Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

# Sudoku – NP-Completude

Vinícius Julião Ramos
viniciusjuliao@dcc.ufmg.br

24 de Novembro de 2019

**Resumo**

O trabalho aqui apresentado consiste na elaboração de um modelo computacional que consiga resolver o problema conhecido como Sudoku. Para isso solicitou-se que fosse realizada uma possível redução desse problema para o problema de coloração de grafos a fim de que aplicar uma heurística capaz de encontrar uma possível solução, visto que tais problemas são são definidos como determinísticos em tempo polinomial. Então, a solução mostrada a seguir não resolverá o problema do tabuleiro para todos os casos.

## 1 Introdução

O jogo Sudoku, é composto por um tabuleiro com dimensões quadradas em que deve-se encontrar uma solução para a organização dos símbolos do tabuleiro. Tal organização parte do princípio da análise de uma matriz com linhas, colunas e quadrantes (ou blocos), em que numa mesma linha nenhum dos símbolos podem se repetir, sendo tal comportamento replicado para as colunas e os quadrantes. Esse jogo é muito famoso em seções de passa tempo dos jornais e revistas, e nem sempre são de fácil solução dada a larga possibilidade de combinações que uma instância do problema pode ter. Em sua versão mais conhecida o tabuleiro é formado por dimensões de $9 \times 9$ células e os símbolos utilizados são os algarismos de 1 a 9.

É possível resolver tal problema de forma a testar todas as possibilidades de combinações no tabuleiro, entretanto seria computacionalmente muito custoso, e para casos de elevados custos computacionais, opta-se por uma solução não determinística. A solução não determinística adotada será o *backtracking*, em que, dada uma redução do Sudoku para coloração de grafos, pode-se testar as possibilidades de forma a caminhar do início do tabuleiro (posição [1,1]) até o fim do tabuleiro (posição [n,n]) a fim de identificar a possível "cor"que determinada célula do grafo poderia receber de acordo com as limitações de linha, coluna e quadrante. Uma vez, que a definição do jogo é dada de forma abstrata, em que deseja-se encontrar uma organização dos símbolos para o tabuleiro, tais símbolos podem ser representados por quaisquer entidades, inclusive cores, desde que haja uma tradução bijetiva, em que nenhum símbolo possa se transformar em outros dois e vice e versa.

Após o desenvolvimento dos algoritmos para solucionar o problema, na implementação prática usou-se da linguagem de programação **C++**, que permite implementar o conceito de orientação a objetos, além de possuir uma biblioteca padrão robusta que contém diversas estruturas de dados já implementadas (fatores que foram essenciais para uma boa implementação dos códigos).

# 2 Modelagem

A partir da descrição dada para o problema, necessitou-se da criação de duas diferentes estruturas, sendo uma delas a estrutura de armazenamento e outra a estrutura de computação da solução. A primeira estrutura foi modelada como uma matriz de elementos inteiros para simular o tabuleiro do jogo, em que a partir da entrada do programa, obtém-se três atributos chave de tal estrutura. Tais atributos são classificados como um $n$ – quantidade de símbolos diferentes que o tabuleiro suporta, por consequência a matriz terá um tamanho $n \times n$ –, *col* e *row* que indicam a quantidade de colunas e linhas, respectivamente, que um bloco do tabuleiro possui. Já a segunda estrutura é composta por um grafo (uma lista de nós), em que cada vértice (ou nó) contém uma lista de adjacências e um atributo cor utilizado para a heurística da solução dada.

Para possibilitar a resolução do Sudoku com o problema de coloração de grafos necessitou-se de uma transformação do problema matricial para o problema de grafos. Entende-se o problema de coloração de grafos, como uma solução de restrições, ou seja, os vértices adjacentes não podem pertencer ao mesmo grupo (cor), pois a aresta define uma restrição entre tais nós. Por isso, modelou-se o problema de forma a criar um nó no grafo para cada célula, sendo que as arestas entre vértices denotam as células pertencem a um mesmo bloco, linha ou coluna do tabuleiro do jogo, ou seja, não podem ter a mesma cor (símbolo ou número). A visualização desse modelo é descrito pela Figura 1 para um tabuleiro de proporção 9×9.

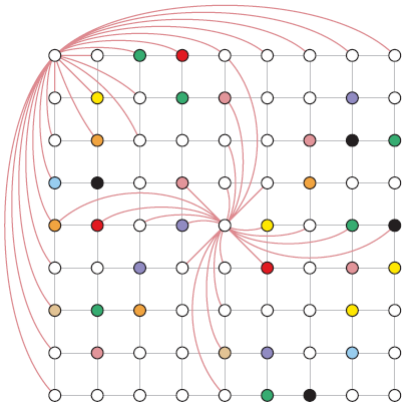

Figura 1: Transformação do sudoku para grafo colorido.

Diante de tal modelo, agora, pode-se criar o cenário ideal para armazenar e computar as respostas baseadas na heurística conhecida como *backtracking*. Porém, é necessário entender como a transformação se deu e como foi feita a montagem da recursão para satisfazer a heurística, as sub seções seguintes demonstrarão como tais algoritmos foram implementados.

## 2.1 Transformação Polinomial

A transformação encontrada para a conversão de uma matriz em um grafo via lista de adjacência, deve cuidar de uma sutileza: não permitir adicionar arestas redundantes. Esse cuidado, leva em consideração que cada vértice possui apenas uma restrição a outro vértice, e na adição de arestas redundantes, é possível que o programa considere retratar a restrição definida pela segunda aresta, mesmo quando já tratara em um momento anterior.

Em resumo, inicialmente, para determinada posição da matriz Sudoku $m_{i,j}$, adiciona-se arestas no grafo para as células da mesma da linha e coluna da posição inicialmente referenciada. Entretanto, ao adicionar as restrições provadas pelo quadrante da matriz, deve-se cuidar para não adicionar, novamente, as células de mesma linha e coluna que $m_{i,j}$. O Algoritmo 1 a seguir demonstra a operação realizada:

O custo de execução do Algoritmo 1 anterior pode ser demonstrado através de uma $f$ que recebe $n$ como único parâmetro. O calculo tomará como base que a matriz possui $n^2$ elementos e isso refere-se em dizer que o tamanho da entrada é quadrático. O primeiro laço do problema, consiste em adicionar cada uma da células

**Algorithm 1** Polynomial Transformation

```
1: function POLYNOMIAL_TRANFORMATION(MatrixM[ ], Integer n, GraphG)
2:     for all Row ∈ M[ ] do
3:         for all Column ∈ M[ ] do
4:             G.add_vertex(M[Row, Column])
5:         end for
6:     end for
7:     for all Row ∈ M[ ] do
8:         for all Column ∈ M[ ] do
9:             for all Element ∈ Column do
10:                G.add_edge(M[Row, Column], Element)
11:            end for
12:            for all Element ∈ Row do
13:                G.add_edge(M[Row, Column], Element)
14:            end for
15:        end for
16:    end for
17:    for all Row ∈ M[ ] do
18:        for all Column ∈ M[ ] do
19:            Block ← M[Row, Column].block
20:            for all Element ∈ Block do
21:                if Element.Row ≠ Row and Element.Column ≠ Column then
22:                    G.add_edge(M[Row, Column], Element)
23:                end if
24:            end for
25:        end for
26:    end for
       return G
27: end function
```

da matriz como um vértice do grafo, isso faz com que seja necessário percorrer toda a estrutura matricial, gerando um custo $n^2 = O(n^2)$. Já o segundo aninhamento de laços, dada uma célula $m_{i,j}$, há necessidade de adicionar cada uma das $n-1$ células da mesma linha e as $n-1$ células da mesma coluna que $m_{i,j}$ como um vértice adjacente ao vértice de $m_{i,j}$ no grafo. Esse processo deverá ser feito para todos os $n^2$ elementos de $M$ e portanto possui um custo de $n^2(2 \times (n-1)) = O(n^3)$.

Já o processo de adicionar as arestas dos quadrantes de cada célula do Sudoku, necessita de uma análise um pouco mais cautelosa. Como o tabuleiro de um Sudoku é quadrado ($n^2$ elementos), os quadrantes devem ser divididos de forma que haja $n$ quadrantes no máximo, por consequência, cada quadrante terá exatamente $n$ elementos. Dado o aninhamento de laços do algoritmo, deve-se validar para que cada um nos $n^2$ elementos de $M$ tenha os $n-1$ elementos de seu quadrante adicionados como vértice adjacente no grafo da transformação. Portanto, os laços aninhados fazem com que se tenha um custo $n-1$ para todos os $n^2$ elementos da matriz. Em resumo, tem-se que o custo dessa etapa é de $n^2(n-1) = O(n^3)$.

Assim, ao final da execução da transformação polinomial, dita polinomial, prova-se que há uma relação cúbica entre a raiz do tamanho da entrada ($\sqrt{n^2}$), gastou-se o equivalente a:

$$O(n^2) + O(n^3) + O(n^3) = O(n^3)$$

## 2.2 Coloração do Grafo

A solução desse problema, envolve uma recursão que analisa várias formas de coloração, "colorindo"o grafo na mesma ordem em que os vértices foram adicionados. Para isso criou-se uma função recursiva capaz de mapear as cores de acordo com números inteiros, sendo que, ao conseguir e adicionar uma cor para o vértice $v$, analisa valida, recursivamente, se é possível adicionar uma cor para $v+1$.

Esse processo criará uma árvore também conhecida como árvore de possibilidades. Em que uma solução só é encontrada se houver um nó folha no nível $n^2$ da árvore. Isso é justificado pelo fato de que cada nível da arvore, é a verificação de coloração do nó em que caso consiga colorir, a arvore ganha mais um nó – do contrário retorna-se para os passos anteriores até que seja possível criar outro ramo da árvore de possibilidades para que se chegue até um nó folha no nível $n^2$. Como visto na transformação polinomial, cada vértice pode ser colorido por no máximo $n$ cores, portanto aqui encontra-se um árvore $n$-ária de possibilidades, em que se configura um heurística conhecida como *Backtracking*.

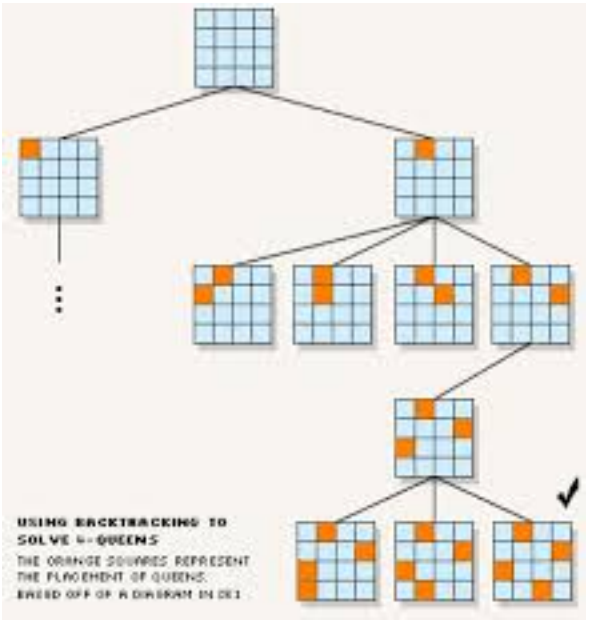


Figura 2: Exemplo de árvore de possibilidades para *backtracking* do Sudoku com $n = 2$

Esse tipo de heurística consiste em validar uma gama de possibilidades para determinada solução. Dado

que a solução de um passo de valor $k$ depende da solução do passo de valor $k-1$. Antes mesmo de realizar a montagem do algoritmo, é possível descrever o custo de execução para esse passo de acordo com a descrição da árvore de possibilidades definida no passo anterior. Cada nó da árvore, representa uma chamada à função recursiva que decide se é ou não possível preencher um nó com determinada cor. Essa validação depende da quantidade de arestas que cada vértice tem, ou seja, valida no máximo uma vez todas as arestas de um nó para cada cor.

Dada a tabela do Sudoku e a transformação realizada, cada vértice do grafo colorido possuirá no máximo $3(n-1)$ arestas, então o validação de cores tem um custo de execução igual a $3(n-1) = O(n)$. Como essa operação é realizada para cada cor, cada nó da árvore de possibilidades tem um custo $O(n)$. Então imaginando uma árvore $n$-ária completa, ou seja, o $i$-ésimo nível da árvore possui $n^(i-1)$ elementos, com cada elemento custando $3(n-1) = O(n)$, o custo de completar toda a árvore pode ser determinado pelo custo de uma progressão geométrica:

$$\sum_{i=1}^{n^2} n^i = \frac{n^{n^2}-1}{n-1} = O(n^{n^2})$$

O algoritmo será descrito por um passo a passo de instruções que a função recursiva *colore(vértice V, cores C[])* possui:

1. se V for o ultimo vértice da lista de vértices do grafo $G$ retorne **VERDADEIRO**
2. para todas as cores $c_i$ em C:

   valide se $v$ pode ser colorido com $c_i$ – custo $O(n)$.

   se o passo anterior retornar verdadeiro execute *colore(V+1, C[])*
3. retorne **FALSO** – afirma que não há coloração possível por aquele ramo da árvore de possibilidades.

Diante da análise do custo de execução em função de $n$, é valido ressaltar que a quantidade de elementos da entrada do problema é igual a $n^2$. O valor de $n$ é referente à quantidade de cores em que deseja-se colorir o grafo. Portanto, sendo $x = n^2$, pode-se analisar o custo de execução de acordo com o tamanho da entrada da seguinte maneira:

$$O(n^{n^2}) = O(n^x) = O(\sqrt{x}^x) = O(x^{\frac{x}{2}})$$

O motivo pelo qual selecionou-se a heurística de *backtracking* é justificada pelo fato da criação de uma grande árvore de possibilidades. Além disso, esse tipo de aplicação é bastante conhecida como uma forma não determinística de apresentar uma solução para grande quantidades de instâncias. Portanto a escolha é justificada pela alta taxa de acerto.

Apesar do elevado custo de execução, dada que as instâncias não possuirão um tamanho tão grande, pode-se aplicar esse algoritmo e obter resposta com boa precisão sem grande espera. Assim, o *backtrackin* é uma boa escolha para solucionar tal problema, desde que as entradas sejam pequenas, como é o caso para esse trabalho.

# 3 Análise de Complexidade

A análise do comportamento da solução seguirá um cenário teórico que servirá de base comparativa para identificar o custo de cada algoritmo utilizado.

## 3.1 Custo de Tempo

Da seção anterior, fez-se um levantamento sobre a execução das duas etapas chave da solução do Sudoku. A partir dessa análise, obteve-se o custo de cada uma das etapas, sendo que a etapa da transformação polinomial teve um custo assintótico de $O(n^3) = O(x^(\frac{1}{2}))$ e para a coloração o custo assintótico superior foi de $O(x^{\frac{x}{2}})$ para $x = n^2$, visto que $x$ é o tamanho da entrada do problema e o grafo possui $x$ vértices.

Então, pode-se analisar o custo de tempo para executar tal trabalho igual a $O(x^{\frac{1}{2}}) + O(x^{\frac{x}{2}})$. O que resulta num custo total de tempo de execução igual a:

$$O(x^{\frac{x}{2}})$$

### 3.2 Custo de Espaço

Para um análise concisa e teórica, definiremos que o custo de uma célula da matriz do Sudoku, de um vértice do grafo colorido e de uma aresta do grafo colorido têm custo igual a 1.

Logo, dado $n$ a matriz do jogo ocupa um espaço igual a $n^2$. Já o grafo, como anteriormente definido, na etapa da transformação, possuirá também um custo de $n^2$ vértices, entretanto, analisando a quantidade de arestas que cada vértice possuirá em sua lista de adjacência, as limitações definidas por linhas, colunas e blocos fazem com que hajam $3(n-1)$ arestas para cada vértice. Portanto, o grafo ocupará uma espaço igual a $n^2 \times 3(n-1)$ arestas e $n^2$ vértices que custam $O(n^3) = O(x^{\frac{1}{2}})$ e para a matriz o custo assintótico é dado por $O(n^2) = O(x)$.

Porém há um ponto que deve ser levado em consideração: a recursão executada pelo *backtracking* que para cada chamada da função ocupa uma quantidade constante $c$ de memória para executar as tarefas. Visto que há uma árvore de recursão de tamanho $O(x^{\frac{x}{2}})$ que corresponde à quantidade de chamadas à função de custo de memória constante $c$, tem-se então que para esse ponto o custo de memória é de $c \times O(x^{\frac{x}{2}}) = O(x^{\frac{x}{2}})$.

Então, o custo total de espaço utilizado pelo programa é dado por:

$$O(x^{\frac{x}{2}}) + O(x^{\frac{1}{2}}) + O(x) = O(x^{\frac{x}{2}})$$

## 4 Análise Experimental

A etapa experimental demandou a criação de *scripts* capazes de criarem instâncias do problema de forma aleatória. Tais instâncias foram executadas quinze vezes, sendo que gerou-se dez instâncias para cada uma das seguintes tamanhos de Sukodu($n$): 4, 6, 8, 9. Após tal etapa, pôde-se criar uma tabela para observar o comportamento das médias e do desvio padrão de cada uma das instâncias, permitindo a validação do teste. Além disso pode-se observar a taxa de acertos geral para as instâncias aleatórias geradas, assim, observando uma taxa de acertos de 100% através da execução do comando de execução *./tp3 entrada.txt -t* que exibe se a solução foi encontrada ou não (s/n)e o tempo de execução em nanosegundos.

Para os dois primeiros gráficos mostrados na Figura 3, é possível notar o comportamento não linear tanto para o gráfico de comportamento médio (esquerda) quando para o gráfico de desvio padrão (direita). Tal fato é um bom argumento justificativo a cerca do provável comportamento exponencial da solução. Em que é possível ainda, inferir uma provável "assíntota"que representa o rápido crescimento dos valores da função para as entradas de tamanho $n$.

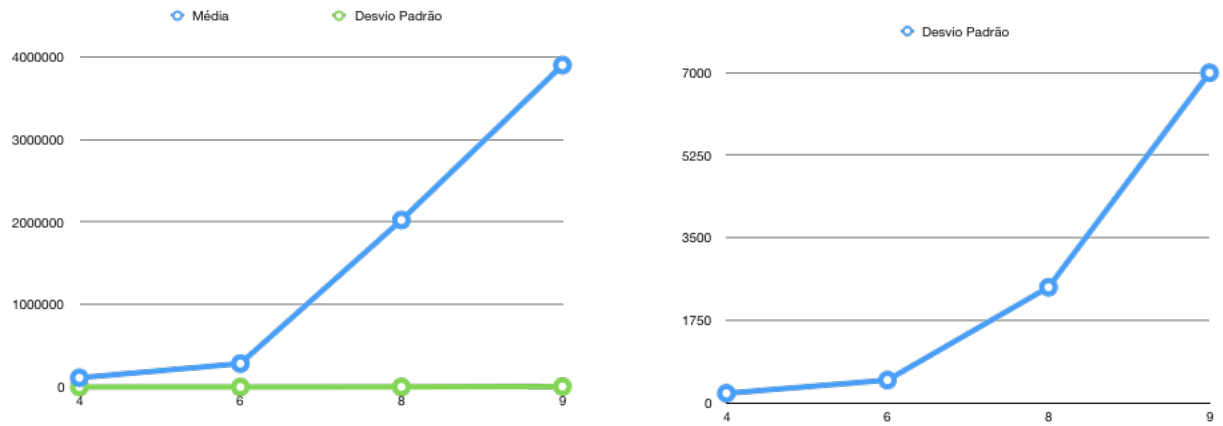


Figura 3: Análise do comportamento médio dos custos e do desvio padrão

Ainda é possível observar que o gráfico de desvio padrão também tem o mesmo comportamento, entretanto para valores muito baixos numericamente, se comparados aos valores médios. Uma vez que os casos de teste foram executados muitas vezes, a variância dos resultados é diminuída, fato explicado pela lei dos grades

números. Isso é suficiente para demonstrar que há um comportamento possivelmente exponencial das solução apresentada.

# 5 Conclusão

Conclui-se portanto, que a escolha do algoritmo de *backtrackin* é boa para entradas pequenas, visto a baixa quantidade de saídas não solucionadas em um tempo aceitável. Entretanto, pela análise dos gráficos, pode observar que essa não é uma boa metodologia de solução caso deseja-se executar uma instancia de tamanho muito grande, dado o comportamento exponencial da solução.

Mostrou-se ainda que é possível resolver um problema do mundo real, como um jogo de Sudoku, através da redução a soluções anteriormente conhecidas outros tipos de casos. É necessário ressaltar que a solução encontrada nesse trabalho pertence à classe de problemas NP-Completo, uma vez que coloração de grafos está nessa mesma classe e foi utilizado como forma de solução.