INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

COLOREADO DE GRAFOS

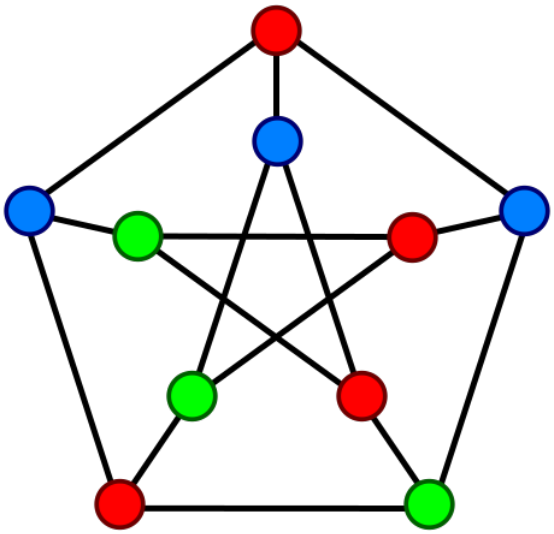
Computer Science and Engineering

(Artificial Intelligence & Machine Learning)

By

Marcos Damián Pool Canul

Oscar Manuel Ruiz Reyes



ING. DATOS E INTELIGENCIA ORGANIZACIONAL

Profesor: Ricardo Armando Ruíz Hernández

February, 2024

TABLE OF CONTENTS

CHAPTER 1

Introducción

La presente documentación se centra en los diversos algoritmos de Coloración de Grafos, una rama intrigante y profundamente relevante de la teoría de grafos. Los grafos, estructuras matemáticas compuestas por vértices y aristas, son fundamentales en numerosas aplicaciones científicas y prácticas, incluyendo la Inteligencia Artificial. En particular, la coloración de grafos, que implica asignar colores a los vértices de un grafo de tal manera que ningún par de vértices adyacentes comparta el mismo color, se revela como una herramienta poderosa en la representación y análisis de mapas geográficos.

El coloreado de grafos es un área fascinante y significativa de la teoría de grafos, crucial en diversas aplicaciones prácticas y científicas, incluyendo la inteligencia artificial. Este documento se centra en explorar varios algoritmos de coloreado de grafos, ofreciendo un análisis detallado de su implementación y eficacia. Comenzando con una revisión de la teoría de grafos y las librerías principales utilizadas, se profundiza en la metodología de evaluación de algoritmos como los de Welsh y Powell, Matula Marble Isaacson, y Brelaz. Se presentan implementaciones detalladas en Python, acompañadas de ilustraciones y análisis exhaustivos, culminando con evaluaciones críticas de cada método.

1.1 Teoría de Grafos

Los vértices o nodos, constituyen uno de los dos elementos que forman un grafo. Se define también como la unidad fundamental de la que están compuestos los grafos. Los vértices son tratados, en la teoría de Grafos, como unidades indivisibles y sin propiedades, aunque pueden tener estructuras adicionales dependiendo del uso del grafo al que pertenecen.

1.1.1 Vértice

Los vértices o nodos, constituyen uno de los dos elementos que forman un grafo. Se define también como la unidad fundamental de la que están compuestos los grafos. Los vértices son tratados, en la teoría de Grafos, como unidades indivisibles y sin propiedades, aunque pueden tener estructuras adicionales dependiendo del uso del grafo al que pertenecen.

1.1.2 Arista

Las aristas, junto con los vértices, forman los elementos principales de un grafo. Se definen como las uniones entre nodos o vértices. Usualmente las aristas denotan relaciones entre los vértices, como el orden, la vecindad o la herencia.

Las aristas además de unir dos vértices suelen tener una dirección establecida. Es decir, que $a \to b$ sería una arista distinta que $a \leftarrow b$, pudiendo existir ambas en el mismo grafo $a \leftrightarrow b$. Siendo estos grafos conocidos como dirigidos.

1.1.3 Grafo

Un grafo es un par de conjuntos, $G = (V, A)$, donde V es un conjunto finito no vacío de elementos llamados vértices y A es un conjunto finito de pares no ordenados de vértices de V, llamados aristas, ambos relacionados mediante la aplicación T, donde $T = V \to A$.

Existen varios tipos de grafos, a continuación, se expondrán los más importantes:

- **Grafo dirigido**: También llamado dígrafo, es aquel grafo en el que la relación entre los elementos considera su dirección, la relación T no es simétrica. Se caracterizan porque cada arista tiene una dirección asignada, expresada como: $a = u \to v$. Su relación se expresa de la siguiente manera: $(u, v) \neq (v, u)$.

- **Grafo no dirigido**: Son llamados grafos, éste no contempla dirección de sus aristas, su principal característica es que sus aristas son pares no ordenados de vértices, lo que significa que la relación T entre ellos es simétrica, es decir, en un grafo $G = (V, A)$ entonces, $(u, v) = (v, u)$.

- **Grafo completo**: Un grafo G es completo si cada vértice tiene un grado igual a $n - 1$, siendo n el número de vértices que componen el grafo. Además, presentan una arista entre cada par de vértices del grafo, es decir que todos los vértices son adyacentes. Esto proporciona un conjunto A de $m = \frac{n(n-1)}{2}$ aristas. Siendo m el número de aristas del conjunto A.

- **Grafo conexo**: Decimos que un grafo es conexo si consiste de una sola pieza.

- **Grafo inconexo**: Si consiste de varios pedazos, a los que se les llama componentes.

- **Grafo regular**: Un grafo cuyos vértices tienen el mismo grado o val

1.2 Librerias principales

Para empezar con este proyecto, necesitamos de ciertas librerías para crear los grafos, graficar, tabular, etc. A continuación se enumeran las principales:

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 from random import randint, choice, random
4 from tabulate import tabulate
```

- `networkx`: Esta librería de Python se especializa en la creación, manipulación y estudio de la estructura, dinámicas y funciones de redes complejas. Es ampliamente utilizada para análisis de redes, ya sean redes sociales, redes de transporte, redes eléctricas, entre otras.

- `matplotlib.pyplot`: Es una sublibrería de 'matplotlib' y se utiliza para la creación de gráficos y visualizaciones estáticas, animadas e interactivas en Python. Es muy versátil y se usa comúnmente para representar datos y resultados en forma de gráficos como histogramas, gráficos de líneas, gráficos de dispersión, etc.

- `random`: Esta librería ofrece una serie de funciones que permiten generar números aleatorios. Esto es especialmente útil en simulaciones, pruebas aleatorias y en cualquier situación donde se requiera la aleatoriedad, como la selección aleatoria de nodos o aristas en una red.

- `tabulate`: Esta librería se utiliza para imprimir tablas en Python. Es muy útil para la presentación de datos en una forma estructurada y fácilmente legible. Puede ser utilizada para mostrar los datos de los nodos, aristas, o cualquier otro conjunto de datos relacionados con la red de una manera clara y organizada.

1.3 Ingresar el número de vértices

Para comenzar la construcción de la red, el primer paso es ingresar el número de vértices. Esto se hace solicitando al usuario que introduzca el número deseado a través de la consola. El código correspondiente en Python sería el siguiente:

```
numero_vertices = int(input("Numero de vertices: "))
print("Numero de vertices: ", numero_vertices)
```

Este código solicita al usuario que ingrese el número de vértices para la red y luego imprime este número para confirmar la entrada. La función input recoge la entrada del usuario, int convierte esta entrada en un entero, y print se utiliza para mostrar el valor ingresado.

1.4 Funciones principales

En esta seccion se explicaran todas las funciones que se utilizaron aparte de los algortitmos de coloracion.

1.4.1 Crear un grafo conectado

```
def grafo_conectado(numero_vertices):
    """
    Crea un grafo conectado con numero_vertices vertices.
    """
    G = nx.Graph()
    G.add_nodes_from(range(numero_vertices))

    # Asegurarnos de que el grafo este conectado
    for i in range(numero_vertices - 1):
        G.add_edge(i, i + 1)

    # Agregar mas aristas aleatoriamente para hacer el grafico
       mas complejo
    additional_edges = numero_vertices // 2
    for _ in range(additional_edges):
        v1, v2 = randint(0, numero_vertices -
                         1), randint(0, numero_vertices - 1)
        while G.has_edge(v1, v2) or v1 == v2:
            v1, v2 = randint(0, numero_vertices -
                             1), randint(0, numero_vertices -
                                 1)
```

```
20            G.add_edge(v1, v2)
21        return G
```

La función `grafo_conectado` realiza lo siguiente:

1. **Inicialización del Grafo**: Crea una instancia de un grafo vacío utilizando la clase `Graph` de la librería `networkx`.

2. **Adición de Vértices**: Añade vértices al grafo. Los vértices son numerados de 0 a `numero_vertices - 1`.

3. **Conexión de Vértices**: Para asegurar que el grafo esté conectado, la función crea una arista entre cada par de vértices consecutivos (es decir, entre el vértice i y $i + 1$), formando una cadena.

4. **Añadiendo Aristas Adicionales**: Para incrementar la complejidad del grafo, la función agrega un número adicional de aristas, que es la mitad del número de vértices. Estas aristas se generan aleatoriamente entre pares de vértices distintos, asegurándose de que no se creen bucles (aristas que conectan un vértice consigo mismo) ni aristas duplicadas.

5. **Retorno del Grafo**: Finalmente, la función retorna el grafo generado.

1.4.2 Colorear el Grafo Inicialmente

Una función importante en la manipulación de grafos es `colores_iniciales`, que se encarga de asignar colores iniciales a los vértices del grafo. A continuación, se presenta la definición de la función y luego se explica su funcionamiento:

```
1  def colores_iniciales(G, colores=['red', 'green', 'blue',
       'yellow']):
2      """
3      Asigna colores iniciales a los vertices del grafo.
4      """
5      for nodo in G.nodes():
6          G.nodes[nodo]['color'] = choice(colores)
7
8      return G
```

Explicación del Funcionamiento

La función `colores_iniciales` realiza las siguientes operaciones:

1. **Parámetros de Entrada**: - `G`: El grafo al cual se le asignarán los colores. - `colores`: Una lista de colores disponibles para asignar a los vértices. Por defecto, se utilizan 'red', 'green', 'blue' y 'yellow'.

2. **Asignación de Colores**: - La función recorre todos los nodos (vértices) del grafo `G`. - Para cada nodo, se elige un color al azar de la lista `colores` utilizando

la función `choice` de la librería `random`. - Este color se asigna al atributo 'color' del nodo correspondiente en el grafo.

3. **Retorno del Grafo**: - Tras asignar colores a todos los nodos, la función devuelve el grafo `G` con los colores de los nodos actualizados.

Esta función es útil para inicializar el estado de un grafo con colores asignados aleatoriamente a sus vértices, lo cual puede ser relevante en algoritmos de coloreado de grafos, visualización, o en problemas que requieran diferenciar nodos por colores.

1.4.3 Dibujar el Grafo

Una función esencial en la visualización de grafos es `dibujar_grafo`, que se encarga de dibujar el grafo con vértices coloreados según los colores asignados previamente. A continuación, se presenta la definición de la función y luego se explica su funcionamiento:

```
1  def dibujar_grafo(G, titulo):
2      """
3      Dibuja el grafo con v rtices coloreados.
4      """
5      plt.figure(figsize=(8, 6))
6      mapa_colores = [G.nodes[nodo]['color'] for nodo in G.
           nodes()]
7      nx.draw(G, with_labels=True, node_color=mapa_colores,
8              node_size=500, font_size=10)
9      plt.title(titulo)
10     plt.show()
```

Explicación del Funcionamiento

La función `dibujar_grafo` realiza las siguientes operaciones:

1. **Parámetros de Entrada**: - `G`: El grafo que se va a dibujar. - `titulo`: El título que se mostrará en la visualización del grafo.

2. **Preparación de la Visualización**: - Se crea una figura con un tamaño específico usando `plt.figure`. - Se genera un mapa de colores, donde cada color corresponde al atributo 'color' asignado a cada nodo en el grafo.

3. **Dibujo del Grafo**: - Se utiliza la función `nx.draw` de la librería `networkx` para dibujar el grafo. - Los parámetros incluyen: - `with_labels=True` para mostrar las etiquetas de los nodos. - `node_color=mapa_colores` para aplicar los colores a los nodos. - `node_size` y `font_size` para ajustar el tamaño de los nodos y la fuente de las etiquetas.

4. **Mostrar el Grafo**: - Se establece el título de la figura con `plt.title`. - Finalmente, se muestra el grafo utilizando `plt.show`.

Esta función es especialmente útil para visualizar la estructura del grafo y la asignación de colores a los nodos, lo cual es importante en análisis de redes, estudios de grafos y en la presentación de resultados de algoritmos de grafos.

1.4.4 Crear Tabla de Grados

Otra función relevante en el análisis de grafos es `crear_tabla_grados`, que genera una tabla para mostrar el grado de cada vértice en el grafo. A continuación, se muestra la definición de la función y luego se proporciona una explicación de su funcionamiento:

```
def crear_tabla_grados(G, orden_inverso=False):
    """
    Crea una tabla para mostrar cada v rtice y su grado.
    """
    # Ordenar los v rtices seg n el grado (el orden
       inverso depende del par metro)
    vertices_ordenados = sorted(
        G.nodes(), key=lambda x: G.degree(x), reverse=
            orden_inverso)

    # Crear una lista de tuplas (v rtice, grado)
    lista_grados_vertices = [(f"V rtice {vertice}", f"
       Grado {G.degree(vertice)}") for vertice in
       vertices_ordenados]

    # Crear una tabla usando tabulate
    tabla = tabulate(lista_grados_vertices, headers=[
                     'V rtice', 'Grado'], tablefmt='grid
                        ')
    return tabla
```

Explicación del Funcionamiento

La función `crear_tabla_grados` realiza las siguientes operaciones:

1. **Parámetros de Entrada**: - `G`: El grafo del cual se calcularán los grados de los vértices. - `orden_inverso`: Un parámetro booleano que determina si la tabla se ordenará en orden inverso de grado (de mayor a menor).

2. **Ordenamiento de Vértices**: - Los vértices del grafo se ordenan según su grado. El orden puede ser directo (de menor a mayor grado) o inverso, dependiendo del valor de `orden_inverso`.

3. **Creación de la Lista de Grados**: - Se crea una lista de tuplas, donde cada tupla contiene el identificador del vértice y su grado correspondiente.

4. **Generación de la Tabla**: - Utilizando la librería `tabulate`, se genera una tabla a partir de la lista de grados. Los encabezados de la tabla son 'Vértice' y 'Grado'.

5. **Retorno de la Tabla**: - La función devuelve la tabla generada, la cual muestra de manera clara y estructurada el grado de cada vértice en el grafo.

Esta función es útil para visualizar y analizar rápidamente la distribución de los grados de los vértices en un grafo, lo cual es importante en estudios de conectividad y estructura de redes.

CHAPTER 2

Algoritmo de Coloración de Wells y Powell

Algoritmo también conocido como "primero el de mayor grado". Por lo que, en este algoritmo los vértices se ordenan de acuerdo a sus grados, de mayor a menor. Se ordena de forma que $d(v_1) \geq d(v_2) \geq \ldots \geq d(v_n)$, donde $d(v)$ representa el grado del vértice.

El algoritmo Welsh-Powell es un método eficiente para colorear vértices de un grafo. Su funcionamiento se detalla a continuación:

1. Primero, se ordenan los vértices del grafo en orden descendente según su grado.
2. Después, cada vértice se colorea de manera individual, utilizando el menor número (color) posible, asegurándose de que los vértices adyacentes no tengan el mismo color.

2.1 Pseudocódigo

1. Abrir un archivo, recibir el número de vértices y aristas, y crear la matriz o lista de adyacencias.
2. Iniciar el Algoritmo Welsh-Powell.
3. Ordenar los vértices de mayor a menor grado, seleccionar el primero en la lista y colorearlo o etiquetarlo con el número 1.
4. Tomar el siguiente vértice y colorearlo o etiquetarlo con el menor número admisible, verificando las adyacencias.
5. Repetir el paso 4 hasta que todos los vértices hayan sido coloreados.
6. Mostrar los resultados.

2.2 Implementación en Python

```
1  def algoritmo_welsh_powell(G, colores=['red', 'green', 'blue',
       'yellow']):
2      vertices_ordenados = sorted(
3          G.nodes(), key=lambda x: G.degree(x), reverse=True)
4      for nodo in vertices_ordenados:
5          colores_adyacentes = {G.nodes[vecino]['color']
6                                for vecino in G.neighbors(nodo)}
7          for color in colores:
8              if color not in colores_adyacentes:
9                  G.nodes[nodo]['color'] = color
10                 break
11     return G
```

2.3 Visualización

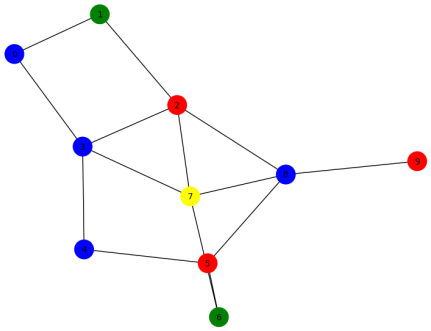


Figure 2.1: Gráfica de colores iniciales

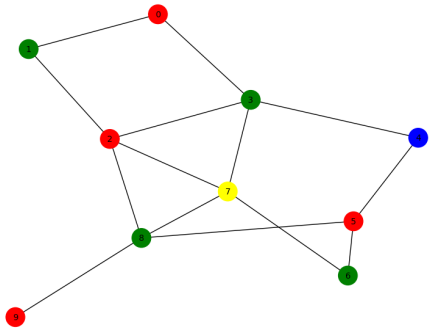


Figure 2.2: Gráfico coloreado con el algoritmo Welsh-Powell

Table 2.1: Grados de los vértices de la Coloración Welsh y Powell

No. de Vértices	Grado del Vértice
Vértice 2	Grado 4
Vértice 3	Grado 4
Vértice 7	Grado 4
Vértice 8	Grado 4
Vértice 5	Grado 3
Vértice 0	Grado 2
Vértice 1	Grado 2
Vértice 4	Grado 2
Vértice 6	Grado 2
Vértice 9	Grado 1

CHAPTER 3

Algoritmo de Coloración de Matula, Marble, Isaacson

"El de menor grado el último"

Esta variante se debe a Marble, Matula e Isaacson. Se ordenan los vértices en orden inverso. Primero se elige v_n como el vértice de menor grado, luego se elige v_{n-1} como el vértice de menor grado en $G - \{v_n\}$, y así se continúa recursivamente, examinando los vértices de menor grado y eliminándolos del grafo.

3.1 Pseudocódigo

```
Algoritmo MenorGradoUltimo(G):
   Inicializar lista L vacía
   Mientras G tenga vértices:
      Encontrar vn, el vértice de menor grado en G
      Añadir vn al inicio de L
      Eliminar vn de G
   Fin Mientras
   Retornar L
Fin Algoritmo
```

3.2 Implementación en Python

```
1  def algoritmo_matula_marble_isaacson(G, colores=['red', 'green'
       , 'blue', 'yellow']):
2      vertices_ordenados = sorted(G.nodes(), key=lambda x: G.
           degree(x))
3      for nodo in vertices_ordenados:
4          colores_adyacentes = {G.nodes[vecino]['color']
5                                for vecino in G.neighbors(nodo)}
6          for color in colores:
7              if color not in colores_adyacentes:
8                  G.nodes[nodo]['color'] = color
9                  break
10     return G
```

3.3 Visualización

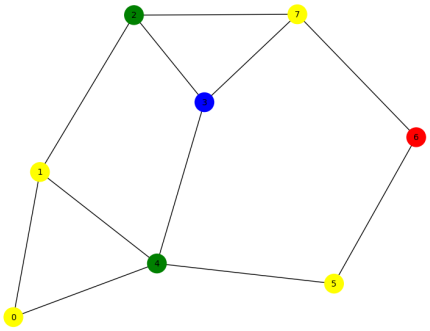


Figure 3.1: Gráfica de colores iniciales

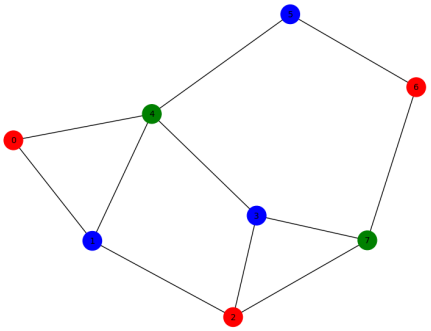


Figure 3.2: Gráfico coloreado con el algoritmo Matula, Marble, Isaacson

Table 3.1: Grados de los vértices de coloración Matula, Marble, Isaacson

No. de vértices	Grado de vértices
Vértice 0	Grado 2
Vértice 5	Grado 2
Vértice 6	Grado 2
Vértice 1	Grado 3
Vértice 2	Grado 3
Vértice 3	Grado 3
Vértice 7	Grado 3
Vértice 4	Grado 4

CHAPTER 4

Algoritmo de Coloración de Brelaz

El algoritmo de Brelaz, conocido comúnmente como DSATUR (Degree of SATURation), es un método heurístico para el coloreo de grafos que busca minimizar el número de colores utilizados para pintar un grafo, asegurando que dos vértices conectados directamente no compartan el mismo color. Este algoritmo se destaca por su enfoque en el grado de saturación (d) de los vértices, una métrica clave que indica cuántos colores diferentes están adyacentes a un vértice dado. Al concentrarse en el grado de saturación, el algoritmo de Brelaz mejora significativamente la eficiencia del proceso de coloreo en comparación con los métodos que solo consideran el grado del vértice. La aplicación del algoritmo de Brelaz es fundamental en diversas áreas que involucran problemas de asignación y organización, como la creación de horarios escolares y universitarios, la asignación de canales en sistemas de comunicación y la resolución de problemas de asignación en redes informáticas.

4.1 Pseudocódigo

```
Algoritmo ColoreoBrelaz(G):
   Ordenar vértices de G por grado decreciente
   Inicializar color de todos los vértices a -1 y dsat a 0
   Colorear vértice de mayor grado con 0

   Mientras haya vértices sin color:
      Seleccionar vértice sin color con máximo dsat (usar
      grado como desempate)
      Marcar colores usados por vecinos
      Asignar menor color disponible al vértice
      Actualizar dsat de vecinos

   Retornar coloreo de G
```

4.2 Implementación en Python

```
1  def brelaz_coloring(graph):
2      # Paso 1
3      vertices = sorted(graph, key=lambda x: len(graph[x]),
           reverse=True)
4      color = {v: -1 for v in vertices}  # -1 significa sin color
5      dsat = {v: 0 for v in vertices}
6
7      # Paso 2
8      color[vertices[0]] = 0
9
10     while -1 in color.values():
11         # Paso 3
12         max_dsat = max(dsat, key=dsat.get)
13         available_colors = [True] * len(vertices)
14
15         for neighbor in graph[max_dsat]:
16             if color[neighbor] != -1:
17                 available_colors[color[neighbor]] = False
18
19         # Paso 4
20         color[max_dsat] = available_colors.index(True)
21
22         for neighbor in graph[max_dsat]:
23             dsat[neighbor] = len([n for n in graph[neighbor] if
                   color[n] != -1])
24
25     # Paso 5
26     return color
```

4.3 Visualización

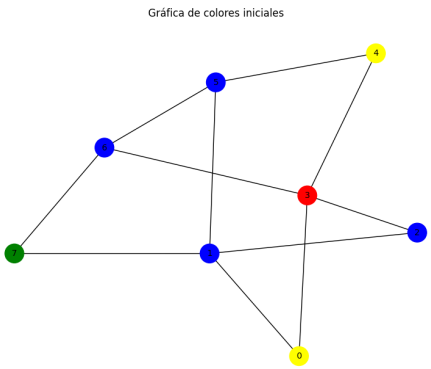


Figure 4.1: Gráfica de colores iniciales

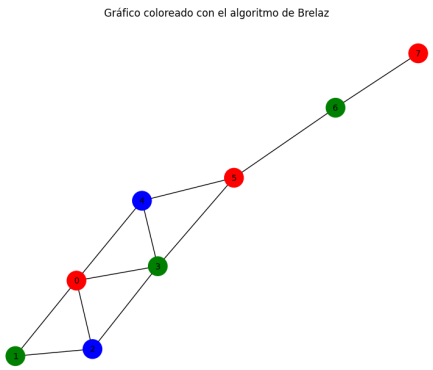


Figure 4.2: Gráfico coloreado con el algoritmo de Brelaz

Table 4.1: Grado de saturación de cada vertices

No. de Vértices	Grado de saturación
Vértice 0	Grado 2
Vértice 1	Grado 2
Vértice 2	Grado 2
Vértice 3	Grado 2
Vértice 4	Grado 2
Vértice 5	Grado 2
Vértice 6	Grado 1
Vértice 7	Grado 1

CHAPTER 5

Conclusiones

En este trabajo, hemos explorado y analizado tres algoritmos fundamentales para el coloreado de grafos: el Algoritmo de Welsh y Powell, el Algoritmo de Matula Marble Isaacson y el Algoritmo de Brelaz. Cada uno de estos algoritmos ofrece una perspectiva única y soluciones eficientes para el problema del coloreado de grafos, que es central en muchas aplicaciones prácticas en ciencia de datos, optimización de redes y teoría de grafos.

El Algoritmo de Welsh y Powell, conocido por su enfoque secuencial y eficiente, se destaca por su simplicidad y eficacia en grafos con una amplia gama de distribuciones de grado. Este algoritmo es particularmente eficiente en grafos donde la distribución del grado es no uniforme, permitiendo una rápida coloración y minimizando el número de colores necesarios.

Por otro lado, el Algoritmo de Matula Marble Isaacson adopta un enfoque inverso, comenzando con los vértices de menor grado. Este método resulta ser especialmente eficaz en grafos donde los vértices de menor grado son menos densos, lo que ayuda a reducir la probabilidad de conflictos de colores y, en consecuencia, el número total de colores utilizados.

Finalmente, el Algoritmo de Brelaz, también conocido como DSATUR, introduce una heurística basada en el grado de saturación, que es una medida del número de colores diferentes a los que está adyacente un vértice. Este enfoque no solo considera el grado del vértice, sino también su contexto dentro del grafo, lo que lo hace altamente efectivo y versátil para una amplia variedad de grafos.

A través de la implementación y el análisis detallado de estos algoritmos, hemos podido apreciar sus fortalezas y limitaciones. Mientras que algunos algoritmos pueden ser más adecuados para ciertos tipos de grafos, otros pueden ser preferibles en diferentes contextos. Esta comprensión no solo enriquece nuestra apreciación de la teoría de grafos, sino que también abre caminos para futuras investigaciones y aplicaciones prácticas en campos que van desde la optimización de redes hasta la ciencia de datos y la inteligencia artificial.

Este trabajo subraya la importancia y la utilidad de los algoritmos de coloreado de grafos. Al comparar y contrastar diferentes enfoques, hemos obtenido una visión más profunda de cómo estas estrategias pueden ser aplicadas en el mundo real, proporcionando una base sólida para futuras investigaciones y desarrollos en este campo fascinante.

Bibliography

[1] Bondy, J. A., & Murty, U. S. R. (2008). *Graph Theory*. Springer.

[2] Diestel, R. (2017). *Graph Theory* (5th ed.). Springer-Verlag.

[3] Welsh, D. J. A., & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 85-86.

[4] Matula, D. W., Marble, G., & Isaacson, J. D. (1972). Graph coloring algorithms. En *Graph Theory and Computing* (pp. 109-122). Academic Press.