = n;
                      }
                if (repetida == 1)
                  {lin = m;
                   lin2 = m;
                  }
               }
            }
         }
      }
   return;
}

void calcula_tamanho_entrada()
{
   // calcula comprimento da entrada
   L = 0;
   for (lin = 1; lin <= m; lin++)
      {soma_mais = 0;
       soma_menos = 0;
       for (col = 1; col <= n; col++)
          if (x[lin][col] != 0)
             {soma_digitos_indice;
              if (x[lin][col] == 1)
                 soma_mais++;
              else
                 if (x[lin][col] == -1)
                    soma_menos++;
             }
       L += soma_mais; // nome da variável
       L += 2*soma_menos; // não  e nome da variável
       L += (soma_mais + soma_menos - 1); // ou
       L += 2; // parenteses
      }
   L += (m - 1); // e
   if (m > 1)
       L += 2; // parenteses inicial e final
   mn = m * n;
   return;
}

void soma_digitos_indice()
{
   if (col < 10)
      {L += 1;
       return;
      }
   if (col < 100)
     {L += 2;
```

```
        return;
       }
    if (col < 1000)
       {L += 3;
        return;
       }
    if (col < 10000)
       {L += 4;
        return;
       }
    if (col >= 10000)
      {printf("Numero de coluna imprevisivel: %d\n", col);
       system("pause");
      }
    return;
}

void ordena_matriz()
{ // ordem crescente: -1 < 0 < +1
    for (lin = 1; lin <= m - 1; lin++)
    for (lin2 = lin + 1; lin2 <= m; lin2++)
    for (col = 1; col <= n; col++)
        if (x[lin][col] > x[lin2][col])
          {permuta(lin, lin2);
           break;
          }
        else
          if (x[lin][col] < x[lin2][col])
             break;
    return;
}

void permuta(int lin, int lin2)
{
  int k, aux;
  for (k = 1; k <= n; k++)
     {aux = x[lin][k];
      x[lin][k] = x[lin2][k];
      x[lin2][k] = aux;
     }
  return;
}

void inicializacoes()
{
    for (lin = 1; lin <= m; lin++)
        col_lin[lin] = 0;
    for (col = 1; col <= n; col++)
       {default_col[col] = 0;
        opcao_unica_col[col] = 0;
        usou_col[col] = 0;
        contradicao[col] = 0;
```

```
      }
   ha_contradicao = 0;
   return;
}

void busca_opcoes_unicas()
{
   for (lin = 1; lin <= m; lin++)
      {preenchidos = 0;
       valor = 0;
       unico = 1;
       for (col = 1; col <= n; col++)
          if (x[lin][col] != 0)
            {preenchidos++;
             if (valor == 0)
                valor = x[lin][col];
             else
                if (valor != x[lin][col])
                   unico = 0;
            }
       if (preenchidos == 1)
         if (unico == 1)
           for (col = 1; col <= n; col++)
              if (x[lin][col] != 0)
                 if (default_col[col] == 0)
                   {default_col[col] = valor;
                    opcao_unica_col[col] = valor;
                   }
                 else
                   if (x[lin][col] != opcao_unica_col[col])
                     {contradicao[col] = 1;
                      ha_contradicao = 1;
                     }
      }
   if (ha_contradicao == 1) // já sei que é NSAT!
      for (col = 1; col <= n; col++)
          if (contradicao[col] == 1)
            {opcao_unica_col[col] = 0;
             default_col[col] = 0;
            }
   for (col = 1; col <= n; col++)
        if (opcao_unica_col[col] == 0)
          {ha_contradicao = 0;
           for (lin = 1; lin <= m; lin++)
             if (x[lin][col] != 0)
               {for (lin2 = lin + 1; lin2 <= m; lin2++)
                 if (x[lin2][col] != 0)
                    if (x[lin2][col] != x[lin][col])
                      {ha_contradicao = 1;
                       break;
                      }
           if (ha_contradicao == 0)
```

```
                {opcao_unica_col[col] = x[lin][col];
                 break;
                }
             }
    }
    return;
}

void calcula_moda()
{
    for (col = 1; col <= n; col++)
       if (default_col[col] == 0)
         {quantos_mais = 0;
          quantos_menos = 0;
          for (lin = 1; lin <= m; lin++)
             if (x[lin][col] == 1)
                quantos_mais++;
             else
                if (x[lin][col] == -1)
                   quantos_menos++;
          if (quantos_mais > quantos_menos)
             default_col[col] = +1;
          else
             default_col[col] = -1;
         }
    return;
}

void programa1()
{   // algoritmo da solucao (Sucessor e Soma Linhas)
    sat = 0;
    cont_tentativas = 0;
    for (lin = 1; lin <= m; lin++)
       {zera;
        inicializa_y_solucao;
        substitui_zeros;
        negativa;
        testa_solucao;
        negativa;
        if (sat == 1)
          {sat1 = sat;
           break;
          }

        for (r = 1; r <= mn; r++)
           {calcula_sucessor;
            negativa;
            testa_solucao;
            if (sat == 1)
               break;
            negativa;
           }
```

```
    if (sat == 1)
       break;

    zera;
    inicializa_soma;
    substitui_zeros;
    negativa;
    testa_solucao;
    if (sat == 1)
       break;
    negativa;
    for (lin2 = lin + 1; lin2 <= m; lin2++)
        {soma;
         negativa;
         testa_solucao;
         if (sat == 1)
            break;
         negativa;
        }
    if (sat == 1)
       break;
   }
sat1 = sat;
if (sat1 != sat2)
  {exibe = 1;
   exibe_resultado;
   exibe = 0;
  }
if (sat == 1)
    if (cont_tentativas == 1)
      {exibe = 1;
       exibe_resultado_PM015;
       exibe = 0;
      }
tentativas_AP = cont_tentativas;
if (cont_tentativas == 1)
    tab_tentativas_AP[0]++;
else
if (cont_tentativas > 10000)
    tab_tentativas_AP[5]++;
else
if (cont_tentativas > 1000)
    tab_tentativas_AP[4]++;
else
if (cont_tentativas > 100)
    tab_tentativas_AP[3]++;
else
if (cont_tentativas > 10)
    tab_tentativas_AP[2]++;
else
    tab_tentativas_AP[1]++;
```

```
    return;
}

void zera()
{
    for (col = 1; col <= n; col++)
          y[col] = 0;
    return;
}

void inicializa_y_solucao()
{
    for (col = 1; col <= n; col++)
       {y[col] = opcao_unica_col[col];
        if (y[col] == 0)
           y[col] = x[lin][col];
       }
    return;
}

void substitui_zeros()
{
    for (col = 1; col <= n; col++)
        if (y[col] == 0)
           for (lin2 = 1; lin2 <= m; lin2++)
               if (x[lin2][col] != 0)
                  {y[col] = x[lin2][col];
                   break;
                  }
    return;
}

void negativa()
{
    for (col = 1; col <= n; col++)
        if (opcao_unica_col[col] == 0)
           y[col] = -y[col];
    return;
}

void testa_solucao()
{
    int i;
    for (i = 1; i <= m; i++)
        col_lin[i] = 0;
    sat = 1;
    for (i = 1; i <= m; i++)
       {for (col = 1; col <= n; col++)
          if (y[col] != 0)
            if (x[i][col] == y[col])
              {col_lin[i] = col;
               col = n;
```

```
                }
          if (col_lin[i] == 0)
            {i = m;
             sat = 0;
            }
         }
     cont_tentativas++;
     return;
}

void calcula_sucessor()
{
   loop_sucessor = 0;
   for (col = n - loop_sucessor; col >= 1; col--)
        if (opcao_unica_col[col] == 0)
           if (y[col] == -1)
             {y[col] = 1;
              col = 0;
             }
           else
             if (y[col] == 1)
                y[col] = -1;
    return;
}

void inicializa_soma()
{
  for (col = 1; col <= n; col++)
       {y[col] = opcao_unica_col[col];
        if (y[col] == 0)
           y[col] = x[lin][col];
       }
  return;
}

void soma()
{
  vai = 0;
  for (col = n; col >= 1; col--)
         if (opcao_unica_col[col] == 0)
            {soma_vai_y;
             soma_y_lin2;
            }
  return;
}

void soma_vai_y()
{
  if (y[col] == -1)
    {if (vai == -1)
         {y[col] = +1;
          vai = 0;
```

```
          }
       else
            if (vai == 0)
                vai = 0;
            else // vai == +1
               {y[col] = -1;
                vai = -1;
               }
     }
  else
       if (y[col] == 0)
           vai = 0;
       else // y[col] = +1;
             if (vai == -1)
                {y[col] = -1;
                 vai = -1;
                }
             else
                 if (vai == 0)
                     vai = 0;
                 else // vai = +1
                     {y[col] = +1;
                      vai = -1;
                     }
  return;
}

void soma_y_lin2()
{
if (y[col] == -1)
   {if (x[lin2][col] == -1)
      {y[col] = +1;
       vai = 0;
      }
    else
        if (x[lin2][col] == 0)
            vai = 0;
        else // x[lin2][col] == +1
           {y[col] = -1;
            vai = -1;
           }
   }
else
   if (y[col] == 0)
      vai = 0;
   else // y[col] = +1;
      if (x[lin2][col] == -1)
        {y[col] = -1;
         vai = -1;
        }
      else
          if (x[lin2][col] == 0)
```

```
                vai = 0;
            else // x[lin2][col] = +1
                {y[col] = +1;
                 vai = -1;
                }
return;
}

void exibe_resultado()
{
    if (exibe == 0)
       return;
    if (exemplo % 1000 == 0)
    {
    if (sat == 0 && m <= 50)
    {for (lin = 1; lin <= m; lin++)
       {printf("\nLIN %3d: ", lin);
        for (col = 1; col <= n; col++)
           {printf("%3d", x[lin][col]);
            if (col_lin[lin] == col)
               printf(". ");
            else
               printf("  ");
           }
        printf("=> COL %3d", col_lin[lin]);
       }
     printf("\n");
    }
    printf("\nMatriz %d X %d ", m, n);
    if (sat == 1)
       printf("SAT!\n");
    else
      {printf("NSAT...\n");
       //system("pause");
      }
    printf("\n");
    }
    return;
}

void exibe_resultado_PM015()
{
    if (exibe == 0)
       return;
    if (n != 25)
        return;
    if ((exemplo % 1000 == 0) || (cont_tentativas == 1))
    {
    if (m <= 50)
    {for (lin = 1; lin <= m; lin++)
       {printf("\n%2d: [", lin);
        for (col = 1; col <= n; col++)
```

```
            {printf("%2d", x[lin][col]);
             if (col_lin[lin] == col)
                printf("* ");
             else
                printf("  ");
            }
         printf("]\n");
         //printf("=> COL %3d", col_lin[lin]);
        }
     printf("\n");
    }
    printf("\nMatriz %d X %d ", m, n);
    if (sat == 1)
       printf("SAT!\n");
    else
      {printf("NSAT...\n");
       //system("pause");
      }
    printf("tentativas: %.0f\n", cont_tentativas);
    printf("\n");
    system("pause");
    }
    return;
}

void programa2()
{   // força bruta 2^n
    int tam, tam_mn; float q, q_mn;
    sat = 0;
    lin = 1;
    inicializa_y;
    substitui_zeros_fb;
    cont_tentativas = 0;
    cont_tentativas_max = pow(2, n);
    if (cont_tentativas_max < 0)
      {printf("Numero de colunas em excesso: %d\n", n);
       printf("Encerre a execucao.\n");
       system("pause");
       return;
      }
    while (cont_tentativas <= cont_tentativas_max) // substituir <= por <
      {testa_solucao;
       if (sat == 1)
          break;
       calcula_sucessor_fb;
      }
    tentativas_FB = cont_tentativas;
    if (cont_tentativas == 1)
        tab_tentativas_FB[0]++;
    else
    if (cont_tentativas > 10000)
        tab_tentativas_FB[5]++;
```

```
else
if (cont_tentativas > 1000)
    tab_tentativas_FB[4]++;
else
if (cont_tentativas > 100)
    tab_tentativas_FB[3]++;
else
if (cont_tentativas > 10)
    tab_tentativas_FB[2]++;
else
    tab_tentativas_FB[1]++;
exibe = 0;
exibe_resultado;
exibe = 0;
tam = L;
tam_mn = m*n;
q = (float) cont_tentativas / tam;
q_mn = (float) cont_tentativas / tam_mn;
if (q > q_max)
   q_max = q;
if (q_mn > q_mn_max)
   q_mn_max = q_mn;
//if ((exemplo % 1000 == 0)
//||  (q >= 2 && sat == 1)
//||  (q_mn >= 2 && sat == 1)
//||  (m == dimLin && n == dimCol)
//||  (cont_tentativas > 1000)
//||  (sat == 0))
if (cont_tentativas > 1000)
{
printf("\nExemplo %d\n", exemplo);
printf("Matriz %d X %d\n", m, n);
        printf("Forca Bruta\n");
        printf("SAT: %d\n", sat);
printf("cont_tentativas: %.0lf\n", cont_tentativas);
printf("tamanho entrada L: %d\n", tam);
printf("tamanho entrada m X n: %d\n", tam_mn);
printf("cont_tentativas_max n, 2^n: %d, %.0lf\n", n, cont_tentativas_max);
printf("cont_tentativas / tam entrada L: %f\n", q);
printf("cont_tentativas / tam entrada m X n: %f\n\n", q_mn);
printf("q_max atual: %f\n\n", q_max);
printf("q_mn_max atual: %f\n\n", q_mn_max);
if ((q >= 4 || q_mn >= 4) && sat == 1)
  {printf("Matriz SAT! Observe q, q_mn >= 4\n\n");
   system("pause");
   //srand(time(NULL));
  }
else
        if ((exemplo < 2500 && m == dimLin && n == dimCol)
        ||  (exemplo < 100  && cont_tentativas > 1)
        ||  (cont_tentativas > 1000))
          {printf("Verifique dimensao maxima, tentativas > 1 e NSAT.\n\n");
```

```
            //system("pause");
            //srand(time(NULL));
              }
    }
    return;
}

void inicializa_y()
{
    for (col = 1; col <= n; col++)
        if (opcao_unica_col[col] == 0)
            y[col] = x[lin][col];
        else
            y[col] = opcao_unica_col[col];
    return;
}

void substitui_zeros_fb()
{
    for (col = 1; col <= n; col++)
        if (y[col] == 0)
            y[col] = 1; //default_col[col];
    return;
}

void calcula_sucessor_fb()
{
    for (col = n; col >= 1; col--)
       if (opcao_unica_col[col] == 0)
          if (y[col] == -1)
            {y[col] = 1;
             col = 0;
            }
          else
            if (y[col] == 1)
               y[col] = -1;
    return;
}
```