TALLER N°2

VERTEX COLORING GRAPH PROBLEM

usach

Taller de programación 2-2024
Fecha: 10/12/2014
Autor: Sebastian De La Fuente

TALLER N°2

VERTEX COLORING GRAPH PROBLEM

Explicación breve del algoritmo

En este informe se aborda el clásico problema del coloreo de grafos, el cual consiste en asignar colores (representados por números) a los vértices de un grafo no dirigido, de manera que ningún par de vértices adyacentes comparten el mismo color. Puesto que, en el peor de los casos, podrían requerirse tantos colores como vértices existen (lo que sucede cuando el grafo es completo), el objetivo fundamental es determinar el menor número de colores necesarios para lograr un coloreo adecuado, minimizando así la cantidad total de colores utilizados. (ver anexo 1 para ver ejemplos de coloreo de grafos.)

Para enfrentar este desafío, el programa combina la heurística DSATUR (Degree of Saturation) con un enfoque de branch and bound. En cada paso, se selecciona el vértice más “restringido”, es decir, aquel con mayor diversidad de colores entre sus vecinos, y se le asigna un color factible. Esto permite focalizar la búsqueda en los vértices que presentan mayores dificultades, reduciendo así el espacio que debe explorarse.

Si en algún punto se detecta que no será posible mejorar la mejor solución obtenida hasta el momento, se "podan" las ramas del árbol de búsqueda, evitando dedicar recursos a exploraciones infructuosas. Además, se establece un límite temporal es decir, si la ejecución se prolonga más de lo previsto, se interrumpe la búsqueda exhaustiva y se adopta un coloreo “greedy” más sencillo. Esta combinación de heurísticas, poda y límite de tiempo busca equilibrar la calidad de la solución con la eficiencia del cómputo, Para así en la mayoría de los casos obtener una solución óptima. Al finalizar, el algoritmo muestra los colores asignados a cada vértice y el número total de colores utilizados. Para una mayor comprensión de la solución propuesta, puede consultarse el pseudocódigo presentado en el Anexo 2.

Heurísticas o técnicas utilizadas

Para abordar el problema del coloreo de grafos, se ha optado por una combinación estratégica de tres técnicas principales: DSATUR, Branch & Bound y un coloreo Greedy. Esta selección no sólo busca hallar la mínima cantidad de colores necesarios, sino también optimizar el tiempo de ejecución y la calidad de las soluciones.

La heurística DSATUR (Degree of Saturation) juega un papel fundamental en la selección de los vértices a colorear. Esta técnica se basa en la información de saturación, es decir, en el número de colores distintos ya asignados a los vecinos de un vértice. Al priorizar la coloración de aquellos vértices con mayor saturación, DSATUR orienta la búsqueda hacia aquellas regiones del grafo más difíciles de colorear, reduciendo el espacio de búsqueda

efectivo. Esto no sólo aumenta la probabilidad de encontrar soluciones de buena calidad en menos tiempo, sino que también orienta las decisiones cromáticas para evitar redundancias o asignaciones ineficientes.

Por otra parte, el método de Branch & Bound proporciona un marco más exhaustivo, pero controlado. Al explorar sistemáticamente el espacio de soluciones, esta técnica emplea cotas superiores e inferiores para descartar rápidamente partes del espacio que no conducirán a mejoras. De esta forma, se evita desperdiciar recursos en caminos infructuosos, reduciendo el tiempo de ejecución sin comprometer la posibilidad de hallar la solución óptima. Este enfoque se asienta en la idea de backtracking, pues retrocede en sus decisiones cuando detecta que es imposible superar la mejor solución hallada hasta el momento.

El coloreo Greedy, por su parte, se presenta como una alternativa rápida y simple para obtener una solución factible. Aunque no garantiza la optimalidad. De esta manera, si el tiempo de ejecución se extiende demasiado, el algoritmo puede detener la búsqueda exhaustiva y conformarse con una solución greedily obtenida, manteniendo la eficiencia cuando las condiciones no permiten una exploración más profunda.

Adicionalmente, otras técnicas complementarias contribuyen a un mejor desempeño de estas estrategias principales. Por ejemplo, la estimación del tamaño de la máxima clique (max clique) proporciona una cota inferior natural al número mínimo de colores requeridos, ayudando así a guiar las decisiones del Branch & Bound. Del mismo modo, el criterio de seleccionar el siguiente vértice a colorear según su mayor grado de saturación fortalece la heurística DSATUR, concentrando los esfuerzos en las secciones más complejas del grafo, mejorando la eficiencia en la búsqueda y evitando explorar configuraciones poco prometedoras.

En conjunto, el uso de heurísticas (como DSATUR y la selección de vértices con mayor saturación), técnicas de poda de la búsqueda (Branch & Bound) y estrategias más sencillas como el Greedy, genera un sistema híbrido capaz de equilibrar la calidad de la solución con la eficiencia computacional. Así, el programa logra dirigirse hacia soluciones prometedoras, descartar con rapidez aquellas rutas sin salida y, en casos de limitaciones temporales, recurrir a un método más directo para asegurar resultados en plazos razonables. Esta combinación optimiza significativamente el proceso de búsqueda, evitando la exploración exhaustiva de todas las posibilidades y maximizando la productividad de los recursos invertidos.

Funcionamiento del programa

Al ejecutar el programa, se presenta un menú con la siguiente estructura:

Menu de Coloreo de Grafos

---

1. Leer y colorear un grafo desde archivo

2. Salir

Seleccione una opcion:

Opción 1: Leer y colorear un grafo desde archivo

Si el usuario selecciona la primera opción, el programa solicita ingresar el nombre de un archivo .txt que contenga la representación del grafo (vértices y conexiones). El archivo debe estar ubicado en la carpeta grafos/, cabe mencionar que ya existen archivos de prueba para este programa . Una vez ingresado el archivo, el programa realiza los siguientes pasos:

1. **Lectura del archivo**:
   Se llama al método readGraphFromFile de la clase Graph, el cual construye la estructura del grafo, cargando todos los vértices y sus respectivas conexiones.
2. **Instancia de la clase Coloring**:
   Se crea un objeto de la clase Coloring, que contiene las implementaciones necesarias para colorear el grafo utilizando las técnicas DSATUR, Branch & Bound y Greedy.
3. **Medición del tiempo de ejecución**:
   Antes de iniciar el proceso de coloreo, se inicia un cronómetro para medir el tiempo total que toma la ejecución.
4. **Ejecución del método DSATUR con Branch & Bound**:
   Se llama al método dsaturBranchAndBound, que utiliza las estrategias descritas previamente para encontrar el número mínimo de colores necesarios para colorear el grafo. Si el tiempo de ejecución supera los 30 segundos, el proceso se interrumpe y se utiliza un coloreo Greedy mejorado.
5. **Visualización de resultados**:
   - Si el proceso se completa dentro del tiempo límite, se muestra la solución óptima encontrada por DSATUR con Branch & Bound, incluyendo los colores asignados a cada vértice y el número mínimo de colores utilizados.
   - Si el tiempo límite es excedido, se muestra la solución aproximada obtenida mediante el coloreo Greedy.

6. **Tiempo total de ejecución**:
   Se muestra el tiempo total empleado en la resolución del problema.

Al finalizar el proceso de coloreo, el programa regresa al menú principal, permitiendo al usuario colorear otro grafo o salir del programa.

Opción 2: Salir

Si el usuario selecciona la opción 2, el programa finaliza mostrando el mensaje:

Saliendo del programa. ¡Hasta luego!.

ver ejemplo de ejecución en el anexo 3.

Aspectos de implementación y eficiencia

El desarrollo del programa para resolver el problema del coloreo de grafos se basó en un enfoque de programación orientado a objetos (POO), estructurando la solución en clases que permiten una mejor organización y reutilización del código. Se definieron las clases principales: Graph y Coloring. Estas clases están diseñadas para descomponer el problema en módulos específicos y manejables, cada uno cumpliendo un propósito claro que contribuye al funcionamiento eficiente del programa.

A continuación, se describen las principales clases y su papel en la solución:

- **Graph**:
  Representa la estructura del grafo, almacenando la información de los vértices y sus conexiones. Esta clase incluye el método readGraphFromFile, que construye el grafo a partir de un archivo de texto proporcionado por el usuario, facilitando la entrada de datos para el programa.

- **Coloring**:
  Implementa los algoritmos necesarios para el coloreo del grafo, incluyendo DSATUR con Branch & Bound y el coloreo Greedy. Esta clase gestiona las heurísticas y la lógica de poda para reducir el espacio de búsqueda, mejorando la eficiencia del proceso de coloreo.

En cuanto a los aspectos de optimización:

- **Cálculo de la máxima clique aproximada**:
  Este procedimiento proporciona una cota inferior para el número mínimo de colores necesarios, ayudando al Branch & Bound a descartar soluciones que no sean óptimas, reduciendo el espacio de búsqueda.

- **Selección de vértices por grado de saturación**:
  Este criterio, central en la heurística DSATUR, asegura que los vértices más restringidos sean coloreados primero, maximizando la efectividad del algoritmo y disminuyendo la probabilidad de errores en asignaciones futuras.

- **Estrategias de poda**:
  El método de Branch & Bound utiliza cotas superiores e inferiores para identificar rápidamente rutas no prometedoras, evitando exploraciones exhaustivas innecesarias.

- **Coloreo Greedy**:
  Sirve como respaldo en caso de que el tiempo de ejecución supere el límite establecido, proporcionando una solución rápida aunque subóptima.

La combinación de estas técnicas y estructuras asegura un manejo eficiente de los recursos, permitiendo que el programa sea capaz de encontrar soluciones óptimas o cercanas a la óptima en un tiempo razonable, incluso para grafos complejos. Este enfoque modular, combinado con heurísticas y estrategias de optimización, facilita el mantenimiento del código y asegura su escalabilidad para futuras extensiones.

En este trabajo se evidenció una mejora significativa en el entendimiento y abordaje del problema del coloreo de grafos. A pesar de ser un desafío de alta complejidad, la solución desarrollada no requirió un número excesivo de estructuras de datos, lo que facilitó su implementación. Esto fue posible gracias al uso de la biblioteca STL, que proporciona estructuras de datos altamente eficientes y bien optimizadas. Su empleo permitió centrar los esfuerzos en la lógica del algoritmo y no en la implementación manual de estructuras auxiliares.

En términos de eficiencia, el programa cumple con lo solicitado, mostrando un desempeño adecuado en grafos simples y medianamente complejos. Sin embargo, se identificaron limitaciones al abordar grafos más grandes y desafiantes, como el "DSJR500.5", para el cual la solución propuesta produjo un resultado de 69 colores, muy lejos del valor aproximado óptimo de 48. Esto demuestra que, aunque la solución funciona bien para problemas menos exigentes, aún presenta oportunidades de mejora para casos de mayor complejidad.

A modo de conclusión, este trabajo refleja un avance significativo respecto a tareas previas, tanto en la comprensión del problema como en la implementación de una solución funcional. Si bien no se alcanzó un nivel de eficiencia óptima en todos los casos, el desarrollo del programa muestra un balance entre simplicidad, funcionalidad y eficiencia, logrando resultados satisfactorios en la mayoría de los casos. Este progreso deja una sensación de satisfacción y marca un paso adelante en el aprendizaje y resolución de problemas algorítmicos complejos.

Ejecución del código

Guía para ejecutar el programa y realizar pruebas

Preparación del proyecto:

1. **Descomprimir el proyecto:**
   Extrae la carpeta del proyecto en una ubicación local en tu sistema.

Compilación del programa principal:

Compilación con Makefile:
Utiliza el Makefile proporcionado para compilar tanto el programa principal como los archivos de prueba. Para ello, ejecuta el siguiente comando desde la terminal dentro de la carpeta del proyecto:

```
make all
```

Este comando generará el ejecutable principal main, junto con otros archivos de prueba: Graphtest, Coloringtest.

Ejecución del programa principal:

Iniciar el programa:
Una vez compilado, ejecuta el programa principal utilizando el ejecutable main generado:
./main

Menú interactivo:
Al iniciar el programa, se mostrará un menú que permite elegir entre las siguientes opciones:

- **Leer archivo y colorear grafo:**
  Esta opción solicita el ingreso del nombre de un archivo .txt que contenga la descripción del grafo (vértices y conexiones). Luego, el programa colorea el grafo utilizando las técnicas descritas y muestra los resultados por pantalla.
- **Salir:**
  Finaliza la ejecución del programa.

Ejecución de archivos de prueba:

Archivos de prueba predefinidos:
Para verificar el correcto funcionamiento del programa, se han generado archivos de prueba que pueden ser ejecutados individualmente. Para ello, utiliza los ejecutables generados durante la compilación:
./testGraph

./testColoring

Cada archivo de prueba evalúa componentes específicos del programa, como la correcta lectura de grafos, las heurísticas aplicadas, y la ejecución de los algoritmos.

Limpieza de archivos generados:

Eliminar archivos temporales:
Para limpiar los archivos objeto y ejecutables creados durante la compilación, ejecuta el siguiente comando:
make clean

Librerías y consideraciones adicionales:

Librerías utilizadas:
El programa hace uso de las siguientes librerías estándar de C++:
#include <iostream>

#include <vector>

#include <chrono>

#include <fstream>

#include <algorithm>

#include <set>

Estas proporcionan funcionalidades esenciales para la gestión del tiempo, entrada/salida de archivos, y manipulación de datos.

Entorno de desarrollo:

- **En Linux:**
  Simplemente asegúrate de tener instalado make y un compilador como g++. Ejecuta el comando make all dentro de la carpeta del proyecto.
- **En Windows:**
  Asegúrate de tener instalado MinGW con las configuraciones adecuadas para C++. Verifica que el comando make esté disponible en el sistema.

Pruebas personalizadas:
La carpeta del proyecto contiene una subcarpeta llamada test con pruebas predefinidas para validar el programa. Si deseas realizar pruebas adicionales, coloca tus propios casos de prueba en formato .txt dentro de esta carpeta.

Este enfoque modular, con soporte para pruebas automatizadas y personalizadas, facilita la evaluación del programa y asegura su correcto funcionamiento en diferentes entornos.

Bibliografía

Furini, F., Gabrel, V., & Ternier, I.-C. (2015). *An improved DSATUR-based Branch and bound for the Vertex Coloring Problem*. https://optimization-online.org/wp-content/uploads/2015/10/5159.pdf

Umamaheswari, K., & Umavathi, A. R. (s/f). *Vertex coloring in graph theory*. Ijsrr.org. Recuperado el 10 de diciembre de 2024, de https://www.ijsrr.org/down_2057.php

S/f). Researchgate.net. Recuperado el 10 de diciembre de 2024, de https://www.researchgate.net/publication/261513432_Recent_Advances_in_Graph_Vertex_Coloring

Graph Coloring Instances. (s/f). Cmu.edu. Recuperado el 10 de diciembre de 2024, de https://mat.tepper.cmu.edu/COLOR/instances.html

Redirect notice. (s/f). Google.com. Recuperado el 10 de diciembre de 2024, de https://www.google.com/url?sa=i&url=http%3A%2F%2Fwww.geocities.ws%2Flcomp2001%2FTarea2.htm&psig=AOvVaw0UhbAJKHmeoxSCAAYAxmVL&ust=1733890898109000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCLCG142tnIoDFQAAAAAdAAAAABAS

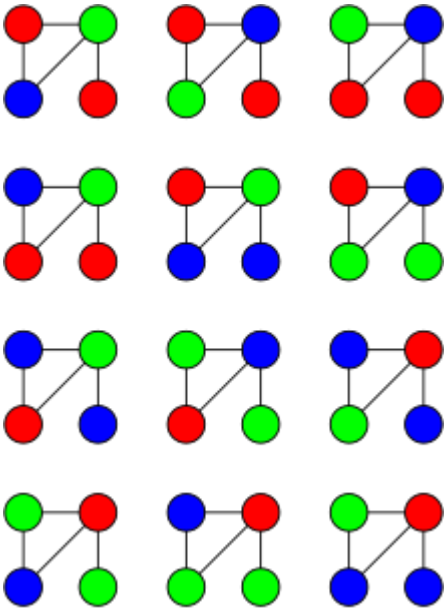
Redirect notice. (s/f-b). Google.com. Recuperado el 10 de diciembre de 2024, de https://www.google.com/url?sa=i&url=https%3A%2F%2Fecfm.usac.edu.gt%2Fnode%2F38&psig=AOvVaw0UhbAJKHmeoxSCAAYAxmVL&ust=1733890898109000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCLCG142tnIoDFQAAAAAdAAAAABAX

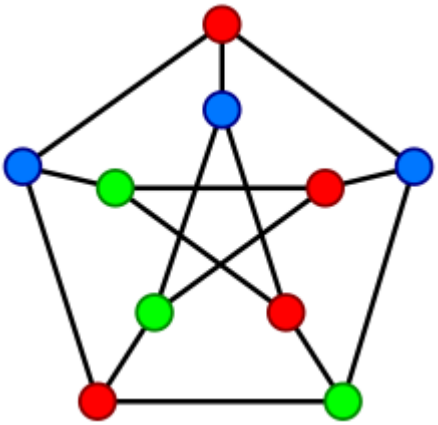
Wikipedia contributors. (s/f). *Coloración de grafos*. Wikipedia, The Free Encyclopedia.

https://es.wikipedia.org/w/index.php?title=Coloraci%C3%B3n_de_grafos&oldid=162904491

Anexos:

1. Grafos coloreados:





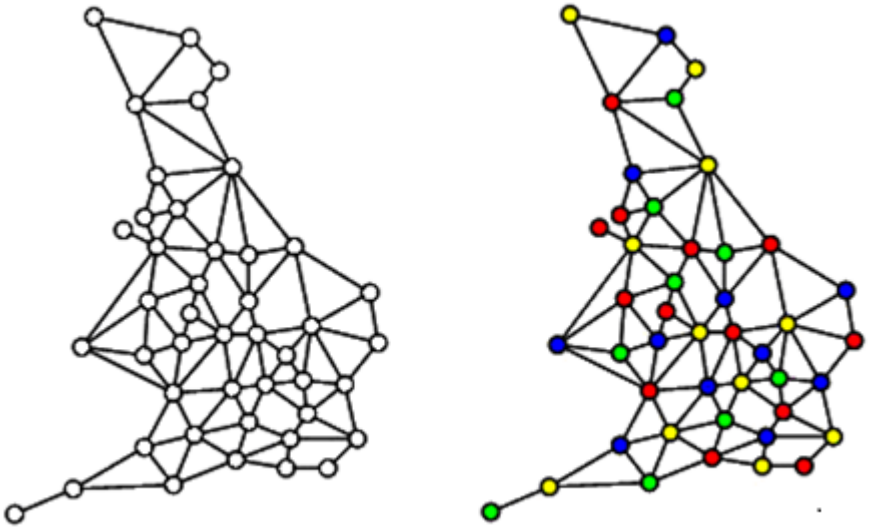
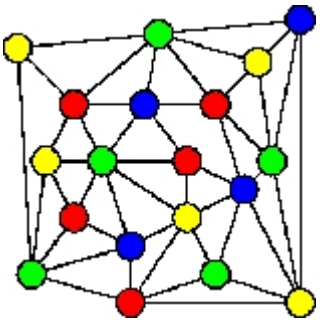


Figura 1



2. Pseudocodigos:

Algorithm 1 DSATUR Branch and Bound

Require: Grafo $G = (V, E)$, referencia a *bestSolution*, referencia a *upperBound*, tiempo *startTime*, referencia a *timeout*

Ensure: Actualiza *bestSolution* y *upperBound* si encuentra una mejor solución

1: $color \leftarrow$ vector de tamaño $|V|$ con -1
2: $saturation \leftarrow$ vector de tamaño $|V|$ con 0
3: $colored \leftarrow$ vector booleano de tamaño $|V|$ en falso
4: $degree \leftarrow$ vector tal que $degree[v] = \text{grado}(v)$
5: **for all** $v \in V$ **do**
6: $\quad degree[v] \leftarrow \text{grado}(v)$
7: **end for**
8: $maxClique \leftarrow \text{aproximarMaxClique}(G)$
9: $\text{DSATUR\_B\_B\_Recursivo}(G, best, upb, 0, 0, c, sn, cl, maxClique, degree, startTime, timeout)$

Algorithm 2 DSATUR_B_B_Recursivo

Require: $G = (V, E)$, ref $bestSolution$, ref $upperBound$, $usedColors$, $step$, $color[]$, $saturation[]$, $colored[]$, $maxClique$, $degree[]$, $startTime$, ref $timeout$
Ensure: Actualiza $bestSolution$ y $upperBound$ si mejora la solución

```
1: now ← tiempoActual()
2: elapsed ← now − startTime
3: if elapsed > 30 s then
4:    timeout ← verdadero
5:    return
6: end if
7: lowerBound ← máx(maxClique, usedColors)
8: if lowerBound ≥ upperBound then
9:    return    // Podar
10: end if
11: if step = |V| then
12:    upperBound ← usedColors
13:    bestSolution ← color
14:    return
15: end if
16: vertex ← seleccionarVertice(saturation, degree, colored)
17: neighborColors ← {}
18: for all w ∈ N(vertex) do
19:    if color[w] ≠ −1 then
20:       neighborColors ← neighborColors ∪ {color[w]}
21:    end if
22: end for
23: for c = 0 to upperBound − 1 do
24:    if c ∈ neighborColors then
25:       continue
26:    end if
27:    color[vertex] ← c
28:    colored[vertex] ← verdadero
29:    for all w ∈ N(vertex) con colored[w] = falso do
30:       saturation[w] ← saturation[w] + 1
31:    end for
32:    DSATUR_B_B_Recursivo(G, best, upbound, máx(usedColors, c+1), step+
       1, c, s, cl, maxClique, degree, startTime, timeout)
33:    if timeout = verdadero then
34:       return
```

```
    end if
    // Backtracking
    color[vertex] ← −1
    colored[vertex] ← falso
    for all w ∈ N(vertex) con colored[w] = falso do
      saturation[w] ← saturation[w] − 1
    end for
  end for
```

3. Compilación

```
seba@DESKTOP-SRO8MLK:/mnt/c/Users/Sebastian/Desktop/lab2tdp$ ./main
--------------------------------------
   Menu de Coloreo de Grafos
--------------------------------------
1. Leer y colorear un grafo desde archivo
2. Salir
Seleccione una opcion: 1
Ingrese el nombre del archivo (sin la carpeta grafos/): test.txt
Mejor solucion encontrada (DSATUR con poda):
Vertice 1: Color 1
Vertice 2: Color 0
Vertice 3: Color 2
Vertice 4: Color 0
Vertice 5: Color 3
Vertice 6: Color 3
Vertice 7: Color 2
Vertice 8: Color 1
Vertice 9: Color 2
Vertice 10: Color 0
Vertice 11: Color 1
Vertice 12: Color 0
Vertice 13: Color 2
Numero minimo de colores encontrados: 4
Tiempo total de ejecucion: 0 ms
--------------------------------------
   Menu de Coloreo de Grafos
--------------------------------------
1. Leer y colorear un grafo desde archivo
2. Salir
Seleccione una opcion: 2
Saliendo del programa. ¡Hasta luego!
seba@DESKTOP-SRO8MLK:/mnt/c/Users/Sebastian/Desktop/lab2tdp$
```

Se utilizó chat gpt para la corrección de ortografía de este informe.