Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Maestría en Ciencias

# Un algoritmo genético para el coloreo de grafos

Matemáticas Discretas: Otoño 2024

Alhely González Luna

Diciembre 9, 2024

# Contents

# 1 Introducción

Los problemas relacionados con la coloración de mapas de regiones, como mapas de partes del mundo, han generado numerosos resultados en la teoría de grafos. Al colorear un mapa, es habitual asignar colores diferentes a dos regiones que comparten una frontera. Una manera de asegurar que dos regiones adyacentes nunca tengan el mismo color es utilizar un color distinto para cada región. Sin embargo, esto resulta ineficiente y, en mapas con muchas regiones, sería difícil distinguir colores similares. Por lo tanto, se debe emplear un número reducido de colores siempre que sea posible. [4]

Consideremos el problema de determinar el número mínimo de colores necesarios para colorear un mapa de manera que las regiones adyacentes nunca compartan el mismo color, dicho problema dio inicio al problema del coloreo. [4]

Tal problema puede verse como uno de combinatoria, en la que los nodos de un grafo se acomodan en cajas de colores, cuidando que nodos adyacentes no se coloquen en la misma caja.

El problema de coloreo de grafos es NP-completo y hasta el día de hoy no existe un algoritmo que lo resuelva en tiempo polinomial, por lo que algoritmos que optimizen la búsqueda son necesarios. [5]

Los algoritmos genéticos son un tipo de algoritmos basados en el concepto de evolución y supervivencia del más apto y son frecuentemente utilizados para optimización en problemas de combinatoria, por tal razón representan un buen candidato para solucionar el problema del coloreo.

En este documento se presenta un algoritmo genérico sencillo desarrollado en python con operadores de cruce y mutación para encontrar el número cromático de algunos grafos y el arreglo más apto de colores que no presenta conflictos (nodos adyacentes del mismo color) si es que existe. El documento se organiza como sigue: en la segunda sección hacemos una revisión más a fondo sobre la teoría de problema del coloreo y algunos de los algoritmos que lo resuelven, en la sección tres se presenta una introducción a algoritmos genéticos. En la sección cuatro se muestra el análisis y diseño del algoritmo así como la justificación del lenguaje de programación y el impacto en el área de computación matemática, en la sección cinco mostramos resultados y por último se presentan las conclusiones.

# 2 El problema del coloreo

Antes de pasar al problema del coloreo daremos algunas definiciones necesarias:

## 2.1 Definiciones

Un grafo G = (V, E) consiste en V, un conjunto no vacío de vértices (o nodos) y E, un conjunto de aristas. Cada arista tiene asociado uno o dos vértices, llamados sus extremos. Se dice que una arista conecta sus extremos.[4]

Si queremos señalar que algún grafo G tiene V como conjunto de vértices y E como conjunto de aristas, escribimos G = (V,E). Si hablamos de algún grafo conocido G y queremos referirnos a su conjunto de vértices, lo denotamos por V(G). De manera similar, escribimos E(G) para el conjunto de aristas de G.[1]

1. Dos vértices u y v en un grafo no dirigido G se llaman **adyacentes** (o vecinos) en G si $u$, $v$ son extremos de una arista $e$ de G. Tal arista se llama **incidente** con los vértices $u$,$v$ y se dice que $e$ conecta $u$ y $v$.[4]

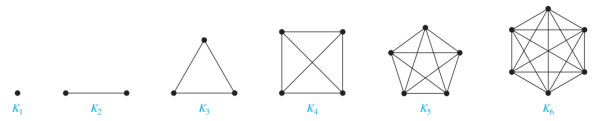
2. El conjunto de todos los vecinos de un vértice v de G = (V, E), denotado por N(v), se llama **vecindario** de $v$. Si A es un subconjunto de V, denotamos por N(A) el conjunto de todos los vértices en G que son adyacentes a al menos un vértice en A.[4]

3. El **grado** de un vértice en un grafo no dirigido es el número de aristas incidentes con él, excepto que un bucle en un vértice contribuye dos veces al grado de ese vértice. El grado del vértice v se denota por deg(v).[4]

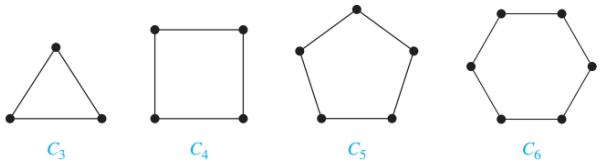
4. Cuando *(u, v)* es una arista del grafo G con aristas dirigidas, se dice que u es adyacente a v y v se dice que es adyacente desde u. El vértice u se llama vértice inicial de *(u, v)*, y v se llama vértice terminal o final de *(u, v)*. El vértice inicial y el vértice terminal de un bucle son iguales [4].

## 2.2 Grafos Especiales

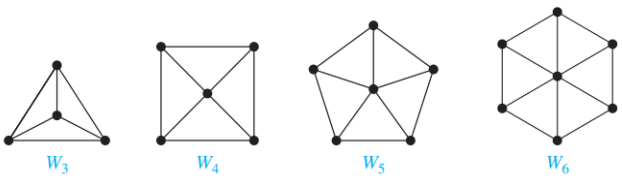
1. **Un grafo completo** con n vértices, denotado por $K_n$, es un grafo simple que contiene exactamente una arista entre cada par de vértices distintos. Los grafos$K_n$, para n = 1, 2, 3, 4, 5, 6, se muestran en la Figura . Un grafo simple para el que hay al menos un par de vértices distintos no conectados por una arista se llama no completo.[4]




2. **Un ciclo** $C_n$, $n >= 3$, consiste en n vértices $v_1, v_2, ..., v_n$ y aristas $\{v_1, v_2\}$, $\{v_2, v_3\}$,... , $\{v_{n-1}, v_n\}$ y $\{v_n, v_1\}$. [4]




3. **Rueda** Obtenemos una rueda $W_n$ cuando agregamos un vértice adicional a un ciclo $C_n$, para $n >= 3$, y conectamos este nuevo vértice a cada uno de los n vértices en $Cn$, mediante nuevas arista [4]



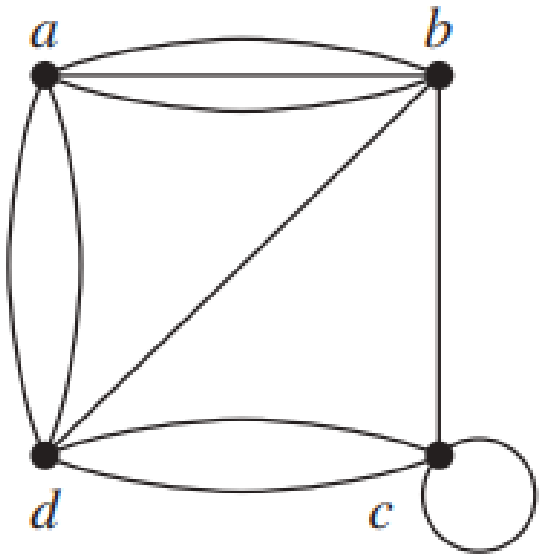

4. **Grafo bipartito** Un grafo simple G se llama bipartito si su conjunto de vértices V se puede dividir en dos conjuntos disjuntos V1 y V2 de modo que cada arista del grafo conecte un vértice en V1 y un vértice en V2 (de modo que ninguna arista en G conecte dos vértices en V1 o dos vértices en V2). Cuando se cumple esta condición, llamamos al par (V1, V2) una bipartición del conjunto de vértices V de G. [4]

## 2.3 Matriz de Adyacencia

Sea G = (V,E) un grafo con n vértices. Denote los vértices por $v_1, v_2, ..., v_n$ (en algún orden arbitrario)[1]. La **matriz de adyacencia** de G, con respecto a la numeración de vértices elegida, es una matriz n×n $A_G = (a_{ij})_{i,j=1}^{n}$ definida por la siguiente regla:

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

Por ejemplo, el grafo




tiene como matriz de adyacencia

$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}.$$

## 2.4 Coloreo de grafos

Una coloración de un grafo simple es la asignación de un color a cada vértice del grafo de manera que ningún par de vértices adyacentes reciba el mismo color. Un grafo puede ser coloreado asignando un color diferente a cada uno de sus vértices. Sin embargo, para la mayoría de los grafos, es posible encontrar una coloración que utilice menos colores que el número total de vértices del grafo. El menor número de colores necesarios para colorear un grafo, de forma que dos vértices adyacentes no compartan el mismo color, se denomina *número cromático del grafo*. El número cromático de un grafo G se denota como $\chi(G)$[4]. Se dice que G es k-coloreable si existe es posible asignarle a cada uno de los vertices de un grafo un arreglo de los k colores sin que nodos adyancentes tengan el mismo color [8].

### 2.4.1 Aplicación en problemas de optimización

La **coloración de grafos** tiene una amplia aplicación en una gran variedad de problemas complejos relacionados con la optimización. En particular, la resolución de conflictos o la partición óptima de eventos mutuamente excluyentes a menudo puede lograrse mediante la coloración de grafos. Ejemplos de tales problemas incluyen [9]:

- La programación de exámenes en el menor número de periodos de tiempo, de forma que ninguna persona tenga que participar en dos exámenes simultáneamente.
- El almacenamiento de químicos en el menor número de estantes, asegurando que no se coloquen juntos dos químicos mutuamente peligrosos (es decir, peligrosos cuando están en presencia uno del otro).

En cada uno de estos problemas, las restricciones suelen expresarse en forma de pares de objetos incompatibles (por ejemplo, pares de químicos que no pueden almacenarse en el mismo estante). Estas incompatibilidades se representan de manera útil mediante la estructura de un grafo, donde cada objeto es representado por un nodo y cada incompatibilidad es representada por una arista que une dos nodos. Una coloración de este grafo corresponde entonces a una partición de los objetos en bloques (o colores) de tal forma que no haya dos objetos incompatibles en el mismo bloque. Por lo tanto, las soluciones óptimas a estos problemas pueden encontrarse determinando coloraciones mínimas de los grafos correspondientes[9].

Sin embargo, esto no siempre puede lograrse en un tiempo razonable, no existe un algoritmo conocido que, para cualquier grafo, encuentre una coloración óptima en un tiempo acotado por un polinomio en el número de nodos. Dado que los algoritmos de tiempo exponencial costosos para problemas a gran escala, se ha centrado mucha atención en el desarrollo de algoritmos heurísticos que, aunque no necesariamente óptimos, suelen producir una buena coloración en un tiempo razonable para cualquier grafo [9].

### 2.4.2 Algoritmos

**2.4.2.1 Algoritmo X (Búsqueda exhaustiva):** Un algoritmo directo para encontrar una coloración de vértices de un grafo consiste en buscar sistemáticamente entre todas las asignaciones posibles desde el conjunto de vértices al conjunto de colores. Esta técnica, a menudo llamada búsqueda exhaustiva o fuerza bruta, puede describirse de la siguiente manera [10]:

Dado un entero $q \geq 1$ y un grafo $G$ con conjunto de vértices $V$, este algoritmo encuentra una coloración de vértices utilizando $q$ colores si existe una:

- **X1 [Bucle principal]**: Para cada asignación $f : V \to \{1, 2, \ldots, q\}$, realizar el paso X2.
- **X2 [Verificar $f$]**: Si cada arista $vw$ satisface $f(v) \neq f(w)$, terminar con $f$ como el resultado.

Este método es computacionalmente costoso porque explora todas las asignaciones posibles, lo que hace que su complejidad crezca exponencialmente con el número de vértices y colores. Sin embargo, sirve como punto de partida para introducir algoritmos más avanzados que optimicen este enfoque básico. Si $G$ tiene $n$ vértices y $m$ aristas, entonces el número de operaciones utilizadas por el Algoritmo X puede ser acotado asintóticamente por $O(q^n(n + m))$ [10].

En este caso:

- $q^n$ es el número de posibles asignaciones de colores a los $n$ vértices del grafo.
- $(n + m)$ representa el costo de verificar si cada asignación de colores cumple con las restricciones del grafo, es decir, asegurarse de que no haya vértices adyacentes con el mismo color. Este costo depende de las aristas $m$ y los vértices $n$.

Por lo tanto, el número total de operaciones es el producto de estas dos cantidades, lo que da como resultado la complejidad temporal asintótica $O(q^n(n + m))$.

**2.4.2.2 Greedy colouring:** Dado un grafo $G$ con grado máximo $\Delta$ y un ordenamiento $v_1, v_2, \ldots, v_n$ de sus vértices, este algoritmo encuentra una coloración de vértices con:

$$\max\left(|\{j < i : v_j v_i \in E\}| + 1\right) \leq \Delta + 1 \text{ colores.}$$

- **G1 [Inicialización]**: Establecer $i = 0$.

- **G2 [Siguiente vértice]**: Incrementar $i$. Si $i = n+1$, terminar con $f$ como el resultado.
- **G3 [Encontrar los colores $N(v_i)$]**: Calcular el conjunto $C = \bigcup_{j<i} f(v_j)$ de los colores ya asignados a los vecinos de $v_i$.
- **G4 [Asignar el color más pequeño disponible a $v_i$]**: Para $c = 1, 2, \ldots$, comprobar si $c \in C$. Si no, asignar $f(v_i) = c$ y regresar al paso G2.

Para el número de colores, es claro que en el paso G4, el valor de $c$ es como máximo $|C|$, que está acotado por el número de vecinos de $v_i$ entre $v_1, v_2, \ldots, v_{i-1}$. En particular, el Algoritmo G establece que $\chi(G) \leq \Delta(G) + 1$.

Para el tiempo de ejecución, observe que tanto los pasos G3 como G4 requieren como máximo $O(1+\deg(v_i))$ operaciones. Sumando sobre todos los $i$, el tiempo total empleado en los pasos G3 y G4 está acotado asintóticamente por:

$$n + (\deg(v_1) + \deg(v_2) + \cdots + \deg(v_n)) = n + 2m.$$

Por lo tanto, el Algoritmo G toma un tiempo de $O(n + m)$ [10].

# 3 Algoritmos genéticos

Los objetivos de crear inteligencia artificial y vida artificial se remontan a los mismos inicios de la era computacional. Los primeros científicos de la computación—Alan Turing, John von Neumann, Norbert Wiener, entre otros—se vieron en gran parte motivados por la visión de dotar a los programas de computadora con inteligencia, con la capacidad de autorreplicarse, y con la habilidad adaptativa de aprender y controlar su entorno. Estos primeros pioneros de la ciencia de la computación estaban tan interesados en la biología y la psicología como en la electrónica, y miraban a los sistemas naturales como metáforas guías para lograr sus visiones. No debería sorprender, entonces, que desde los primeros días las computadoras se aplicaran no solo para calcular trayectorias de misiles y descifrar códigos militares, sino también para modelar el cerebro, imitar el aprendizaje humano y simular la evolución biológica. Estas actividades de computación motivadas por la biología han crecido y disminuido con el paso de los años, pero desde principios de la década de 1980 han experimentado un resurgimiento en la comunidad de investigación computacional. La primera ha dado lugar al campo de las redes neuronales, la segunda al aprendizaje automático, y la tercera a lo que ahora se llama "computación evolutiva", de la cual los algoritmos genéticos son el ejemplo más destacado [6].

## 3.1 Cómputo suave y algoritmos genéticos

El cómputo suave es una técnica basada en Inteligencia Artificial utilizada para resolver problemas NP-completos en escenarios del mundo real. Se categoriza principalmente en: Lógica Difusa y Algoritmos Genéticos. [11].

## 3.2 Desarrollo y aplicaciones

Los algoritmos genéticos son algoritmos de optimización numérica inspirados tanto en la selección natural como en la genética. Los algoritmos son fáciles de entender y el código informático requerido es fácil de escribir. El concepto de que la evolución, a partir de poco más que un "desorden" químico, generó la biodiversidad que vemos a nuestro alrededor hoy en día es un paradigma poderoso, por no decir impresionante, para resolver cualquier problema complejo [12].

Un algoritmo genético típico consite en lo siguiente [12]:

1. Un número, o población, de conjeturas de la solución al problema
2. Una forma de calcular qué tan buenas o malas son las soluciones individuales dentro de la población
3. Un método para mezclar fragmentos de las mejores soluciones para formar nuevas soluciones, en promedio aún mejores
4. Un operador de mutación para evitar la pérdida permanente de diversidad dentro de las soluciones

Ejemplos de aplicaciones son la optimización combinatoria a gran escala (como el diseño de tuberías de gas) y las estimaciones de parámetros de valores reales (como el registro de imágenes) dentro de espacios de búsqueda complejos llenos de óptimos locales. Es esta capacidad para abordar espacios de búsqueda con múltiples óptimos locales lo que representa una de las principales razones por las cuales cada vez más científicos e ingenieros están utilizando tales algoritmos [12].

Entre los numerosos problemas prácticos y áreas en los que los algoritmos genéticos (GAS, por sus siglas en inglés) se han aplicado con éxito, se encuentran [12]:

- Procesamiento de imágenes
- Predicción de estructuras tridimensionales de proteínas
- Diseño de circuitos electrónicos de muy gran escala (VLSI)
- Tecnología láser
- Medicina
- Trayectorias de naves espaciales
- Análisis de series temporales
- Física del estado sólido
- Aeronáutica
- Cristales líquidos
- Robótica

- Redes de agua
- Reglas de autómatas celulares evolutivos
- Aspectos arquitectónicos del diseño de edificios
- Evolución automática de software de computadora
- Estética
- Programación de tareas en talleres
- Reconocimiento facial
- Entrenamiento y diseño de sistemas de inteligencia artificial como redes neuronales artificiales
- Control

## 3.3 Operadores

La forma más simple de un algoritmo genético involucra tres tipos de operadores: selección, cruce y mutación [6].

### 3.3.1 Selección

Este operador selecciona los cromosomas en la población para su reproducción. Cuanto más apto es un cromosoma, más veces será probable que sea seleccionado para reproducirse [6].

### 3.3.2 Cruce

Este operador elige aleatoriamente un locus y intercambia las subsecuencias antes y después de ese locus entre dos cromosomas para crear dos descendientes. Por ejemplo, los strings 10000100 y 11111111 podrían cruzarse después del tercer locus de cada uno para producir los dos descendientes 10011111 y 11100100. El operador de cruce imita de forma aproximada la recombinación biológica entre dos organismos haploides de un solo cromosoma [6].

### 3.3.3 Mutación

Este operador cambia aleatoriamente algunos de los bits de un cromosoma. Por ejemplo, el string 00000100 podría ser mutado en su segunda posición para dar lugar a 01000100. La mutación puede ocurrir en cada posición de bit de un string con alguna probabilidad, usualmente muy pequeña (por ejemplo, 0.001) [6].

### 3.3.4 Ejemplo del funcionamiento de un GA

Dado un problema claramente definido y una representación de cadenas de bits para las soluciones candidatas, un algoritmo genético simple funciona de la siguiente manera [6]:

1. Comienza con una población generada aleatoriamente de $n$ cromosomas de $l$-bits (soluciones candidatas al problema).

2. Calcula la aptitud $f(x)$ de cada cromosoma $x$ en la población.
3. Repite los siguientes pasos hasta que se hayan creado $n$ descendientes:
   (a) Selecciona un par de cromosomas padres de la población actual, siendo la probabilidad de selección una función creciente de la aptitud. La selección se realiza "con reemplazo", lo que significa que un mismo cromosoma puede ser seleccionado más de una vez para ser un padre.
   (b) Con una probabilidad $p_c$ (la "probabilidad de cruce" o "tasa de cruce"), realiza un cruce en el par de padres en un punto elegido aleatoriamente (con probabilidad uniforme) para formar dos descendientes. Si no ocurre cruce, se forman dos descendientes que son copias exactas de sus respectivos padres. (Cabe señalar que aquí la tasa de cruce se define como la probabilidad de que dos padres realicen un cruce en un solo punto. También existen versiones del algoritmo genético con "cruce multipunto", en las cuales la tasa de cruce para un par de padres es el número de puntos en los que se realiza un cruce).
   (c) Mutación: Mutar los dos descendientes en cada locus con una probabilidad $p_m$ (la probabilidad de mutación o tasa de mutación) y colocar los cromosomas resultantes en la nueva población. Si $n$ es impar, se puede descartar aleatoriamente un miembro de la nueva población.
4. Reemplaza la población actual con la nueva población.
5. Regresa al paso 2.

Cada iteración de este proceso se denomina una generación. Un algoritmo genético se repite típicamente entre 50 y 500 generaciones o más. El conjunto completo de generaciones se llama una ejecución. Al final de una ejecución, suelen existir uno o más cromosomas altamente aptos en la población. Dado que la aleatoriedad juega un papel importante en cada ejecución, dos ejecuciones con semillas de números aleatorios diferentes generalmente producirán comportamientos detallados distintos. [6].

# 4 Desarrollo

En este trabajo se desarrolló un algoritmo genético sencillo para el coloreo de grafos con python, como se describió en las secciones anteriores, los algoritmos genéticos permiten optimizar la busqueda en el espacio de soluciones de colores asignables al grafo.

## 4.1 Justificación del lenguaje de programación

El algoritmo fue desarollado en Python, principalmente porque cuenta con la librería https://networkx.org/documentation/stable/index.html que permite graficar los grafos de manera rápida y eficiente. Además ofrece la librería numpy https://numpy.org/, que permite trabajar de manera rápida y eficiente con vectores. En este programa cada individuo es representado por un vector, además de que se utiliza la matriz de adyacencia para el calculo de la funcion de fitness.

Por último Python permite escribir y ejecutar código escrito desde bloc de notas hasta jupyter notebook, cada una con la ventaja de mayor o menor visibilidad del proceso para el usuario, en lo personal, utilizo jupyter notebook al inicio del desarrollo del algoritmo porque soy capaz de visualizar el resultado de forma inmediata y secuencial.

## 4.2 Algoritmo

La idea principal del algoritmo genético es encontrar al cromosoma más apto evaluado mediante la función de fitness. La entrada del algoritmo será el conjunto de vértices del grafo y con esto el programa calculará la matriz de adyacencia, el número de nodos y dibujará el grafo.

A continuación se muestra el pseudocódigo del funcionamiento general de un algoritmo genético

```
GA()
   initialize population
   find fitness of population

   while (termination criteria is reached) do
      parent selection
      crossover with probability pc
      mutation with probability pm
      decode and fitness calculation
      survivor selection
      find best
   return best
```

## 4.3 Inicialización de la población

Se inicializa de forma aleatoria la población inicial en donde cada cromosoma (individuo):

1. Es un vector de tamaño n+1.
2. Los primeros n elementos del vector representan cada uno de los nodos ordenados de menor a mayor.
3. El último elemento guarda el valor de la función de fitness del individuo.
4. A cada uno de los primeros n elementos, se le asigna un número entre n y n-1 de forma aleatoria, dicho número representa el color asignado al nodo n.
5. Por ejemplo, si se tienen cinco nodos $v_i$, el individuo $k$ de la población antes de la asginación será

$$[v_1, v_2, v_3, v_4, v_5, f_k] \tag{1}$$

como se tienen cinco nodos, se asignarán colores del 1 al 4 de forma aleatoria al individuo

$$eq1[1, 2, 3, 4, 2, f_k] \tag{2}$$

Que representa que al nodo $v_1$ se le asigna el color 1, al nodo $v_2$ se le asigna el color 2 y así sucesivamente.

En este algoritmo la población consiste de 1,500 individuos.

El pseudocódigo para la generación de la población inicial es :

```
Algoritmo GeneracionAleatoria(SizeOfPopulation, NumberOfNodes,
   NumberOfColors):
    Crear una lista vacia llamada generation
    Para i desde 1 hasta SizeOfPopulation hacer:
        Crear una lista vacia llamada cromosome
        Para j desde 1 hasta NumberOfNodes hacer:
            Agregar un numero aleatorio entre 1 y NumberOfColors a
               cromosome
        Fin Para
        Agregar cromosome a generation
    Fin Para
    Retornar generation
Fin Algoritmo
```

## 4.4 Función de aptitud (fitness)

La función de aptitud nos permite evaluar a cada cromosoma para seleccionar a los mas aptos de cada generación, el objetivo del algoritmo es encontrar el individuo con el arreglo que no asigne colores a nodos adyacentes, para ello necesitamos la vecindad de cada nodo, que puede ser calculada de la matriz de adyacencia. El cáculo de la función de fitness es como sigue:

Para cada gen de cada individuo:

1. Obtener la vecindad de cada gen (nodo) utilizando la matriz de adyancencia.
2. Inicializar el número de confictos en cero.
3. Si alguno de los nodos de la vecindad del gen tienen asignado el mismo color que el gen actual, sumar uno al número de conflictos.
4. Para cada cromosoma, calcular el número de colores únicos y sumarlo al número de conflictos. Esta suma representa la aptitud del individuo o cromosoma.

El pseudocodigo para calcular el fitness es:

```
1
2  Algoritmo FitnessPorIndividuo(nodes, gene, G_):
3      Inicializar j como 0
4      Inicializar conflict como 0
5
6      Mientras j < nodes hacer:
7          Obtener adyacent_nodes como la lista de vecinos de j en G_
8          Para cada nodo a en adyacent_nodes hacer:
9              Si gene[j] es igual a gene[a] entonces:
10                 Incrementar conflict en 1
11             Fin Si
12         Fin Para
13         Incrementar j en 1
14     Fin Mientras
15
16     Calcular unique_colors como el numero de colores Ãºnicos en gene
17     Calcular fitness como 1 / (unique_colors + conflict)
18     Retornar fitness
19 Fin Algoritmo
```

## 4.5 Selección

Para crear las nuevas generaciones es necesario seleccionar a dos individuos que serán los padres, existen muchos métodos para realizar dicha selección :

1. Selección: Se ordenan la población usando el valor de fitness y se selecciona el 50% de la población.
2. Ruleta: La selección proporcional a la aptitud, también conocida como selección tipo rueda de ruleta, es una técnica común en algoritmos evolutivos. En este método, la probabilidad de que un individuo sea seleccionado es directamente proporcional a su aptitud. Esta técnica asegura que los individuos con mayor aptitud tengan una mayor probabilidad de ser seleccionados, aunque no excluye a aquellos con menor aptitud, lo que mantiene la diversidad genética en la población. [12]
3. Torneo: La selección por torneo ejerce presión selectiva al realizar un torneo entre $S$ competidores, donde $S$ es el tamaño del torneo. El ganador del torneo es el individuo con la mayor aptitud entre los $S$ competidores. Este ganador será uno de los padres. [3]

Este método asegura que los individuos con mayor aptitud tienen una mayor probabilidad de ser seleccionados, manteniendo la diversidad genética dentro de la población.

En este algoritmo, utilizamos la selección por ruleta, por lo que se toma el inverso de la función de fitness, ya que entre mayor sera el valor de la función de aptitud para el individuo es menos apto porque significa que tiene mayor numero de conflictos y colores. Al tomar

el inverso, el individuo con el menor valor de fitness es más apto, después se siguen los siguientes pasos [12]:

1. **Calcular la suma total de aptitudes:** Sumar las aptitudes de todos los miembros de la población y llamar a este valor S.
2. **Elegir un número aleatorio:** Generar un número aleatorio $P_s$ tal que $0 \leq P_s \leq$ sum_fitness.
3. **Recorrer la población:** Ir sumando las aptitudes de los individuos de la población uno por uno, deteniéndose cuando la suma acumulada supere el valor de $P_s$.
4. **Seleccionar el individuo:** El último individuo añadido al total acumulado es el seleccionado.

```
Algoritmo SeleccionRuleta(S, poblacion):
    Seleccionar un numero aleatorio random_select entre 0 y S
    Inicializar P como 0

    Para cada individuo en la poblacion hacer:
        Sumar la aptitud del individuo actual al acumulado P
        Si P > random_select entonces:
            Detener el bucle
        Fin Si
    Fin Para

    Retornar el individuo seleccionado
Fin Algoritmo
```

## 4.6 Operador de cruce (crossover)

El operador de *crossover* elige aleatoriamente una posición y realiza un intercambio de subsecuencias antes y después de dichas posición entre dos cromosomas para crear dos descendientes. Este operador imita de manera aproximada la recombinación biológica entre dos organismos haploides (de un solo cromosoma)[6]. Existen diferentes tipos de operadores de cruza, en este trabajo se utiliza el crossover de dos puntos, en el que se seleccionan dos posiciones de cada padre y se intercambian los valores para formar dos hijos

Por ejemplo, si tenemos los cromosomas:

Padre 1: 10000100
Padre 2: 11111111,

y seleccionamos la **posición 2 y 7**, los descendientes después del *crossover* serán:

Hijo 1: 10111111
Hijo 2: 11000100.

Este proceso asegura la mezcla de material genético de ambos cromosomas, fomentando la diversidad en las generaciones sucesivas.

```
Algoritmo CrossoverMultipunto(nodes, padre1, padre2):
    Inicializar cromosoma1 como padre1
    Inicializar cromosoma2 como padre2

    Elegir aleatoriamente random_cutoff_1 entre 1 y nodes // 2
    Elegir aleatoriamente random_cutoff_2 entre random_cutoff_1 + 1 y
        nodes - 1

    Dividir cromosoma1 en tres cortes:
        first_cutoff_p1 = cromosoma1[0:random_cutoff_1]
        second_cutoff_p1 = cromosoma1[random_cutoff_1:random_cutoff_2
           ]
        third_cutoff_p1 = cromosoma1[random_cutoff_2:nodes]

    Dividir cromosoma2 en tres cortes:
        first_cutoff_p2 = cromosoma2[0:random_cutoff_1]
        second_cutoff_p2 = cromosoma2[random_cutoff_1:random_cutoff_2
           ]
        third_cutoff_p2 = cromosoma2[random_cutoff_2:nodes]

    Generar descendientes:
        offspring1 = first_cutoff_p1 + second_cutoff_p2 +
           third_cutoff_p1
        offspring2 = first_cutoff_p2 + second_cutoff_p1 +
           third_cutoff_p2

    Retornar offspring1, offspring2
Fin Algoritmo
```

Para elegir que padres se cruzan se utiliza una probabilidad de cruce, se genera un número aletorio entre 0 y 1 para cada par de padres, si dicho número es mayor que la probabilidad de cruce, el operador crossover se lleva a cabo, de otra forma, se pasan los padres como la nueva generación. En este caso elegimos una **probabilidad de cruce de 0.5**.

## 4.7 Operador de mutación

Este operador invierte aleatoriamente algunos de los bits en un cromosoma. Por ejemplo, la cadena 00000100 podría mutarse en su segunda posición para obtener 01000100 [6].

En este caso utilizamos mutación de dos elementos de cada cromosoma, el pseudocódigo es el siguiente

```
Algoritmo MutacionPorIntercambio(nodes, cromosoma):
    Elegir aleatoriamente random_idx1 entre 0 y nodes // 2
    Elegir aleatoriamente random_idx2 entre random_idx1 + 1 y nodes -
        1

    Realizar el intercambio en la secuencia del cromosoma:
        mutated_cromosoma = cromosoma[0:random_idx1] +
                            [cromosoma[random_idx2]] +
                            cromosoma[random_idx1+1:random_idx2] +
                            [cromosoma[random_idx1]] +
                            cromosoma[random_idx2+1:]

    Retornar mutated_cromosoma
Fin Algoritmo
```

De forma similar, para elegir a los hijos a los cuales se les aplicará la mutación, se define una probabilidad de mutación, se genera un número aletorio entre 0 y 1 para cada hijo generado despues de cruce (si aplica), si dicho número es mayor que la probabilidad de mutación, el operador demutación se lleva a cabo, de otra forma, se pasan el hijo sin mutar como la nueva generación. Se definió una **probabilidad de mutación de 1/tamaño de la población**.

## 4.8 Selección de la nueva generación

Después de llevar a cabo la selección de padres, los operadores de crossover y mutación elegimos se calcula la aptitud para cada nuevo hijo, y la nueva generación se selecciona tomando el mismo número de individuos de la población original usando a los individuos más aptos de la nueva y anterior generación.

El pseudocódigo que combina todos los operadores y dada una generación crea una nueva es el siguiente

```
Algoritmo UnaIteracion(nodes, pop_size, pc, pm, population, G, S):
    ## Calcular la aptitud de toda la poblacion
    # S = la suma de todas las aptitudes.

    k = 0

    ## Seleccion de padres
    Mientras k < pop_size // 2:
        p1 = SeleccionRuleta(S, poblacion[:pop_size])
        p2 = SeleccionRuleta(S, poblacion[:pop_size])

        Si p1 == p2:
            p1 = SeleccionRuleta(S, poblacion[:pop_size])

```

```
15          r_pc = numero aleatorio entre 0 y 1
16
17          Si r_pc > pc:
18              xs_offspring1, xs_offspring2 = CrossoverMultipunto(nodes,
                    p1[:nodes], p2[:nodes])
19          Sino:
20              xs_offspring1, xs_offspring2 = p1[:nodes], p2[:nodes]
21
22          r_pm = numero aleatorio entre 0 y 1
23
24          Si r_pm > pm:
25              mutated_offspring1 = MutacionPorIntercambio(nodes,
                   xs_offspring1)
26              mutated_offspring2 = MutacionPorIntercambio(nodes,
                   xs_offspring2)
27          Sino:
28              mutated_offspring1, mutated_offspring2 = xs_offspring1,
                   xs_offspring2
29
30          mutated_offspring1_fit = AptitudPorIndividuo(nodes,
               mutated_offspring1, G)
31          mutated_offspring1.append(mutated_offspring1_fit)
32          mutated_offspring2_fit = AptitudPorIndividuo(nodes,
               mutated_offspring2, G)
33          mutated_offspring2.append(mutated_offspring2_fit)
34
35          Si (mutated_offspring1 no esta en poblacion[:pop_size]):
36              poblacion.append(mutated_offspring1)
37              k += 1
38
39          SinoSi (mutated_offspring2 no esta en poblacion[:pop_size]):
40              poblacion.append(mutated_offspring2)
41              k += 1
42          Sino:
43              Pasar
44
45      poblacion.sort(key=lambda x: x[-1], reverse=True)
46      siguiente_generacion_mas_apta = poblacion[:pop_size]
47
48      Retornar siguiente_generacion_mas_apta
49  Fin Algoritmo
```

## 4.9 Condición de paro

El algoritmo elige al individuo más apto después de cada iteración, dicho individuo tendrá una función de fitness definida como:

$$f = \frac{1}{\text{numero de colores} + \text{numero de conflictos}} \tag{3}$$

El individiduo será una solución si el número de conflictos es igual a cero, que se calcula como

$$\frac{1}{f} - \text{numero de colores} = \text{numero de conflictos} \tag{4}$$

Si no se encuentra una solución despues de **500** generaciones, se regresa la mejor solución de la última generación junto con el número de conflictos y colores de la misma.

## 4.10 Algoritmo final

[1]:
```
import numpy as np
import networkx as nx
import random
import matplotlib.colors as mcolors
import matplotlib.pyplot as plt
```

[2]:
```
def dibujar_grafo(relacion):
    g_object = nx.Graph()
    g_object.add_edges_from(list(relacion))
    return g_object

def dibujar_grafo_dirigido(relacion):
    g_object = nx.DiGraph()
    g_object.add_edges_from(list(relacion))
    return g_object
```

[3]:
```
def randomGeneration(NumberOfPopulation,NumberOfNodes,NumberOfColors):␣
↪#number of row is number of population
    generation_list = []
    for i in range(NumberOfPopulation):
        gene = []
        for j in range(NumberOfNodes):
            gene.append(random.randint(1,NumberOfColors)) # generate the␣
↪row number for each individual
        # gene.append(0)
        generation_list.append(gene)
    return generation_list
```

[4]:
```
# Fitness of initial population
```

[5]:
```
def fitness_by_individual(nodes, gene, G_):
    j = 0
    conflict = 0
    while j < nodes:
        adyacent_nodes = [n for n in nx.neighbors(G_, j)]
        for a in adyacent_nodes:
            if gene[j]==gene[a]:
                conflict += 1
        j += 1

    unique_colors = len(set(gene))
    fitness = 1/(unique_colors + conflict)
    return fitness
```

[6]:
```
def fitness_pop(nodes, population, G):
    for i in range(len(population)):
        fit = fitness_by_individual(nodes, population[i], G)
        population[i].append(fit)
    return population
```

[7]:
```
# Roulette selection
```

[8]:
```
def roulette_selection(S, population):
    random_select = random.randint(0,S)
    P = 0
    ## Starting from the top of the population, keep adding the finesses␣
 ↪to the partial sum P, till P<S.
    for i in range(len(population)):
        P += population[i][-1]
        if P > random_select:
            break
    return population[i]
```

[9]:
```
# Multipoint crossover
```

[10]:
```
def multipoint_cross_over(nodes,parent1, parent2):

    cromosome1_n = parent1
    cromosome2_n = parent2

    random_cutoff_1 = random.randint(1,nodes//2)
    random_cutoff_2 = random.randint(random_cutoff_1+1,nodes-1)
```

```
    first_cuttof_p1 =  cromosome1_n[0:random_cutoff_1]
    second_cuttof_p1 =  cromosome1_n[random_cutoff_1:random_cutoff_2]
    third_cuttof_p1 =  cromosome1_n[random_cutoff_2:nodes]

    first_cuttof_p2 =  cromosome2_n[0:random_cutoff_1]
    second_cuttof_p2 =  cromosome2_n[random_cutoff_1:random_cutoff_2]
    third_cuttof_p2 =  cromosome2_n[random_cutoff_2:nodes]

    offspring1 = first_cuttof_p1+second_cuttof_p2+third_cuttof_p1
    offspring2 = first_cuttof_p2+second_cuttof_p1+third_cuttof_p2


    return offspring1, offspring2
```

[11]:
```
# Swap mutation
```

[12]:
```
def swap_mutation(nodes,cromosome):

    random_idx1 = random.randint(0,nodes//2)
    random_idx2 = random.randint(random_idx1+1,nodes-1)

    mutated_cromosome = cromosome[0:
 ↪random_idx1]+[cromosome[random_idx2]]+cromosome[random_idx1+1:
 ↪random_idx2]+[cromosome[random_idx1]]+cromosome[random_idx2+1:]

    return  mutated_cromosome
```

[13]:
```
# Next generation selection
```

[27]:
```
def one_iteration(nodes,pop_size, pc,pm, population, G,S):
    ## fitness of whole population
    # Calculate S = the sum of all fitnesses.


    k = 0

    ### select parents

    while k < pop_size//2:
        p1 = roulette_selection(S, population[:pop_size])
        p2 = roulette_selection(S, population[:pop_size])

        if p1 == p2:
```

```
            p1 = roulette_selection(S, population[:pop_size])

        r_pc = random.randint(0,1)

        if r_pc > pc:
            xs_offspring1, xs_offspring2 = multipoint_cross_over(nodes,␣
↪p1[:nodes], p2[:nodes])
        else:
            xs_offspring1, xs_offspring2 =  p1[:nodes], p2[:nodes]

        r_pm = random.randint(0,1)

        if r_pm > pm:
             mutated_offspring1 = swap_mutation(nodes, xs_offspring1)
             mutated_offspring2 = swap_mutation(nodes, xs_offspring2)
        else:
            mutated_offspring1, mutated_offspring2 = xs_offspring1,␣
↪xs_offspring2

        mutated_offspring1_fit = fitness_by_individual(nodes,␣
↪mutated_offspring1, G)
        mutated_offspring1.append(mutated_offspring1_fit)
        mutated_offspring2_fit = fitness_by_individual(nodes,␣
↪mutated_offspring2, G)
        mutated_offspring2.append(mutated_offspring2_fit)

        if (mutated_offspring1 not in population[:pop_size]):
            population.append(mutated_offspring1)
            k += 1

        elif (mutated_offspring2 not in population[:pop_size]):
            population.append(mutated_offspring2)
            k += 1
        else:
            pass

    population.sort(key=lambda x: x[-1], reverse=True)

    next_generation_most_fit = population[:pop_size]

    return next_generation_most_fit
```

[33]:
```
def one_step(relacion_set, population_size, max_iterations, pc,␣
↪name_of_colors):
```

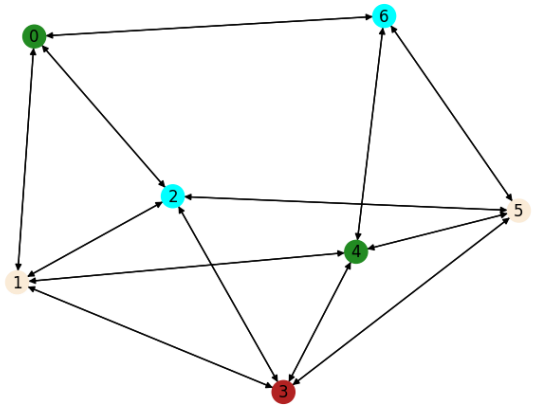
```
    ## create graph from adjacency matrix
    g_object = nx.DiGraph()
    g_object.add_edges_from(list(relacion_set))
    matrix_adj = nx.to_numpy_array(g_object)
    G = nx.from_numpy_array(matrix_adj, create_using=nx.DiGraph)
    ## get number of nodes and number of colors
    number_of_nodes = len(matrix_adj)
    number_of_colors = number_of_nodes-1
    # mutation probability based on the size of the population
    pm = 1/population_size
    ### color mapping
    cmap_vec={}
    for i in range(number_of_nodes):
        cmap_vec[i] = name_of_colors[i]

    ### actual seach
    current_generation = randomGeneration(population_size,␣
↪number_of_nodes, number_of_colors) # initialize pop
    current_generation = fitness_pop(number_of_nodes,current_generation,␣
↪G)
    S = int(np.sum(current_generation, axis = 0)[-1])
    epoch = 1
    flag  = False  ## to check for a solution
    while epoch < max_iterations:
     #   print("-----------------------------------------------------")
    #    print("Epoch ",epoch)

        next_generation = one_iteration(number_of_nodes,population_size,␣
↪pc,pm, current_generation, G,S)
        current_generation = next_generation
        current_generation =␣
↪fitness_pop(number_of_nodes,current_generation, G)

        best_element = current_generation[0][:number_of_nodes]
        unique_colors = len(set(best_element))
        best_element_fitness = current_generation[0][number_of_nodes]
        best_element_conflicts = (1/best_element_fitness) - unique_colors

        if best_element_conflicts == 0:
            print("The graph is " +str(unique_colors)+" colorable : ",␣
↪best_element)
            flag = True
            break
```

```
        epoch+=1


    if flag == False:
        print("Coulnd't find a solution after "+str(max_iterations)+'␣
↪iterations')
        print("Best Solution: ", best_element)
        print("Number of conflicts: ", best_element_conflicts)
        print("Number of colors: ", unique_colors)
    ### returned colored graph

    color_map = []
    for node_color in best_element:
        color_map.append(name_of_colors[node_color])

    nx.draw(G, node_color=color_map, with_labels=True)
    plt.show()


    return #best_element, flag
```

[34]:
```
population_size_ = 1500
max_iterations_ = 500
pc_ = 0.6
name_of_colors = [k for k in mcolors.CSS4_COLORS.keys()]
random.shuffle(name_of_colors)
```

[35]:
```
relacion_str = input("Introduce las aristas del grafo:" )
relacion_set_ = eval(relacion_str)
one_step(relacion_set_, population_size_, max_iterations_, pc_,␣
↪name_of_colors)
```

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (0, 6), (1, 0), (1, 2),␣
↪(1,
3), (1, 4), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5),␣
↪(4,
1), (4, 3), (4, 5), (4, 6), (5, 2), (5, 3), (5, 4), (5, 6), (6, 0), (6, 4),␣
↪(6,
5)}

The graph is 4 colorable :  [6, 2, 3, 5, 6, 2, 3]
```




## 4.11 Complejidad en tiempo

# Análisis de la Complejidad Temporal

### 4.11.1 Construcción del grafo:

- `g_object.add_edges_from(list(relacion_set))`: Este paso agrega todas las aristas al grafo. Si $E$ es el número de aristas en el grafo, este paso tiene una complejidad de $O(E)$.
- `matrix_adj = nx.to_numpy_array(g_object)`: Esta función convierte el grafo a una matriz de adyacencia. Dado que el número de nodos es $n$ y estamos creando una matriz $n \times n$, la complejidad es $O(n^2)$.
- `G = nx.from_numpy_array(matrix_adj, create_using=nx.DiGraph)`: Esto crea nuevamente un grafo a partir de la matriz de adyacencia. La complejidad también es $O(n^2)$ debido a la conversión de la matriz de adyacencia en el grafo.

### 4.11.2 Inicialización de la población:

- `current_generation = randomGeneration(population_size, number_of_nodes, number_of_colors)`: Esta función genera una población aleatoria. Tiene una complejidad de $O(\text{population\_size} \times n)$, ya que está creando una población de tamaño `population_size` y cada cromosoma tiene $n$ nodos.

### 4.11.3 Cálculo de la aptitud de la población:

- `current_generation = fitness_pop(number_of_nodes, current_generation, G)`: La función `fitness_pop` calcula la aptitud para cada individuo en la población. Dependiendo de cómo esté implementada esta función, si realiza un recorrido

por el grafo o comprueba las conexiones de cada nodo, su complejidad será de $O(\text{population\_size} \times n)$, ya que se calcula la aptitud de `population_size` individuos con $n$ nodos en cada uno.

#### 4.11.4 Iteración principal (bucle principal):

El bucle `while epoch < max_iterations` se ejecuta hasta que se alcanza el número máximo de iteraciones (`max_iterations`) o se encuentra una solución. Cada iteración tiene los siguientes pasos:

- **Selección de los padres**: `next_generation = one_iteration(...)`. Dentro de la función `one_iteration`, se realiza la selección de los padres y la generación de la siguiente población. En la función `one_iteration`, se hace una selección de ruleta que toma $O(\text{population\_size})$ por cada selección de padres. Esto se repite $\frac{\text{population\_size}}{2}$ veces, lo que resulta en una complejidad de $O(\text{population\_size})$ para esta parte.
- **Crossover y mutación**: Estas operaciones se ejecutan para cada par de padres y descendientes, lo que tiene una complejidad de $O(\text{population\_size} \times n)$, ya que cada operación de crossover y mutación involucra a $n$ nodos por cada individuo.
- **Cálculo de la aptitud**: Después de cada iteración de crossover y mutación, la aptitud de la población se recalcula, lo cual tiene una complejidad de $O(\text{population\_size} \times n)$.
- **Bucle de `one_iteration`**: Dentro de `one_iteration`, la selección de ruleta, el crossover y la mutación tienen una complejidad total de $O(\text{population\_size} \times n)$ por iteración, dado que cada individuo tiene $n$ nodos y cada operación se repite para todos los individuos de la población.

#### 4.11.5 Ordenamiento de la población:

- `population.sort(key=lambda x: x[-1], reverse=True)`: Esta operación ordena la población de acuerdo con la aptitud, lo que tiene una complejidad de $O(\text{population\_size} \log \text{population\_size})$.

#### 4.11.6 Complejidad Total:

Si combinamos todos estos pasos, la complejidad total por iteración del bucle principal (hasta `max_iterations`) sería:

- Generación de la población: $O(\text{population\_size} \times n)$
- Cálculo de la aptitud: $O(\text{population\_size} \times n)$
- Selección, crossover y mutación: $O(\text{population\_size} \times n)$
- Ordenamiento de la población: $O(\text{population\_size} \log \text{population\_size})$

Dado que el bucle principal se ejecuta hasta `max_iterations` veces, la complejidad total sería:

$O(\text{max\_iterations} \times \text{population\_size} \times n + \text{max\_iterations} \times \text{population\_size} \log \text{population\_size})$

En resumen, la complejidad temporal total es:

$O(\text{max\_iterations} \times \text{population\_size} \times n + \text{max\_iterations} \times \text{population\_size} \log \text{population\_size})$

# 5 Resultados

Se usaron los siguientes grafos como conjunto de prueba y sus coloreo conocido [4, 13]

| **Grafo** | **Aristas** | **Coloreo** |
|---|---|---|
| G | (0, 1), (0, 2), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5), (4, 1), (4, 3), (4, 5), (4, 6), (5, 2), (5, 3), (5, 4), (5, 6), (6, 4), (6, 5) | 3 |
| H | (0, 1), (0, 2), (0, 6), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5), (4, 1), (4, 3), (4, 5), (4, 6), (5, 2), (5, 3), (5, 4), (5, 6), (6, 0), (6, 4), (6, 5) | 4 |
| C5 | (0, 1), (0, 4), (1, 2), (2, 3), (3, 4) | 3 |
| C6 | (0, 1), (0, 5), (1, 2), (2, 3), (3, 4), (4, 5) | 2 |
| Bull | (0, 1), (0, 2), (1, 2), (1, 3), (2, 4) | 3 |
| S6 | (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6) | 2 |
| Petersen | (0, 1), (0, 4), (0, 5), (1, 2), (1, 6), (2, 3), (2, 7), (3, 4), (3, 8), (4, 9), (5, 7), (5, 8), (6, 8), (6, 9), (7, 9) | 3 |
| W5 | (0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4) | 3 |

Como el algoritmo se inicializa de forma aleatoria, es posible que encuentre soluciones distintas en cada corrida o que no encuentre solución alguna, es por eso que para cada grafo de prueba se corrió el algoritmo 10 veces, los resultados se reportan en la siguiente tabla

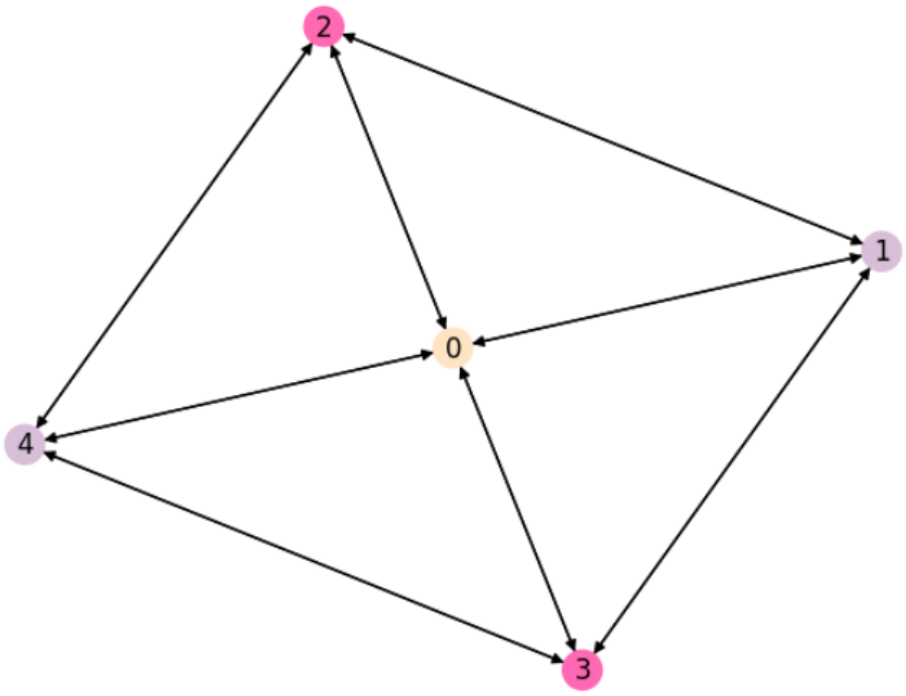
| **Grafo** | **Coloreo** | **Iteraciones correctas** |
|---|---|---|
| G | 3 | 8/10 |
| H | 4 | 10/10 |
| C5 | 3 | 10/10 |
| C6 | 2 | 10/10 |
| Bull | 3 | 10/10 |
| S6 | 2 | 5/10 |
| Petersen | 3 | 2/10 |
| W5 | 3 | 10/10 |

Por lo que el algoritmo encuentra el numero de colores correcto un 65% de las veces.

## 5.1 Ejemplos

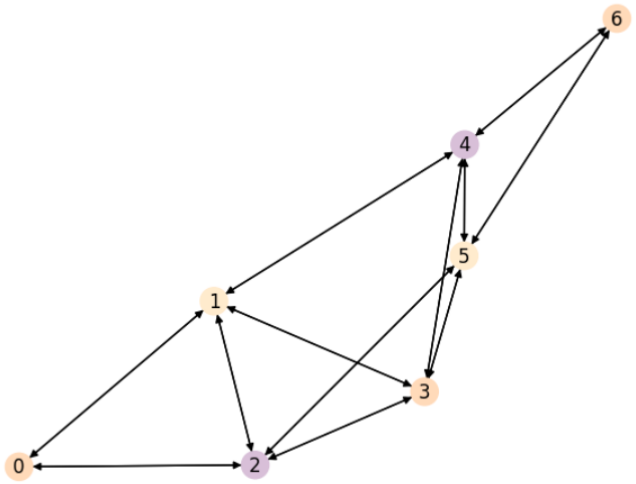
El coloreo para el grafo W5

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4)}
The graph is 3 colorable :  [1, 4, 3, 3, 4]
```



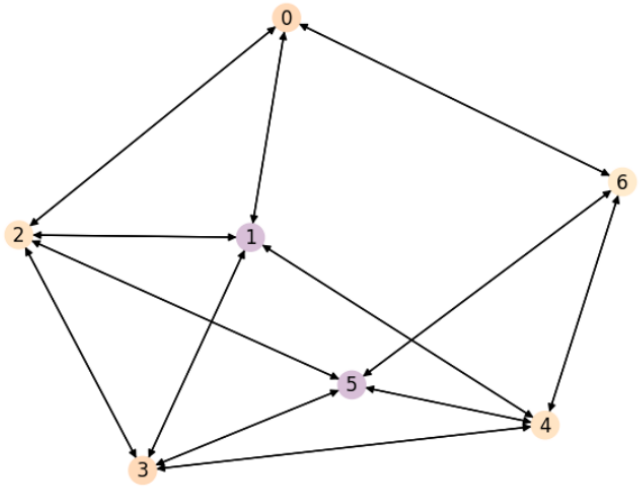

El coloreo para el grafo G

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5), (4,
1), (4, 3), (4, 5), (4, 6), (5, 2), (5, 3), (5, 4), (5, 6), (6, 4), (6, 5)}
The graph is 3 colorable :  [2, 5, 4, 2, 4, 5, 2]
```



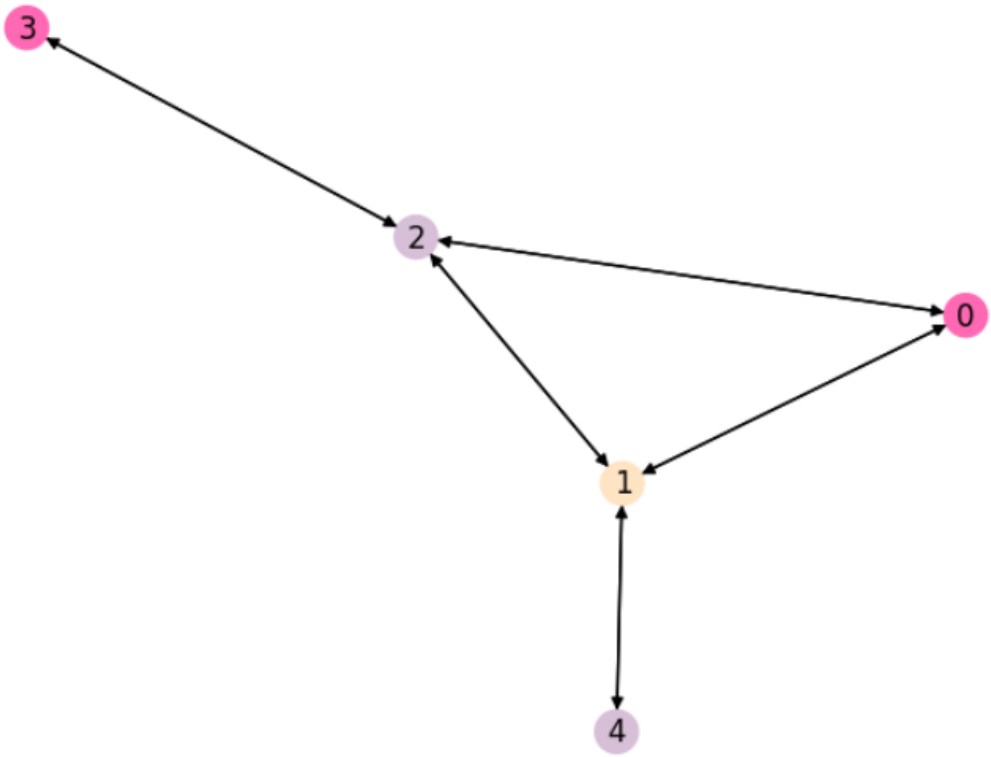
El coloreo para el grafo H

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (0, 6), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3,
5), (4, 1), (4, 3), (4, 5), (4, 6), (5, 2), (5, 3), (5, 4), (5, 6), (6, 0), (6, 4), (6, 5)}
The graph is 4 colorable :  [2, 4, 1, 2, 1, 4, 5]
```



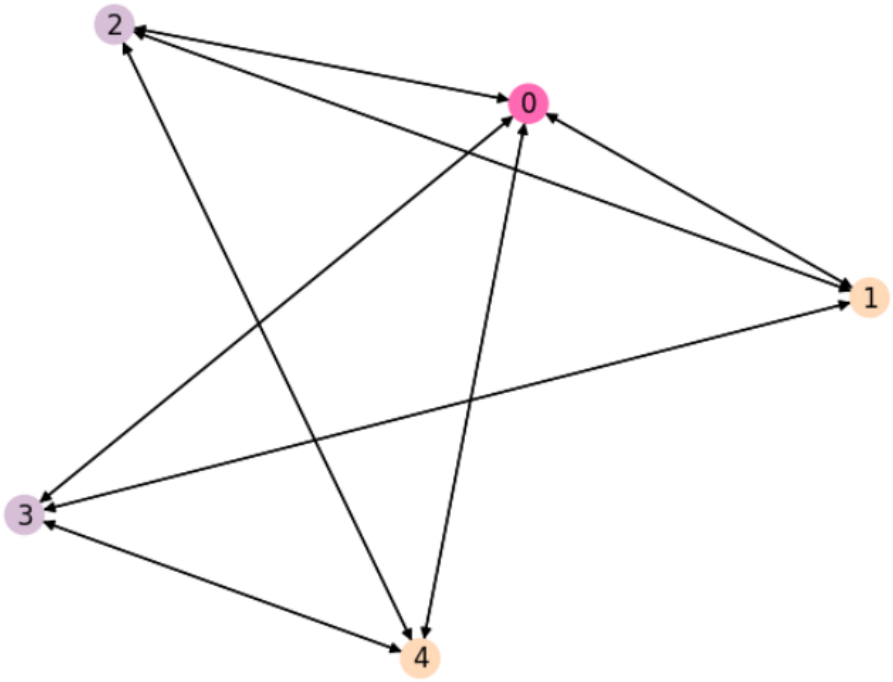

El coloreo para el grafo Bull

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (1, 2), (1, 3), (2, 4)}
The graph is 3 colorable :  [3, 1, 4, 3, 4]
```



El coloreo para el grafo W5

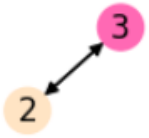
```
Introduce las aristas del grafo: {(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4)}
The graph is 3 colorable :  [3, 2, 4, 4, 2]
```




## 5.2 Limitaciones

Como puede verse en la tabla de resultados, el algoritmo aun puede mejorarse, principalmente para grafos mas complejos como el Petersen, así como tambien para grafos pequeños. El siguiente grafo se cicla en algunas iteraciones

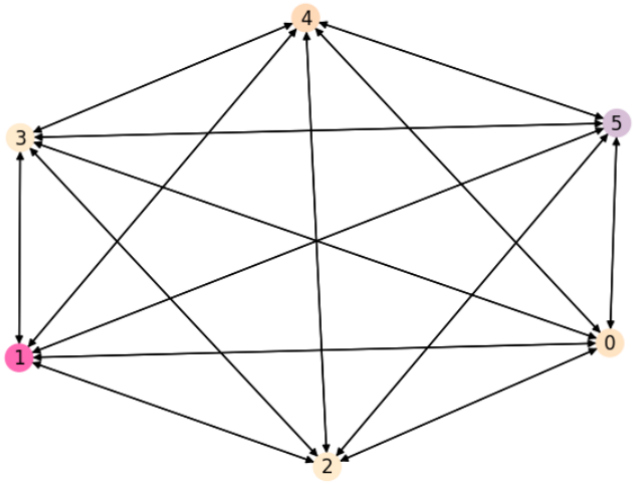
```
Introduce las aristas del grafo: (0, 3), (1, 2), (2, 1), (3, 0)
The graph is 2 colorable :  [1, 3, 1, 3]
```


Debido a que el número de permutaciones no es suficiente para generar nuevos individuos dependiendo de la población inicial.

El siguiente grafo no es coloreable, el algoritmo corre las 500 iteraciones pero no es capaz de encontrar una solución

```
Introduce las aristas del grafo: {(0, 1), (0, 2), (0, 4), (0, 5), (1, 0), (1, 2), (1, 3), (2, 1), (2, 3), (2, 4), (2, 5), (3, 0), (3, 1), (3, 2), (3,
4), (3, 5), (4, 1), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 4)}
Coulnd't find a solution after 500 iterations
Best Solution:  [1, 3, 5, 5, 2, 4]
Number of conflicts:  2.0
Number of colors:  5
```




## 5.3 Conclusiones

El algoritmo genético presentado en este trabajo es muy sencillo en cuanto a los operadores de cruce y mutación, las probabilidades se eligieron a través de experimentación en el proceso de desarrollo, se demostró que aunque es sencillo, es capaz de encontrar el coloreo correcto en el 65% de las corridas.

# References


[1] J. Matousek, J. Nesetril, Invitation to Discrete Mathematics, Oxford University Press. Second edition 2008.

[2] R. Johnsonbaugh, Discrete Mathematics, Pearson. 8th edition 2017. https://broman.dev/download/Discrete%20Mathematics%208th%20Edition.pdf

[3] Miller, B. L., & Goldberg, D. E. (1995). Genetic Algorithms, Tournament Selection, and the Effects of Noise. Complex Systems, 9. https://dblp.uni-trier.de/db/journals/compsys/compsys9.html#MillerG95

[4] Rosen, K. H. (2019). Discrete mathematics and its applications / Kenneth H. Rosen, Monmouth University (and formerly AT&T Laboratories)(8.a ed.). McGraw-Hill Education.

[5] Marappan, R., & Sethumadhavan, G. (2013). A New Genetic Algorithm for Graph Coloring. 2013 Fifth International Conference on Computational Intelligence, Modelling and Simulation. https://doi.org/10.1109/cimsim.2013.17

[6] Mitchell, M. (1996). An introduction to genetic algorithms. https://doi.org/10.7551/mitpress/3927.001.0001

[7] Badwaik Jyoti S. Recent Advances in Graph Theory and its Applications., Int. Res. Journal of Science & Engineering, February, 2020, Special Issue A7 :533-538.

[8] Aho, A. V., & Hopcroft, J. E. (1974). The design and analysis of computer algorithms. http://archives.umc.edu.dz/handle/123456789/114627

[9] Leighton, F. (1979). A graph coloring algorithm for large scheduling problems. Journal of Research of the National Bureau of Standards, 84(6), 489. https://doi.org/10.6028/jres.084.024

[10] Husfeldt, T. (2015). Graph colouring algorithms. In Cambridge University Press eBooks (pp. 277–303). https://doi.org/10.1017/cbo9781139519793.016

[11] A Systematic Review of Quality of Service in Wireless Sensor Networks using Machine Learning: Recent trend and Future Vision. (n.d.). Journal of Network and Computer Applications. https://doi.org/10.1017/cbo9781139519793.016

[12] Coley, D. A. (1999). An introduction to genetic algorithms for scientists and engineers.

[13] Wolfram Research, Inc. (s. f.). Chromatic Number – from Wolfram MathWorld. https://mathworld.wolfram.com/ChromaticNumber.html https://doi.org/10.1142/3904