

# Solucionador de Sudoku utilizando coloração em grafos

José Davi Viana Francelino  
Thiago de Oliveira Cordeiro

outubro, 2023

## Resumo

Este artigo visa apresentar uma modelagem em grafos para a resolução do jogo Sudoku e um algoritmo aproximativo, uma heurística, que soluciona o Sudoku genérico  $n \times n$ . O Sudoku é um problema NP-completo e um desafio de raciocínio lógico que é relacionado com o quadrado latino. Ademais, será utilizado a coloração de grafos simples nessa heurística, haja vista que a coloração é uma ferramenta de rotulação de grafos relevante, e, a partir dela, é possível solucionar o jogo Sudoku.

**Palavras-chaves:** grafos. coloração de grafos. sudoku.

## Introdução

O Sudoku é um popular quebra-cabeça que desafia os jogadores a preencher uma grade  $n \times n$ , com dígitos de 1 a  $n$ , com  $n = k^2$ , para algum inteiro positivo  $k$ , onde algumas posições já vem previamente preenchidas, de forma que cada linha, coluna e subgrade  $k \times k$  contenha todos os dígitos exatamente uma vez. Uma abordagem comumente utilizada para resolver o Sudoku é através de métodos exatos, como a busca exaustiva ou backtracking. No entanto, esses métodos podem ser computacionalmente intensivos, especialmente para Sudoku mais difíceis.

Uma abordagem alternativa interessante é a aplicação de técnicas de coloração em grafos para resolver o Sudoku de forma aproximativa. Na teoria dos grafos, um grafo é um conjunto de vértices e arestas, onde os vértices representam entidades e as arestas indicam conexões entre essas entidades. Ademais, a coloração de grafos é um conceito fundamental nessa área e trata-se da atribuição de cores aos vértices de um grafo de modo que vértices adjacentes não tenham a mesma cor. Neste contexto, cada célula do Sudoku pode ser mapeada como um vértice em um grafo e as arestas vão representar a adjacência entre as células, ou seja, se estão na mesma linha, coluna ou subgrade.


Este trabalho se concentra na implementação de um algoritmo aproximativo que utiliza a coloração de grafos para abordar o problema do Sudoku. A ideia é explorar a aplicação prática da teoria dos grafos para encontrar uma solução viável e eficaz para o Sudoku.

## 1 Descrição do problema

O problema abordado neste artigo consiste em gerar um solucionador de instâncias do jogo Sudoku. Por sua vez, esse jogo foi concebido por Howard Garns, um arquiteto e criador de quebra-cabeças. O Sudoku foi, possivelmente, baseado em uma estrutura matemática denominada "quadrado latino", desenvolvida pelo matemático suíço Leonhard Euler [3].

O objetivo do jogo Sudoku é encontrar uma configuração final válida, que atende as regras do jogo, para um dado tabuleiro. Isso implica em estender a grade inicial de forma a preencher todas as células em branco com números de 1 a  $n$ , garantindo que não haja repetições de números em nenhuma linha, coluna ou bloco. O Sudoku é comumente jogado em um tabuleiro  $9 \times 9$ , que é subdividido em blocos menores de  $3 \times 3$ . No início do jogo, algumas células do tabuleiro já estão preenchidas, enquanto outras permanecem em branco, desafiando os jogadores a resolver o quebra-cabeça atentando as regras do jogo. A [Figura 1] ilustra como é o jogo Sudoku e uma possível solução para a instância exibida na figura.

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



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

Figura 1 – Exemplo de uma instância de Sudoku  $9 \times 9$ .

Para modelar o Sudoku como um grafo, pode-se pensar em cada célula do tabuleiro como um vértice, onde dados dois vértices  $v$  e  $w$ , eles são adjacentes se e somente se  $v$  e  $w$  pertencem a mesma linha, mesma coluna ou mesmo bloco/subgrade. Deste modo, como existe uma aresta conectando cada vértice aos outros contidos no bloco, coluna e linha, podemos realizar a aplicação da coloração de vértices. Para cada dois vértices adjacentes, eles não são coloridos com a mesma cor. Cabe destacar que no contexto deste trabalho, de solucionar Sudoku, as cores serão representadas por números. Foi selecionado a modelagem por meio de grafo simples, porém também poderia ser modelado por hipergrafos. Na figura baixo é ilustrado a modelagem do grafo, mostrando um vértice e seus adjacentes no tabuleiro:

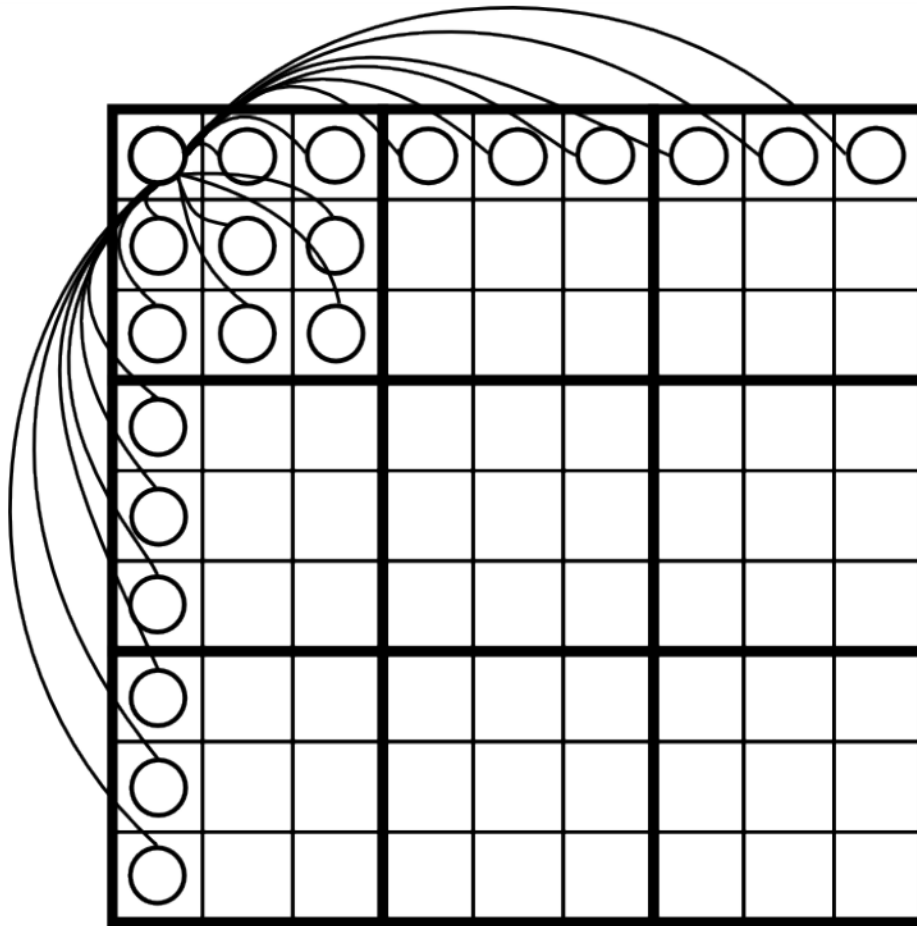


Figura 2 – Modelagem de um vértice do Sudoku  $9 \times 9$  em grafos.

## 2 Complexidade do problema

Em [2], é demonstrado que o problema do Sudoku pertence a classe dos problemas NP-difícil.

## 3 Heurísticas construtivas e algoritmos aproximativos existentes

Nesta seção serão expostas algumas heurísticas e algoritmos aproximativos presentes na literatura.

- Forward Checking: este método sugere que a busca por uma solução seja encerrada, ou que ocorra a poda da sub-árvore de busca, no momento em que alguma variável se torna inviável, ou seja, quando seu conjunto de valores possíveis se torna vazio. Isso significa que não há mais opções viáveis para essa variável. No cenário do Sudoku, sempre que o algoritmo de backtracking se deparar com uma célula não preenchida na qual não haja mais possibilidade de inserir qualquer valor, a busca em andamento é finalizada. [1]
- Minimum Remaining Values (MRV): essa heurística sugere que é preferível selecionar variáveis com um quantitativo menor de opções disponíveis, ou seja, escolher a

variável mais restrita. Dentro do contexto do Sudoku, isso se traduz na seleção da célula que possui o menor número de candidatos toda vez que se faz necessário atribuir um valor a uma nova célula que ainda não foi preenchida. [1]

- Heurística do Valor Menos Restritivo: essa heurística sugere que dada uma variável, o recomendado é escolher os seus valores a partir do menos restritivo, ou seja, proporciona maior flexibilidade para as próximas atribuições. No contexto do Sudoku, envolve a seleção prioritária de células com o menor número de candidatos possíveis. Isso é feito para simplificar a resolução, uma vez que preencher células com menos restrições pode levar a menos conflitos e, portanto, a uma busca mais eficiente por uma solução válida. [1]

## 4 Banco de instância

Inicialmente, para gerar as instâncias do Sudoku, foi utilizado [um gerador](#) de instâncias. No entanto, não foram encontrados geradores de puzzles maiores do que 9x9 e os puzzles gerados por ele, mesmo no nível mais difícil, eram resolvidos de forma correta pelo algoritmo.

Desejava-se realizar a implementação de um gerador de instâncias de Sudoku. Entretanto, seria necessário implementar um algoritmo exato para fazer isso, o que foi avaliado como fora do escopo para esse trabalho, mas que será realizado posteriormente.

Portanto, usou-se instâncias extraídas do site [sudoku.com](#) para os puzzles 9x9 e [sudoku-pluzzes-online](#) para os puzzles 16x16 e 25x25.

Por conta do processo manual de transformar as imagens dos puzzles em uma entrada que o algoritmo entenda, não foi possível utilizar tantas instâncias do puzzle, principalmente para os Sudoku maiores. Foram gerados aleatoriamente 3 puzzles de cada dificuldade (easy, medium, hard, expert, evil) para o 9x9, 1 de cada dificuldade (beginner, confirmed, expert) para o 16x16 e 1 de cada dificuldade beginner e expert para o 25x25.

## 5 Implementação de algoritmo aproximativo

Utilizando a linguagem de programação C++, implementamos um algoritmo aproximativo (anexo I) para resolver o Sudoku. O algoritmo é baseado na heurística MRV, que escolhe a célula com mais restrições para escolher um valor primeiro.

O Sudoku foi representado simplesmente como um vetor com  $n^4$  elementos, sendo os  $n^2$  primeiros a primeira linha, os próximos  $n^2$  a segunda e assim por diante.

Definimos a saturação de uma célula como a quantidade de números diferentes nas células adjacentes, se assemelha à saturação do algoritmo DSatur.

O algoritmo consiste em visitar sempre a célula com maior saturação e colocar o menor número possível respeitando as condições do Sudoku.

## 6 Análise do algoritmo

Considerando  $n$  como o número de células no Sudoku, o tempo para achar a célula com maior saturação é  $O(n)$ , como precisamos fazer isso até todas as células vazias estarem disponíveis, então o algoritmo em si possui complexidade  $O(n^2)$ .

## 7 Aplicação do algoritmo

Segundo as restrições do Sudoku, cada linha, coluna ou subgrid deve ter os valores de 1 a  $n$ . A exemplo, numa instância do jogo de tabuleiro com dimensões  $9 \times 9$ , os valores em cada célula deve estar entre 1 e 9. Desta forma, dentro da modelagem apresentada neste projeto, aplicando a coloração de vértices no Sudoku, temos que o número cromático, menor número de cores necessário para obter uma coloração do grafo, será igual a  $n$ . Portanto, em um algoritmo exato, aplicado em um Sudoku  $9 \times 9$ , terá até o valor numérico 9 em suas células. A heurística implementada neste projeto aplicada nas instâncias nem sempre conseguiu atingir o número cromático.

## 8 Resultado do algoritmo

Por estar no contexto de algoritmos aproximativos, podemos calcular a taxa de aproximação do algoritmo, que trata-se da taxa que o resultado do algoritmo se afasta da solução ótima ( $p(n)$ ).

Nas instâncias de Sudoku  $9 \times 9$  utilizadas neste trabalho, obtidas do site [sudoku.com](http://sudoku.com), a taxa de aproximação observada na maioria das execuções foi a de  $p(n) = \frac{10}{9} = 1.\bar{1}$ , sendo que apenas nos puzzle de dificuldade evil o algoritmo gerou uma solução utilizando 11 números.

Nas instâncias de Sudoku  $16 \times 16$  utilizadas neste trabalho, obtidas do site [sudoku-pluzzes-online](http://sudoku-pluzzes-online), o algoritmo gerou o resultado ótimo para o puzzle de dificuldade beginner, usou três números a mais para a conformed e dois números a mais para a expert.

Nas instâncias de Sudoku  $25 \times 25$  utilizadas neste trabalho, obtidas do site [sudoku-pluzzes-online](http://sudoku-pluzzes-online), o algoritmo gerou o resultado ótimo para o puzzle de dificuldade beginner e usou quatro números a mais para a expert.

Portanto, é factível inferir que o algoritmo afastou-se, em valores percentuais, 10% do resultado ideal esperado pelo algoritmo exato.

## 9 Links importantes

[Como jogar Sudoku: estratégia, dicas e regras.](#)

O link acima fornece um panorama geral sobre o jogo abordado neste projeto. Seu funcionamento, regras e afins.

[O que é um grafo?](#)

Este segundo link apresenta conceitos, definições e exemplos da teoria de grafos, abordando conceitos importantes para entendimento satisfatório deste trabalho tais como adjacência e matriz de adjacência.

[O problema da coloração dos vértices de um grafo](#)

No link acima é mostrado o que é o problema da coloração de vértices e também mostra heurística para solucionar.

[Repositório de implementação da heurística deste projeto](#)

O link acima trata-se do repositório onde o código desenvolvido durante este projeto está hospedado.

## Conclusão

Por meio da elaboração deste projeto, foi possível consolidar o conhecimento obtido na disciplina de Projetos e Análise de Algoritmos (PAA), colocando em prática conteúdos estudados na disciplina, em especial o conteúdo de complexidade de problemas e algoritmos aproximativos e heurísticas, além de revisar conceitos relevantes como o de grafos simples e coloração de grafos.

# Sudoku solver using graph coloring

José Davi Viana Francelino  
Thiago de Oliveira Cordeiro

outubro, 2023

## Abstract

This article aims to present graph modeling for solving the Sudoku game and an approximate algorithm, a heuristic, that solves the generic Sudoku  $n \times n$ . Sudoku is an NP-complete problem and logical reasoning challenge that is related to the Latin square. Furthermore, simple graph coloring will be used in this heuristic, given that coloring is a relevant graph labeling tool, and, using it, it is possible to solve the Sudoku game.

**Key-words:** graphs. graph coloring. sudoku.

## Referências

Figura 1. [http://jogadamais.blogspot.com.br/2013\\_11\\_01\\_archive.html](http://jogadamais.blogspot.com.br/2013_11_01_archive.html). Acessado em: 2023- 10-17.

Figura 2. <http://vigusmao.github.io/manuscripts/sudoku.pdf>. Acessado em: 2023-10-17.

[1] Rosa, J. L. G. Algoritmos Avançados. <http://wiki.icmc.usp.br/images/6/62/SCC210Cap7.pdf>. Acessado em: 2023- 10-15.

[2] Takano, K; Freitas, R. de; Sá, V. G. P. de. O jogo de lógica Sudoku: modelagem teórica, NP-completude, algoritmos e heurísticas. 2015.

[3] SANTOS, R. P. dos; VASCONCELLOS, L. A. dos S. A matemática por trás do sudoku. C.Q.D.– Revista Eletrônica Paulista de Matemática, Bauru, v. 12, p. 26-46, jul. 2018.

Byskov, J. M. (2002). Chromatic number in time  $O(2.4023n)$  using maximal independent sets. B RICS Report Series, 9(45).

McDiarmid, C. (1979). Determining the chromatic number of a graph. SIAM Journal on Computing, 8(1):1–14.

## Anexos

### Anexo 1

```
1 void Graph::solve() {
2     auto is_empty = [](Node &node) { return node.value == DOT; };
3
4     while (true) {
5         auto empty = nodes | std::views::filter(is_empty);
6
7         auto max = std::max_element(
8             empty.begin(), empty.end(), [](auto node_a, auto node_b) {
9                 return node_a.saturation() <= node_b.saturation();
10            });
11
12         if (max == empty.end()) {
13             break;
14         }
15
16         max->value = max->next_free(rank * rank * rank * rank);
17
18         for (auto j{0}; j < rank * rank; j++) {
19             if (j != max->j) {
20                 auto node = get_node(max->i, j);
21                 node->adjacent[max->value - 1] = true;
22             }
23         }
24         for (auto i{0}; i < rank * rank; i++) {
25             if (i != max->i) {
26                 auto node = get_node(i, max->j);
27                 node->adjacent[max->value - 1] = true;
28             }
29         }
30         int startRow = (max->i / rank) * rank;
31         int startCol = (max->j / rank) * rank;
32
33         for (int i = startRow; i < startRow + rank; ++i) {
34             for (int j = startCol; j < startCol + rank; ++j) {
35                 if (i != max->i && j != max->j) {
36                     auto node = get_node(i, j);
37                     node->adjacent[max->value - 1] = true;
38                 }
39             }
40         }
41     }
42 }
43 }
```