Problema NP Completo - Sudoku

Luis Sebastian Arroyo Pinto
Sebastian André Paz Ballón
Sebastian Ugarte Concha
Sharon Daniela Valdivia Begazo

June 2022

Índice

1. Introducción

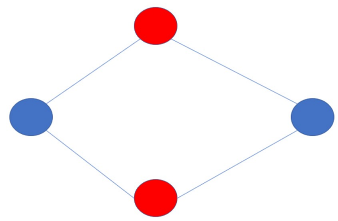
En el siglo XVIII el matemático Leonhard Euler creó un sistema de probabilidades para representar una serie de números sin repetir. Debido a esto, Euler se considera el inventor de este juego. Sin embargo, no se hizo conocido hasta 1970, cuando Walter MacKey lo público en la revista *Math Puzzles and Logic Problems*, en la sección llamada *Number Place* (El lugar de los números).
El nombre de este juego proviene de una publicación de 1984 en el periódico japonés *Monthly Nikolist*, publicó una sección de pasatiempos llamada *Sūji wa dokushin ni kagiru*, este título se abrevió a Sudoku, es decir, *Su* = número y *Doku* = solo (Solo números). [1]

Este juego se hizo conocido internacionalmente en 2005 y toda persona que haya intentado resolver un tablero de Sudoku, ha buscado la forma de hacerlo con mayor facilidad, ya sea a su manera o guiándose de varios métodos de resolución, sin embargo, cuando va aumentando la dificultad es más complicado seguir estas formas, por lo que terminan probando diversas combinaciones de números hasta dar con la respuesta, o en otras palabras, por el método de fuerza bruta. Por lo que nos preguntamos, ¿Habrá alguna estrategia más inteligente para resolver este problema? O por el contrario ¿El sudoku es un juego dificil de resolver intrínsecamente?
En este documento demostraremos que el Sudoku es un problema NP-completo

2. Demostración

Para demostrar que el Sudoku es un problema Np-completo, se demostrará que puede ser resuelto a partir del problema de coloración de grafos, el cual pertenece a 3 SAT, grupo correspondiente al

conjunto de problemas Np-completos.
En un problema de coloreo de grafos - G, tenemos que encontrar si un grafo se puede colorear por optimización, donde se busca hallar un número mínimo de colores, o por decisión, con un conjunto de colores predefinidos 'G', también conocido como el número cromático de un grafo y se denota como X(G).
Por ejemplo, en el siguiente grafo tenemos que $G = 2$: [3]



Vemos que $G = 2$ porque es el número mínimo de colores que puede aceptar el grafo.

Los problemas NP-completos son problemas difíciles o duros, que aunque podemos resolver velozmente con el concurso del no determinismo, por el momento solo sabemos resolver determinísticamente aplicando una búsqueda exhaustiva (fuerza bruta). Si aprendiéramos a resolver en menor tiempo alguno de estos problemas, conseguiríamos entonces a resolver rápidamente todo problema en NP. [6]
Un problema L es NP-completo si y solo si:

1. L $\in$ NP;
2. Si T $\in$ NP, entonces T es karp-reductible a L

Una vez que conocemos la definición de los problemas NP-completos, podemos probar que el problema del sudoku pertenece a este conjunto, por lo cual mostraremos que existe un problema NP-completo que es Karp-reductible a *Sudoku*, usando el problema de *Coloreo de grafos*.

1. Mostrar que existe un algoritmo no determinista con complejidad polinomial para el problema del Sudoku.

Listing 1: Example C++

```
bool funcion(int **matriz,int columna,int fila)
{

    if (!EncontrarValor0(matriz,fila,columna))
        return true;

    for (int num = 1; num <= 9; num++)
    {
        if (Valido(matriz,fila,columna, num))
        {
            matriz[fila][columna] = num;

            if (funcion(matriz))
                return true;
            matriz[fila][columna] = 0;
        }
    }
    return false;
}
```

2. Demostrar que el problema de Coloreo de grafos es reducible para poder solucionar el problema del Sudoku.

a) Entrada: Grafo G(v,a) donde v son los vertices y a las aristas del grafo.

b) Problema: Decida si G es reducible al problema de Sudoku.
Teniendo en cuenta que G es regular $(n^4, \frac{3n^6}{2} - n^5 - \frac{n^4}{2})$, de grado $3n^2 - 2n - 1$[2]

Para poder hacer el coloreo, primero se necesita probar el teorema para poder colorear un grafo.
Teorema: Para todo número natural n, existe una coloración adecuada del grafo de Sudoku X_n usando n^2 colores. El número cromático de X_n es n^2. Esto se debe a que en la fila, la columna y el cuadrante de una posición en específico, con un número asignado, este no se repetirá en las ubicaciones anteriormente mencionadas.

a	b	c	d
e	f		
i			
m			

Prueba: Notemos primero que todas las celdas de la esquina superior izquierda de la cuadrícula nn son adyacentes uno con otro y esto forma un grafo completo isomorfo de K_{n^2}. El número cromático de K_{n^2} es n^2 y por lo tanto, X_n necesitaría al menos n^2 colores para una coloración adecuada. Ahora, mostraremos que puede ser coloreado usando n^2 colores como se comentó anteriormente, es conveniente etiquetar los vértices (i, j) con $0 \leq i,\ j \leq n^2 1$. Considere el módulo de clases de residuos n^2. Para $0 \leq i \leq n^2 1$, escribimos $i = t_i n + d_i$ con $0 \leq d_i \leq n1y0 \leq t_i \leq n1$ y de manera similar para $0 \leq j \leq n^2 1$, también. Asignamos el "color" $c(i,j) = d_i n + t_i + nt_j + d_j$, módulo reducido n^2, a la (i,j)-ésima posición en la $n^2 x n^2$ cuadrícula. Nosotros afirmamos que se trata de una coloración adecuada. Para ver esto, nosotros deberíamos mostrar que dos coordenadas adyacentes cualesquiera (i,j) y (i,j) tienen colores distintos. De hecho, si $i = i$, entonces debemos mostrar que $c(i,j) \neq c(i,j)$ a no ser que $j = j$. Si $c(i,j) = c(i,j)$, entonces $nt_j + d_j = nt_{j'} + d_j$ lo que significa $j = j'$. Del mismo modo, si $j = j$, después $c(i,j) \neq c(i,j)$ a menos que $i = i$. Si ahora, $[i/n] = [i/n]$ y $[j/n] = [j/n]$, entonces $d_i = d_i y dj = dj'$. Si $c(i,j) = c(i,j)$, después

$$t_i + ntj = t_{i'} + nt_{j'}$$

Reduciendo el módulo n se tiene $t_i = t_{i'}$. Por lo tanto, $t_j = t_{j'}$ así que $(i,j) = (i',j')$ en este caso también. Por lo tanto, esta es una coloración adecuada.[4]
El problema del Sudoku puede resolverse usando el algoritmo de coloreo. Primero se escoge el número cromático que resulta ser igual a n.
La idea para aplicar el algoritmo del coloreo es asignar un color a cada uno de los nodos. Por seguridad se recomienda revisar si los colores se repiten con respecto a los vértices adyacentes. Luego, si es posible asignar un color, se asigna el color como parte de la solución. Si no es posible asignar un color, se aplica backtracking y retorna falso.
El algoritmo es el siguiente:

1. Cree una función recursiva que tome el índice de vértice actual, el número de vértices y la matriz de colores de salida como argumentos.

2. Si el índice de vértice actual es igual al número de vértices. Devuelve True e imprime la configuración de color en la matriz de salida.

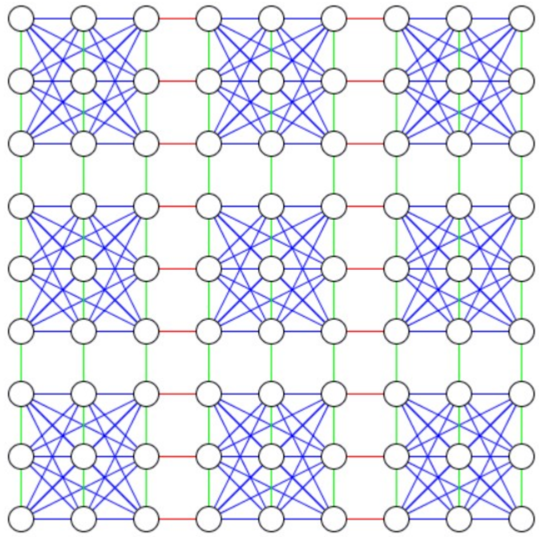
3. Asignar color a un vértice (1 a m).

4. Para cada color asignado, verifique si la configuración es segura (es decir, verifique si los vértices adyacentes no tienen el mismo color) llame recursivamente a la función con el siguiente índice y número de vértices

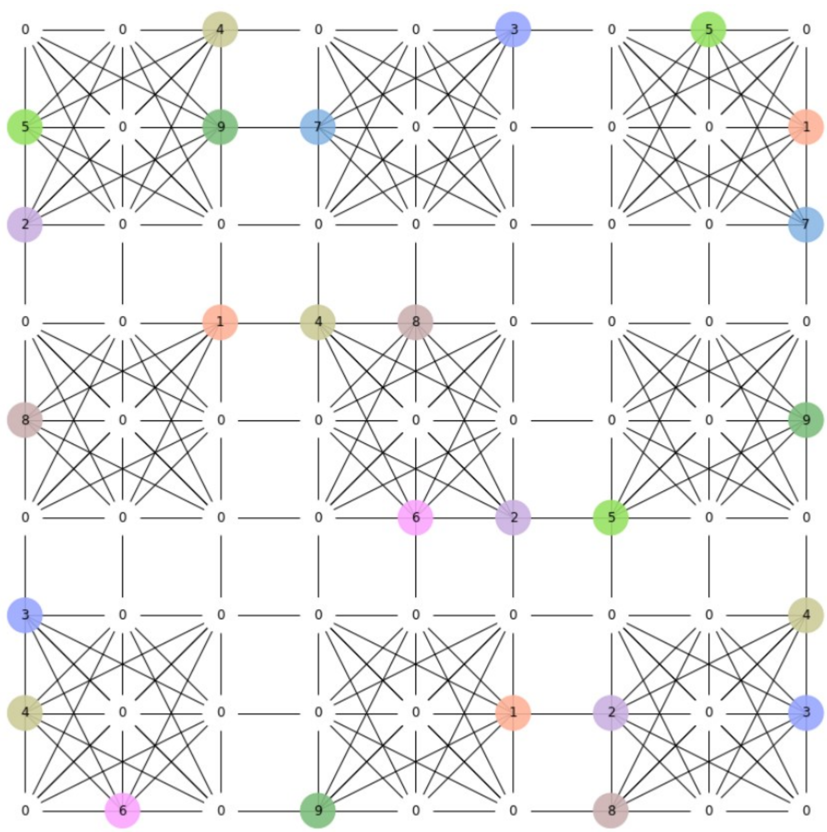
5. Si alguna función recursiva devuelve verdadero, rompa el ciclo y devuelva verdadero.

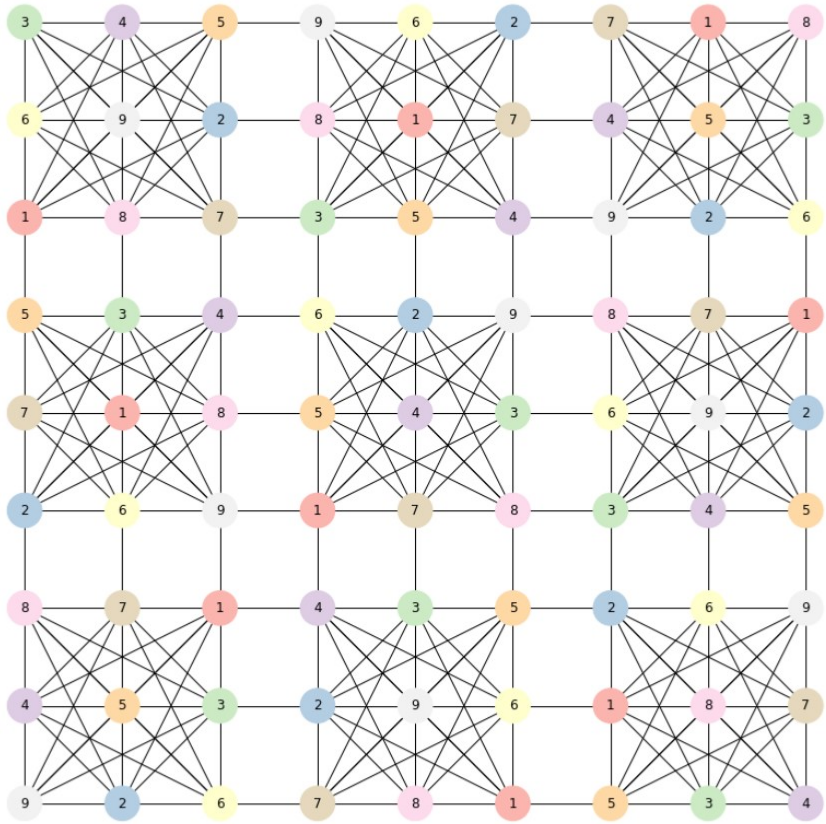
6. Si ninguna función recursiva devuelve verdadero, devuelve falso.

Resolver el Sudoku con coloreo
Si n=9, entonces se tendrían 81 nodos, donde cada celda del Sudoku puede representarse como nodo de un grafo.



Podemos ver que el Sudoku puede verse como un grafo, por lo tanto, puede ser resuelto con coloreo de grafos.
A continuación se muestra el coloreo del grafo asignando los valores para hallar la solución del Sudoku.





Una vez definidos todos los colores del grafo, podemos transformalo nuevamente a la matriz del Sudoku, reemplazando cada color por su equivalente en número, y así tendriamos resuelto nuestro tablero. Demostrando finalmente que a partir de un problema de *coloreo de grafos*, pordemos resolver el problema del *Sudoku*, por lo tanto este también pertenece al conjunto de problemas *NP-completo*.

3. Algoritmos

3.1. Algoritmo de Fuerza Bruta

La búsqueda por fuerza bruta es una técnica trivia que a menudo, que consiste en enumerar sistematicamente todos los posibles caminos para la solucion de un problema, con el de fin chequear si dicho camino satisface la solución al problema[5].
La búsqueda por fuerza bruta al buscar por todos los caminos posibles convella a tener un coste de ejecución proporcional al número de soluciones posibles, por ello se recomienda usar este método cuando el número de soluciones posibles no es tan elevado.

Explicación de las funciones del algoritmo de fuerza bruta para solucinar el problema del Sudoku.

- La función Valido verifica si la posición donde se coloca un número es segura y no colisiona con un valor igual al seleccionado en las filas, columnas o cuadrante.

- La función Sudoku se encarga de aplicar la recursión, comprobando todas las opciones posibles para cada valor de cada casilla.

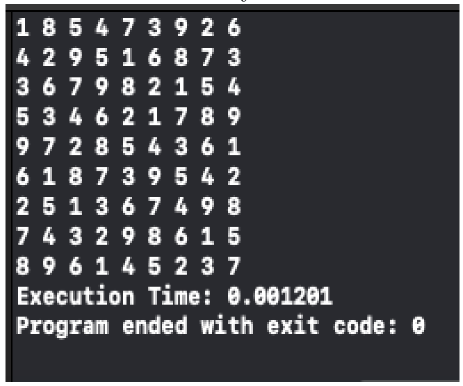
- La función printSudo imprime la matriz del Sudoku resuelto.

Listing 2: Fuerza Bruta

```
#include <iostream>
#include <time.h>
using namespace std;
#define N 4
#define R 2
void printSudo(int sudo[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << sudo[i][j] << " ";
        cout << endl;
    }
}
bool Valido(int sudo[N][N], int fil, int col, int num)
{
    for (int x = 0; x < N; x++)
        if (sudo[fil][x] == num)
            return false;
    for (int x = 0; x < N; x++)
        if (sudo[x][col] == num)
            return false;
    int FilaIni = fil - fil % R,
            ColIni = col - col % R;

    for (int i = 0; i < R; i++)
        for (int j = 0; j < R; j++)
            if (sudo[i + FilaIni][j +
                            ColIni] == num)
                return false;

    return true;
}
bool Sudoku(int sudo[N][N], int fil, int col)
{
    if (fil == N - 1 && col == N)
        return true;
    if (col == N) {
        fil++;
        col = 0;
    }
    if (sudo[fil][col] > 0)
        return Sudoku(sudo, fil, col + 1);

    for (int num = 1; num <= N; num++)
    {
        if (Valido(sudo, fil, col, num))
        {
            sudo[fil][col] = num;
            if (Sudoku(sudo, fil, col + 1))
                return true;
        }
        sudo[fil][col] = 0;
    }
    return false;
}
int main()
{
    unsigned t0, t1;
    int sudo[N][N] = { { 1, 0, 0, 0},
                    { 0, 0, 1, 0},
                    { 2,0, 0, 0},
                    { 0, 3, 0, 0}};
    t0=clock();
    if (Sudoku(sudo, 0, 0))
        printSudo(sudo);
    else
```

```
        cout << "No existe soluci n" << endl;
    t1 = clock();
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
    cout << "Execution Time: " << time << endl;
    return 0;
}
```

Algoritmo de fuerza bruta corriendo y mostrando el tiempo de ejecución



3.2. Algoritmo Aproximado

Listing 3: Clases Nodo y Grafo

```
class Node :

    def __init__(self, idx, data = 0) : # Constructor
        self.id = idx
        self.data = data
        self.connectedTo = dict()

    def addNeighbour(self, neighbour , weight = 0) :
    ##Define nodos adyacentes
        if neighbour.id not in self.connectedTo.keys() :
            self.connectedTo[neighbour.id] = weight
    #getter
    def getConnections(self) :  ## Devuelve nodos adyacentes
        return self.connectedTo.keys()
class Graph :

    totalV = 0 # total vertices in the graph

    def __init__(self) :
        self.allNodes = dict()

    def addNode(self, idx) :
        if idx in self.allNodes :
            return None

        Graph.totalV += 1
        node = Node(idx=idx)
        self.allNodes[idx] = node
        return node

    def addEdge(self, src, dst, wt = 0) :
        self.allNodes[src].addNeighbour(self.allNodes[dst], wt)
        self.allNodes[dst].addNeighbour(self.allNodes[src], wt)

    def isNeighbour(self, u, v) :
        if u >=1 and u <= 81 and v >=1 and v<= 81 and u !=v :
            if v in self.allNodes[u].getConnections() :
                return True
```

```
        return False

    def getAllNodesIds(self) :
        return self.allNodes.keys()
```

Listing 4: Clase Sudoku Connections

```
from graph import Graph

class SudokuConnections :
    def __init__(self) :

        self.graph = Graph()

        self.rows = 9
        self.cols = 9
        self.total_blocks = self.rows*self.cols #81

        self.__generateGraph()
        self.connectEdges() ##Conecta aristas
        self.allIds = self.graph.getAllNodesIds() ##Crea la lista de los ids
    def __generateGraph(self) :
        for idx in range(1, self.total_blocks+1) :
            _ = self.graph.addNode(idx)

    def connectEdges(self) :
        matrix = self.__getGridMatrix()
        head_connections = dict()

        for row in range(9) :
            for col in range(9) :
                head = matrix[row][col]
                connections = self.__whatToConnect(matrix, row, col)
                head_connections[head] = connections

        self.__connectThose(head_connections=head_connections)

    def __connectThose(self, head_connections) :
    ##Agrega los nodos adyacentes
        for head in head_connections.keys() :
            connections = head_connections[head]
            for key in connections :
                for v in connections[key] :
                    self.graph.addEdge(src=head, dst=v)


    def __whatToConnect(self, matrix, rows, cols) :
        connections = dict()
    ##Define los nodos adyacentes, en fila, columna y cuadrante
        row = []
        col = []
        block = []

        # ROWS
        for c in range(cols+1, 9) :
            row.append(matrix[rows][c])
        connections["rows"] = row

        # COLS
        for r in range(rows+1, 9):
            col.append(matrix[r][cols])
        connections["cols"] = col

        # BLOCKS

        if rows%3 == 0 :

            if cols%3 == 0 :
```

```python
                block.append(matrix[rows+1][cols+1])
                block.append(matrix[rows+1][cols+2])
                block.append(matrix[rows+2][cols+1])
                block.append(matrix[rows+2][cols+2])

            elif cols%3 == 1 :

                block.append(matrix[rows+1][cols-1])
                block.append(matrix[rows+1][cols+1])
                block.append(matrix[rows+2][cols-1])
                block.append(matrix[rows+2][cols+1])

            elif cols%3 == 2 :

                block.append(matrix[rows+1][cols-2])
                block.append(matrix[rows+1][cols-1])
                block.append(matrix[rows+2][cols-2])
                block.append(matrix[rows+2][cols-1])

        elif rows%3 == 1 :

            if cols%3 == 0 :

                block.append(matrix[rows-1][cols+1])
                block.append(matrix[rows-1][cols+2])
                block.append(matrix[rows+1][cols+1])
                block.append(matrix[rows+1][cols+2])

            elif cols%3 == 1 :

                block.append(matrix[rows-1][cols-1])
                block.append(matrix[rows-1][cols+1])
                block.append(matrix[rows+1][cols-1])
                block.append(matrix[rows+1][cols+1])

            elif cols%3 == 2 :

                block.append(matrix[rows-1][cols-2])
                block.append(matrix[rows-1][cols-1])
                block.append(matrix[rows+1][cols-2])
                block.append(matrix[rows+1][cols-1])

        elif rows%3 == 2 :

            if cols%3 == 0 :

                block.append(matrix[rows-2][cols+1])
                block.append(matrix[rows-2][cols+2])
                block.append(matrix[rows-1][cols+1])
                block.append(matrix[rows-1][cols+2])

            elif cols%3 == 1 :

                block.append(matrix[rows-2][cols-1])
                block.append(matrix[rows-2][cols+1])
                block.append(matrix[rows-1][cols-1])
                block.append(matrix[rows-1][cols+1])

            elif cols%3 == 2 :

                block.append(matrix[rows-2][cols-2])
                block.append(matrix[rows-2][cols-1])
                block.append(matrix[rows-1][cols-2])
                block.append(matrix[rows-1][cols-1])

        connections["blocks"] = block
        return connections
```

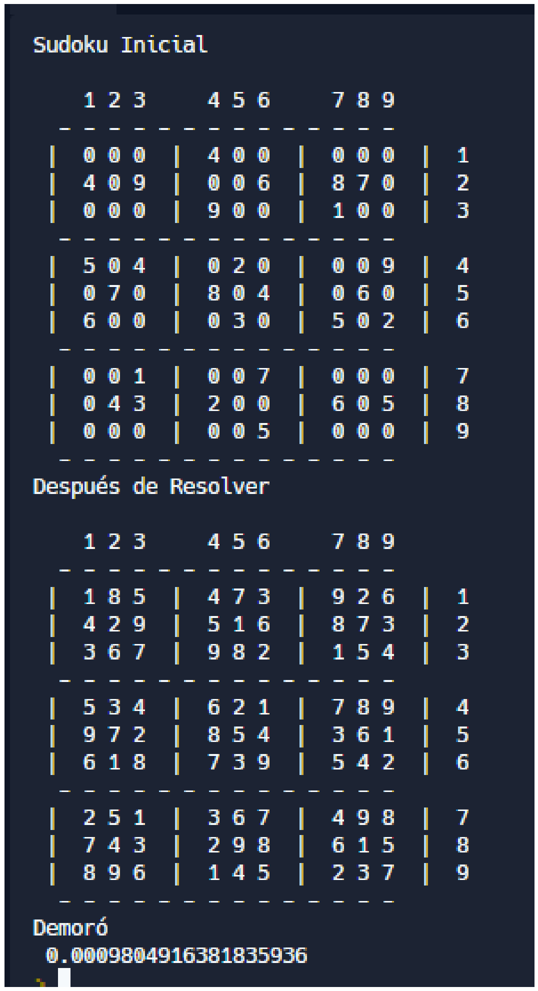
```python

    def __getGridMatrix(self) :
    ##Matriz enumerada por posiciones
        matrix = [[0 for cols in range(self.cols)]
        for rows in range(self.rows)]
        count = 1
        for rows in range(9) :
            for cols in range(9):
                matrix[rows][cols] = count
                count+=1
        return matrix
```

Listing 5: Main

```python

from sudoku_connections import SudokuConnections
import time


class SudokuBoard :
    def __init__(self) :

        self.board = self.getBoard() ##Lectura del sudoku inicial

        self.sudokuGraph = SudokuConnections()
        ## Genera el grafo, verifica las conexiones y las crea
        self.mappedGrid = self.__getMappedMatrix()

    def __getMappedMatrix(self) :  ##Matriz de 0 a 81
        matrix = [[0 for cols in range(9)]
        for rows in range(9)]

        count = 1
        for rows in range(9) :
            for cols in range(9):
                matrix[rows][cols] = count
                count+=1
        return matrix

    def getBoard(self) :

        board = [
            [0,0,0,4,0,0,0,0,0],
            [4,0,9,0,0,6,8,7,0],
            [0,0,0,9,0,0,1,0,0],
            [5,0,4,0,2,0,0,0,9],
            [0,7,0,8,0,4,0,6,0],
            [6,0,0,0,3,0,5,0,2],
            [0,0,1,0,0,7,0,0,0],
            [0,4,3,2,0,0,6,0,5],
            [0,0,0,0,0,5,0,0,0]
        ]
        return board

    def printBoard(self) :
        print("    1 2 3     4 5 6     7 8 9")
        for i in range(len(self.board)) :
            if i%3 == 0  :#and i != 0:
                print("  - - - - - - - - - - - - - - ")

            for j in range(len(self.board[i])) :
                if j %3 == 0 :#and j != 0 :
                    print(" |  ", end = "")
                if j == 8 :
                    print(self.board[i][j]," | ", i+1)
                else :
                    print(f"{ self.board[i][j] } ", end="")
        print("  - - - - - - - - - - - - - - ")
```

```


    def graphColoringInitializeColor(self):
    ##Crea un array color, donde se guardaran todos los colores del nodo
        color = [0] * (self.sudokuGraph.graph.totalV+1)
        given = []  ##Array que almacena los nodos que tienen un valor preasignado en el tablero
        for row in range(len(self.board)) :
            for col in range(len(self.board[row])) :
                if self.board[row][col] != 0 :
                    idx = self.mappedGrid[row][col]
                    color[idx] = self.board[row][col]
                    given.append(idx)
        return color, given

    def solveGraphColoring(self, m =9) :
        color, given = self.graphColoringInitializeColor()
        if self.__graphColorUtility(m =m, color=color, v =1, given=given) is None :
            print(":(")
            return False
        count = 1
        for row in range(9) :
            for col in range(9) :
                self.board[row][col] = color[count]
                count += 1
        return color

    def __graphColorUtility(self, m, color, v, given) :
    ##Define los colores de los nodos, con recursividad
        if v == self.sudokuGraph.graph.totalV +1 :
            return True
        for c in range(1, m+1) :
            if self.__isSafeColor(v, color, c, given) == True :
                color[v] = c
                if self.__graphColorUtility(m, color, v+1, given) :
                    return True
            if v not in given :
                color[v] = 0


    def __isSafeColor(self, v, color, c, given) :
    ## Verifica que el color que estamos asignando sea seguro
        if v in given and color[v] == c:
            return True
        elif v in given :
            return False

        for i in range(1, self.sudokuGraph.graph.totalV+1) :
            if color[i] == c and self.sudokuGraph.graph.isNeighbour(v, i) :
                return False
        return True


def test() :
    s = SudokuBoard()
    print("Sudoku Inicial \n")
    s.printBoard()
    print("Despu s de Resolver \n")
    s.solveGraphColoring(m=9)
    s.printBoard()

start = time.time()
test()
end = time.time()
print("Demor  \n", (end-start)/1000)
```

Algoritmo Aproximado corriendo y mostrando el tiempo de ejecución



```
Sudoku Inicial

   1 2 3    4 5 6    7 8 9
  - - - - - - - - - - - - - - -
 | 0 0 0  | 4 0 0  | 0 0 0  | 1
 | 4 0 9  | 0 0 6  | 8 7 0  | 2
 | 0 0 0  | 9 0 0  | 1 0 0  | 3
  - - - - - - - - - - - - - - -
 | 5 0 4  | 0 2 0  | 0 0 9  | 4
 | 0 7 0  | 8 0 4  | 0 6 0  | 5
 | 6 0 0  | 0 3 0  | 5 0 2  | 6
  - - - - - - - - - - - - - - -
 | 0 0 1  | 0 0 7  | 0 0 0  | 7
 | 0 4 3  | 2 0 0  | 6 0 5  | 8
 | 0 0 0  | 0 0 5  | 0 0 0  | 9
  - - - - - - - - - - - - - - -
Después de Resolver

   1 2 3    4 5 6    7 8 9
  - - - - - - - - - - - - - - -
 | 1 8 5  | 4 7 3  | 9 2 6  | 1
 | 4 2 9  | 5 1 6  | 8 7 3  | 2
 | 3 6 7  | 9 8 2  | 1 5 4  | 3
  - - - - - - - - - - - - - - -
 | 5 3 4  | 6 2 1  | 7 8 9  | 4
 | 9 7 2  | 8 5 4  | 3 6 1  | 5
 | 6 1 8  | 7 3 9  | 5 4 2  | 6
  - - - - - - - - - - - - - - -
 | 2 5 1  | 3 6 7  | 4 9 8  | 7
 | 7 4 3  | 2 9 8  | 6 1 5  | 8
 | 8 9 6  | 1 4 5  | 2 3 7  | 9
  - - - - - - - - - - - - - - -
Demoró
 0.0009804916381835936
```

Referencias

[1] Colaboradores de los proyectos Wikimedia (2005, Junio 20). ”Sudoku - Wikipedia, la enciclopedia libre”[online]. Wikipedia, la enciclopedia libre. Disponible: https://es.wikipedia.org/wiki/Sudoku

[2] M. Jhajharia (2022, January, 21). Sudoku and Graph Coloring. [online]. Available:https://mjhajharia.com/post/2022/01/21/sudoku-and-graph-coloring/

[3] I. Gupta (2020, Juny, 15). Sudoku Solver - Graph Coloring [online]. Available:https://medium.com/code-science/sudoku-solver-graph-coloring-8f1b4df47072

[4] Agnes M. Herzberg and M. Ram Murty (2007, Juny) Sudoku Squares and Chromatic Polynomials. [online]. Available: https://www.ams.org/notices/200706/tx070600708p.pdf

[5] Colaboradores de los proyectos Wikimedia. ”Búsqueda de fuerza bruta - Wikipedia, la enciclopedia libre”. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Búsqueda_de_fuerza_brutaFuerza_bruta_lógica.

[6] J. Montoya (2006) La dificultad de jugar sudoku. [online]. Available: file:///C:/Users/u1/Downloads/Ladificultaddejugarsudoku.pdf