Resolvendo Sudokus Generalizados através de algoritmos de *backtracking* e de *Dancing Links*

Allan K. B. S. da Cruz[1], **João D. de S. Almeida**[1]

[1]Programa de Pós-Graduação Doutorado em Ciência da Computação
Associação UFMA-UFPI (DCCMAPI)
Universidade Federal do Manhão (FMA) – São Luís, MA – Brazil

allankassio@gmail.com, joao.dallyson@ufma.br

Abstract. *This paper presents an implementation for solving generalized Sudoku puzzles using backtracking and Dancing Links algorithms. Two algorithms were implemented, one using the backtracking technique and the other using the Dancing Links technique. The algorithms were tested on Sudokus of different sizes (9x9, 16x16 and 25x25) and the execution time was measured. The results show that the Dancing Links algorithm is more efficient than the backtracking algorithm for solving generalized Sudoku puzzles.*

Resumo. *Este trabalho apresenta uma implementação da resolução de sudokus generalizados através de algoritmos de backtracking e Dancing Links. Foram implementados dois algoritmos, um utilizando a técnica de backtracking e outro utilizando a técnica de Dancing Links. Os algoritmos foram testados em diferentes tamanhos de sudoku (9x9, 16x16 e 25x25) e o tempo de execução foi medido. Os resultados mostraram que o algoritmo de Dancing Links é mais eficiente que o algoritmo de backtracking para a resolução de sudokus generalizados.*

1. Introdução

O Sudoku é um jogo de lógica que consiste em um quadro 9x9 dividido em subquadros 3x3, no qual cada subquadro contém os dígitos de 1 a 9 sem repetição e cada linha e coluna do quadro também contém os dígitos de 1 a 9 sem repetição [Felgenhauer and Jarvis 2006]. O objetivo do jogo é preencher os quadros com os dígitos de 1 a 9 de tal forma que cada dígito apareça apenas uma vez em cada linha, coluna e subquadro. O tabuleiro inicia com alguns espaços previamente preenchidos (geralmete com inteiros de 1 a n). A Figura 1 mostra uma intância do Sudoku 9x9 e uma possível solução.

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

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

Figure 1. Grid Sudoku 9x9 com solução. Adaptado de: [Dreamstime 2018]

A história dos quebra-cabeças de Sudoku provavelmente tem suas raízes no conceito matemático de quadrados latinos [McKay and Wanless 2005]. Eles são um tipo de quadrado mágico, uma tabela quadrada contendo números naturais, tal que a soma dos números em cada linha, coluna, diagonal e quadrado mágico principal é a mesma. A Figura 2 mostra um exemplo de um quadrado latino de ordem 4.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Figure 2. Quadrado latino de ordem 4. Adaptado de: [Nogueira 2015]

Os quadrados latinos foram introduzidos pela primeira vez pelo matemático suíço Leonhard Euler em 1782. Em 1892, Euler apresentou o problema de determinar se existe um quadrado latino de ordem 9 [Cota 2011]. Esse problema permaneceu sem resposta até 1915, quando o matemático estadunidense Sam Loyd publicou um quadrado latino de ordem 9 em sua revista americana de matemática, Mathematical Puzzles [Costa 2014].

No entanto, Loyd afirmou que esse quadrado latino de ordem 9 era um problema de ”brinquedo”, e não uma demonstração de um teorema matemático. O tabuleiro de Sudoku que conhecemos hoje, no entanto, é uma prova de um teorema matemático. Em 1979, o matemático japonês Tetsuya Nishio e seus colegas demonstraram que existe um número infinito de tabuleiros de Sudoku que satisfazem as propriedades do jogo [Tsao 2019].

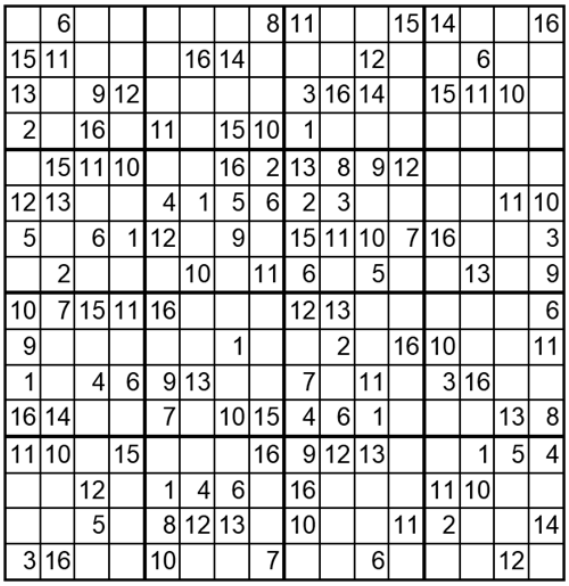
Assim pode-se dizer que o Sudoku é um jogo relativamente novo. O nome do jogo é uma contração da frase em japonês suuji wa dokushin ni kagiru, que significa ”os números devem estar sozinhos”. O jogo ganhou popularidade nos Estados Unidos nos anos 2000 e, desde então, tem sido um dos jogos de lógica mais populares do mundo [Maia 2012].

1.1. Sudoku Generalizado

Um sudoku generalizado é um sudoku em que as dimensões da grade de jogo são ajustáveis [Haynes 2008]. Isso significa que, em vez de uma grade 9x9, um sudoku generalizado pode ser uma grade de qualquer tamanho, como 16x16 (Figura 3), 25x25 ou 36x36. Além disso, as regras podem ser ajustadas para permitir que os jogadores usem qualquer número de 1 a 9, ou até mesmo letras ou símbolos diferentes.

O Sudoku generalizado é um problema de otimização NP-completo, o que significa que, em princípio, não existe um algoritmo que possa resolver o problema de forma eficiente para todos os possíveis casos [Maji and Pal 2014]. Isto é devido à natureza de exponencial do problema, uma vez que aumentando o tamanho da grade do Sudoku, aumenta-se exponencialmente o número de possíveis soluções.

Um problema NP-difícil é um problema para o qual não existe um algoritmo de polynomial para resolvê-lo. Isso significa que, se um problema é NP-difícil, então não



	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

Figure 3. Grid Sudoku 16x16. Adaptado de: [Schmidl et al. 2014]

existe um algoritmo que possa resolvê-lo em tempo polinomial [Rai et al. 2018]. No entanto, existem algoritmos que podem resolver problemas NP-difíceis em tempo subpolinomial, mas esses algoritmos são considerados ineficientes [Yildirim et al. 2008]. Para tentar aproximar-se do resultado ideal são utilizadas heurísticas para prover soluções.

Uma heurística é uma técnica que busca uma solução aproximada para um problema de otimização NP-completo, de forma a encontrar uma solução que seja ”suficientemente boa”. Não existe uma heurística única que seja capaz de resolver todos os problemas de Sudoku, uma vez que cada problema é único e pode requerer uma abordagem diferente [Takano et al. 2015].

Uma heurística comum para o Sudoku é a de tentar preencher as células vazias com os valores que restam, de forma a minimizar o número de valores que faltam para completar a grade [Dittrich et al. 2010]. Outra heurística é a de procurar as células que têm o menor número de valores possíveis e tentar preenchê-las com esses valores [Takano et al. 2015].

Existem muitas outras heurísticas que podem ser usadas para resolver problemas de Sudoku, e é importante experimentar várias delas para encontrar aquela que melhor se adapta ao problema em questão.

2. Algoritmos de Resolução de Sudoku Generalizado

Existem vários algoritmos diferentes que podem ser usados para resolver um Sudoku Generalizado, mas a maioria dos algoritmos segue os mesmos princípios básicos. O primeiro passo é determinar o tamanho do quadrado e, em seguida, preencher as linhas e colunas com os números de 1 a N. Em seguida, é necessário determinar os blocos e, finalmente, preenchê-los com os números restantes.

Uma vez que o quadrado estiver completamente preenchido, é possível verificar se a solução é correta, comparando-a com as regras do Sudoku Generalizado. Se a solução

estiver correta, é possível parar o algoritmo. Se a solução não estiver correta, é necessário verificar se há algum erro na solução e, se houver, corrigi-lo.

No entanto a solução para um Sudoku se enquandra como um problema da cobertura exata. Que é um problema de otimização combinatória que trata de encontrar um subconjunto de um determinado conjunto tal que os elementos do subconjunto cubram todos os elementos do conjunto dado, e dois elementos do subconjunto não cubram o mesmo elemento do dado conjunto. definir [Kapanowski 2010]. O problema de cobertura exata é um exemplo de problema de satisfação de restrição. Na maioria dos casos, quando são instâncias de solução possíveis, são resolvidos por algoritmos recursivos com backtracking. Dentre os algoritmos que resolvem o sudoku e se encaixam na cobertura exata podemos destacar os seguintes:

- **Minimum Remaining Values:** defende que e mais viável selecionar variáveis com menos possibilidades de valores [Abuluaih et al. 2018]. Ou seja, para o Sudoku, significa selecionar a celula com menos candidatos de inserção para toda vez que for necessário procurar uma nova celula.
- **Forward Checking:** propõe o término quando finaliza a busca caso alguma variável ainda não testada não tenha valores dentro de seu domínio [Simonis 2005]. Para o caso do Sudoku, toda vez que o Backtracking encontrar uma célula em que não será possível inserir mais nenhum valor, a busca é finalizada e testada para um próximo estado [Skiena 2008].
- **Manipulação de Bits:** técnica utilizada para representação das estruturas de dados do Sudoku e manipulação das unidades (linha, coluna e bloco) com consulta em tempo constante [Takano et al. 2015]. Implementa o modelo teorico de coloração em hipergrafos, aplicando-se a ideia de que as unidades são hiperarestas, e assim o acesso para verificação de um dígito nas unidades durante a execução do método é O(1). Para que a Manipulação de Bits ocorra é necessário que se guarde todos os dígitos pre-preenchidos do tabuleiro do Sudoku em vetores. Cada dígito do tabuleiro é representado por um algarismo de um numero presente no vetor. A verificação é a remoção de dígitos e feita através dos vetores criados.
- **Constraint Propagation:** uma técnica que remove valores de um domínio de variaveis da qual não iráo participar de nenhuma solução [Russell 2010]. Enquanto o algoritmo for executado atraves da busca em profundidade, a técnica irá remover os candidatos do tabuleiro da qual seriam impossíveis de ser inseridas.
- **Dancing Links:** técnica proposta por Donald Knuth [knu 2000], também conhecida como DLX, para implementar de forma eficiente seu algoritmo X. O algoritmo X e um tipo de backtracking com podas, uma busca em profundidade recursiva nao-determinística, que encontra todas as soluões para realizar a cobertura exata do problema.

Assim neste trabalho foi escolhido implementar o algoritmo de Dancing Links e paralelamente, a título de comparação de performance, um algoritmo trivial baseado backtracking.

2.1. Backtracking

Como todos os outros problemas de Backtracking, o Sudoku pode ser resolvido passo a passo, atribuindo números às células vazias. Antes de atribuir um número, devemos

verificar se podemos atribuir. Verifica-se se o mesmo número não está presente na linha atual, coluna atual e subgrade atual. Depois de verificar a segurança, devmos atribuir um número e verificar recursivamente se essa atribuição leva a uma solução ou não. Se a atribuição não levar a uma solução, devemos tentar o próximo número para a célula vazia atual. E se nenhum dos números (1 a 9) levar a uma solução, devemos retornar falso e imprimir que não existe solução (nesse caso, provavelmente há um problema com a grade original gerada).

É necessário criar uma função que verifique, após atribuir o índice atual, se a grade se torna insegura ou não. Devemos manter um hashmap para uma linha, coluna e caixas. Se algum número tiver uma frequência maior que 1 no hashmap, retornamos falso, caso contrário, retornamos verdadeiro (o hashmap pode ser evitado usando loops).

Em resumo, o algoritmo segue essa ordem:

1. Crie uma função recursiva que recebe uma grade.
2. Verifique se há algum local não atribuído.
3. Se presente, atribua um número de 1 a n^2, verifique se atribuir o número ao índice atual torna a grade insegura ou não.
4. Se for segura, chame recursivamente a função para todos os casos seguros de 0 a n^2.
5. Se qualquer chamada recursiva retornar true, finaliza o loop e retornar true.
6. Se nenhuma chamada recursiva retornar true, então retorne false.
7. Se não houver local não atribuído, retorne true.

2.1.1. Análise de Complexidade do Backtraking

Complexidade de tempo: $O((n^2)^{(n*n)})$. Para cada índice não atribuído, existem n^2. opções possíveis, então a complexidade de tempo é $O((n^2)^{(n*n)})$. A complexidade de tempo permanece a mesma, mas haverá algumas podas iniciais, de modo que o tempo gasto será muito menor do que o algoritmo ingênuo, mas a complexidade de tempo do limite superior permanece a mesma.

Complexidade espacial: $O(n * n)$, pois para armazenar a saída é necessária uma estrutura do tipo matriz.

2.2. Dancing Links

Programas para resolver o Sudoku, geralmente caem no problema da cobertura total ou cobertura exata. Esse tipo de problema pode ser resolvido com o algoritmo conhecido como “Algoritmo X” [knu 2000].

Dada uma coleção S de subconjuntos do conjunto X, uma cobertura exata é o subconjunto S* de S tal que cada elemento de X contido é exatamente um subconjunto de S* [knu 2000]. Deve satisfazer as seguintes duas condições:

- A interseção de quaisquer dois subconjuntos em S* deve estar vazia. Ou seja, cada elemento de X deve estar contido em no máximo um subconjunto de S*
- A união de todos os subconjuntos em S* é X. Isso significa que a união deve conter todos os elementos do conjunto X. Então podemos dizer que S* cobre X.

Exemplo (representação padrão):

Seja

$$S = \{ A, B, C, D, E, F \}$$

$$e$$

$$X = \{1, 2, 3, 4, 5, 6, 7\}$$

tal que:

$A = \{1, 4, 7\}$, $B = \{1, 4\}$, $C = \{4, 5, 7\}$, $D = \{3, 5, 6\}$, $E = \{2, 3, 6\ 7\}$, $F = \{2, 7\}$.

Então $S^* = \{B, D, F\}$ é uma cobertura exata, porque cada elemento em X está contido exatamente uma vez nos subconjuntos $\{B, D, F\}$. Se unirmos subconjuntos, obteremos todos os elementos de X:

$$B \bigcup D \bigcup F = \{1, 2, 3, 4, 5, 6, 7\}$$

O problema da cobertura exata é um problema de decisão para determinar se a cobertura exata existe ou não. É considerado um problema NP-Completo. O problema pode ser representado na forma de uma matriz onde a linha representa os subconjuntos de S e as colunas representam o elemento de X.

2.2.1. Algoritmo X

O Algoritmo X foi proposto visando poder encontrar todas as soluções para o problema de cobertura exata. O Algoritmo X pode ser implementado eficientemente pela técnica de Dancing Links (ou links dançantes), proposta pelo mesmo autor, chamada DLX [knu 2000].

É um algoritimo recursivo, primeiro em profundidade, e um algoritmo que usa o conceito de backtraking. É de natureza não determinística, o que significa que, para a mesma entrada, pode exibir comportamentos diferentes em uma execução diferente.

A seguir está o pseudocódigo para o Algoritmo X:

1. Se a matriz A não tiver colunas, a solução parcial atual é uma solução válida; terminar com sucesso.
2. Caso contrário, escolha uma coluna c (deterministicamente).
3. Escolha uma linha r tal que A[r] = 1 (não deterministicamente).
4. Inclua a linha r na solução parcial.
5. Para cada coluna j tal que A[r][j] = 1,
    (a) para cada linha i tal que A[i][j] = 1,
        i. exclua a linha i da matriz A.
    (b) exclua a coluna j da matriz A.
6. Repita este algoritmo recursivamente na matriz reduzida A.
7. Escolha não determinística de r significa, o algoritmo copia a si mesmo no subalgoritmo. Cada subalgoritmo herda a matriz original A, mas a reduz em relação ao r escolhido (veremos isso em breve no exemplo)

O subalgoritmo forma uma árvore de busca com o problema original na raiz e cada nível k tem um subalgoritmo que corresponde às linhas escolhidas no nível anterior (assim como o espaço de busca) [knu 2000].

Se a coluna escolhida C for totalmente zero, então não há subalgoritmos e o processo terminou sem sucesso. [knu 2000] sugere que devemos escolher a coluna com o número mínimo de 1s. Se não sobrar nenhuma coluna, então sabemos que encontramos nossa solução.

A técnica do Dancing Links baseia-se na ideia de lista encadeada duplamente circular. Conforme discutido por [knu 2000], transforma-se o problema de cobertura exata em forma de matriz de 0 e 1. Aqui cada "1" na matriz é representado por um nó de lista encadeada e toda a matriz é transformada em uma malha de nós conectados de 4 vias. Cada nó contém os seguintes campos:

- Ponteiro para o nó à esquerda dele
- Ponteiro para o nó direito a ele
- Ponteiro para o nó acima dele
- Ponteiro para o nó abaixo dele
- Ponteiro para listar o nó do cabeçalho ao qual pertence

Cada linha da matriz é, portanto, uma lista circular vinculada entre si com ponteiros para a esquerda e para a direita e cada coluna da matriz também será uma lista circular vinculada a cada uma acima e abaixo com os ponteiros para cima e para baixo. Cada lista de colunas também inclui um nó especial chamado "nó de cabeçalho de lista". Este nó de cabeçalho é como um nó simples, mas tem poucos campos extras:

- Código da coluna
- Contagem de nós na coluna atual

Podemos ter dois tipos diferentes de nós, um para as colunas que possui o "atributo tamanho" e um para os nós "dançantes", que possui o atributo que identifica o "cabeçalho da coluna".

3. Conclusão

Como dito anteriormente as soluções escolhidas foram a de Backtracking e de Dancing Links. Cada um desses algoritmos foi implementado como uma classe separada. Adicionamente foram implementadas classes para o Iterador, para o Nó, para representar a Matriz do Sudoku, para gerar o sudoku a partir dos datasets e para validação do sudoku. Os códigos do sistema estão disponíveis no Anexo deste trabalho e no repositório GitHub: `https://github.com/allankassio/generalized-sudoku-solver`.

3.1. Base de dados utilizada

A base de dados utilizada consiste em quatro aqruivos CSV que modelam e representam os puzzles do soduko como linhas únicas. Cada coluna é composta por dois dígitos (permitindo assim casas maiores do que 9). Para cada tamanho N do rank representado, os arquivos possuem $2(N^2)$ dígitos por linha.

Para tabuleiros de 9x9 a base de dados possui 3000 entradas. Para 16x16, 25x25 e 36x36 são 10 entradas para cada um. Já 49x49 possui 6 entradas, enquanto 64x64 tem 5 entradas e 81x81 tem apenas 1 entrada. A Figura 4 mostra um exemplo de 3 entradas de dados do tipo 9x9.

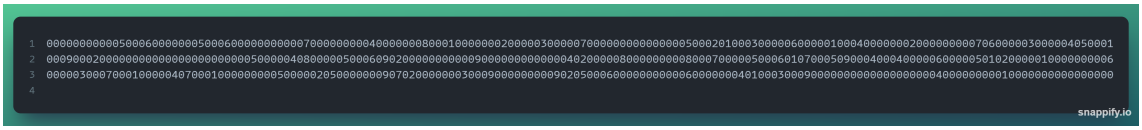


Figure 4. Exemplo de base de dados utilizada.

3.2. Exemplos de saídas

As saídas são compostas do número de iterações necessárias para resolver o sudoku, o tempo levado para a resolução e se aquela resposta é válida. A Figura 5 mostra um exemplo de saídas no console.

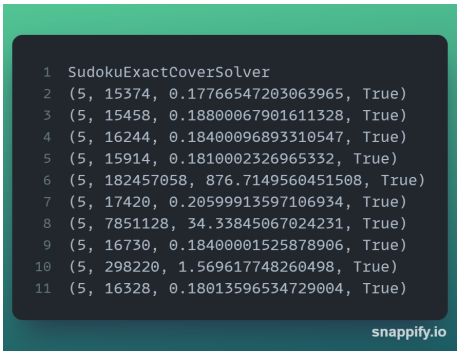


Figure 5. Exemplo de saída no console para entradas com n=5.

3.3. Comparação dos algoritmos

Para comparar o algoritmo de backtracking com o de Dancing Links foram utilizados os seguintes critérios: comparação da média dos 10 primeiros resultados para matrizes 9x9, 16x16 e 25x25 tanto tem termo de iterações quanto em termo de tempo médio. Ambos algoritmos utilizaram a mesma base de dados como entrada. Na Figura 6 podemos vemos a comparação para n=3. Na Figura 7 podemos vemos a comparação para n=4. Na Figura 8 podemos vemos a comparação para n=5. Os dados do Backtracking para n=4 e n=5 foram inviáveis, e por isso não foram gerados.

Dancing Links		
3	2112	0,005000352859497070
3	1992	0,004998445510864250
3	2154	0,005249023437500000
3	2160	0,003999710083007810
3	2010	0,003999471664428710
3	2106	0,004999637603759760
3	2070	0,003999948501586910
3	2244	0,005005598068237300
3	2118	0,005000352859497070
3	2166	0,006001472473144530

Backtracking		
3	170126	10,657027721405000000
3	130052	7,676674604415890000
3	3079	0,203737020492553000
3	47673	3,103725910186760000
3	3106	0,192025184631347000
3	46140	2,963670492172240000
3	16683	1,085922479629510000
3	40344	2,709635019302360000
3	87861	5,951405286788940000
3	329971	22,379093408584500000

N	Iterações	Tempo
3	**2113,2**	**0,004825401306152340**

N	Iterações	Tempo
3	**87503,5**	**5,692291712760910000**

Figure 6. Comparação de desempenho com n=3.

O algoritmo de Backtracking apresenta um crescimento exponencial muito rápido a medida que aumentamos o valor de n. Em contrapartida, o Dancing Links, mesmo também apresentando crescimento exponencial, apresenta para a grande maioria dos casos tempos abaixo de 1 segundo.

Dancing Links		
4	36114	0,115999698638916000
4	223784	0,669000148773193000
4	35288	0,108999967575073000
4	11212	0,045999765396118100
4	6802	0,035000085830688400
4	6248	0,033999443054199200
4	137750	0,371982574462890000
4	5780	0,034000158309936500
4	25254	0,085000276565551700
4	6440	0,033999919891357400

N	Iterações	Tempo
4	**49467,2**	**0,153398203849792000**

Figure 7. Comparação de desempenho com n=4.

Dancing Links		
5	15374	0,1776654720306390
5	15458	0,1880006790161130
5	16244	0,1840009689331050
5	15914	0,1810002326965330
5	182457058	876,7149560451500000
5	17420	0,2059991359710690
5	7851128	34,3384506702423000
5	16730	0,1840000152587890
5	298220	1,5696177482604900
5	16328	0,1801359653472900

N	Iterações	Tempo
5	**19071987,4**	**91,392382693290600000**

Figure 8. Comparação de desempenho com n=5.

Com isso é possível perceber que Dancing Links pode ser mais eficiente que backtracking para resolver o sudoku, dependendo da implementação, dependendo do tamanho do sudoku e da entropia da entrada. Dancing links consegue resolver sudoku de forma eficiente porque ao contrário de backtracking, dancing links não precisa fazer muitas buscas caso existam várias soluções. Isso é possível pois ele utiliza listas encadeadas circulares para representar os dados.

References

(2000). Dancing links. Number: arXiv:cs/0011047 arXiv:cs/0011047.

Abuluaih, S., Mohamed, A., Annamalai, M., and Iida, H. (2018). Fog of Search Resolver for Minimum Remaining Values Strategic Colouring of Graph. In *International Conference on Soft Computing in Data Science*, pages 201–215. Springer.

Costa, O. d. (2014). *A matemática recreativa no ensino Básico*. PhD Thesis.

Cota, A. C. d. S. (2011). Euler e os números pentagonais. Master's thesis, Universidade Federal do Rio Grande do Norte.

Dittrich, M., Preußer, T. B., and Spallek, R. G. (2010). Solving Sudokus through an Incidence Matrix on an FPGA. In *2010 International Conference on Field-Programmable Technology*, pages 465–469. IEEE.

Dreamstime (2018). Sudoku com solução livra para usar se.

Felgenhauer, B. and Jarvis, F. (2006). Mathematics of Sudoku I.

Haynes, I. W. (2008). *Analysis of generalized sudoku puzzles: A mixture of discrete techniques*. PhD Thesis, University of South Carolina.

Kapanowski, A. (2010). Python for education: the exact cover problem. *arXiv preprint arXiv:1010.5890*.

Maia, P. A. d. A. (2012). Jogos na aprendizagem matemática: Uma proposta com Sudokus, Malba Tahan e Tangram. Publisher: Matemática.

Maji, A. K. and Pal, R. K. (2014). Sudoku solver using minigrid based backtracking. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 36–44. IEEE.

McKay, B. D. and Wanless, I. M. (2005). On the Number of Latin Squares. *Annals of Combinatorics*, 9(3):335–344.

Nogueira, R. (2015). Quadrados mágicos.

Rai, D., Ingle, M., and Chaudhari, N. (2018). POLYNOMIAL 3-SAT REDUCTION OF SUDOKU PUZZLE. *International Journal of Advanced Research in Computer Science*, 9(3).

Russell, P. N. (2010). Artificial Intelligence: A Modern Approach by Stuart. *Russell and Peter Norvig contributing writers, Ernest Davis...[et al.]*.

Schmidl, D., Terboven, C., an Mey, D., and Müller, M. S. (2014). Suitability of Performance Tools for OpenMP Task-Parallel Programs. In Knüpfer, A., Gracia, J., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2013*, pages 25–37. Springer International Publishing, Cham.

Simonis, H. (2005). Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer.

Skiena, S. (2008). The Algorithm Design Manual. Springer Publishing Company.

Takano, K., de Freitas, R., and de Sá, V. G. P. (2015). O jogo de lógica Sudoku: modelagem teórica, NP-completude, algoritmos e heurısticas. *XXXIV Concurso de Trabalhos de Iniciação Científica da Sociedade Brasileira de Computação*.

Tsao, H.-p. (2019). EVOLUTIONARY MATHEMATICS AND ARTS FOR TALK ELEGANCY INVESTIGATION. *Evolutionary Progress in Science, Technology, Engineer-*

ing, Arts, and Mathematics (STEAM), pages 1–68. Publisher: Lenox Institute Press, Newtonville, NY, 12128-0405, USA.

Yildirim, H., Krishnamoorthy, M., and Deo, N. (2008). A study of the Sudoku graph family. In *Proceedings of the Thirty-Ninth Southeastern International Conference on Combinatorics, Graph Theory and Computing. Congr. Numer*, volume 192, pages 85–95.

ANEXOS

Listing 1. main.py

```python
import csv
import time

from app.SudokuBackTrackingSolver import SudokuBackTrackingSolver
from app.SudokuExactCoverSolver import SudokuExactCoverSolver
from app.SudokuGenerator import SudokuGenerator
from app.SudokuValidator import SudokuValidator


def run_exact_cover_solver():
    for n in range(3, 6):
        print(f'n={n}')
        main_solver(
            solver=SudokuExactCoverSolver,
            n=n,
            write_to_csv=True
        )


def run_backtracking_solver():
    for n in range(3, 6):
        print(f'n={n}')
        main_solver(
            solver=SudokuBackTrackingSolver,
            n=n,
            write_to_csv=True
        )


def main_solver(solver, n, num_puzzles=None, write_to_csv=False):

    print(f'{solver.__name__}')
    results = [('num_updates', 'time_taken', 'is_valid')]

    sudoku_generator = SudokuGenerator(n)
    sudoku_matrices = sudoku_generator.generate_puzzles(num_puzzles)

    for sudoku_matrix in sudoku_matrices:
        num_updates, time_taken = solve_puzzle(solver, sudoku_matrix)
        sudoku_validator = SudokuValidator(sudoku_matrix)
        is_valid = sudoku_validator.validate()
        print((n, num_updates, time_taken, is_valid))
        results.append((num_updates, time_taken, is_valid))

    if write_to_csv:
        write_data_to_csv(results, f'output/sudoku_rank_{n}_{solver.__name__}.csv')
    else:
        [print(r) for r in results]

    return results


```

```
def solve_puzzle(solver, sudoku_matrix):
    s = solver(sudoku_matrix)

    start = time.time()
    s.solve()
    end = time.time()
    num_updates = s.get_num_updates()
    return num_updates, end - start


def write_data_to_csv(results, csv_file):
    with open(csv_file, 'w') as csv_file:
        writer = csv.writer(csv_file)
        [writer.writerow([r for r in result]) for result in results]


def main():
    # Rodar esses dois m todos vai permitir executar
    # todos os testes dos ranks 3 at  5 (9x9, 16x16 e 25x25)
    # ============================================================
    # run_exact_cover_solver()
    # run_backtracking_solver()

    # Rodar dessa forma permite definir qual dataset vai
    # ser utilizado apenas mudando o N, desde que n>=3 e n<=9.
    # Para cada valor de n, o tamanho da matriz s e r  de n**2
    # ============================================================
    # main_solver(solver=SudokuBackTrackingSolver, n=4, write_to_csv=
    True)
    main_solver(solver=SudokuExactCoverSolver, n=5, write_to_csv=True)
    main_solver(solver=SudokuBackTrackingSolver, n=4, write_to_csv=True
    )
    main_solver(solver=SudokuBackTrackingSolver, n=5, write_to_csv=True
    )

if __name__ == "__main__":
    main()
```

Listing 2. Node.py

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.up = None
        self.down = None
        self.row_id = None
        self.column_id = None


class DancingNode(Node):
    def __init__(self, value):
        super().__init__(value)
        self.column_header = None


class ColumnNode(Node):
    def __init__(self, value):
        super().__init__(value)
        self.size = 0
```

Listing 3. Iterator.py

```
class LeftIterable(object):

    def __init__(self, node):
        self.node = node
        self.original_node = node

    def __iter__(self):
        return self

    def __next__(self):
        self.node = self.node.left

        if self.node == self.original_node:
            raise StopIteration

        return self.node


class RightIterable(object):

    def __init__(self, node):
        self.node = node
        self.original_node = node

    def __iter__(self):
        return self

    def __next__(self):
        self.node = self.node.right

        if self.node == self.original_node:
```

```
            raise StopIteration

        return self.node


class UpIterable(object):

    def __init__(self, node):
        self.node = node
        self.original_node = node

    def __iter__(self):
        return self

    def __next__(self):
        self.node = self.node.up

        if self.node == self.original_node:
            raise StopIteration

        return self.node


class DownIterable(object):

    def __init__(self, node):
        self.node = node
        self.original_node = node

    def __iter__(self):
        return self

    def __next__(self):
        self.node = self.node.down

        if self.node == self.original_node:
            raise StopIteration

        return self.node
```

Listing 4. SudokuMatrix.py

```
from tabulate import tabulate


class SudokuMatrix:

    def __init__(self, n):
        self.n = n  # classifica o (rank)
        self.k = 0  # n mero de pistas (n meros preenchidos)
        self.sudoku_matrix = [[0 for _ in range(n ** 2)] for _ in range
    (n ** 2)]
        self.EMPTY_CELL = 0

    def __str__(self):
        return tabulate(self.get_rows(), tablefmt="fancy_grid")

    def get(self, row, column):
        return self.sudoku_matrix[row][column]

    def get_rows(self):
        return self.sudoku_matrix

    def get_row(self, row):
        return self.sudoku_matrix[row]

    def get_columns(self):
        return [*zip(*self.sudoku_matrix)]

    def get_column(self, column):
        return self.get_columns()[column]

    def get_boxes(self):
        indices = [(x % self.n, int(x / self.n)) for x in range(self.n
    ** 2)]
        boxes = [[self.sudoku_matrix[x * self.n + xx][y * self.n + yy]
    for xx, yy in indices] for x, y in indices]
        return boxes

    def get_box(self, row, column):
        box_index = self.get_box_index(row, column)
        return self.get_boxes()[box_index]

    def get_box_index(self, row, column):
        indices = [(x % self.n, int(x / self.n)) for x in range(self.n
    ** 2)]

        xx = row % self.n
        yy = column % self.n

        x = (row - xx) / self.n
        y = (column - yy) / self.n

        return indices.index((x, y))

    def get_rank(self):
        return self.n
```

```
52
53      def get_num_clues(self):
54          return self.k
55
56      def get_empty_cells(self):
57          empty_cells = []
58
59          for row in range(self.n ** 2):
60              for column in range(self.n ** 2):
61                  if self.is_empty_cell(row, column):
62                      empty_cells.append((row, column))
63
64          return empty_cells
65
66      def has_empty_cells(self):
67          for row in range(self.n ** 2):
68              for column in range(self.n ** 2):
69                  if self.is_empty_cell(row, column):
70                      return True
71
72          return False
73
74      def is_empty_cell(self, row, column):
75          return self.get(row, column) == self.EMPTY_CELL
76
77      def set(self, row, column, value):
78          self.sudoku_matrix[row][column] = value
79
80          if value != self.EMPTY_CELL:
81              self.increment_num_clues()
82
83      def set_if_valid(self, row, column, value):
84          if self.is_valid(row, column, value):
85              self.set(row, column, value)
86              return True
87
88          return False
89
90      def make_cell_empty(self, row, column):
91          self.set(row, column, self.EMPTY_CELL)
92
93      def increment_num_clues(self):
94          self.k += 1
95
96      def is_valid(self, row, column, value):
97          if value in self.get_row(row):
98              return False
99
100         if value in self.get_column(column):
101             return False
102
103         if value in self.get_box(row, column):
104             return False
105
106         return True
```

Listing 5. SudokuValidator.py

```
from app.SudokuMatrix import SudokuMatrix


def contains_duplicates(arr):
    if len(set(arr)) != len(arr):
        return True

    return False


class SudokuValidator:

    # :type sudoku_matrix: SudokuMatrix
    def __init__(self, sudoku_matrix):

        self.sudoku_matrix = sudoku_matrix

    def validate(self):

        fully_filled = not self.sudoku_matrix.has_empty_cells()
        validated_rows = self._validate_rows()
        validated_columns = self._validate_columns()
        validated_boxes = self._validate_box()

        return fully_filled and validated_rows and validated_columns
    and validated_boxes

    def _validate_rows(self):
        for row in self.sudoku_matrix.get_rows():
            if contains_duplicates(row):
                return False

        return True

    def _validate_columns(self):
        for column in self.sudoku_matrix.get_columns():
            if contains_duplicates(column):
                return False

        return True

    def _validate_box(self):
        for box in self.sudoku_matrix.get_boxes():
            if contains_duplicates(box):
                return False

        return True
```

Listing 6. SudokuGenerator.py

```
import csv

from app.SudokuMatrix import SudokuMatrix


class SudokuGenerator:

    def __init__(self, n):
        self.n = n
        self.csv_file = f'datasets/sudoku_rank_{self.n}.csv'

    def generate_puzzles(self, num_puzzles=None):
        puzzles = self._read_sudoku_csv(num_puzzles)
        return [self._convert_string_to_matrix(puzzle) for puzzle in
    puzzles]

    def generate_puzzle_from_id(self, i):
        with open(self.csv_file) as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            sudoku_puzzle = list(csv_reader)[i][0]

        return [self._convert_string_to_matrix(sudoku_puzzle)]

    def _read_sudoku_csv(self, num_puzzles=None):
        sudoku_puzzles = []
        with open(self.csv_file) as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            next(csv_reader, None)  # pula o cabe alho

            for i, row in enumerate(csv_reader):
                if num_puzzles and len(sudoku_puzzles) == num_puzzles:
                    break

                sudoku_puzzles.append(row[0])

        return sudoku_puzzles

    def _convert_string_to_matrix(self, sudoku_string):
        sudoku_matrix = SudokuMatrix(self.n)

        c = 0
        for i in range(self.n ** 2):
            for j in range(self.n ** 2):
                if sudoku_string[c] == '.':
                    sudoku_matrix.set(i, j, 0)
                else:
                    sudoku_matrix.set(i, j, int(sudoku_string[c:c + 2])
    )
                c += 2

        return sudoku_matrix
```

Listing 7. DancingLinks.py

```
from app.Node import DancingNode, ColumnNode


class DancingLinks:
    def __init__(self, matrix):
        self.matrix = matrix
        self._pad_matrix()

    # Atualiza a matriz de entrada adicionando cabe alhos de coluna
    # e matriz de preenchimento com 0s para mant -lo um quadrado
    perfeito
    def _pad_matrix(self):
        for row in self.matrix:
            row.insert(0, 0)

        column_headers = []
        for j in range(len(self.matrix[0])):
            if j == 0:
                # inserir n   de cabe alho
                column_headers.append('H')
            else:
                # inserir cabe alhos de coluna
                column_headers.append(f'C{j}')
        self.matrix.insert(0, column_headers)

    # M todo usado para conectar todos os  n  s usando listas
    duplamente vinculadas
    def create_dancing_links(self):
        nodes = self._create_nodes()
        self._create_links_between_nodes(nodes)

    # Converte todos os cabe alhos de coluna e c lulas com 1s em  n  s
    def _create_nodes(self):
        nodes = []
        for i in range(len(self.matrix)):
            for j in range(len(self.matrix[i])):
                value = self.matrix[i][j]
                # nada a ser feito quando    0
                if value == 0:
                    continue
                node = None
                # converter todos os 1 para DancingNode
                if value == 1:
                    node = DancingNode(value)
                # converter todos os cabe alhos de coluna para
    ColumnNode
                if value != 1 and value != 0:
                    node = ColumnNode(value)
                node.row_id = i
                node.column_id = j
                nodes.append(node)
                self.matrix[i][j] = node

        return nodes

```

```
    # Cria um link entre n s que est o conectados    esquerda,
    # direita, para cima e para baixo.
    # Al m disso, cada DancingNode    referenciado a um ColumnNode
    def _create_links_between_nodes(self, nodes):

        for node in nodes:
            node.left = self._get_left(node.row_id, node.column_id)
            node.right = self._get_right(node.row_id, node.column_id)
            # o n   de cabe alho  n o precisa de links para cima ou
    para baixo
            if node.value != 'H':
                node.up = self._get_up(node.row_id, node.column_id)
                node.down = self._get_down(node.row_id, node.column_id)
            # criar refer ncia ao cabe alho da coluna
            if node.value == 1:
                node.column_header = self._get_column_header(node.
    column_id)
                node.column_header.size += 1

    # Retorna o n      esquerda do n   em (linha, coluna)
    def _get_left(self, row, column):
        j = (column - 1) % len(self.matrix[row])
        while self.matrix[row][j] == 0:
            j = (j - 1) % len(self.matrix[row])
        return self.matrix[row][j]

    # Retorna o n      direita do n   em (linha, coluna)
    def _get_right(self, row, column):
        j = (column + 1) % len(self.matrix[row])
        while self.matrix[row][j] == 0:
            j = (j + 1) % len(self.matrix[row])
        return self.matrix[row][j]

    # Retorna o n   acima do n   em (linha, coluna)
    def _get_up(self, row, column):
        i = (row - 1) % len(self.matrix)
        while self.matrix[i][column] == 0:
            i = (i - 1) % len(self.matrix)
        return self.matrix[i][column]

    # Retorna o n   abaixo do n   em (linha, coluna)
    def _get_down(self, row, column):
        i = (row + 1) % len(self.matrix)
        while self.matrix[i][column] == 0:
            i = (i + 1) % len(self.matrix)
        return self.matrix[i][column]

    # Retorna o cabe alho da coluna do n   na coluna
    def _get_column_header(self, column):

        return self.matrix[0][column]
```

Listing 8. ExactCoverSolver.py

```
import sys

from app.Iterator import DownIterable, RightIterable, LeftIterable,
    UpIterable
from app.DancingLinks import DancingLinks


class ExactCoverSolver:

    def __init__(self, exact_cover_matrix):
        self.exact_cover_matrix = exact_cover_matrix
        DancingLinks(exact_cover_matrix).create_dancing_links()
        self.header = self.exact_cover_matrix[0][0]
        self.num_updates = 0
        self.answer = []

    # Busca do Algoritmo X
    def search(self, k, o):

        if self.header.right == self.header:
            self.answer.append(o.copy())
            return

        c = self._choose_column()
        self._cover(c)

        for r in DownIterable(c):
            o[k] = r

            for j in RightIterable(r):
                self._cover(j.column_header)

            self.search(k + 1, o)

            r = o.pop(k, None)
            c = r.column_header

            for j in LeftIterable(r):
                self._uncover(j.column_header)

        self._uncover(c)
        return

    def get_num_updates(self):
        return self.num_updates

    def get_answer(self):
        return self.answer

    # Retorna a coluna com o menor n mero de 1s.
    def _choose_column(self):

        min_size = sys.maxsize
        column_selected = None

```

```
        for c in RightIterable(self.header):
            if c.size < min_size:
                min_size = c.size
                column_selected = c

        return column_selected

    def _cover(self, c):
        self._unlinkLR(c)

        for i in DownIterable(c):
            for j in RightIterable(i):
                self._unlinkUD(j)
                j.column_header.size -= 1

    def _uncover(self, c):
        for i in UpIterable(c):
            for j in LeftIterable(i):
                j.column_header.size += 1
                self._relinkUD(j)

        self._relinkLR(c)

    def _unlinkUD(self, x):
        x.down.up = x.up
        x.up.down = x.down
        self.num_updates += 1

    def _relinkUD(self, x):
        x.down.up = x
        x.up.down = x
        self.num_updates += 1

    def _unlinkLR(self, x):
        x.right.left = x.left
        x.left.right = x.right
        self.num_updates += 1

    def _relinkLR(self, x):
        x.right.left = x
        x.left.right = x
        self.num_updates += 1
```

Listing 9. SudokuExactCoverSolver.py

```
from app.ExactCoverSolver import ExactCoverSolver


class SudokuExactCoverSolver:

    # :type sudoku_matrix: SudokuMatrix
    def __init__(self, sudoku_matrix):

        self.sudoku_matrix = sudoku_matrix
        self.n = sudoku_matrix.get_rank()
        self.exact_cover_matrix, self.possibilities = self.
    _create_exact_cover_matrix()
        self.exact_cover_solver = ExactCoverSolver(self.
    exact_cover_matrix)

    def solve(self):
        self.exact_cover_solver.search(k=0, o=dict())
        solutions = self.exact_cover_solver.get_answer()

        for solution in solutions[0].values():
            row, column, value = self.possibilities[solution.row_id -
    1]
            if self.sudoku_matrix.is_empty_cell(row, column):
                self.sudoku_matrix.set(row, column, value)

    def get_num_updates(self):
        return self.exact_cover_solver.get_num_updates()

    def _create_exact_cover_matrix(self):
        possibilities = self._create_possibilities()
        constraints = self._create_constraints()
        exact_cover_matrix = []

        for possibility in possibilities:
            m = []
            for constraint in constraints:
                m.append(self.
    _handle_possibility_constraint_combination(possibility, constraint)
    )
            exact_cover_matrix.append(m)

        return exact_cover_matrix, possibilities

    def _create_possibilities(self):
        possibilities = []

        for row in range(self.n ** 2):
            for column in range(self.n ** 2):
                if self.sudoku_matrix.is_empty_cell(row, column):
                    for i in range(1, self.n ** 2 + 1):
                        if self.sudoku_matrix.is_valid(row, column, i):
                            possibilities.append((row, column, i))
                else:
                    possibilities.append((row, column, self.
    sudoku_matrix.get(row, column)))
```

```
50
51          return possibilities
52
53      def _create_constraints(self):
54          constraints = []
55
56          # restri  o de linha-coluna
57          for row in range(self.n ** 2):
58              for column in range(self.n ** 2):
59                  constraints.append(('rc', row, column))
60
61          # restri  o de n mero de linha
62          for row in range(self.n ** 2):
63              for number in range(1, self.n ** 2 + 1):
64                  constraints.append(('rn', row, number))
65
66          # restri  o de n mero de coluna
67          for column in range(self.n ** 2):
68              for number in range(1, self.n ** 2 + 1):
69                  constraints.append(('cn', column, number))
70
71          # restri  o de n mero de caixa
72          for box in range(self.n ** 2):
73              for number in range(1, self.n ** 2 + 1):
74                  constraints.append(('bn', box, number))
75
76          return constraints
77
78      def _handle_possibility_constraint_combination(self, possibility,
    constraint):
79          row, column, value = possibility
80          constraint_type, x, y = constraint
81
82          if constraint_type == 'rc':
83              return 1 if row == x and column == y else 0
84
85          if constraint_type == 'rn':
86              return 1 if row == x and value == y else 0
87
88          if constraint_type == 'cn':
89              return 1 if column == x and value == y else 0
90
91          if constraint_type == 'bn':
92              box_index = self.sudoku_matrix.get_box_index(row, column)
93              return 1 if box_index == x and value == y else 0
```

Listing 10. SudokuBackTrackingSolver.py

```
from app.SudokuMatrix import SudokuMatrix


class SudokuBackTrackingSolver:

    # :tipo sudoku_matrix: SudokuMatrix
    def __init__(self, sudoku_matrix):

        self.sudoku_matrix = sudoku_matrix
        self.n = sudoku_matrix.get_rank()
        self.num_backtracks = 0  # contador para medir a performance do
    algoritmo

    def solve(self):

        if not self.sudoku_matrix.has_empty_cells():
            return True

        current_row, current_column = self.sudoku_matrix.
    get_empty_cells()[0]

        for i in range(1, self.n ** 2 + 1):

            if self.sudoku_matrix.set_if_valid(current_row,
    current_column, i):
                if self.solve():
                    return True

                # Caso chegue nessa parte, faz o backtracking
                self.sudoku_matrix.make_cell_empty(current_row,
    current_column)
                self.num_backtracks += 1

        return False

    def get_num_updates(self):
        return self.num_backtracks
```