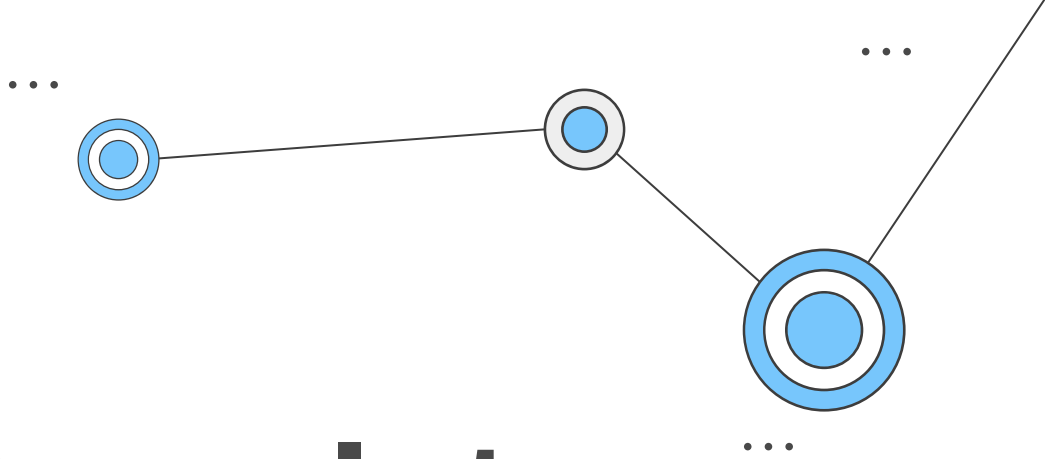


Problema NP-Completo
Sudoku

Integrantes:

-Luis Arroyo
-Sebastian Paz
-Sebastian Ugarte
-Sharon Valdivia

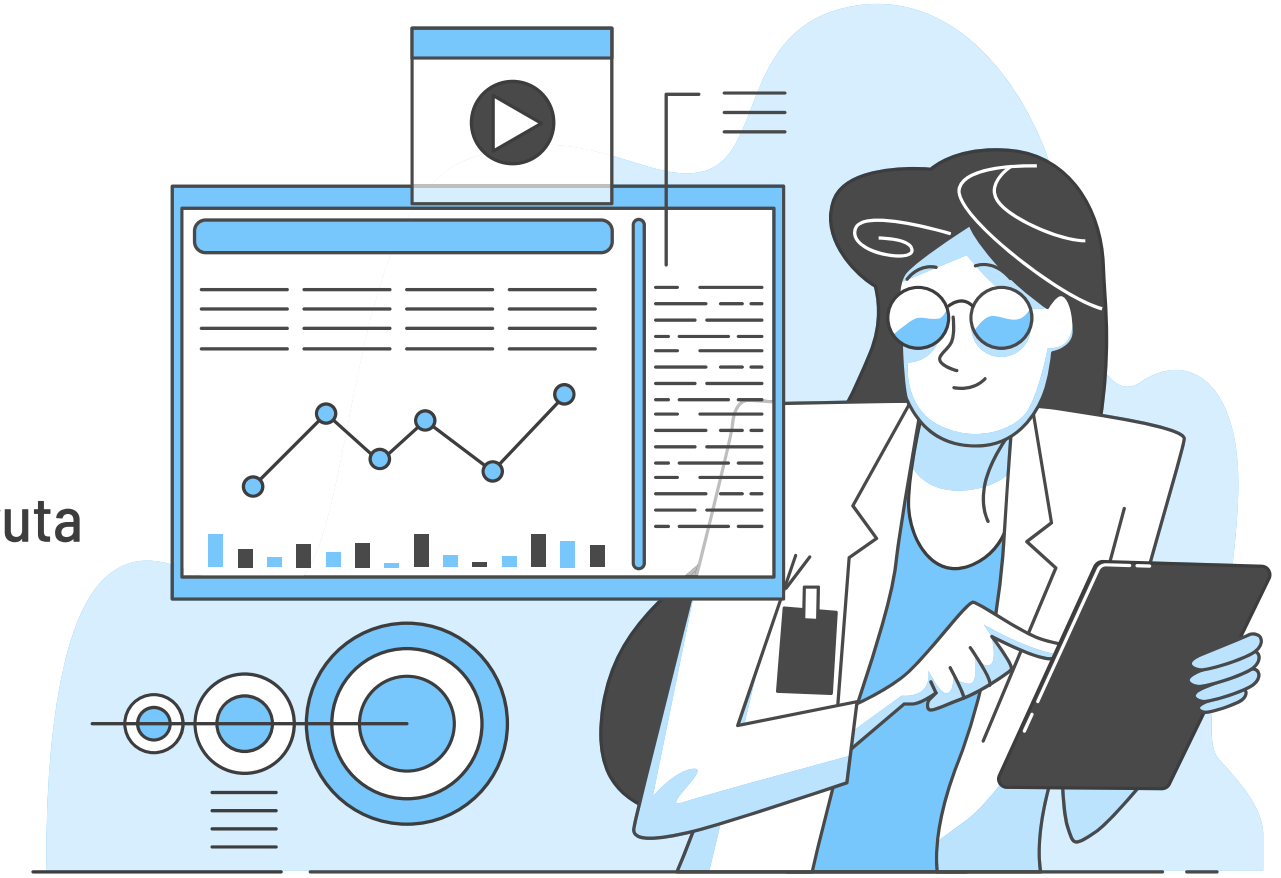
01 Introducción

02 Demostración

03 Algoritmos

Algoritmo por Fuerza Bruta

Algoritmo Aproximado



01

Introducción

Introducción

Leonhard Euler

Siglo XVIII
Crea un sistema de probabilidades para números sin repetir

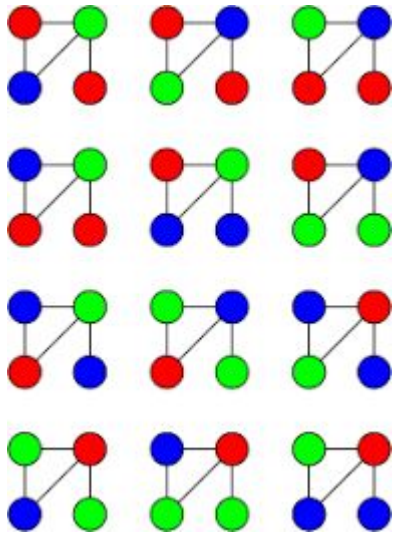
Kaji Maki

1986
Aplica las reglas actuales del sudoku como un número restringido de números y las celdas deben ser simétricas

SU = número DOKU = solo

Introducción

Para resolver este problema del **Sudoku** se demostrará que puede ser resuelto a partir del problema de **coloración de grafos** el cual es un problema 3SAT que pertenece al grupo de NP-Completos

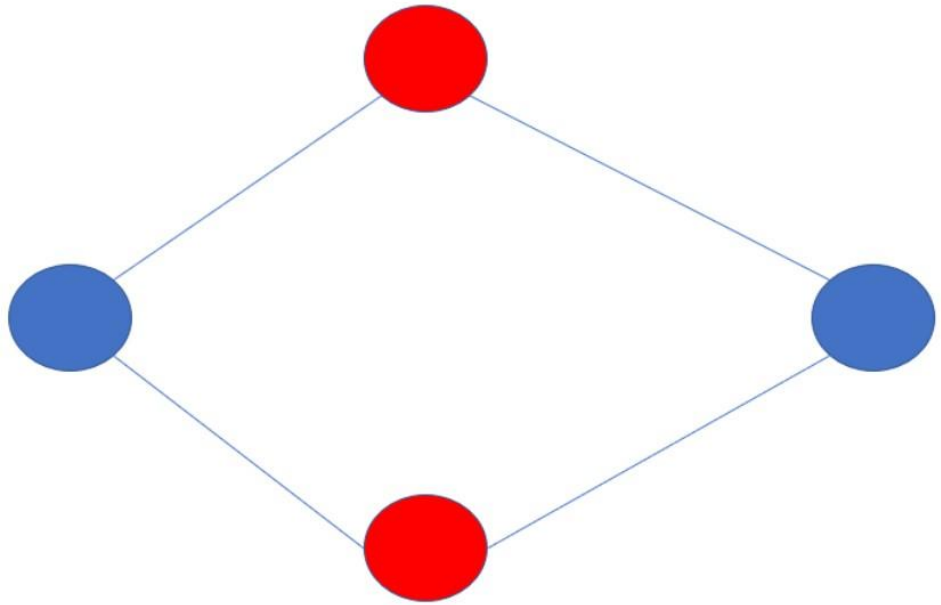


02

Demostración

Demostración

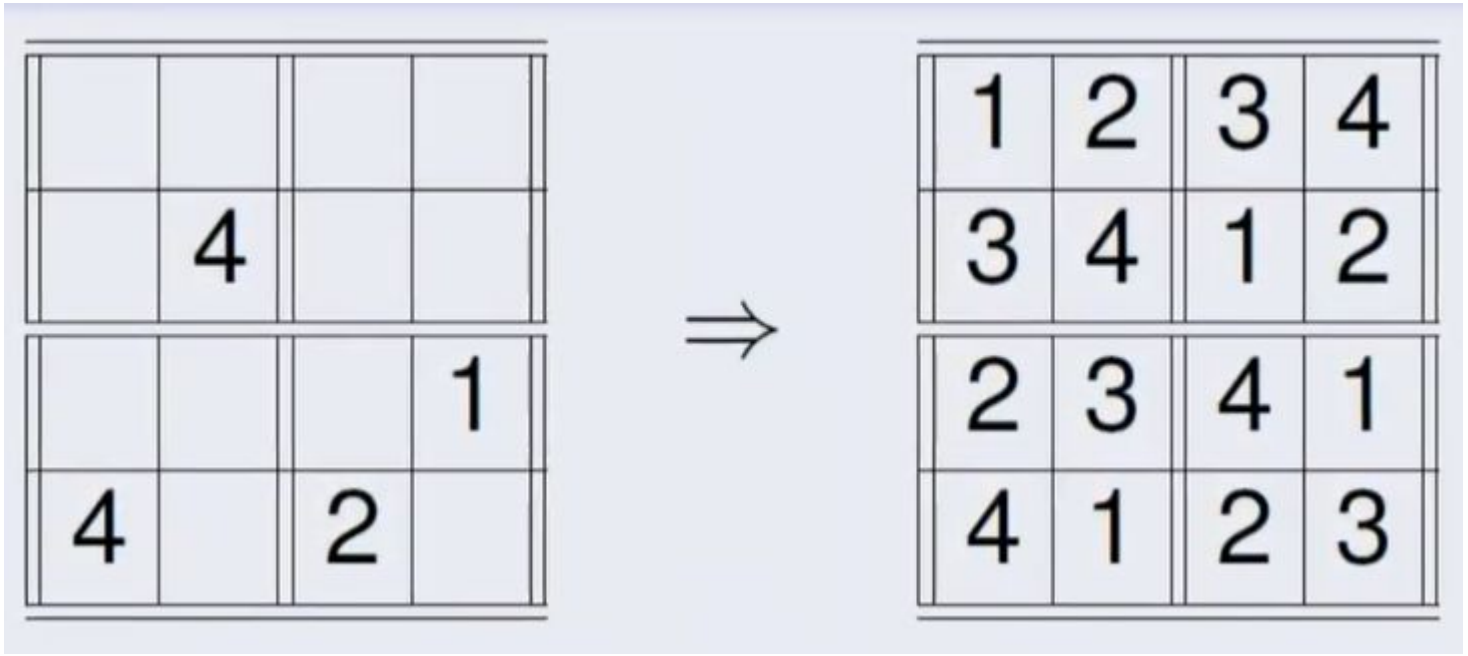
En un problema de coloreo de grafos - G, tenemos que encontrar si un grafo se puede colorear por optimización, donde se busca hallar un número mínimo de colores, o por decisión, con un conjunto de colores predefinidos 'G', también conocido como el número cromático de un grafo y se denota como X(G). Por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



G es igual a 2 ya que es el mínimo número de colores que puede tener el grafo.

Demostración

Un problema NP-Completo es un problema muy complicado y difícil, que puede llegar a resolverse transformándolo a otro problema NP-Completo, en este caso se resolverá transformando el problema del sudoku al coloreo de Grafos, siguiendo el siguiente ejemplo 4x4.



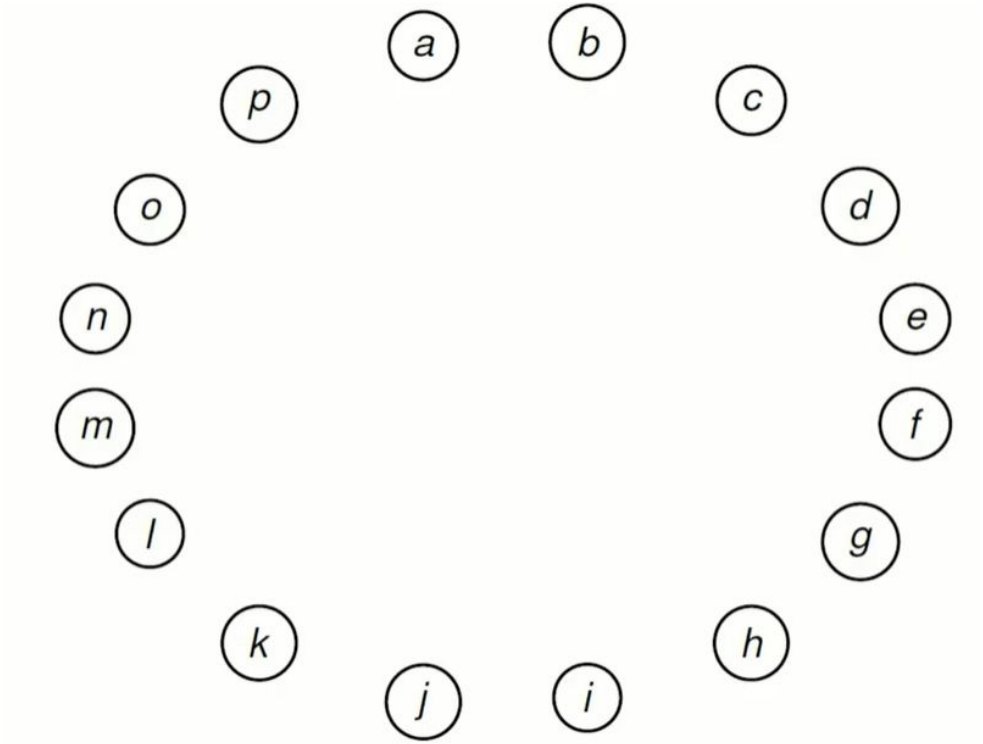
Demostración

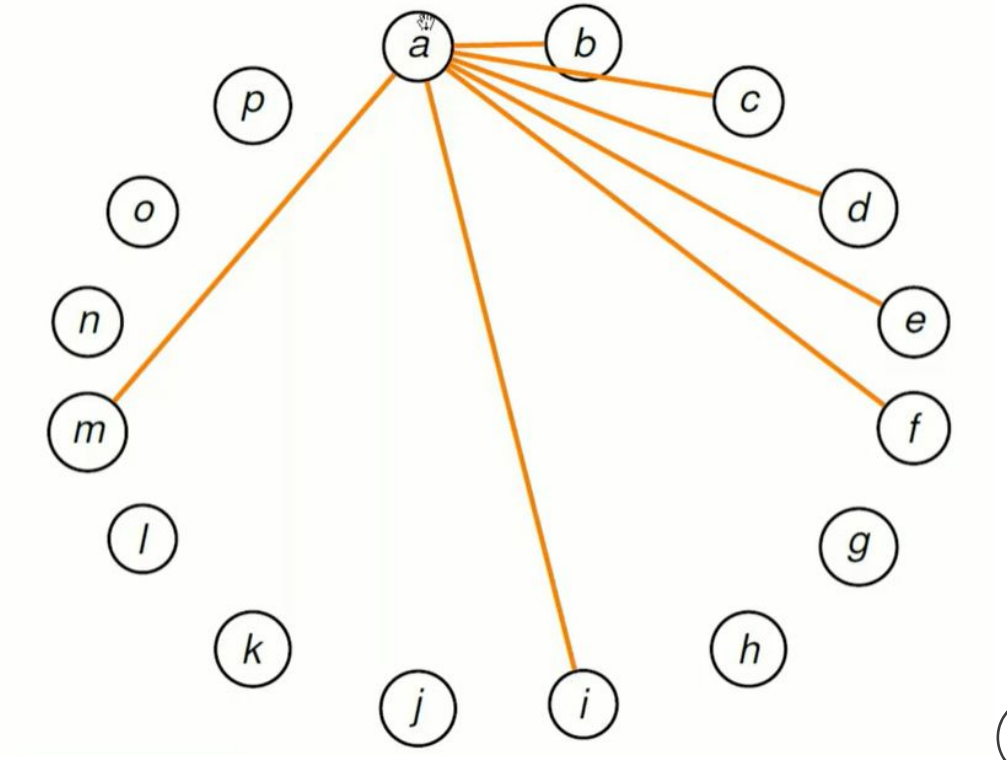
Primero se nombran cada una de las casillas de izquierda a derecha y de arriba a abajo, el orden se eligió para no dificultar la lectura, por lo que empezando con la a esta debe tener un color diferente a cualquiera otra posición en la misma fila, columna o cuadrado.

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d
e	f		
i			
m			

Demostración

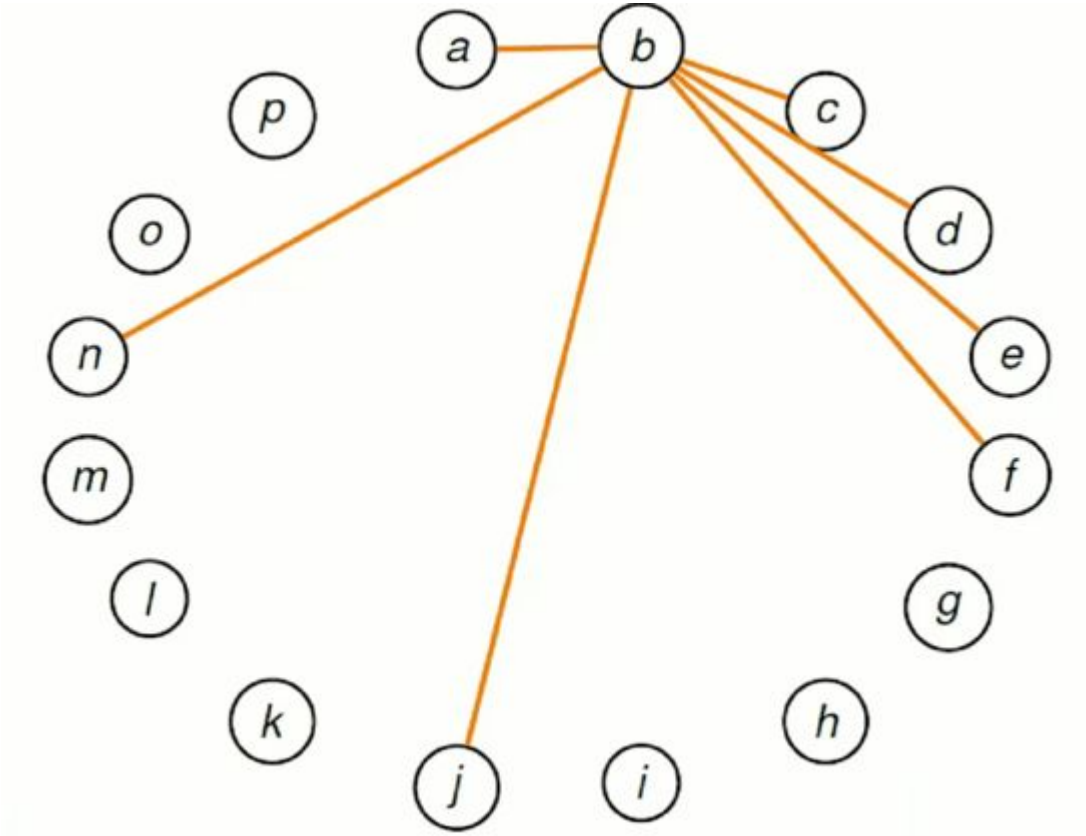




Demostración

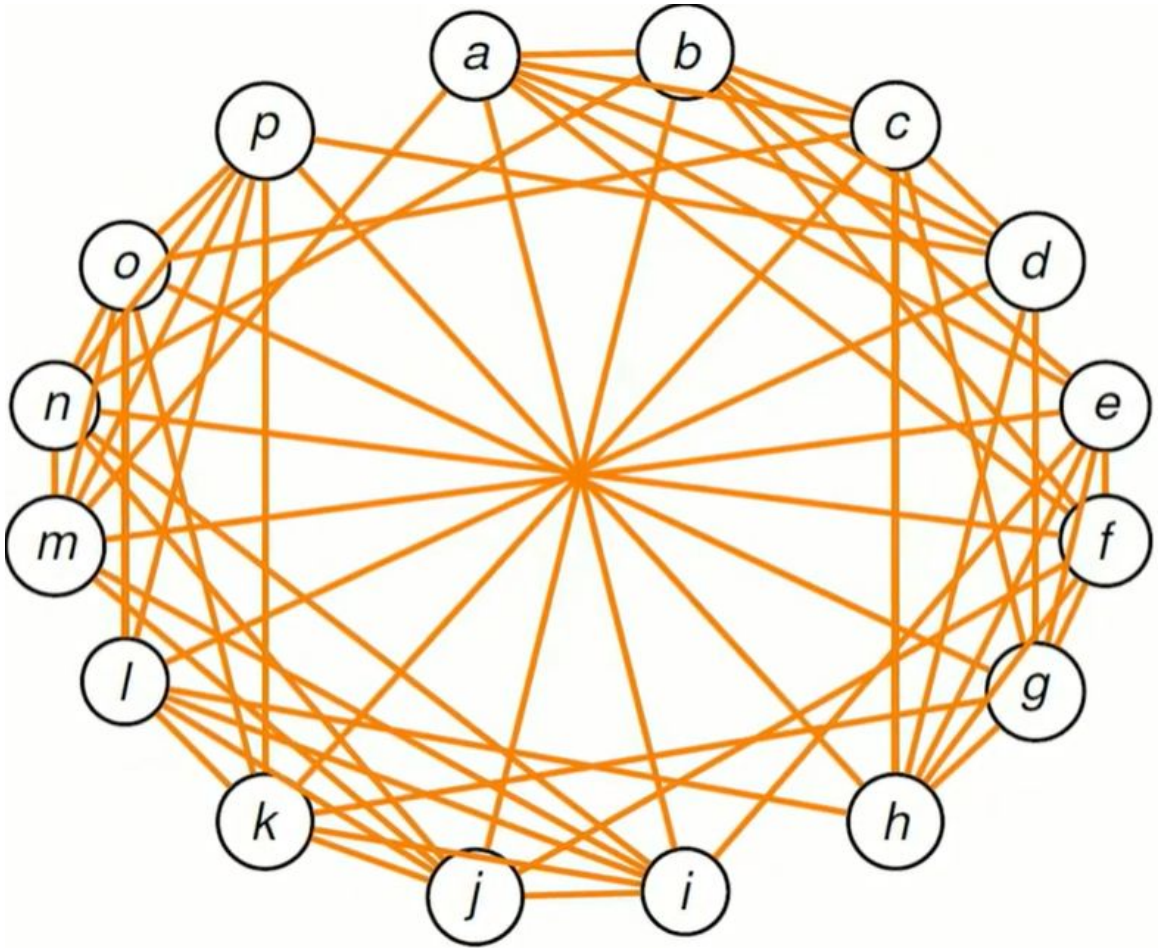
Cada casilla representa una posición que se tiene que unir a los demás vértices que estén en la misma fila, columna y cuadrado, mediante aristas ignorando las que ya tengan conexión. Continuando con todas las posiciones del sudoku.

a	b	c	d
e	f		
	j		
	n		



Demostración

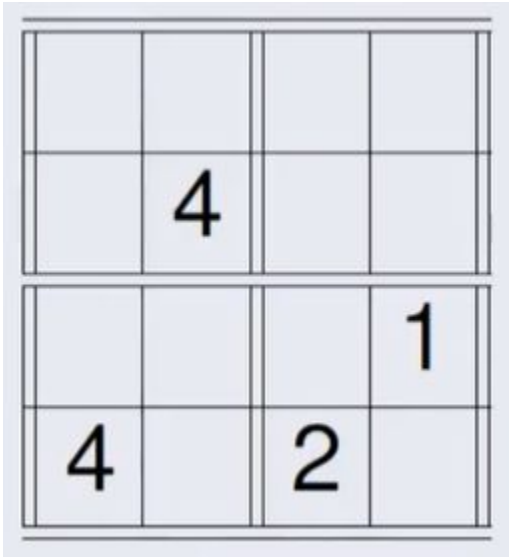
Se continua hasta que todos los vértices están unidos a través de aristas, una vez obtenido el grafo se empezará el coloreo.



Demostración

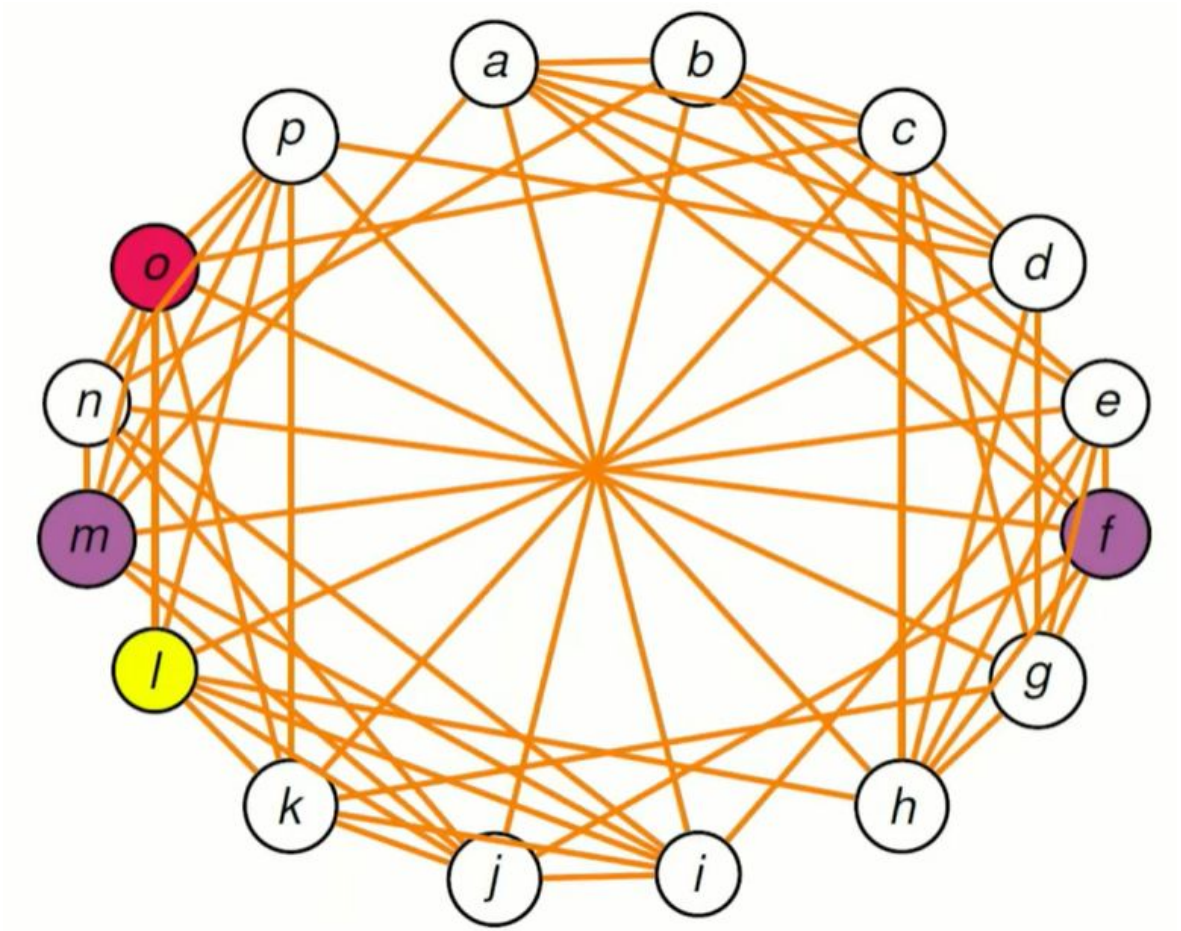
Ahora es un problema de colorear el grafo con exactamente 4 colores que se corresponden a los números 1, 2, 3 y 4:

Para este ejemplo:



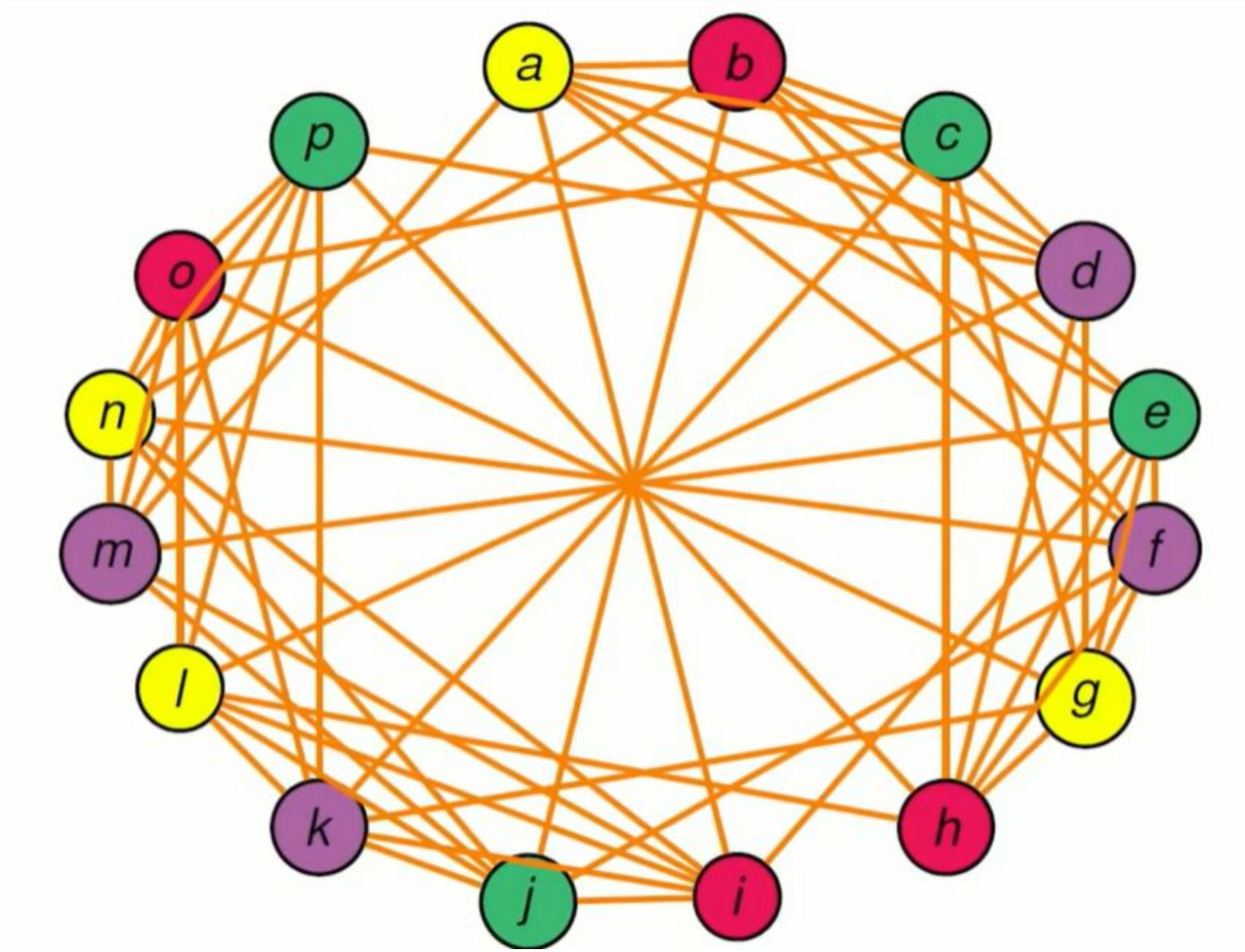
Demostración

Se elige un vértice del mayor grado V y se buscan un conjunto de vértices no adyacentes a ese vértice, de ese conjunto se elige el vértice N con el máximo de conexiones en común con V y se pinta del mismo color, se elimina N del grupo de adyacencia y se repite con el resto de elementos del grupo, Luego se elimina V del grafo y se empieza desde el paso del inicio con el siguiente vértice.

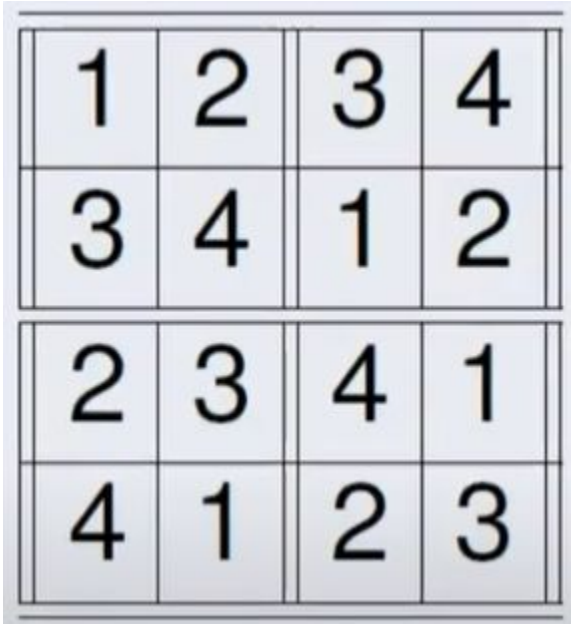


Demostración

Una vez visitado todos los vértices,todos tendrán un color asignado y el grafo final equivale a la solución del sudoku, solo faltaria asignar un número a cada color siguiendo las reglas del sudoku y colocarlo según sus posiciones.



Sudoku resuelto



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

03
Algoritmos

Algoritmo por Fuerza Bruta

Filas

Columnas

Cuadrante

```
15  bool Valido(int sudo[N][N], int fil, int col, int num)
16  {
17      for (int x = 0; x < N; x++)
18          if (sudo[fil][x] == num)
19              return false;
20      for (int x = 0; x < N; x++)
21          if (sudo[x][col] == num)
22              return false;
23      int FilaIni = fil - fil % R,
24          ColIni = col - col % R;
25
26      for (int i = 0; i < R; i++)
27          for (int j = 0; j < R; j++)
28              if (sudo[i + FilaIni][j +
29                               ColIni] == num)
30                  return false;
31
32      return true;
33  }
```

```
34  bool Sudoku(int sudo[N][N], int fil, int col)
35  {
36      if (fil == N - 1 && col == N)
37          return true;
38      if (col == N) {
39          fil++;
40          col = 0;
41      }
42      if (sudo[fil][col] > 0)
43          return Sudoku(sudo, fil, col + 1);
44
45      for (int num = 1; num <= N; num++)
46      {
47          if (Valido(sudo, fil, col, num))
48          {
49              sudo[fil][col] = num;
50              if (Sudoku(sudo, fil, col + 1))
51                  return true;
52          }
53          sudo[fil][col] = 0;
54      }
55      return false;
56  }
```

Algoritmo Aproximado

```python
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 :
```

Algoritmo Aproximado

```
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()
```

Algoritmo Aproximado

```
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)
```

Algoritmo Aproximado

```python
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 :
```

Algoritmo Aproximado

```python
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]
```

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

Algoritmo Aproximado

```
    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
```

Algoritmo Aproximado

```
    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
```

Algoritmo Aproximado

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

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

```
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
>
```