

Solucionador de Sudoku utilizando coloração em grafos

José Davi Viana Francelino
Thiago de Oliveira Cordeiro

novembro, 2023

Resumo

Este artigo visa apresentar uma modelagem em grafos para a resolução do jogo Sudoku e um algoritmo exato implementado na linguagem de programação C++ 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 modelagem do problema, 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 vêm 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.

Uma abordagem interessante é a aplicação de técnicas de coloração em grafos para resolver o Sudoku de forma exata. 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. Outrossim, 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 exato que utiliza a coloração de grafos e o backtracking 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 9×9 , que é subdividido em blocos menores de 3×3 . 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 9×9 .

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:

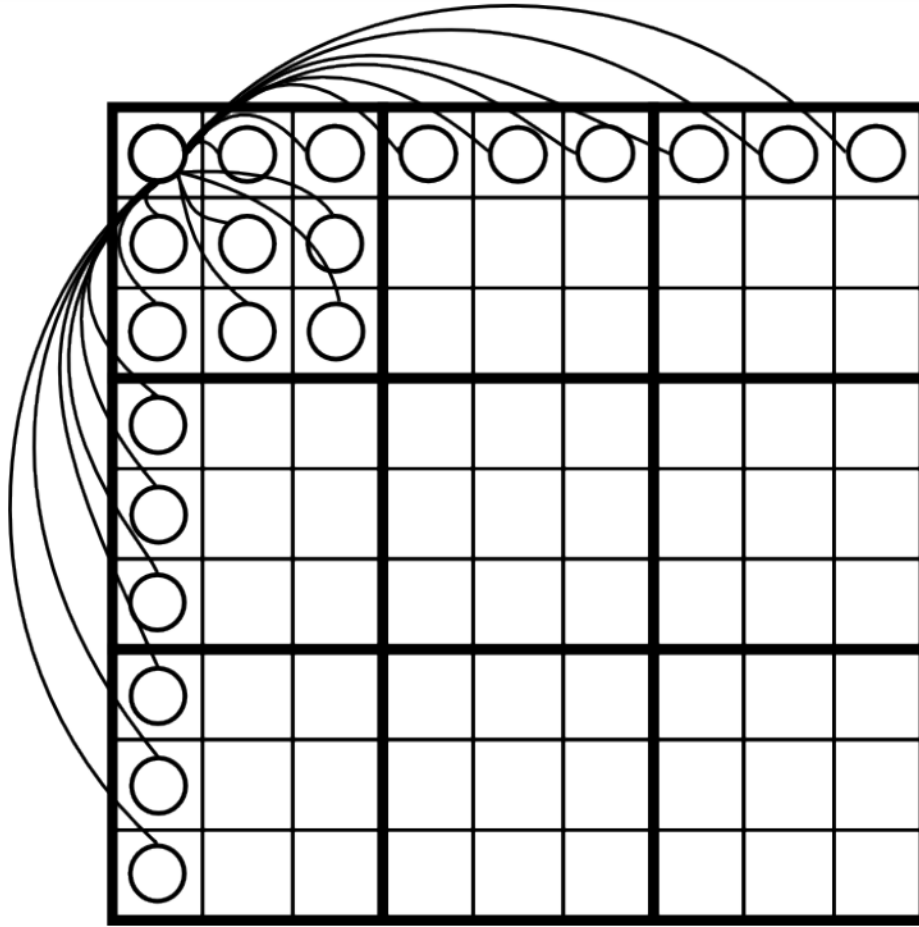


Figura 2 – Modelagem de um vértice do Sudoku 9×9 em grafos.

2 Backtracking

O *backtracking* é uma técnica utilizada para resolver problemas computacionais que envolvem a busca sistemática por soluções em espaços de decisão complexos. Ou seja, é uma forma de testar várias sequências de decisões, até encontrar a que fornece uma solução ou a solução ótima.

A essência do backtracking reside na construção passo a passo de uma solução candidata para o problema, avaliando continuamente se ela atende aos requisitos estabelecidos. Ao identificar uma solução parcial, o algoritmo toma decisões cruciais, optando por avançar na exploração dessa solução ou retroceder para considerar alternativas. Essa abordagem é fundamental para evitar uma busca exaustiva por soluções inviáveis, resultando em economia de tempo computacional.

A recursão desempenha um papel vital nessa técnica, proporcionando a construção incremental da solução e a manutenção de um estado do problema em cada nível de recursão. Essa estrutura facilita o processo de retrocesso, permitindo que o algoritmo reavalie escolhas anteriores e explore outras possibilidades quando necessário.

Dessa forma, o backtracking se destaca como uma estratégia eficaz para resolver problemas complexos, aproveitando a recursividade para uma abordagem sistemática e

eficiente na busca por soluções viáveis. E foi esta a estratégia selecionada neste trabalho, o *backtracking* (puro, sem podas).

3 Complexidade do problema

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

4 Estado da arte

Há algumas abordagens que já foram apresentadas para a resolução de sudokus genéricos a partir de algoritmos exatos e aproximativos. Sendo alguns destes:

Primeiramente, alguns algoritmos exatos:

- Algoritmo de *backtracking* implementado por Skiena, o qual faz uso da técnica de Forward Checking e a heurística de Minimum Remaining Values. Neste algoritmo, há a interrupção da busca pela solução, ou seja a poda da sub-árvore de busca, quando uma variável ainda não testada não possui mais valores possíveis em seu domínio. Dentro do cenário do Sudoku, quando o algoritmo de *backtracking* identifica uma célula não preenchida na qual não é mais possível inserir nenhum valor, a busca atual é finalizada. [4]
- Algoritmo proposto por Norvig baseado em propagação de restrições. Neste algoritmo, é usado uma técnica que efetua a remoção de valores dos domínios das variáveis que, com certeza, não podem mais ser atribuídos. Durante a execução do algoritmo, que se dá por meio da busca em profundidade (*backtracking* padrão). É realizada a exclusão de candidatos das células do tabuleiro à medida que se infere que esses valores não podem mais ser inseridos. [6]
- Algoritmo de *backtracking* implementado fazendo uso da técnica de manipulação de Bits. Neste algoritmo, foi-se usado como base o algoritmo de propagação de restrição do Norvig e neste algoritmo os candidatos são representados por meio de vetores de bits, utilizando manipulação de bits para realizar de forma eficiente operações como inserção, remoção e verificação de candidato. [2]

Agora, 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]

5 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.

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 das dificuldades beginner e expert para o 25x25.

6 Implementação de algoritmo exato

Utilizando a linguagem de programação C++, implementamos um algoritmo exato (anexo I) para resolver o Sudoku. O algoritmo utiliza a técnica *backtracking* para solucionar o problema abordado neste trabalho.

Para resolver o Sudoku, passamos por cada célula do Sudoku e atribuímos o menor número possível a ela. Se em algum momento não tiver nenhum número disponível para aquela célula, voltamos para a célula anterior e colocamos o próximo número disponível.

7 Análise do algoritmo

Consideramos n como a quantidade de números que pode estar em uma célula no Sudoku. O algoritmo, no pior caso, vai ter que testar $n!$ possibilidades para cada linha, já que a cada número que ele escolhe para uma célula, diminui o número de possibilidades para as próximas células. Como o tabuleiro tem n linhas, nosso algoritmo é $O(n \cdot n!)$.

7.1 Prova de corretude

Mostraremos, por indução, que após k chamadas recursivas o algoritmo retorna *true* para soluções válidas e *false* para inválidas.

Base de Indução:

Para $k = 0$, o tabuleiro encontra-se no estado inicial, que é garantido ser válido pois ocorre uma verificação antes do algoritmo ser executado.

Passo de indução:

Suponha que após k chamadas recursivas o algoritmo gere um tabuleiro válido. Agora, na $(k + 1)$ -ésima chamada recursiva, a função buscará preencher a próxima célula vazia.

Se não existir célula vazia, quer dizer que o tabuleiro está todo preenchido e, pela hipótese de indução, é uma solução válida. Dessa forma, o algoritmo retorna *true* para indicar que a solução foi encontrada.

Se existir uma célula, o algoritmo vai procurar o menor número que pode ocupar aquela célula com base nas restrições do Sudoku, ou seja, um número que não apareça nas células adjacentes. Caso esse número exista, ele é colocado na célula, mantendo o Sudoku num estado válido. Caso não exista, a função retornará *false*, indicando que não é possível gerar um Sudoku válido com aquela configuração.

Portanto, está demonstrado que o algoritmo sempre termina sua execução e retorna saídas corretas para todas as instâncias do problema.

8 Resultado do algoritmo

Na versão de algoritmo aproximativo deste trabalho, foi possível calcular a taxa de aproximação do algoritmo ($\mathfrak{p}(n)$), que trata-se da taxa que o resultado do algoritmo se afasta da solução ótima.

Nas instâncias de Sudoku 9×9 utilizadas neste trabalho, obtidas do site sudoku.com, o algoritmo exato conseguiu resolver o *puzzle* em tempo hábil, sempre gerando a solução exata, enquanto o aproximativo, mesmo com tempo semelhante, não entregou o melhor resultado para todos os *puzzles*, tendo uma taxa de aproximação por volta de $\mathfrak{p}(n) = \frac{10}{9} = 1.\bar{1}$.

Nas instâncias de Sudoku 16×16 e 25×25 utilizadas neste trabalho, obtidas do site sudoku-pluzzes-online, o algoritmo exato, mesmo após horas executando, não conseguiu gerar um resultado. Enquanto o aproximativo, gerou o resultado ótimo para os *puzzle* de dificuldade beginner, e usou no máximo 4 números a mais nas outras dificuldades.

Portanto, com base nos resultados apresentados, é factível inferir que a abordagem aproximativa utilizada no trabalho para resolver Sudoku demonstrou eficácia em diversas situações, especialmente em cenários desafiadores onde o algoritmo exato enfrentou dificuldades computacionais.

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 do algoritmo 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, backtracking 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

novembro, 2023

Abstract

This article aims to present graph modeling for solving the Sudoku game and an exact algorithm implemented in the C++ programming language 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 modeling of the problem, 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. 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.

[4] Skiena, S. S. (2008). Algorithm Design Manual. Springer Publishing.

[5] Norvig, P. (2006). Solving every sudoku puzzle. <http://norvig.com/sudoku.html>. Acessado em 09 de novembro de 2023.

Byskov, J. M. (2002). Chromatic number in time $O(2.4023^n)$ 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 bool Graph::try_solve() {
2     // recupera a primeira célula vazia do sudoku
3     auto first = nodes.end();
4     for (auto it{nodes.begin()}; it < nodes.end(); it++) {
5         if (it->value == DOT) {
6             first = it;
7             break;
8         }
9     }
10
11     // se não tem células vazias, é porque o sudoku foi resolvido
12     if (first == nodes.end()) {
13         return true;
14     }
15
16     // guarda quais números já são usados por células adjacentes
17     std::vector<bool> used(rank * rank, false);
18
19     // verifica os adjacentes na coluna
20     for (auto j{0}; j < rank * rank; j++) {
21         if (j != first->j) {
22             auto node = get_node(first->i, j);
23             if (node->value != DOT) {
24                 used[node->value - 1] = true;
25             }
26         }
27     }
28     // verifica os adjacentes na linha
29     for (auto i{0}; i < rank * rank; i++) {
30         if (i != first->i) {
31             auto node = get_node(i, first->j);
32             if (node->value != DOT) {
33                 used[node->value - 1] = true;
34             }
35         }
36     }
37     // verifica os adjacentes no bloco
38     int startRow = (first->i / rank) * rank;
39     int startCol = (first->j / rank) * rank;
40
41     for (int i = startRow; i < startRow + rank; ++i) {
42         for (int j = startCol; j < startCol + rank; ++j) {
43             if (i != first->i && j != first->j) {
44                 auto node = get_node(i, j);
45                 if (node->value != DOT) {
46                     used[node->value - 1] = true;
47                 }
48             }
49         }
50     }
51
52     // passa por todos os números não usados
```

```

53  for (auto i{0}; i < rank * rank; i++) {
54      if (!used[i]) {
55          // coloca o valor da célula para o valor não usado
56          first->value = (Cell)(i + 1);
57          // caso consiga resolver com esse número, acaba o algoritmo,
           se não continua no loop
58          if (try_solve()) {
59              return true;
60          }
61      }
62  }
63
64  // se nenhum dos números resolveu o Sudoku, o problema está mais
           atrás. Limpa a célula e retorna que não foi possível resolver
65  first->value = DOT;
66  return false;
67  }

```