Resolución De Sudokus Mediante La Teoría De Grafos

Gabriel S. Fandiño H., Carlos A. Galán P, Rosa A. Galicia J.
Escuela de Ingeniería, Ciencia y Tecnología
Universidad del Rosario

I. Introducción.

Se dice que las primeras nociones del popular puzle del Sudoku se empezaron a escuchar desde el año 1783 por el matemático Leonhard Euler. Sin embargo, hasta 1970 Walter MacKey publicó formalmente este rompecabezas en la revista "Math Puzzles and Logic Problems". En el cual presentó un desafío para las matemáticas, que proponía rellenar una cuadrícula de tamaño $n^2 \times n^2$ con dígitos del 1 al n^2, buscando asegurar que cada columna, fila y bloque (subcuadrícula) contenga todos dígitos exactamente una vez. [1]

En la actualidad, existe diversidad de métodos que se pueden emplear para la solución del Sudoku y sus variaciones. En este contexto, se encontraron dos posibles enfoques para darle solución haciendo uso de la teoría de grafos que ofrece una perspectiva matemáticamente más rigurosa respecto a algunos métodos comunes.

Uno de los enfoques que se trabajará en este proyecto es el concepto de emparejamiento de grafos, en donde se observó que el proceso termina siendo mucho más eficaz y "elegante" que otros métodos, además está respaldado por el teorema de Berge [2]. Por otro lado, la coloración de grafos es otro método para hallar los valores que deben ir en cierta casilla, al representar los números con colores e ir coloreando el grafo que representa al Sudoku [3].

II. Descripción del problema.

El problema que aborda este proyecto es la resolución de Sudokus aplicando principios fundamentales de la teoría de grafos. Para ello, se emplearán dos enfoques distintos.

- En primer lugar, se representará cada fila del Sudoku como un grafo bipartito, junto con una fila auxiliar que contiene los números del 1 al n^2. Esta representación permite la aplicación del teorema de Berge para encontrar un emparejamiento máximo, esencial para resolver el rompecabezas.

- En segundo lugar, se llevará a cabo una representación alternativa del Sudoku mediante un grafo más completo, donde cada arista está asociada con las filas, columnas y bloques del tablero. Esta representación ampliada facilita la aplicación de algoritmos de coloración de grafos. De esta forma, al colorear adecuadamente este grafo, se logrará una solución válida para el Sudoku.

III. Objetivos.

El objetivo general de este proyecto es comparar dos algoritmos que utilizan la teoría de grafos para resolver Sudokus, incorporando diferentes conceptos como emparejamiento y coloración de grafos, respaldados por el teorema de Berge y diversos algoritmos de coloración. Además de este objetivo principal, se han identificado objetivos adicionales que se detallan a continuación:

- Representar los Sudokus como grafos, para poder aplicar los algoritmos que se proponen en ese documento.
- Diseñar e implementar un algoritmo que emplee el emparejamiento de grafos y el teorema de Berge para resolver Sudokus de manera efectiva.
- Modificar e implementar uno de varios algoritmos de coloración de grafos con el propósito de encontrar una solución para Sudokus.
- Comparar el rendimiento y la eficacia de los dos algoritmos propuestos en términos de tiempo de ejecución de las soluciones obtenidas para Sudokus de diferentes niveles de dificultad.

IV. Marco teórico.

Definición 1.1 Un grafo consiste en un conjunto de vértices V(G),un conjunto de aristas E(G) y una relación que asocia a cada arista un par de vértices(extremos) no necesariamente distintos. [4]

Definición 1.2 Un grafo simple G = (V, E) es un grafo sin bucles ni aristas múltiples, donde E es un conjunto de pares no ordenados de vértices.[4]

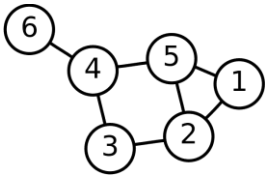


Figura 1. Ejemplo de grafo simple.

En el gráfico anterior, el conjunto de vértices V = {1,2,3,4,5,6} y el conjunto de aristas E = {12, 23, 34, 46, 45, 51,52}.

Definición 1.3 Dos vértices se dice que son adyacentes si existe una arista que les une. En tal caso, se dice que ambos vértices son incidentes en dicha arista. [4]

Se dice que un vértice es aislado si no incide en ninguna arista del grafo. Dos aristas se dice que son adyacentes si tienen un extremo en común.

Definición 1.4 Grado del vértice v, se denota por d(v), al número de aristas incidentes en v, entendiendo que un lazo aporta 2 al grado.[4]

Un grafo se dice regular si todos los vértices tienen el mimo grado. Decimos por tanto que un grafo es k-regular si todos sus vértices tienen grado k.

Definición 1.5 Un grafo G es bipartito si V(G) es la unión de dos conjuntos disyuntos independientes denominados conjuntos partitos de G. [9]

Definición 1.6 Un camino es un grafo simple cuyos vértices pueden ordenarse en una lista de tal manera que dos vértices son adyacentes si y solo si son consecutivos en la lista.[9]

Definición 1.7 Un ciclo es un grafo simple con el mismo número de vértices y aristas cuyos vértices pueden ubicarse alrededor de un círculo de tal manera que dos vértices son adyacentes si y solo s aparecen de manera consecutiva sobre el circulo.

Definición 2.1 Un emparejamiento en un grafo G es un conjunto de aristas M de G sin vértices comunes, es decir, un subgrafo donde todos los vértices tienen grado menor o igual que 1.[6]

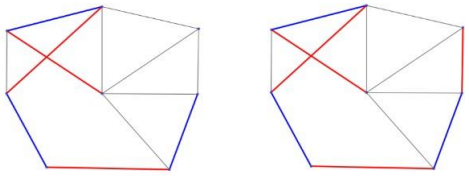
Los vértices incidentes a las aristas de un emparejamiento M son saturados por M, los demás vértices son insaturados. (M-saturado, M-insaturado).

Definición 2.2 emparejamiento máximo a aquel que contiene el número máximo posible de aristas, esto es, que no hay emparejamientos con más aristas. Un emparejamiento es perfecto si todos los vértices del grafo forman parte de él.[6]

Definición 2.3 Diferencia simétrica: Si G y H son grafos con conjunto de vértices V, entonces la diferencia simétrica G△H es el grafo con conjunto de vértices V y aristas E(G)△E(H). [6]

Definición 2.4 M-camino alternado: Sea un grafo G = (V, E) y un emparejamiento M de G. Se llama camino alternado a un camino en el cual sus aristas alternativamente pertenecen y no pertenecen a M. [6]

Definición 2.5 M-camino aumentante es un camino alternado que comienza y termina en un vértice no saturado. [6]



(a) Camino alternado (b) Camino aumentador

Figura 2. Ejemplos de caminos alternantes y aumentantes.

Teorema de Berge, Sea M un emparejamiento de un grafo G = (V, E). M es máximo si y solo si G no contiene M-caminos aumentantes.

Para la solución del problema el teorema se interpreta como: Una arista pertenece a alguna pero no a todos los emparejamientos máximos si y sólo sí. Dado un emparejamiento máximo, pertenece a un ciclo alterno par.

Definición 2.6: un componente fuertemente conectado de un dígrafo (grafo dirigido) es un subconjunto de vértices tal que existe un camino dirigido entre cualquier par de vértices dentro de ese subconjunto.

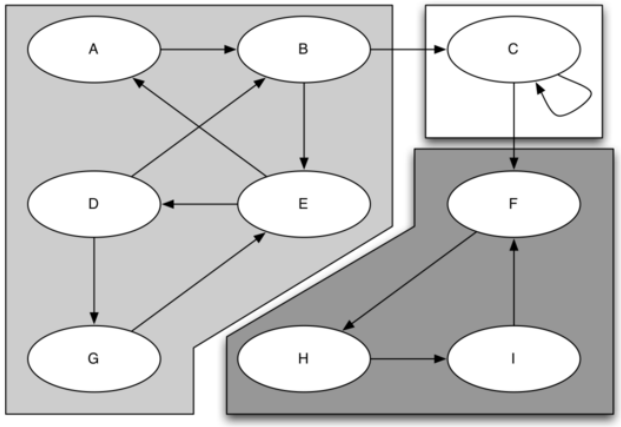


Figura 3. Componentes fuertemente conexas.

Definición 3.1: La coloración de grafos consiste en asignar distintos colores a los vértices de un grafo de tal forma que no haya dos vértices adyacentes que compartan el mismo color.[7]

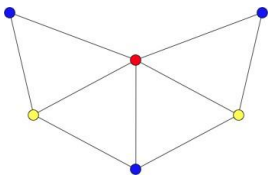


Figura 4. Ejemplo de un grafo coloreado.

Definición 3.2: El número cromático es el número mínimo de colores necesarios para colorear los vértices del grafo, denotado como $\chi(G)$ [7]. Por ejemplo, el número cromático para el grafo presente en la figura 3 es de $\chi(G) = 3$.

Definición 3.3: El grado cromático de un vértice v, también conocido como grado de saturación [10], que se representa como $sat(v)$, es el número de colores distintos utilizados para colorear sus vecinos. Por ejemplo, el vértice rojo en la figura 3 tiene un $sat(v) = 2$, a pesar de que $d(v) = 5$.

V. Modelamiento del problema.

Para comparar los dos algoritmos que se proponen, se han seleccionado dos Sudokus de niveles (n) 2 y 3 (ver figuras 1 y 2).

Esta elección se fundamenta en la necesidad de llevar a cabo una comparativa exhaustiva que abarque una variedad de escenarios. Al tomar Sudokus de distintos niveles, se amplía el alcance de la comparación, lo que nos permite evaluar la eficacia de cada algoritmo en situaciones diversas y desafiantes.

	4		
		2	
	2		
		1	

Figura 5. Sudoku de tamaño 4x4 (nivel 2).

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

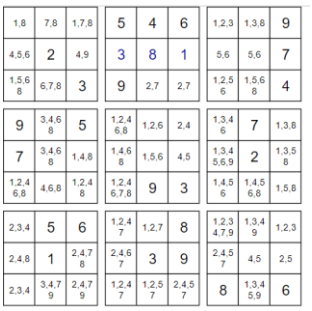
Figura 6. Sudoku de tamaño 9x9 (nivel 3).

Al incluir Sudokus de niveles 2 y 3, se busca determinar cuál algoritmo demuestra un rendimiento superior en cada uno de estos niveles por separado. Esto nos permite identificar posibles fortalezas y debilidades de cada algoritmo en relación con la complejidad del problema. Además, al comparar los resultados obtenidos en Sudokus de diferentes niveles, podemos discernir cuál algoritmo es más adecuado para resolver Sudokus de nivel 2 y cuál es más eficiente en la resolución de Sudokus de nivel 3. De esta manera, obtenemos una evaluación más completa y detallada del desempeño de cada algoritmo en diferentes contextos de dificultad.

Una vez definido los Sudokus que se utilizaran para la prueba y ejecución de los algoritmos que se proponen más adelante, es necesario la representación de los rompecabezas como grafos. De este modo, se asegura el correcto funcionamiento de los algoritmos y se establece la posibilidad de encontrar alguna solución óptima. Para ello, dado que se proponen dos algoritmos, se realizarán dos construcciones de grafos diferentes para cada una de las técnicas:

A. Primera técnica: Emparejamientos.

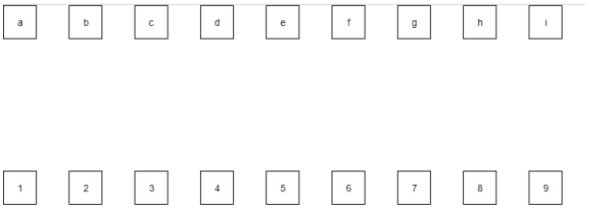
1. Dada una condición inicial con k entradas se le va a asociar a cada casilla sus posibles candidatos, lo anterior se puede visualizar como:



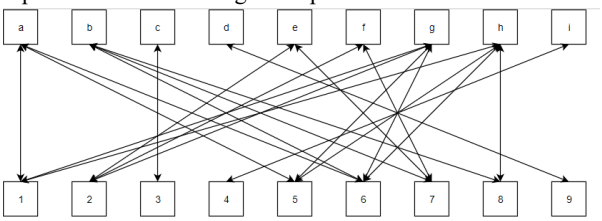
2. En principio se toma una fila, en donde todas las casillas se van a interpretar como los nodos superiores del grafo bipartito. Estos nodos serán etiquetados con letras.

3. Por otro lado en el caso del sudoku 9x9 los posibles valores que puede tomar cualquier casilla vacía estarán representados como los nodos inferiores del grafo bipartito (valores de 1-9)

Lo anterior se puede representar como:

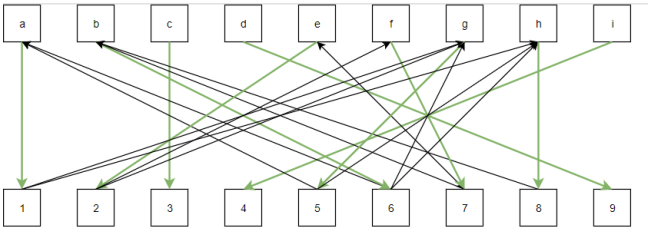


4. Cada nodo superior (casilla) será adyacente a sus posibles candidatos y tendrá una flecha de entrada y salida, es decir, se representará como un dígrafo bipartito.



Ahora usaremos los conocimientos de emparejamiento máximo y con ello verificar si el modelo es factible.

5. Para obtener una coincidencia máxima, solo queremos que la mayor cantidad de aristas que podamos encontrar cumplan los criterios de coincidencia, para ello en la librería networkx existe un método que permite encontrar un emparejamiento máximo.



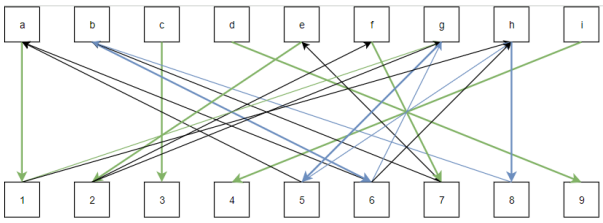
6. Se observa que cada arista del nuevo dígrafo que está en la coincidencia máxima se dirige hacia abajo y las otras aristas se dirigen hacia arriba.

Aquí es factible porque podemos encontrar una coincidencia que tenga tamaño 9 (9 aristas). Eso significa que para cada posición podemos asignar un valor para que todas las posiciones tengan un

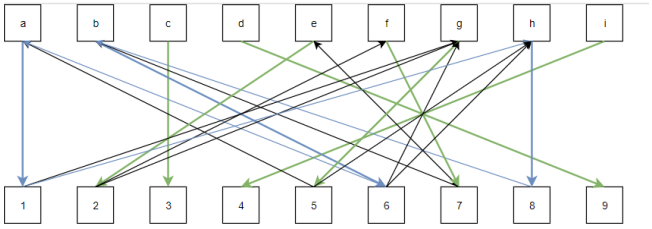
valor diferente. (En caso contrario se mostrará un error de insatisfacible)
7. La idea general ahora es encontrar todas esas coincidencias máximas y comprobar si hay aristas que no aparecen en ninguna de ellas. Dado que no es fácil ni computacionalmente factible encontrar todas las coincidencias máximas, podemos hacer uso del lema de Berge que definimos en el marco teórico.

8. Queremos saber si una arista es parte de algún emparejamiento. Y además tenemos un lema que nos dice que hay una condición que se cumple para cada arista que forma parte de al menos una coincidencia máxima.
Por Berge sabemos que, una arista pertenece a una coincidencia máxima si pertenece a un ciclo alterno par. Como nos encontramos en dígrafos bipartitos, podemos buscar componentes fuertemente conectados, lo que nos garantiza la existencia de un ciclo. Ademas cabe resaltar que el beneficio de usar un digrafo en este caso, nos garantiza una alternativa de no tener que verificar las restricciones de alternancia.

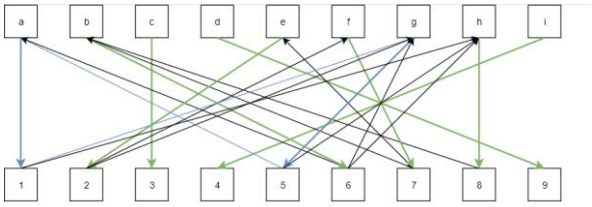
9. Ahora sólo tenemos que encontrar componentes fuertemente conectados que tengan un ciclo uniforme.
A continuación, se muestran las iteraciones hasta terminar de encontrar ciclos pares. En cada iteración se muestra en azul el ciclo par alternante encontrado.
Primera iteración:



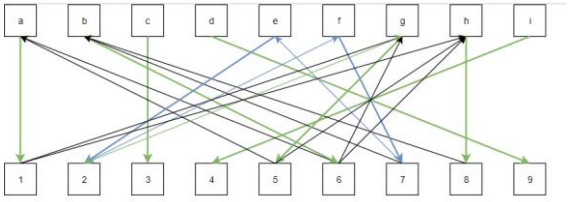
Segunda iteración:



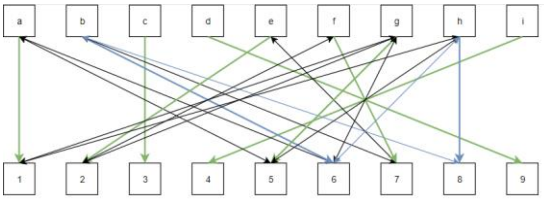
Tercera iteración:



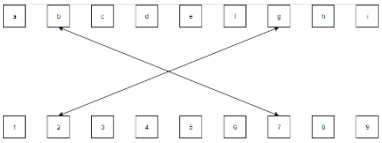
Cuarta iteración:



Quinta iteración:



10. Ahora se eliminarán todas las aristas usadas de la primera coincidencia máxima(coloreadas en verde), así como las aristas de los ciclos alternantes pares encontrados(coloreados de color azul). Quedando así exactamente los dos bordes que queríamos eliminar:



En este caso a las casillas b y g de la fila se le deberá quitar de su lista de candidatos posibles los valores 7 y 2 respectivamente. Y se volverá a repetir el proceso para todas las filas del sudoku.

B. Segunda técnica: Coloración de grafos.

En esta segunda parte, es necesario construir un grafo para cada uno de los Sudokus propuestos, de modo que cada vértice contenga la información suficiente para determinar el color más adecuado para pintarlo. Para ello, se ha generalizado la construcción de los grafos, con el fin de agilizar el proceso de creación, como se detalla a continuación:

- Vértices: cada casilla A corresponde a un vértice A en el grafo G asociado al Sudoku S.

- Aristas: si dos casillas A y B en el Sudoku S pertenecen a la misma fila, columna o bloque, sus vértices homólogos A y B en el grafo G serán adyacentes.

- Colores: si el valor de la casilla A en el Sudoku S es nulo, entonces el color del vértice asociado A en el grafo G será de color blanco. Si el valor es diferente de 0, entonces se sigue la lista de colores por cada valor posible a continuación:

    1. #FF0000 (Rojo).
    2. #00FF00 (Verde).
    3. #008FFF (Azul).
    4. #FFFF00 (Amarillo).
    5. #FF00FF (Rosa).
    6. #00FFFF (Turquesa).
    7. #FF8F00 (Naranja).
    8. #FF008F (Salmon).
    9. #8F00FF (Morado).

Teniendo en cuenta la generalización en la construcción de grafos y utilizando las bibliotecas networkx y matplotlib de Python, se logró visualizar los grafos asociados a cada Sudoku antes de ser resueltos por el algoritmo de coloración. Esta generalización permitió crear grafos de manera más eficiente, con cada vértice conteniendo la información necesaria para determinar el color adecuado.

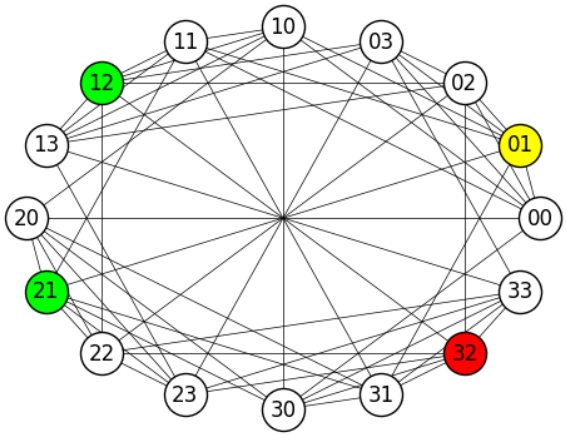


Figura 6. Grafo asociado al Sudoku 4x4.

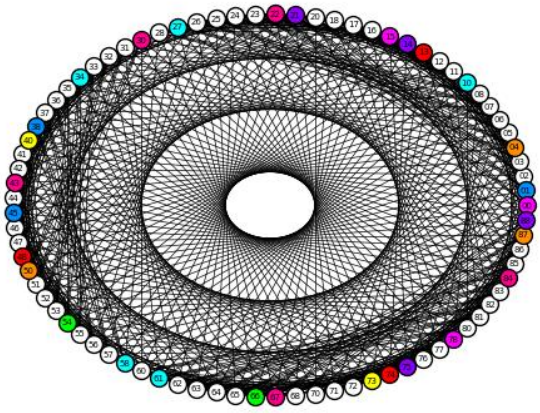


Figura 7. Grafo asociado al Sudoku 9x9.

VI. Solución propuesta.

Para comenzar, el primer paso es representar los Sudokus de una manera estructurada y manejable. Para lograr esto, empleamos nuestros conocimientos en programación orientada a objetos para crear una representación eficiente y flexible.

Comenzamos diseñando una clase llamada "Casilla", la cual encapsula tanto la posición de una casilla en el tablero como el valor numérico que contiene. Si una casilla aún no tiene un valor asignado, su valor se establece en 0. Además, implementamos métodos que permiten modificar el valor de una casilla según sea necesario.

A continuación, damos paso a la clase "Sudoku", la cual representa un juego de Sudoku completo. Esta clase no solo almacena el nivel de dificultad del Sudoku en cuestión, sino que también gestiona conjuntos de casillas y colores. En estos conjuntos se almacenan todas las casillas del rompecabezas y los colores asociados a cada número, respectivamente.

Además de gestionar la información básica del Sudoku, la clase "Sudoku" incluye dos funciones esenciales. La primera función se encarga de actualizar el estado del Sudoku en función de los cambios realizados en las casillas. Por otro lado, la segunda función permite identificar los posibles valores que puede tomar una casilla que aún no tiene asignado ningún valor, ayudando así al jugador a avanzar en la resolución del rompecabezas.

Gracias a esta estructura de clases, podemos representar de manera más intuitiva y eficiente los Sudokus como grafos. Cada casilla se convierte en un vértice del grafo, y las relaciones entre las casillas se modelan mediante aristas. Esto nos brinda una base sólida para aplicar algoritmos de búsqueda y resolución sobre los Sudokus, facilitando su análisis y resolución.

Para abordar la resolución del puzle, es crucial implementar algoritmos efectivos. En este sentido, adoptaremos dos enfoques distintos: uno basado en emparejamientos y otro en la coloración de grafos.

A. Primer método: Emparejamientos.

En este método, nuestra estrategia principal consiste en representar cada fila con una serie de números auxiliares. No buscamos simplemente los valores de cada casilla, sino identificar valores que, aunque posibles, no son viables para la solución del Sudoku. Para lograrlo, debemos revisar continuamente el rompecabezas y actualizar los valores de las casillas cuyos valores actuales son 0 y solo tienen una posibilidad de elección. Utilizaremos la función "actualizar" de la clase Sudoku para este propósito. Además, el Teorema de Berge nos será de utilidad para determinar qué posibilidades no son viables para la solución, especialmente al identificar aristas que no pertenecen a un camino o ciclo alternantes, lo que indica que no forman parte de un emparejamiento máximo [2].

B. Segundo método: Coloración de Grafos.

En este enfoque, utilizamos la teoría de grafos para determinar el número cromático del grafo que representa un Sudoku. Recordemos que el número cromático de un grafo es el número mínimo de colores necesarios para pintar todos sus vértices de manera que dos vértices adyacentes no compartan el mismo color. En nuestro caso, al representar el Sudoku como un grafo, intuimos que su número cromático es 4 o 9, dependiendo del contexto, ya que asignamos un color por cada valor posible en una casilla.

Para alcanzar este objetivo, partimos del algoritmo DSatur (Degree of Saturation). Este algoritmo ordena los vértices de un grafo por su grado de saturación, de forma descendente. Luego, recorre todos los vértices en el orden determinado y los colorea utilizando diferentes colores a los utilizados por sus vecinos adyacentes. [10]

Sin embargo, ningún algoritmo de coloración de grafos garantiza un coloreado óptimo. Por ello, modificamos el algoritmo DSatur para ampliar su aplicabilidad en la resolución de diferentes variantes de Sudoku. En lugar de seguir un orden fijo, el algoritmo recorre el grafo libremente, seleccionando el vértice con el mayor grado de

saturación en cada paso, lo que permite una coloración más flexible y efectiva.

VII. Implementación de la solución.

La implementación de los dos algoritmos propuesto en este proyecto fue realizada en Python, puesto que este lenguaje de programación está enfocado en la programación orientada a objetos, ofreciendo diferentes herramientas útiles en la creación de gráficos, como networkx y matplotlib, además de contar con una sintaxis sencilla y fácil de entender. De este modo, a continuación, se detalla la implementación de cada algoritmo:

A. Algoritmo por emparejamiento.

En comparación a como se implementó el algoritmo de coloración, la implementación del método de emparejamiento máximo se planteó de otra manera:
En donde se creó una clase de error que mostrará un mensaje en el terminal en caso en el que el grafo no sea factible.
También se creó la clase Model() en donde se va a ir actualizando el sudoku mediante la aplicación del algoritmo que se creó a partir del emparejamiento máximo y la búsqueda de ciclos alternantes.
Todo el desarrollo del algoritmo se encuentra en la función check_constraint() en donde para asegurar que cuando haya un vértice libre se ejecute el error de insatisfabilidad se implementó:

```
for n in GM.nodes():
    if str(n)[:2] != "x_" and len(list(GM.predecessors(n))) == 0:
        print("Vertice libre: ", n)
        raise InfeasibleError("Vertice libre no debería existir")
```

Aquí conectamos en el grafo bipartito, para cada casilla en la fila que estamos estudiando las aristas a sus valores corresponiendes:

```
G = nx.MultiDiGraph()

already_know = {}
for i in range(len(values)):
    if 'values' in values[i]:
        for j in values[i]['values']:
            G.add_edge('x_' + str(i), j)
    else:
        G.add_edge('x_' + str(i), values[i]['value'])
        already_know[i] = 1
```

Además afortunadamente la librería networkx contiene métodos como máximum_matching y strongly_conected_components que fueron implementados en el mismo método check_constraint() de la siguiente manera:

```
matching = nx.bipartite.maximum_matching(G, top_nodes=["x_" + str(i) for i in range(len(values))])
n_matching = []
GM = nx.DiGraph()
```

```
scc = nx.strongly_connected_components(GM)
for component in scc:
    for node in component:
        if str(node)[:2] != 'x_':
            pass
        else:
            idx = int(node[2:])  # Obtiene el índice de la variable
            if 'values' not in possible[idx]:
                possible[idx] = {'values': set()}
            for edge in component:
                if str(edge)[:2] != 'x_':
                    possible[idx]['values'].add(edge)
new_possible = []
new_knowledge = [False] * len(values)
i = 0
for p in possible:
    l = list(p['values'])
    if len(l) == 1:
        new_possible.append({'value': l[0]})
        if i not in already_know:
            new_knowledge[i] = True
    else:
        new_possible.append({'values': l[:]})
        if len(l) < len(values[i]['values']):
            new_knowledge[i] = True

    i += 1
```

```
old_changed = self.changed.copy()
self.changed[ss_idx] = new_knowledge
self.changed = np.logical_or(self.changed, old_changed)
self.search_space[ss_idx] = new_possible
```

Por utlimo es importante resaltar que los sudokus que se quieren ejecutar deberán estar en formato txt y en el main se leeran línea por línea de la siguiente manera:

```
grid = [[0] * 9 for i in range(9)]
f = open(sys.argv[1], "r")
lines = f.readlines()
f.close()
c = 0
for line in lines:
    grid[c] = list(map(int, line.split()))
    c += 1

grid = np.array(grid)
```

B. Algoritmo de coloración de grafos.

El siguiente seudo código representa el funcionamiento normal del algoritmo:

Input: un grafo G asociado a un Sudoku S.
Output: el grafo G con todos sus vértices coloreados.
Algoritmo:
1. Mientras que exista un vértice sin colorear
   a. c = colores
   b. v = primer vértice
   c. Por cada vértice u en el grafo:
      i. Si sat(u) > sat(v) y u no está coloreado: v = u
   d. Por cada vértice w en los vecinos de v:
      i. Eliminar el color de w en c.
   e. Asignar a v el primer color en c.

El seudo código anterior proporciona una idea general de cómo funciona nuestro algoritmo. Al ejecutarlo sobre un grafo que

representa un Sudoku, el algoritmo puede colorear todos los vértices del grafo si el Sudoku tiene una solución relativamente sencilla. Al final, cada color en cada vértice determina qué valor debe ir en la casilla asociada al vértice, siguiendo el esquema de colores definido en la modelación del problema.

Sin embargo, como se mencionó previamente, este algoritmo no siempre garantiza una coloración óptima. Por lo tanto, si se ejecuta el algoritmo sobre un grafo que representa un Sudoku con una solución compleja, el algoritmo puede fallar y no ser capaz de colorear todos los vértices del grafo.

VIII. Resultados obtenidos.

A. Método de emparejamientos.

B. Método de coloración de grafos.

Al implementar el algoritmo propuesto para la coloración de grafos obtenemos los siguientes resultados:

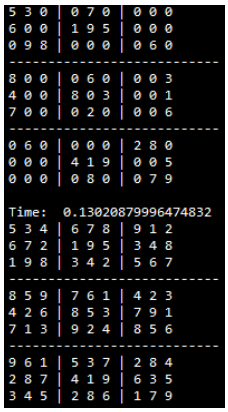


Figura y. Resultados del código para el Sudoku 9x9

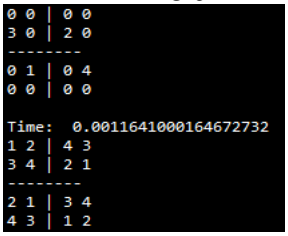


Figura x. Resultados del algoritmo para el Sudoku 4x4

Al analizar los tiempos de ejecución para cada uno de los dos rompecabezas, se nota una clara diferencia entre los niveles 2, tardando aproximadamente 1.16 milisegundos en resolverlo; y 3, resolviéndolo en 130.2 milisegundos. Estos resultados evidencian la clara dificultad que tiene resolver un Sudoku de 9x9, frente a la dificultad que puede tener uno de 4x4. A continuación se los grafos coloreados que representan estas dos soluciones:

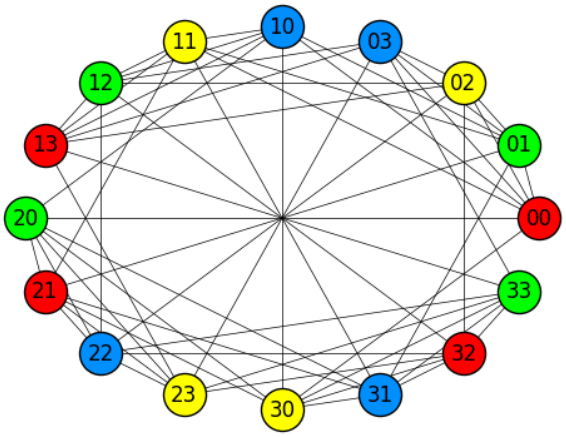


Figura z. Grafo asociado al Sudoku 4x4 coloreado.

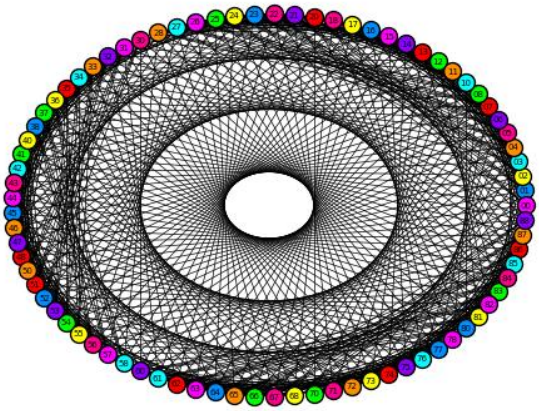


Figura z. Grafo asociado al Sudoku 9x9 coloreado.

IX. Conclusiones.

A. Método de emparejamientos.

B. Método de coloración de grafos.

Ventajas de implementar el algoritmo de coloración de grafos:

- Es capaz de resolver Sudokus de diferentes niveles.
- La implementación es mucho más sencilla que usando emparejamiento.

Desventajas de implementar este algoritmo:

- Requiere de más 30 entradas para poder resolver Sudokus 9x9.
- Tarda más tiempo en dar una respuesta.

X. Bibliografía.

[1]. A. Duran, 'Solving Sudoku puzzles with Graph Theory', 17 July-2023. [Online]. available in: https://acortar.link/Qmlsy2.

[2]. Solving Sudokus Like A Human, (5 de septiembre de 2019). Accedido: 12 de abril de 2024. [OnlineVideo]. Disponible en: https://acortar.link/tmttMf.
[3]. Teoría de grafos: cómo resolver sudokus, (4 de marzo de 2019). Accedido: 20 de abril de 2024. [En línea Video]. Disponible en: https://acortar.link/zAPWkb.
[4]. Unipamplona, 'Teoria de Grafos'[Online] Disponible en: http://surl.li/svefg
[5]. Graph Matchings[Online slides]: http://surl.li/svdzo
[6]. P. Santamaría, 'EMPAREJAMIENTOS ESTABLES', Octubre – 2020.[Online tesis]. Disponible en: http://surl.li/sveac
[7]. Graph Coloring and Chromatic Numbers, Brilliant.org tomado de: http://surl.li/sveca
[8]. O. Garcia, 'Algoritmo para determinar la K-Coloración de un grafo'[Online tesis] Disponible em: http://surl.li/svecs
[9]. Diapositivas de clase.
[10]. S. P. Seijas, «El Problema de Coloración de Grafos».