Universidade Estadual de Campinas
Instituto de Matemática e Computação Científica

Projeto da disciplina MS728

Sudoku: uma abordagem por programação inteira.

Autores:
Bryan Alves Do Prado Raimundo - RA 195171
Flávia Godo Tartare - RA 108175
Thiago Felipe Castro Carrenho - RA 224831
Vinicius Jameli Cabrera - RA 225414

Resumo

É apresentado um modelo de programação inteira binária para a resolução de *Sudokus* tradicionais e diagonais quaisquer de dimensão 9×9, ambos implementados em MATLAB, além uma breve descrição do método de otimização *Branch and bound* e dos parâmetros do solver utilizado. Por fim, a análise de convergência dos problemas para casos de *Sudokus* com múltiplas soluções e para *Sudokus* sem solução.

23 de junho de 2021

Sumário

Lista de Figuras

1 Introdução

"Suuji wa dokushinn ni kagiru", numa tradução livre do Japonês, "Os número devem ser únicos", não foi o primeiro nome do famoso quebra-cabeça mundialmente conhecido como *Sudoku*; um acrônimo do nome anterior. O nome sugere que sua origem tenha se dado no país do sol nascente, mas não se engane! A primeira aparição do *puzzle* foi no final dos anos 70 na cidade de Nova York, sob o título *"Number Place"* projetado por Howard Garns. E então, só em 1984 a empresa Japonesa *Nikoli* tomou conhecimento do jogo e resolveu levá-lo àquele país; batizando-o com o nome nipônico.

Apesar de sua popularidade no Japão, o *Sudoku* passou para a escala mundial quando foi publicado no fim de 2004 pelo jornal britânico *The times*. Chamando a atenção não só dos jogadores casuais, mas sim daqueles que pouco estão interessados em resolver o enigma no papel: os matemáticos.

O *Sudoku* tradicional é composto de uma grade com 9 linhas e 9 colunas, um quadrado de 9×9 células, dividas em 9 regiões; cada uma com 9 células, quadrados 3×3 de células. Inicialmente, a maioria das células está vazia, enquanto as outras células estão preenchida com os algarismos de 1 à 9 formando o *Sudoku* inicial. Um exemplo de cada pode ser visto nas figuras 1a e 1b.

O objetivo do jogo é tentar é completar todas as células em branco inserindo estes números em cada uma. Todavia, existem três restrições para a colocação; formando as regras do jogo:

1. Toda linha contém todos os algarismos de 1 a 9.
2. Toda coluna contém todos os algarismos de 1 a 9.
3. Toda região contém todos os algarismos de 1 a 9.

				8			2	
					7	3		4
		5						9
	4	9	2					
	3		1			8	6	
5			8	2				6
	9			5			1	
		4		1				2

(a) Exemplo de um *Sudoku* tradicional 9×9

4	7	8	6	3	2	5	9	1
9	5	3	4	8	1	6	2	7
6	2	1	5	9	7	3	8	4
1	6	5	3	7	8	2	4	9
8	4	9	2	6	5	1	7	3
7	3	2	1	4	9	8	6	5
5	1	7	8	2	4	9	3	6
2	9	6	7	5	3	4	1	8
3	8	4	9	1	6	7	5	2

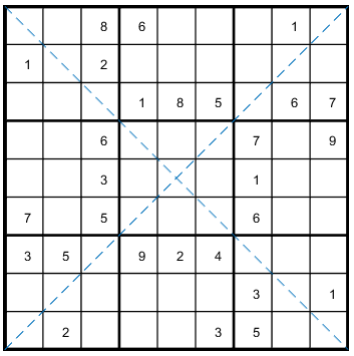
(b) Solução do *Sudoku* tradicional ao lado.

Figura 1: Exemplo de *Sudoku* tradicional

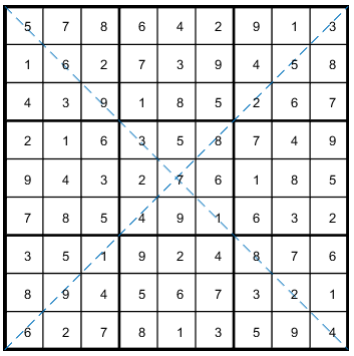
Em geral, qualquer *Sudoku* $n \times n$ com $\{m \in \mathbb{N} | n = m^2\}$ ainda é válido. Não é a toa que existe uma grande variedade de *Sudokus*, diferenciando-se em tamanho e na adição de novas regras, com todos eles facilmente modelados matematicamente. Um exemplo é o *Sudoku* diagonal, que além das três restrições acima, contém a restrição de que "ambas as diagonais contêm todos os algarismos de 1 a 9."

Perceba que a solução do *Sudoku* no exemplo acima, Figura 1b, essa nova restrição não é satisfeita, pois existem algarismos repetidos na mesma diagonal. Além disso, o *Sudoku* da Figura 1a tem solução única, mas nem todos gozam de tal propriedade, alguns não têm solução alguma, outros aceitam mais de uma solução. Desta forma, ao incluir a restrição da diagonal ao *Sudoku* da Figura 1a, o mesmo não teria solução.

Um exemplo de *Sudoku* diagonal é apresentado abaixo, no qual as linhas tracejadas azuis ajudam a perceber as diagonais.



(a) Exemplo de um *Sudoku* diagonal 9×9



(b) Solução do *Sudoku* diagonal ao lado.

Figura 2: Exemplo de *Sudoku* diagonal

Nas seguintes seções, é feita a formulação deste problema utilizando programação inteira binária, visando a solução do mesmo apenas por factibilidade, além de uma descrição do método de resolução do problema inteiro *Branch and bound* e detalhes das implementações computacionais feitas.

Mas antes de prosseguir, vale ressaltar que: de fato, já existem implementações que calculam explicitamente a solução do *Sudoku* sem necessariamente passarem por um problema de otimização. E que sem sombra de dúvidas são mais eficientes, tanto em rapidez, quanto em uso de memória. No entanto, estas necessitam de uma experiência muito maior no jogo para serem feitas, uma vez que a lógica para que a solução seja encontrada é transformada em código. Em contrapartida, o modelo de otimização apenas incorpora as regras básicas do jogo, não requisitando uma experiência prévia.

2 Modelo matemático

2.1 *Sudoku* Tradicional

Tome $a(i,j)$ o valor da célula na linha i e na coluna j, por definição, $a(i,j) \in A$, onde $A = \{1, 2, \cdots, 9\}$, e se $a(i,j) = k$ isso implica que as outras células da linha, da coluna e da região de $a(i,j)$ devem assumir valores diferentes de k, isto é, uma única célula assume esse valor em cada linha, coluna e região.

Isto sugere uma abordagem binária: ou $a(i,j) = k$ ou $a(i,j) \neq k$, então tome a variável binária de decisão $x(i,j,k)$ definida abaixo.

$$x(i,j,k) = \begin{cases} 0 & a(i,j) \neq k \\ 1 & a(i,j) = k \end{cases} \tag{1}$$

Como $a(i,j)$ deve assumir um único valor, cria-se a primeira restrição, que obriga a uma $x(i,j,k)$, variando k, ser 1 e todas as outras, 0.

$$\sum_{k=1}^{9} x(i,j,k) = 1 \quad \forall\, i,j \in A. \tag{2}$$

Perceba que, com essa definição, pode-se achar o valor de $a(i,j)$ por:

$$a(i,j) = \sum_{k=1}^{9} k x(i,j,k). \tag{3}$$

Por ter $i,j,k \in A$, as variáveis de decisão formam uma matriz $9 \times 9 \times 9$, ilustre essa matriz como camadas, em que o *Sudoku* está na superfície superior, e cada camada desse cubo representa um

valor de 1 a 9 (cada camada é um valor de k distinto), uma coluna referente a uma célula de valor 5 no *Sudoku* estará preenchida na camada 5 e vazia em todas as outras camadas.

Fazendo essa ilustração para o *Sudoku* (resolvido) da Figura 1b, obtém-se a seguinte representação em camadas.

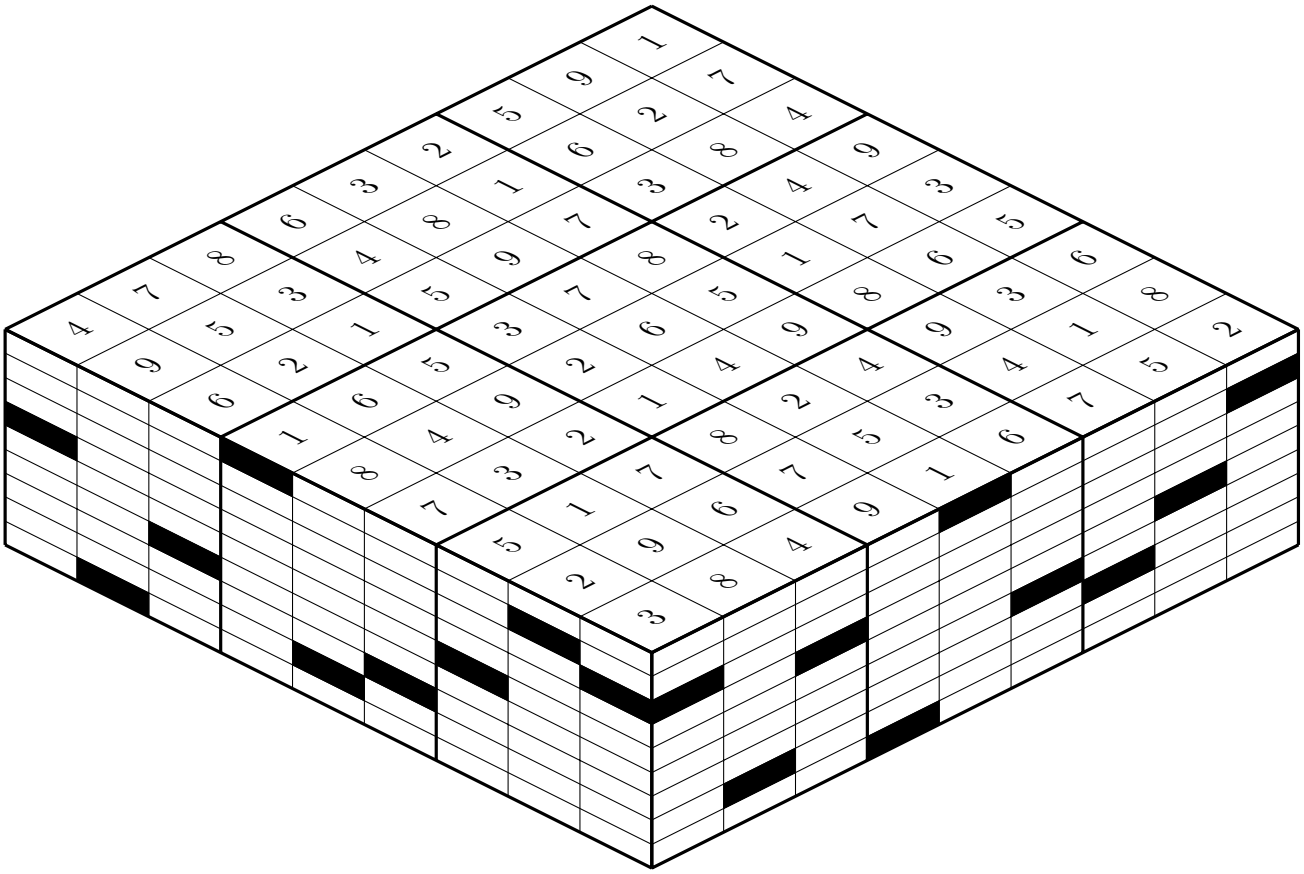


Figura 3: Ilustração do *Sudoku* como 9 camadas com células binárias

Perceba que cada corte paralelo ao lado mostrado à esquerda ressaltará a visão de uma coluna do *Sudoku*, e cada corte paralelo ao lado mostrado à direita destacará a visão de uma linha do *Sudoku*. Além disso, é possível ver a veracidade da primeira restrição, equação (2), pois cada célula tem apenas uma camada preenchida sob ela.

Tome agora uma linha i, com i fixo, tem-se que, para cada k, há apenas um $a(i,j) = k$, isto é, apenas um $x(i,j,k) = 1$. Como isto acontece para toda linha, chega-se à nossa segunda restrição para o problema:

$$\sum_{j=1}^{9} x(i,j,k) = 1 \quad \forall\ i,k \in A, \tag{4}$$

que modela a regra que trata das linhas do *Sudoku*.

Ilustrativamente, olhando a linha do *Sudoku* representada na Figura 3 (o corte mostrado do lado direito), confirma-se esta restrição ao mostrar que para cada camada há apenas uma célula preenchida.

De forma análoga, encontra-se a restrição que modela a regra referente à coluna, agora para cada coluna j e valor k há apenas uma célula $x(i,j,k) = 1$, que nos confere a restrição:

$$\sum_{i=1}^{9} x(i,j,k) = 1 \quad \forall\ j,k \in A, \tag{5}$$

que possui a mesma interpretação na ilustração, apenas alterando a visão para a coluna, isto é, o corte mostrado do lado esquerdo.

Por fim, dentre as regras, só falta modelar a referente às regiões. Olhe separadamente para uma delas, a que contém a célula $a(1,1)$, contém também toda célula $a(i,j)$ para todo $i,j \in \{1,2,3\}$, neste caso a restrição é

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x(i,j,k) = 1 \ \ \forall\ k \in A. \tag{6}$$

Agora olhe separadamente para a região da célula $a(4,8)$, que contém toda célula $a(i,j)$ para $i \in \{4,5,6\}, j \in \{7,8,9\}$, ou então, reescrevendo, toda célula $a(i+3, j+6)$ para todo $i,j \in \{1,2,3\}$. Com isso pode-se unir as restrições de todas as regiões na seguinte restrição:

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x(i+u, j+v, k) = 1 \ \ \forall\ k \in A,\ \ u,v \in \{0,3,6\}. \tag{7}$$

Dessa forma, tem-se o conjunto de restrições formado pelas equações (2), (4), (5) e (7), mas, qual é a função objetivo que o problema busca otimizar? Um *Sudoku* bem feito deve ter solução única, isto é, o espaço factível conteria apenas uma solução inteira, solução esta que deve ser encontrada independentemente da função objetivo que for colocada.

Entretanto, uma função objetivo bem escolhida pode ajudar a encontrar a solução de maneira mais rápida, por exemplo, destruindo a simetria inerente do problema. Para isto, use a seguinte função objetivo:

$$\text{Minimizar } Z = \sum_{i=1}^{9}\sum_{j=1}^{9}\sum_{k=1}^{9} kx(i,j,k). \tag{8}$$

Assim, o modelo do nosso problema binário para *Sudokus* tradicionais é:

$$\begin{aligned}
&\text{Minimize } Z = \sum_{i=1}^{9}\sum_{j=1}^{9}\sum_{k=1}^{9} kx(i,j,k) \\
&\text{Sujeito a:} \begin{cases}
\sum_{k=1}^{9} x(i,j,k) = 1 & \forall\ i,j \in A \\
\sum_{i=1}^{9} x(i,j,k) = 1 & \forall\ j,k \in A \\
\sum_{j=1}^{9} x(i,j,k) = 1 & \forall\ i,k \in A \\
\sum_{i=1}^{3}\sum_{j=1}^{3} x(i+u, j+v, k) = 1 & \forall\ k \in A,\ u,v \in \{0,3,6\} \\
x(i,j,k) = 1 & \forall\ i,j,k \text{ tal que } a(i,j) = k \text{ no } \textit{Sudoku} \text{ inicial} \\
x \in \mathbb{B}^{9\times 9\times 9}
\end{cases}
\end{aligned} \tag{9}$$

2.2 *Sudoku* Diagonal

Ao trabalhar com *Sudoku* diagonal, incluem-se as restrições das diagonais principal e secundária. As células da diagonal principal são as $a(i,i)$ com $i \in A$ e as células da diagonal secundária são as $a(i, 10-i)$ com $i \in A$.

Sendo assim, as restrições são para garantir que as somas das variáveis binárias $x(i,i,k)$ e $x(i, 10-i, k)$ ambas são 1, obtendo-as como:

$$\sum_{i=1}^{9} x(i,i,k) = 1 \ \ \forall\ k \in A, \tag{10}$$

$$\sum_{i=1}^{9} x(i, 10-i, k) = 1 \quad \forall\ k \in A. \tag{11}$$

E adicionando as restrições (10) e (11) ao modelo do problema para *Sudokus* tradicionais, (9), obtém-se a formulação do modelo do problema binário para *Sudokus* diagonais, apresentada a seguir.

$$\text{Minimize } Z = \sum_{i=1}^{9}\sum_{j=1}^{9}\sum_{k=1}^{9} kx(i,j,k)$$

$$\text{Sujeito a:} \begin{cases} \sum_{k=1}^{9} x(i,j,k) = 1 & \forall\ i,j \in A \\ \sum_{i=1}^{9} x(i,j,k) = 1 & \forall\ j,k \in A \\ \sum_{j=1}^{9} x(i,j,k) = 1 & \forall\ i,k \in A \\ \sum_{i=1}^{3}\sum_{j=1}^{3} x(i+u, j+v, k) = 1 & \forall\ k \in A,\ u,v \in \{0,3,6\} \\ \sum_{i=1}^{9} x(i,i,k) = 1 & \forall\ k \in A \\ \sum_{i=1}^{9} x(i,10-i,k) = 1 & \forall\ k \in A \\ x(i,j,k) = 1 & \forall\ i,j,k \text{ tal que } a(i,j) = k \text{ no } \textit{Sudoku} \text{ inicial} \\ x \in \mathbb{B}^{9\times 9\times 9} \end{cases} \tag{12}$$

3 Branch and Bound

3.1 Método Branch and Bound

Os algoritmos de *Branch-and-Bound* surgiram na década de 50, sendo um método enumerativo na busca por soluções usadas em problemas de otimização combinatórias, mas os pesquisadores perceberam sua eficiência nos resultados tornando-se o algoritmo de otimização mais utilizado.

O método, também chamado de $B\&B$, em que o termo *branch* refere-se ao fato de que o método efetua partições no espaço das possíveis soluções, já o termo *bound* diz que a prova da otimalidade da solução utiliza-se limites calculados ao longo da enumeração, tem como função ser um método enumerativo em que o problema original é repetidamente decomposto em subproblemas menores até que a solução seja obtida.

Este método é operacionalizado em, basicamente, cinco passos. Começa-se resolvendo o problema de programação inteira relaxado (PIR), se a solução ótima for inteira, esta é a solução do problema inteiro (PI), caso contrário, a solução do PIR é o *limite superior* da solução ótima do PI. Em seguida, escolhe-se uma variável de decisão fracionária em z^* do PIR criando dois subproblemas sendo o número inteiro menor que esse número fracionário definindo uma ramificação e o número inteiro maior que esse número fracionário definirá outra ramificação, assim criando subproblemas menores que podem ser resolvidos até chegar ao valor ótimo. O terceiro passo é a definição do método de resolver essa árvore hierárquica formada pelos subproblemas.Esse processo é repetido até todos os nós da árvore serem podados e chega-se ao ponto ótimo.

Os nós podem ser podados de três formas diferentes, sendo otimalidade, limitante ou infactibilidade até podar todos os nós sobrando a solução ótima.

3.2 Método Branch and Bound no *Sudoku*

No caso dos *Sudokus* o modelo tem apenas restrições de igualdade, sendo assim, a relaxação linear não necessitará do uso de variáveis de folga, pois obtém-se 279 variáveis extremamente restritivas, em que cada restrição define um hiperplano em $\mathbb{R}^{279}$. Portanto, a região factível do PI será um único ponto inteiro, sendo a única solução para o *Sudoku*. Entretanto, esse ponto pode não ser o único ponto factível do PIR, já que existirão pontos factíveis que não são inteiros, considerando apenas o caso comum do *Sudoku* que possui solução única.

O caso da relaxação usada para o *Sudoku* é feito para um problema binário, o que significa que os resultados que pode-se encontrar para as variáveis, que são de apenas 0 ou 1 para o problema original, se tornam de entre 0 e 1 para o problema relaxado. Sendo assim, os cortes feitos por $B\&B$ são feitos nas variáveis não inteiras do ponto ótimo, assim, serão adicionadas restrições de ≤ 0 ou ≥ 1 nos problemas filho.

Unindo essa restrição com a restrição de ter variáveis entre zero e um possuem desigualdades, podemos considerar os cortes do $B\&B$ como cortes de igualdade.

Ao supor que uma variável é igual a 1, olhando a Figura 3 nota-se que, por exemplo, na camada 3 a variável que se refere à linha e também sua coluna da célula com valor 3 está preenchida , e todas as outras variáveis envolvendo sua linha, coluna e região terão que ser 0, ou seja, estarão vazias por aceitar apenas uma posição como 3. Além disto, todos os outros espaços de mesma linha e mesma coluna, mas nas camadas diferentes devem estar vazios, já que a célula aceita apenas um único valor, sendo o 3.

Isto é, se $x(i', j', k') = 1$, temos que $x(i, j', k') = 0$, $x(i', j, k') = 0$ e $x(i', j', k) = 0$ para todo $i \neq i'$, $j \neq j'$ e $k \neq k'$, além de que para toda célula (i, j) na região de (i', j'), que não seja o próprio (i', j'), tem $x(i, j, k') = 0$. Isto ocorre para obedecer as restrições derivadas das regras básicas citadas acima.

Dessa forma, o próprio problema supõe 28 restrições iguais a 0 para apenas aquela restrição de valor 1 totalizando, portanto, 29 restrições utilizadas.

Uma particularidade do problema em questão é que sua função objetivo terá o mesmo valor de igualdade de 405, o que remete que nenhum nó será podado por limitante, já que tem-se para toda solução factível o mesmo valor para a função objetivo. Portanto, todos os outros nós serão podados por infactibilidade, sendo a única exceção o nó com a solução ótima que será podado por otimalidade.

4 Implementação

4.1 Considerações iniciais

As implementações foram feitas em *MATLAB* utilizando uma rotina nativa do mesmo para o problema do *Sudoku*. Detalhes do programa podem ser encontrados em MATLAB [2014], e a versão adaptada utilizada pelos autores pode ser encontrada nos arquivos enviados pelos nomes *solver_Mathworks.mat* e *drawSudoku.mat*.

Detalhes sobre como as rotinas funcionam estão na forma de comentários nos algoritmos, e mais exemplos de *Sudokus* com diferentes níveis de dificuldade e suas respectivas soluções podem ser encontrados no site https://sudokuonline.pt/ de forma gratuita.

4.2 Detalhes do Solver

Na rotina *solver_Mathworks.mat*, é utilizado o comando nativo *solve* do *MATLAB*, que identifica qual o tipo de otimização em questão. Uma vez identificado que se trata de um problema de otimização inteira, é escolhido o solver *intlinprog*.

Cada solver disponibiliza parâmetros diferentes que podem ser alterados, de forma a modificar o processo de otimização. Alguns dos principais parâmetros padrões do solver *intlinprog* que foram utilizados na resolução do problema são explicitados abaixo:

- *AbsoluteGapTolerance* - Diferença entre o limitante superior e inferior: 0.
- *BranchRule* - Regra para a escolha da ramificação: 'reliability', escolhe o valor fracionário com maior custo relativo.
- *ConstraintTolerance* - Discrepância que as restrições lineares podem ter e ainda se considerarem satisfeitas: 10^{-4}
- *Heuristics* - busca de pontos iniciais factíveis: 'basic'
- *IntegerTolerance* - Discrepância que uma componente do vetor solução x pode ter para ser considerada um número inteiro: 10^{-5}
- *LPMaxIterations* - O número máximo de iterações do simplex por nó do branch and bound: 30000
- *LPPreprocess* - Tipo de pré-processamento para a resolução do PL relaxado: 'basic'
- *MaxTime* - Tempo máximo de execução: 2 horas
- *RootLPAlgorithm* - Algoritmo para a resolução dos PL's: 'dual-simplex'

Para mais informações, visite o site https://www.mathworks.com/help/optim/ug/intlinprog.html#btv2x05.

Felizmente, a relevância destes parâmetros para resolver o problema é ínfima, pois o mesmo é pequeno, com apenas 729 variáveis binárias, e converge à solução num tempo médio de aproximadamente 0,04 segundos.

4.3 Análise do Algoritmo

Para a análise de eficiência do algoritmo, toma-se uma rotina de testes onde um dado *Sudoku*, retirado do site https://sudokuonline.pt/, é resolvido e comparado o resultado do site com o resultado obtido pelo algoritmo, então é constatado que os resultados são o mesmo. Com isso, o processo foi repetido cinco vezes, registrando o tempo em cada um e também o número de células iniciais dadas como dica. Por fim, toma-se como tempo geral do processo a média dos cinco registros, apresentados nas tabelas abaixo em segundos.

O processo que acabou de ser citado foi realizado para as seis dificuldades de *Sudokus* tradicionais presentes no site, variando de muito fácil até diabólico, e para as três dificuldades de *Sudokus* diagonais, fácil, moderado e difícil. As entradas e resultados finais estão presentes nos próximos tópicos. O resultado destes testes são explicitados nas tabelas abaixo:

Tradicional Muito Fácil	Tradicional Fácil	Tradicional Moderado
0.022601	0.022545	0.033084
0.020929	0.022825	0.027578
0.020740	0.020138	0.021619
0.022368	0.021435	0.036784
0.023484	0.024388	0.022260
Média: 0.022024	Média: 0.022266	Média: 0.028265
Células: 26	Células: 25	Células: 26

Tradicional Difícil	Tradicional Muito Difícil	Tradicional Diabólico
0.022640	0.028269	0.048750
0.043245	0.024706	0.036977
0.022738	0.023595	0.036404
0.024505	0.025881	0.035928
0.022104	0.025679	0.035878
Média: 0.027046	Média: 0.025626	Média: 0.038787
Células: 25	Células: 23	Células: 24

Diagonal Fácil	Diagonal Moderado	Diagonal Difícil
0.023209	0.027615	0.054306
0.025747	0.026992	0.055879
0.023605	0.024569	0.054765
0.023108	0.026499	0.054270
0.024914	0.024281	0.053400
Média: 0.024117	Média: 0.025991	Média: 0.054524
Células: 50	Células: 26	Células: 19

Com os resultados obtidos, observa-se que o algoritmo é altamente efetivo, e obtém com total êxito os resultados buscados em um curto espaço de tempo. Para fins de registro, a máquina utilizada para os testes é um Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz com 8GB de RAM e sem nenhum outro programa rodando simultaneamente ao *MATLAB*.

4.4 Exemplos de *Sudokus* Tradicionais

9			5				8	
7			9	8				
	9			3		7		
			8	7		6	9	
	8		2				3	1
2		3	1					6
6						5		3
					6	8		

(a) Exemplo de *Sudoku* de dificuldade muito fácil.

9	6	1	5	2	4	3	8	7
7	2	5	9	8	3	1	6	4
8	3	4	6	1	7	2	5	9
1	9	6	4	3	5	7	2	8
3	4	2	8	7	1	6	9	5
5	8	7	2	6	9	4	3	1
2	5	3	1	4	8	9	7	6
6	1	8	7	9	2	5	4	3
4	7	9	3	5	6	8	1	2

(b) Solução do *Sudoku* ao lado.

7			4	3			9	5
		5			6			4
						1	7	
		2				3	4	
		7						6
1				6				
	4							
							6	3
6	1	3			2	9		

(a) Exemplo de *Sudoku* de dificuldade fácil.

7	2	1	4	3	8	6	9	5
9	8	5	7	1	6	2	3	4
4	3	6	9	2	5	1	7	8
8	6	2	1	5	7	3	4	9
3	9	7	2	8	4	5	1	6
1	5	4	3	6	9	7	8	2
5	4	9	6	7	3	8	2	1
2	7	8	5	9	1	4	6	3
6	1	3	8	4	2	9	5	7

(b) Solução do *Sudoku* ao lado.

							9	
	1		3					
6		3					8	2
		2	6		1	9	7	
		1		2		8		4
		9	8					
9							4	
2		4	1			7	6	3

(a) Exemplo de *Sudoku* de dificuldade moderada.

5	2	7	4	8	6	3	9	1
4	1	8	3	9	2	6	5	7
6	9	3	5	1	7	4	8	2
8	4	2	6	3	1	9	7	5
7	6	1	9	2	5	8	3	4
3	5	9	8	7	4	2	1	6
9	7	5	2	6	3	1	4	8
1	3	6	7	4	8	5	2	9
2	8	4	1	5	9	7	6	3

(b) Solução do *Sudoku* ao lado.

3							1	8
	4			3		2		
5			8		1		9	
	9	2						7
				1	9			
8		1	2		5			4
			1	8				
			4					
				9				2

(a) Exemplo de *Sudoku* de dificuldade difícil.

3	2	9	5	7	4	6	1	8
1	4	8	9	3	6	2	7	5
5	6	7	8	2	1	4	9	3
6	9	2	3	4	8	1	5	7
4	3	5	7	1	9	8	2	6
8	7	1	2	6	5	9	3	4
2	5	6	1	8	7	3	4	9
9	8	3	4	5	2	7	6	1
7	1	4	6	9	3	5	8	2

(b) Solução do *Sudoku* ao lado.

						3		6
6					9		1	
		4	7				2	
2								
							4	7
1	7				2	5		
					4		7	1
		6	2				9	
				3	1			

(a) Exemplo de *Sudoku* de dificuldade muito difícil.

7	9	1	4	2	5	3	8	6
6	5	2	3	8	9	7	1	4
8	3	4	7	1	6	9	2	5
2	4	5	6	9	7	1	3	8
9	6	8	1	5	3	2	4	7
1	7	3	8	4	2	5	6	9
3	2	9	5	6	4	8	7	1
5	1	6	2	7	8	4	9	3
4	8	7	9	3	1	6	5	2

(b) Solução do *Sudoku* ao lado.

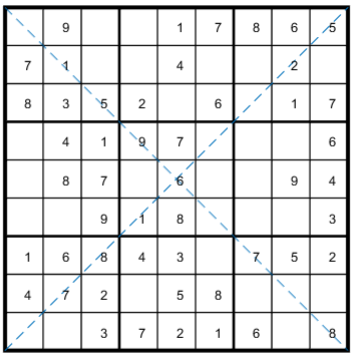
3								5
9		4	8	6				
						4	8	
			1				3	6
		8						4
						1	7	
4				9		5		3
	6			7				8
		9			3			

(a) Exemplo de *Sudoku* de dificuldade diabólica.

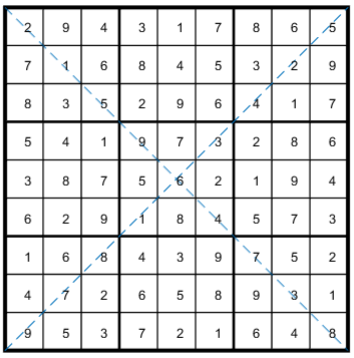
3	8	7	9	1	4	6	2	5
9	5	4	8	6	2	3	1	7
2	1	6	5	3	7	4	8	9
7	9	2	1	4	5	8	3	6
1	3	8	7	2	6	9	5	4
6	4	5	3	8	9	1	7	2
4	7	1	2	9	8	5	6	3
5	6	3	4	7	1	2	9	8
8	2	9	6	5	3	7	4	1

(b) Solução do *Sudoku* ao lado.

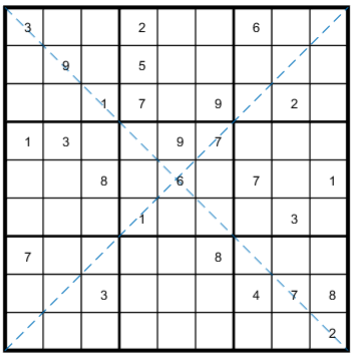
4.5 Exemplos de *Sudokus* Diagonais



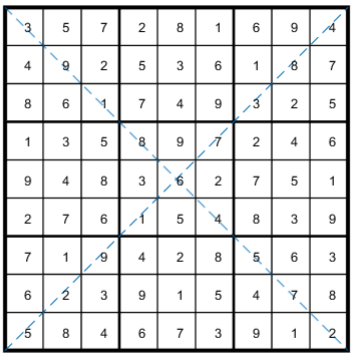
(a) Exemplo de *Sudoku* de dificuldade fácil.



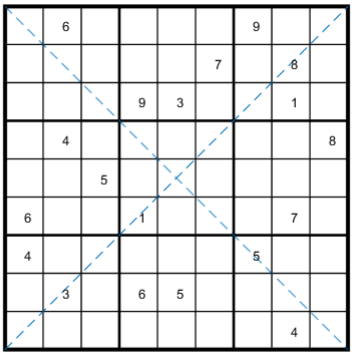
(b) Solução do *Sudoku* ao lado.



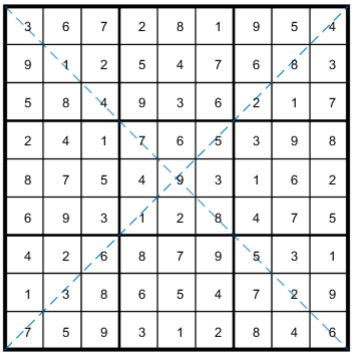
(a) Exemplo de *Sudoku* de dificuldade moderada.



(b) Solução do *Sudoku* ao lado.



(a) Exemplo de *Sudoku* de dificuldade difícil.



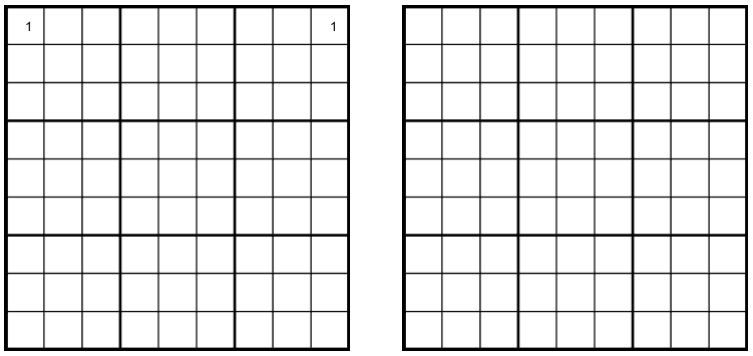
(b) Solução do *Sudoku* ao lado.

4.6 Múltiplas soluções

Um *Sudoku* bem montado deve ter solução única para que o jogador tenha uma resolução prazerosa, mas nem sempre esse é o caso, existem inúmeros *Sudokus* em que se aceita mais de uma solução para dadas condições iniciais, ou ainda, casos em que nenhuma solução é aceita, estes casos são apresentados abaixo.

4.6.1 Casos infactíveis

O algoritmo devolve um *Sudoku* vazio quando ele é infactível, como pode-se conferir no caso abaixo, Figura 4a, que claramente fere a restrição da linha com dois valores 1 na primeira linha, recebe resposta vazia, Figura 4b.

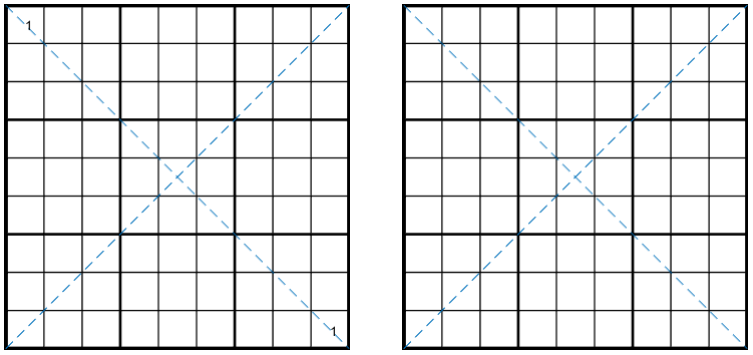


(a) *Sudoku* tradicional infactível

(b) Resposta do algoritmo

Figura 4: *Sudoku* tradicional infactível

Vê-se que o mesmo ocorre no caso do *Sudoku* Diagonal, o exemplo da Figura 5a é factível com as regras tradicionais, mas se torna infactível com a restrição da diagonal, por ter dois elementos 1 na diagonal principal, e também recebe como resposta do algoritmo um *Sudoku* vazio, como mostra a Figura 5b.



(a) *Sudoku* diagonal infactível

(b) Resposta do algoritmo

Figura 5: *Sudoku* diagonal infactível

4.6.2 Caso Vazio

Foi também realizado o experimento em que não foi dada nenhuma condição inicial, vazio, no qual todo *Sudoku* que é possível de ser montado, é factível, as respostas deste caso são mostradas na figura abaixo.

Respostas tais que não aparentam nenhuma escolha específica do algoritmo por alguma ordem crescente ou decrescente.

5	4	2	6	9	8	1	3	7
9	1	8	3	4	7	6	5	2
6	7	3	2	5	1	9	4	8
1	5	7	8	6	3	4	2	9
4	8	6	9	2	5	3	7	1
2	3	9	7	1	4	8	6	5
7	9	4	5	8	6	2	1	3
8	6	5	1	3	2	7	9	4
3	2	1	4	7	9	5	8	6

(a) *Sudoku* solução para o *Sudoku* vazio tradicional

9	3	6	1	2	7	8	4	5
5	2	8	9	6	4	1	3	7
4	1	7	5	3	8	9	6	2
1	7	4	3	9	6	2	5	8
3	6	5	8	1	2	4	7	9
2	8	9	7	4	5	3	1	6
7	5	2	4	8	1	6	9	3
6	4	3	2	5	9	7	8	1
8	9	1	6	7	3	5	2	4

(b) *Sudoku* solução para o *Sudoku* vazio diagonal

Figura 6: *Sudokus* solução para o *Sudoku* vazio

4.6.3 Comparando o caso tradicional com o diagonal

Como o problema para o caso diagonal tem todas as restrições do tradicional com a inclusão de mais duas restrições, o conjunto de soluções factíveis do caso diagonal é um subconjunto do conjunto de soluções factíveis do caso tradicional.

Isso mostra casos como o da Figura 5a, que, apesar de infactível para o caso diagonal, tem solução (e inúmeras soluções) para o caso tradicional, e a resposta encontrada pelo algoritmo para este caso é apresentada na Figura 7 abaixo.

1	4	9	7	6	5	2	8	3
8	6	7	3	9	2	5	1	4
3	5	2	4	1	8	9	6	7
5	9	8	6	7	3	1	4	2
6	2	1	8	4	9	7	3	5
4	7	3	5	2	1	8	9	6
7	1	6	9	5	4	3	2	8
2	8	5	1	3	6	4	7	9
9	3	4	2	8	7	6	5	1

Figura 7: Solução do *Sudoku* diagonal infactível com as restrições do caso tradicional

Outro fenômeno que acontece é o *Sudoku* "perder a graça", isto é, ter uma solução única no caso diagonal, o que confere uma resolução agradável ao jogador, mas inúmeras soluções no caso tradicional. Tome, por exemplo, o *Sudoku* da Figura 2a que tem solução única no caso, mas quando se coloca as mesmas condições iniciais com as restrições tradicionais obtém-se a resolução abaixo.

5	7	8	6	4	2	9	1	3
1	6	2	3	7	9	4	8	5
4	3	9	1	8	5	2	6	7
2	1	6	5	3	8	7	4	9
9	4	3	2	6	7	1	5	8
7	8	5	4	9	1	6	3	2
3	5	1	9	2	4	8	7	6
8	9	4	7	5	6	3	2	1
6	2	7	8	1	3	5	9	4

Figura 8: Solução do *Sudoku* diagonal exemplo, agora com as restrições do caso tradicional

Perceba que este, Figura 8, é diferente da solução apresentada na Figura 2b, pois esta tem elementos repetidos em ambas as diagonais.

4.6.4 Função objetivo

Tomando em consideração que existem casos com múltiplas soluções e que o algoritmo devolve apenas uma, fica a dúvida: se houver mais de uma solução factível, qual o algoritmo escolhe? E por que esta é a escolhida?

O problema em questão é de minimização, então é esperado que seja escolhida a solução com a menor função objetivo, porém, a função objetivo em questão chega ao mesmo valor para todo *Sudoku* factível, 405, pois, como se tem para cada k, 9 células com esse valor, obtém-se

$$Z = \sum_{i=1}^{9}\sum_{j=1}^{9}\sum_{k=1}^{9} kx(i,j,k) = \sum_{k=1}^{9} k \times 9 = 405.$$

Então, como a função objetivo é constante para os *Sudokus*, a maneira que o algoritmo escolhe a função ótima é de acordo com a forma que ele faz as escolhas no *Branch-and-Bound*, de qual filho analisar primeiro, de se encontrar outro resultado com mesma função objetivo ele escolhe a anterior ou o novo resultado, e de como ramificar.

Neste caso, é até possível incluir que ao encontrar um ponto ótimo, forçar o método a escolhê-lo como solução e parar, pois já se sabe que ele será uma solução factível do *Sudoku*, e terá o mesmo valor funcional dos outros *Sudokus*.

Entretanto, se outra função objetivo é escolhida, pode-se privilegiar uma solução específica em detrimento de outra, por exemplo, se alterar a função objetivo para

$$Z = -x(1,1,1),$$

como é um problema de minimização, as soluções que tenham $a(1,1) = 1$ serão privilegiadas em relação às outras.

Lembrando que isto não é uma restrição, mas sim uma preferência: se houver duas ou mais soluções, ele escolhe a que possui $a(1,1) = 1$, se nenhuma delas tiver isto, o programa escolhe qualquer outra, sem preferência.

Veja isto ocorrer na prática, tomando o *Sudoku* abaixo, Figura 9, que tem duas soluções claras, basta escolher 1 ou 2 em uma célula, que as opções para as outras células são únicas, completando, assim, o *Sudoku*.

		6	9	5	8	4	3	7
8	7	5	6	4	3	9	1	2
3	4	9	1	2	7	8	6	5
4	8	1	2	9	5	3	7	6
9	3	7	8	1	6	2	5	4
6	5	2	3	7	4	1	8	9
		8	7	6	9	5	4	3
7	9	4	5	3	1	6	2	8
5	6	3	4	8	2	7	9	1

Figura 9: *Sudoku* tradicional com exatamente duas soluções

Então, pode-se escolher as funções objetivo

$$Z_1 = -x(1,1,1)$$

ou

$$Z_2 = x(1, 1, 1),$$

e, para cada função, uma solução ótima é obtida, como mostra a Figura 10, abaixo.

1	2	6	9	5	8	4	3	7
8	7	5	6	4	3	9	1	2
3	4	9	1	2	7	8	6	5
4	8	1	2	9	5	3	7	6
9	3	7	8	1	6	2	5	4
6	5	2	3	7	4	1	8	9
2	1	8	7	6	9	5	4	3
7	9	4	5	3	1	6	2	8
5	6	3	4	8	2	7	9	1

(a) *Sudoku* solução com a função objetivo Z_1

2	1	6	9	5	8	4	3	7
8	7	5	6	4	3	9	1	2
3	4	9	1	2	7	8	6	5
4	8	1	2	9	5	3	7	6
9	3	7	8	1	6	2	5	4
6	5	2	3	7	4	1	8	9
1	2	8	7	6	9	5	4	3
7	9	4	5	3	1	6	2	8
5	6	3	4	8	2	7	9	1

(b) *Sudoku* solução com a função objetivo Z_2

Figura 10: Soluções distintas para funções objetivo distintas

5 Conclusão

De um ponto de vista da atuação do algoritmo, podemos concluir que o mesmo sempre dará a solução correta como resposta (quando a mesma existir, claro) para o caso tradicional e para o caso diagonal, e em relação a questão de tempo de execução, conseguimos observar que o algoritmo é consideravelmente rápido, variando em um geral entre 0,02 e 0,05 segundos, sendo destacável os resultados obtidos para o caso tradicional com dificuldade diabólica e o caso diagonal com dificuldade difícil, em relação a esses, podemos concluir que a complexidade da resolução acaba por fazer com que o número de processos executados pelo computador no método Branch-and-Bound é maior, o que se traduz em um maior tempo para a obtenção da resolução.

Observa-se que o número de células iniciais oferecidas como dica não tem uma grande interferência no processo de resolução por parte do algoritmo, e que as dificuldades em ambos os casos de *Sudoku* (excluindo-se o tradicional diabólico e o difícil diagonal) não interferem no tempo de resolução do algoritmo, elas são principalmente dificultadores para a resolução manual do *Sudoku*, o algoritmo não é afetado pelas mesmas.

Conclui-se também que, no caso de múltiplas soluções, o algoritmo não tem nenhum tipo de preferência, e o *Sudoku* a ser escolhido depende de como são feitas as ramificações e os cortes no método Branch-and-Bound. Porém, alterando a função objetivo, é possível dar preferência a uma ou a outra solução. Mas estes casos são exceções, o comum é ter uma única solução, ou então o *Sudoku* é tratado como "quebrado", além de ser decepcionante resolver o puzzle até chegar a um estágio que qualquer opção é válida, sem critério para alcançar a opção certa.

Referências

Andrew Bartlett, Timothy Chartier, Amy Langville, and Timothy Rankin. An integer programming model for the sudoku problem. 04 2008.

MATLAB. Solve sudoku puzzles via integer programming: Problem-based, 2014. URL https://www.mathworks.com/help/optim/ug/sudoku-puzzles-problem-based.html.