Proyecto #2

Coloración Probabilística de Grafos con Búsqueda

Profesor: Joss Rayn Pecou Johnson

Estudiantes:

Axel López Cruz

Tayler Wynta Rodríguez

Maikel Flores Navarro

Curso: IC3002 - Análisis de algoritmos

Grupo: 60

II Semestre

Año: 2025

Tabla de contenido

Descripción General del Proyecto

Este proyecto implementa un sistema completo para la coloración probabilística de grafos mediante distintos enfoques de búsqueda. Incluye una interfaz web interactiva construida con React + Vite, conectada a un backend desarrollado en Python, que ejecuta los algoritmos de simulación y análisis.

El objetivo principal es asignar colores a los nodos de un grafo evitando conflictos (dos nodos adyacentes no pueden compartir color), utilizando para ello algoritmos probabilísticos y métodos de búsqueda. El sistema permite generar grafos aleatorios, cargarlos manualmente, visualizar el progreso de la coloración y obtener métricas del rendimiento del algoritmo.

Visión General del Sistema

El sistema permite:

1. **Generar grafos aleatorios según:**
   - Cantidad de nodos
   - Probabilidad de conexión entre nodos
   - Número de colores permitidos
2. **Realizar coloración probabilística mediante:**
   - Búsqueda aleatoria
   - Métodos heurísticos
   - Enfriamiento simulado (si está implementado en tu proyecto)
   - Correcciones por conflicto
3. **Visualizar:**
   - El grafo inicial
   - El estado del grafo durante la coloración
   - El grafo final coloreado
   - Conflictos presentes durante la ejecución
4. **Obtener estadísticas como:**
   - Cantidad de iteraciones
   - Tiempo total de ejecución
   - Número de conflictos detectados
   - Movimientos realizados

5. **Cargar / guardar grafos**

El sistema cuenta con funcionalidades para cargar y guardar grafos, permitiendo al usuario trabajar con estructuras previamente generadas o conservar los resultados obtenidos tras la ejecución de los algoritmos de coloración.

Estas funciones son especialmente útiles para:

- Repetir pruebas sin necesidad de generar un grafo nuevo.
- Comparar distintos algoritmos usando el mismo grafo base.
- Compartir grafos entre integrantes del grupo.
- Conservar grafos coloreados como evidencia o para análisis posteriores.

Arquitectura del Proyecto

El sistema desarrollado para la Coloración Probabilística de Grafos con Búsqueda está compuesto por dos capas principales:
Frontend interactivo creado con React + Vite, y un backend encargado del procesamiento de los algoritmos de coloración. Ambas partes se comunican mediante peticiones HTTP, lo que permite separar completamente la interfaz de usuario de la lógica algorítmica.

A continuación, se describe la arquitectura general del proyecto.

Frontend React + Vite

El frontend se encuentra dentro del directorio principal del repositorio, específicamente en la carpeta src: Su función principal es permitir que el usuario interactúe con el sistema de manera sencilla e intuitiva, mostrando los grafos, los colores asignados y los resultados del algoritmo.

Responsabilidades principales del frontend

- Renderizar grafos en pantalla usando layouts visuales.
- Permitir ingresar datos del usuario (nodos, probabilidad, colores, etc.).
- Generar grafos aleatorios y enviarlos al backend.
- Ejecutar la coloración según el algoritmo seleccionado.
- Visualizar la evolución del algoritmo y sus resultados.

- Mostrar métricas de rendimiento y conflictos finales.
- Gestionar la carga y guardado de grafos.

Componentes principales

A continuación, se listan los componentes más relevantes del frontend, con sus responsabilidades:

GenerarGrafoAleatorio.jsx

- Permite elegir:
    - Número de nodos
    - Probabilidad de conexión
    - Cantidad de colores
- Llama internamente al backend para crear el grafo inicial.
- Redirige a la pantalla de previsualización.

EntradaNumerica.jsx

- Componente reutilizable para entradas numéricas.
- Valida que los datos sean correctos (números positivos, enteros, dentro de rango).

Boton.jsx

- Componente estándar de botones.
- Aplica estilos uniformes a toda la interfaz.

Previsualizar.jsx

- Muestra el grafo generado.
- Permite ejecutar el algoritmo de coloración.
- Muestra el estado inicial del grafo antes de ser coloreado.

Menu.jsx

- Pantalla principal del sistema.
- Desde aquí se accede a todas las funcionalidades:
    - Generar grafo

    - Cargar grafo
    - Ver documentación
    - Ejecutar coloración
    - Ajustar parámetros

Tecnologías clave utilizadas

- React (useState, useEffect)
- Vite
- CSS para estilos del sistema

Backend — Algoritmos y Procesamiento

El backend maneja toda la lógica del problema de coloración de grafos.

Responsabilidades del backend

- Generar grafos aleatorios.
- Representar aristas y nodos en estructuras internas.
- Ejecutar algoritmos probabilísticos de coloración.
- Detectar y corregir conflictos entre nodos adyacentes.
- Controlar criterios de parada (sin conflictos, límite de iteraciones, etc.).
- Mediación entre frontend y resultados de cálculo.
- Gestión de carga/guardado de grafos en formato JSON.

Procesos principales gestionados por el backend

Generación del grafo

- Se crea un grafo basado en:
    - Número de nodos
    - Probabilidad de conexión
    - Número de colores permitidos
- Se construyen listas de adyacencia internas.

Algoritmo de coloración probabilística

El backend implementa los métodos de búsqueda:

- Selección aleatoria de nodos.
- Asignación de colores con heurísticas probabilísticas.
- Detección y corrección de conflictos.
- Registro de métricas de ejecución.

Comunicación con el frontend

- Recibe datos enviados desde React.
- Procesa los algoritmos y responde con JSON:
    - Estados del grafo
    - Colores asignados
    - Cantidad de conflictos
    - Tiempos de ejecución

Descripción del Algoritmo de Coloración Probabilística

El algoritmo de coloración probabilística implementado en este proyecto busca asignar colores a los nodos de un grafo de manera que ningún par de nodos adyacentes comparta el mismo color.

A diferencia de métodos determinísticos clásicos, esta técnica utiliza aleatoriedad y heurísticas probabilísticas para explorar el espacio de soluciones y minimizar los conflictos presentes en el grafo.

El proceso se basa en la construcción de una secuencia de estados del grafo donde, en cada iteración, se selecciona un nodo, se asigna un color y se evalúa su impacto en los conflictos actuales. El algoritmo continúa hasta encontrar una coloración válida o hasta alcanzar un límite definido de iteraciones.

Visión general del algoritmo

El algoritmo consta de cuatro componentes principales:

1. **Representación interna del grafo**

2. **Selección probabilística de nodos**
3. **Asignación aleatoria o heurística de colores**
4. **Corrección de conflictos y criterio de parada**

Representación interna del grafo

El grafo es almacenado como una **lista de adyacencia**, donde:

- Cada nodo se representa por un índice entero.
- Cada nodo mantiene una lista de nodos vecinos conectados por una arista.
- El color de cada nodo se representa como un número entero dentro del rango [0, k-1], siendo *k* la cantidad de colores permitidos.

Esta estructura permite evaluar conflictos, verificar restricciones y actualizar coloraciones de forma eficiente.

Selección probabilística del nodo

En cada iteración del algoritmo, se elige un nodo para intentar recolorarlo. Esta selección sigue una estrategia probabilística que aumenta la diversidad en la exploración del grafo.

Estrategias comunes de selección:

- **Selección uniforme aleatoria:** todos los nodos tienen la misma probabilidad de ser elegidos.
- **Selección ponderada por conflictos:** nodos con más conflictos tienen mayor probabilidad de ser seleccionados.
- **Selección dirigida:** se selecciona siempre algún nodo actualmente en conflicto.

Dependiendo de la implementación, una combinación de estas estrategias puede ser utilizada para equilibrar velocidad y precisión.

Asignación de colores

Una vez seleccionado un nodo, se evalúan los colores posibles {0, 1, ..., k-1}.

Métodos posibles de asignación:

- **Color aleatorio puro:** el nodo se colorea con un color elegido al azar.
- **Color heurístico:** se elige el color que menor número de conflictos ocasiona.
- **Color probabilístico:** cada color recibe una probabilidad inversamente proporcional al número de conflictos que generaría. El color con mayor peso tiene más probabilidad de ser elegido, pero no de forma determinística, lo que permite escapar de óptimos locales.

Evaluación y corrección de conflictos

Un conflicto ocurre cuando un nodo tiene el mismo color que alguno de sus vecinos.

Cálculo de conflictos:

Para cada nodo u:

$$\text{Conflictos}(u) = \text{cantidad de vecinos } v \text{ donde } \text{color}(v) == \text{color}(u)$$

Después de asignar un color, el algoritmo recalcula los conflictos del nodo y sus vecinos, actualizando la métrica global:

- Conflictos totales
- Conflictos por nodo
- Iteración actual

Si un movimiento aumenta los conflictos, el algoritmo puede:

- Aceptar el cambio con probabilidad baja (exploración), o
- Revertirlo si supera un límite tolerable.

Si el algoritmo no logra eliminar los conflictos, se aplican límites como:

- Máximo número de iteraciones
- Máximo número de recoloraciones fallidas
- Estancamiento sin mejora
- Tiempo máximo de ejecución

Cuando se alcanza alguno, se detiene el proceso y devuelve el mejor estado encontrado.

El algoritmo se detiene exitosamente cuando:

Conflictos totales == 0

Descripción de los algoritmos utilizados

Monte Carlo

Descripción general

El algoritmo Monte Carlo implementado en este proyecto es un enfoque probabilístico para el problema de coloración de grafos. La solución se basa en la ejecución repetida de coloraciones completamente aleatorias sobre un mismo grafo con el objetivo de identificar configuraciones con bajo número de conflictos, o preferiblemente sin conflictos.

Cada ejecución consiste en asignar colores aleatoriamente a todos los nodos del grafo y posteriormente evaluar si existen conflictos, definidos como aristas cuyos nodos extremos comparten el mismo color. Este proceso se repite una cantidad fija de veces indicada por el usuario. Durante toda la ejecución se mantienen métricas como la cantidad de intentos exitosos, el tiempo total de ejecución y un historial detallado por intento.

A diferencia de algoritmos deterministas, este enfoque no garantiza una solución óptima en una única ejecución, pero permite explorar múltiples configuraciones de manera rápida, confiando en la probabilidad de alcanzar soluciones válidas mediante repetición estocástica.

Idea principal del algoritmo

- Realizar múltiples coloraciones aleatorias independientes.
- Evaluar el número de conflictos tras cada coloración.
- Registrar los resultados de cada intento en un historial.
- Contabilizar cuántas ejecuciones resultan en soluciones sin conflictos.
- Calcular métricas de rendimiento (tiempo total, tiempo promedio).
- Permitir inspección e incluso corrección manual de resultados dentro de la interfaz.

Funcionamiento interno basado en la implementación

El algoritmo está implementado en la función MonteCarloColoracion, la cual recibe un grafo preconstruido y una cantidad de iteraciones. Para cada intento, se ejecutan los siguientes pasos:

1. El grafo asigna colores aleatorios a todos sus nodos mediante la función colorear_grafo.
2. Se obtiene el mapa actual de colores usando obtener_colores.
3. Se calcula el número total de conflictos usando total_conflictos.
4. Se registra el intento en una estructura de historial.
5. Si el número de conflictos es cero, se incrementa el contador de éxitos.
6. Se continúa hasta completar todas las iteraciones.
7. Se mide el tiempo total de ejecución usando performance.now.

Al finalizar, la función retorna:

- El historial completo,
- El tiempo total transcurrido,
- La cantidad total de soluciones válidas encontradas.

Parámetros de entrada y salida

Entrada

Parámetro	Tipo	Descripción
grafo	Grafo	Grafo que contiene nodos y aristas
iteraciones	number	Cantidad de ejecuciones aleatorias

Salida

Resultado	Tipo	Descripción
historial	arreglo	Registros de cada intento
tiempoTotal	number	Tiempo total de ejecución en ms
exitos	number	Cantidad de soluciones válidas

Complejidad temporal y espacial

- **Complejidad temporal:**

  $O(n \times i)$

  Donde:

  - n = número de nodos
  - i = número de iteraciones

- **Complejidad espacial:**
  O(n)

  Ya que se almacena una asignación de color por nodo en cada intento y un historial global.

Las Vegas

Descripción general

El algoritmo Las Vegas implementado en este proyecto es un algoritmo probabilístico que, a diferencia del enfoque Monte Carlo, garantiza entregar una solución válida siempre que exista y dentro de un número máximo de intentos. La característica principal del algoritmo es que no se detiene cuando se agotan las iteraciones, sino que finaliza únicamente cuando encuentra una coloración sin conflictos.

Durante la ejecución, el algoritmo recolorea completamente el grafo en cada intento, evaluando los conflictos producidos. Si el número de conflictos es cero, el algoritmo se detiene inmediatamente y retorna la solución encontrada. Para evitar ciclos infinitos, se establece un límite máximo de intentos.

El desempeño del algoritmo se mide en términos del número de intentos necesarios, el tiempo total de ejecución y la calidad de la solución final.

Este algoritmo fue modificado levemente para que solamente se coloreen los nodos conflictivos dentro de la ejecución para poder tener una solución en tiempo polinomial

Idea principal del algoritmo

- Repetir coloraciones aleatorias.
- Evaluar conflictos tras cada intento.
- Terminar cuando se encuentre una solución válida.
- Registrar cada intento en el historial.
- Calcular métricas de desempeño.
- Garantizar solución válida si existe dentro del límite de intentos.

Funcionamiento interno basado en la implementación

El algoritmo se ejecuta invocando la función LasVegasColoracion(grafo) desde el componente visual. Esta función gestiona internamente el proceso iterativo hasta cumplir la condición de parada.

En cada ejecución:

1. Se recolorea el grafo aleatoriamente.
2. Se calcula el número de conflictos.
3. Se guarda el resultado en el historial.
4. Si hay cero conflictos, el algoritmo finaliza.
5. Si no, intenta de nuevo hasta alcanzar maxIntentos.

El componente visual no controla el ciclo del algoritmo, solo recupera:

- El historial de intentos,
- El tiempo total,
- La cantidad de intentos utilizados.

Parámetros de entrada y salida

Entrada

Parámetro	**Tipo**	**Descripción**
grafo	Grafo	Grafo a colorear
maxIntentos	number	Cantidad máxima de intentos

Salida

Resultado	**Tipo**	**Descripción**
historial	arreglo	Intentos realizados
tiempoTotal	number	Tiempo total en ms
intentos	number	Número de intentos

Complejidad temporal y espacial

- **Complejidad temporal:**

  $O(n \times i)$

  Donde:

  - n = cantidad de nodos
  - i = intentos necesarios hasta encontrar solución
- **Complejidad espacial:**

  $O(n)$

  Se almacena un mapa de colores por intento.

Instrucciones de ejecución

Dependencias

1. **Descargar el repositorio**

a. Ir al enlace del repositorio oficial del proyecto:

https://github.com/MaikelFn/Proyecto-2-Coloracion-Probabilistica-de-Grafos-con-Busqueda

b. Seleccionar **"Code"** y presionar **"Download ZIP"** para descargar todo el proyecto. Alternativamente, se puede clonar desde Git:

git clone https://github.com/MaikelFn/Proyecto-2-Coloracion-Probabilistica-de-Grafos-con-Busqueda.git

c. Descomprimir el archivo ZIP o abrir el directorio generado al clonar el proyecto.

2. **Herramientas / Dependencias**

Para ejecutar el proyecto de **Coloración Probabilística de Grafos**, es necesario contar con las siguientes herramientas instaladas:

a. **Node.js 18.0+**
b. **npm** (incluido con Node.js)
c. **Vite** (se instala automáticamente al ejecutar npm install)
d. Un navegador moderno (Chrome, Edge, Firefox)

3. **Instalar dependencias del proyecto**

Dentro del /Codigo del proyecto, ejecutar en la terminal:

```
npm install
```

Este comando instalará todas las dependencias necesarias para que el sistema funcione.

Ejecución

a. Abrir una terminal y navegar al directorio donde se encuentra el proyecto descargado o clonado. Por ejemplo:

```
cd Proyecto-2-Coloracion-Probabilistica-de-Grafos-con-Busqueda
```

b. Instalar las dependencias necesarias ejecutando:

```
npm install
```

c. Una vez finalizada la instalación, ejecutar con:

```
npm run dev
```

d. La terminal mostrará un mensaje similar a:

```
VITE vX.X.X  ready in XXXX ms

Local:  http://127.0.0.1:5173/
```

e. Una vez que Vite esté ejecutándose, abrir el navegador e ingresar a la dirección:

http://127.0.0.1:5173/

Manual de usuario

Tras ejecutar y abrir el proyecto, se mostrará esta interfaz inicial

En la cual se verán 2 opciones, "Generar Grafo Aleatorio" y "Generar Grafo Manual"

Configuración inicial

1. **Grafo manual**

Tras presionar "Generar grafo manual" saldrá esta interfaz en donde el usuario tendrá que ingresar los datos necesarios utilizando el botón "Agregar Nodo" a la izquierda y luego asignándole aristas/conexiones en la parte derecha. (Ningún nodo puede quedar sin conexiones y todos tienen que estar conectados entre sí a modo de puente, además, el mínimo son 20, de lo contrario, no se habilitará el botón "Generar Grafo")

2. Grafo Aleatorio

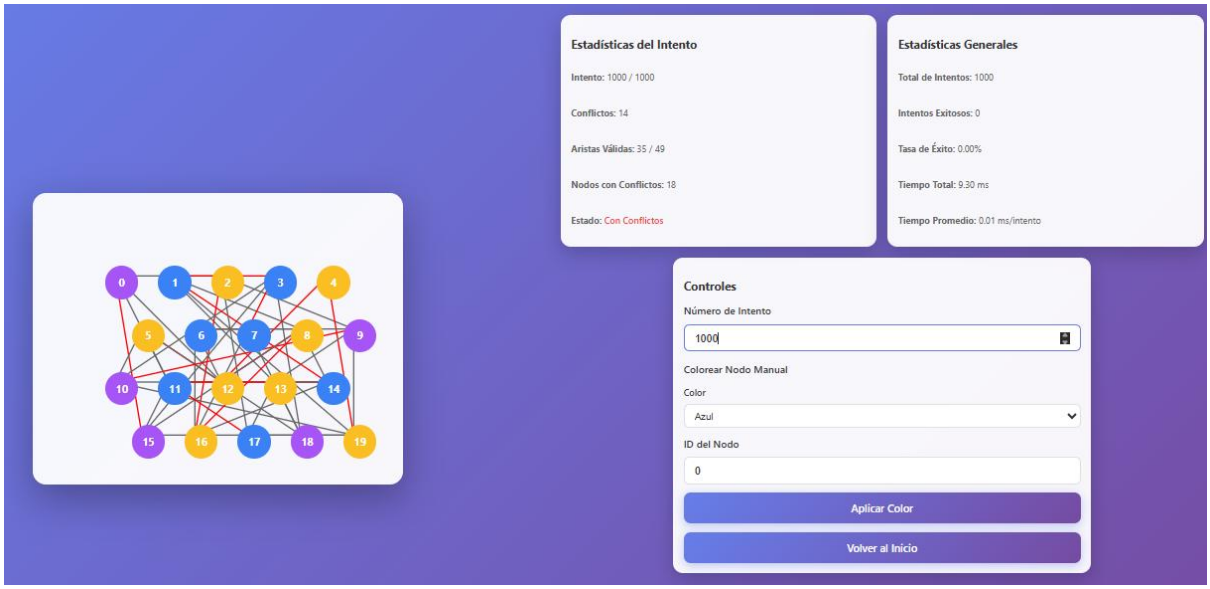
Tras presionar “Generar aleatorio” saldrá esta interfaz en donde el usuario tendrá que ingresar los datos necesarios utilizando por medio de los campos de texto, indicando la cantidad de nodos y la probabilidad de conexión.

Tras completar la configuración de los nodos, se presentará esta interfaz, que es en donde se podrá visualizar el grafo con los nodos introducidos y a la derecha un panel para resolver con “Monte Carlo” o “Las Vegas”.

Flujos generales

1. Monte Carlo

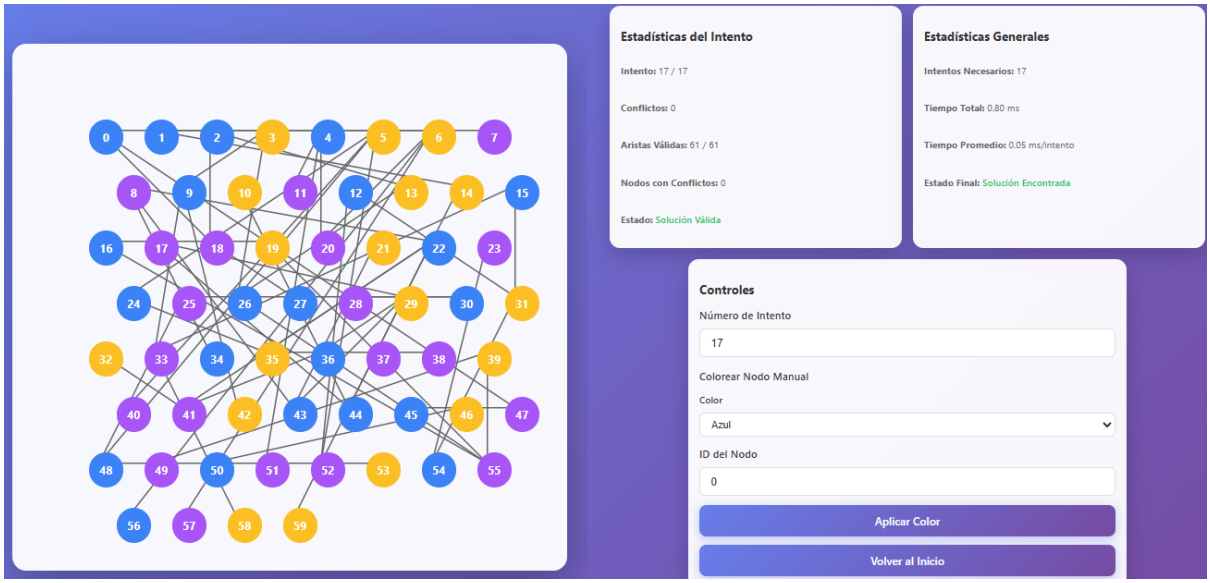
Si se escogió la opción Monte Carlo, se mostrará esta interfaz en donde el usuario tendrá que ingresar la cantidad de intentos para resolver el grafo.



Tras presionar “Ejecutar” se mostrará el último intento por defecto del grafo junto con sus estadísticas del intento seleccionado y las generales del grafo.

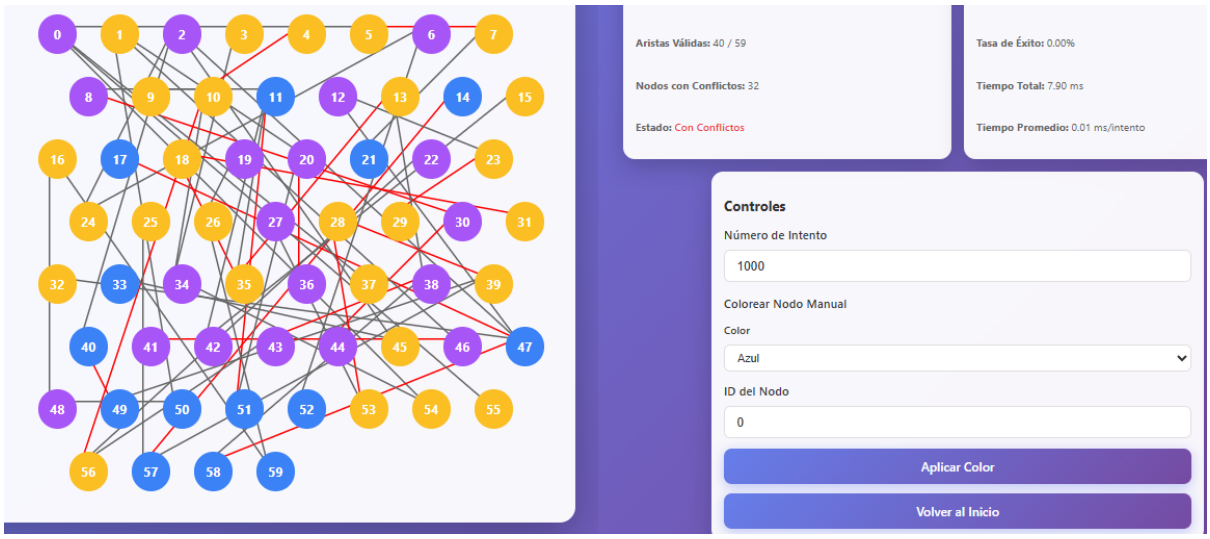
2. Las Vegas

Si se escogió la opción Las Vegas, se mostrará esta interfaz en donde el usuario solamente tendrá que presionar “Ejecutar” para iniciar la resolución.

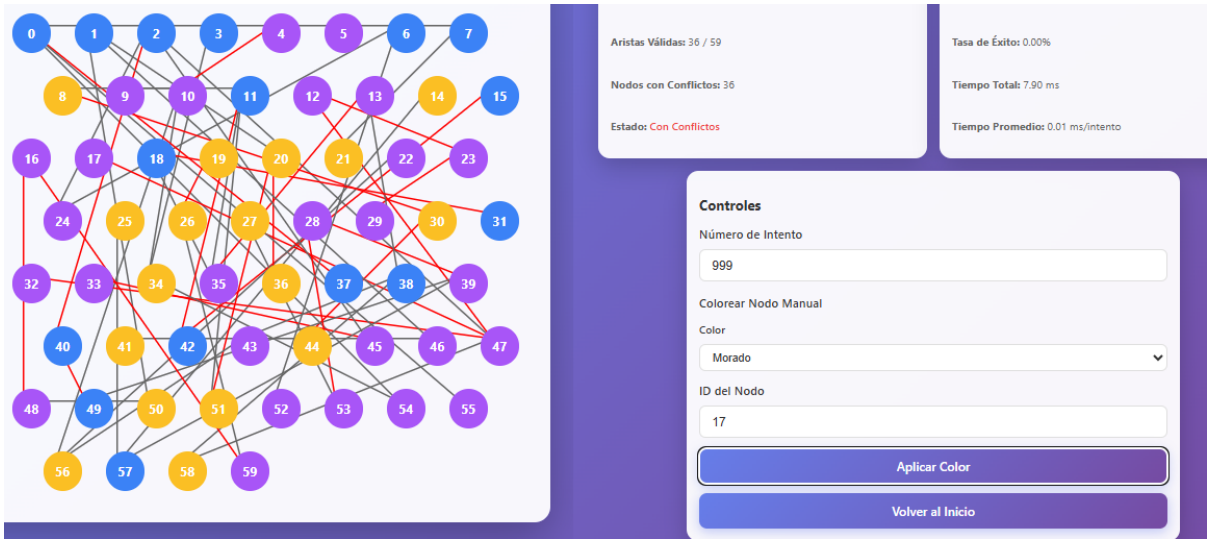


El algoritmo las vegas se seguirá ejecutando hasta encontrar una solución, en la mayoría de los casos esta no podrá ser encontrada, activando una bandera de 100000 intentos para evitar el colapso de la página.

Historial



El historial se guarda con el número de intentos, el usuario puede subir o bajar ese número para consultar la información que tiene el grafo en cada intento, además, se pueden cambiar los colores de los nodos para que el algoritmo recalcule la solución, de forma que se actualice el estado final del grafo.



Interfaz tras haber cambiado el nodo id 17 a color morado en el intento 999

Análisis de resultados

Comparativa entre Las Vegas y Monte Carlo

Algoritmo	Intentos configurados	Intentos usados	Intentos exitosos	Tasa de éxito	Tiempo total	Tiempo promedio	Resultado final
Las Vegas	— (no configurable)	17	1 (en el intento 17)	100% *(cuando detiene)*	0.80 ms	0.05 ms/intento	Solución encontrada
Monte Carlo	1000	1000	0	0.00%	7.90 ms	0.01 ms/intento	No encontró solución

- En este grafo (60 nodos, p = 0,02), Las Vegas encontró una solución válida en tan solo 17 intentos, lo cual es extremadamente eficiente.
- El tiempo total fue 0.80 ms, demostrando que, cuando el grafo es relativamente sencillo de colorear, Las Vegas converge muy rápido.
- Por el contrario, Monte Carlo realizó 1000 coloraciones aleatorias y no produjo ninguna solución válida, resultando en una tasa de éxito de 0%.
- Aunque Monte Carlo es ligeramente más rápido por intento, no garantiza éxito, incluso cuando se ejecuta cientos o miles de veces.
- Las Vegas garantiza solución *si el grafo es 3-coloreable*, y en este caso lo demostró claramente al converger rápido.

En los experimentos realizados se observó que el algoritmo Las Vegas logró encontrar una coloración válida de manera rápida y consistente cuando el grafo era 3-coloreable, mientras que Monte Carlo, aun ejecutando cientos o miles de intentos, no garantizó ninguna solución válida. Ambos algoritmos presentan tiempos por intento muy bajos, pero difieren ampliamente en su confiabilidad: Monte Carlo ofrece un costo fijo de ejecución con resultados inciertos, mientras que Las Vegas garantiza éxito, pero puede requerir una cantidad impredecible de intentos.

Ventajas y desventajas

Algoritmo Las Vegas

Ventajas

- Garantiza encontrar una solución si esta existe.
- Converge muy rápido cuando el grafo no es denso o tiene una estructura favorable.
- Produce resultados completamente válidos sin necesidad de posprocesamiento.

Desventajas

- El número de intentos no es predecible (tiempo probabilístico).
- En grafos muy complejos puede necesitar muchísimos intentos.
- No es ideal cuando se requiere un límite estricto de tiempo.

Algoritmo Monte Carlo

Ventajas

- Tiempo de ejecución totalmente controlado (se fija el número de intentos).
- Fácil de implementar y analizar.
- Útil para estimar tasas de éxito o tendencias estadísticas.

Desventajas

- No garantiza encontrar solución, aunque esta exista.
- Puede requerir muchos intentos para obtener resultados de calidad.

- Su rendimiento disminuye en grafos medianamente densos o con muchas restricciones.

Eficiencia

Cuándo usar Monte Carlo

- Cuando se necesita tiempo fijo o computación acotada.
- Cuando se desea obtener estimaciones estadísticas, no soluciones exactas.
- En escenarios donde la solución perfecta no es indispensable, como:
  - simulaciones,
  - estimación de probabilidad de coloración,
  - análisis de comportamientos promedio del grafo.

Cuándo usar Las Vegas

- Cuando se requiere una solución válida sí o sí.
- En tareas donde la corrección es más importante que el tiempo exacto, por ejemplo:
  - validación de coloración,
  - preprocesamiento de grafos,
  - aplicaciones donde los errores no son aceptables.
- Ideal cuando sabemos o sospechamos que el grafo sea coloreable con k colores.

Enlaces

Github

https://github.com/MaikelFn/Proyecto-2-Coloracion-Probabilistica-de-Grafos-con-Busqueda

Netlify (Despliegue)

https://keen-fairy-3b3dbf.netlify.app/