Evaluación 2 - Taller de Programación
Vertex Coloring Problem

Prof: Pablo Román

Noviembre 05 2024

1 Objetivo

Desarrollar una aplicación eficiente y heurística para resolver el problema de coloreo mínimo en un grafo (Vertex Coloring Problem, VCP). Para estos fines se utilizará la estrategia de Ramificación y Acotamiento. Se implementará en el lenguaje C++, el uso de makefile, la estructuración y buenas prácticas de un código orientado al objeto. Se utilizarán las librerías de C++ STL para incrementar la eficiencia.

2 Problema a resolver: Vertex Coloring Problem (VCP)

El problema en parte consiste en disponer de un grafo con N vértices $\{v_k\}$ a los cuales se le deben asignar colores (`https://en.wikipedia.org/wiki/Graph_coloring`). Un color es simplemente un número c_k asignado al vértice v_k, pueden haber como máximo N colores distintos, que en ese caso es un color asociado unicamente a cada vértice. **El problema es encontrar cual es la mínima cantidad de colores que deben ser utilizados en un grafo de forma de que se cumpla que no hayan dos vértices adyacentes con el mismo color**. Recordemos que en un grafo no direccional cada vértice v_k se conecta con sus vecinos v_j con aristas a_{kj}. Esta cantidad mínima χ de colores se denomina **número cromático (Chromatic Number)** (`https://mathworld.wolfram.com/ChromaticNumber.html`).

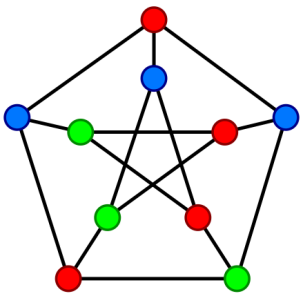

Figure 1: Mínima cantidad de colores ($\chi = 3$) para este grafo (Fuente: Wikipedia Commons).

El problema de coloreo de grafos es antiguo. Por ejemplo se conoce su estudio por el problema de colorear un mapa, donde se tiene el teorema de 4 colores que dice que para grafos planares (cuyo dibujo en una hoja de papel no tiene intersecciones de aristas) siempre existen 4 colores independiente de la cantidad de vértices (`https://mathworld.wolfram.com/Four-ColorTheorem.html`). Para grafos generales ya no se cumple y pueden haber muchos más colores. Este problema es de complejidad computacional exponencial en la cantidad de vértices.

Este problema tiene grandes aplicaciones en la práctica (`https://doi.org/10.1111/j.1475-3995.2009.00696.x`). En la generación de código de máquina se dispone de una cantidad finita de registros en un procesador, cuyo número es claramente inferior a las variables presentes en programas comunes. Las variables que falten por asignar a registros se llevan a memoria principal que es mas lenta que la circuitería presente en el procesador además que deben realizarse procesos de copia de ida y vuelta. El hacer eficiente la asignación de registros corresponde a un problema VCP donde el color corresponde a los registros y los vértices corresponden a las variables del programa. Las aristas del grafo corresponden a la relación de dependencia de una variable en con otra. Minimizar la cantidad

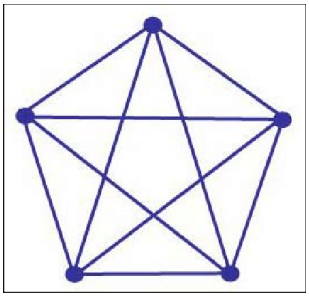

Figure 2: Cinco colores es el mínimo en este caso.

de registros permite un mejor uso de recursos y mayor eficiencia del código de máquina generado. Otra aplicación corresponde a la planificación de tareas en la industria. Se dispone de muchas tareas y unos pocos trabajadores que pueden realizarlas. Además hay tareas que no pueden compartir un trabajador con otras y esto genera un grafo de tareas. Es de interés para la empresa colorear dicho grafo con trabajadores de forma mínima haciendo más eficiente la producción. En la detección de clique en un grafo conforma parte de importantes algoritmos heurísticos.

3 Algoritmos conocidos para resolver VCP

Para un grafo grande el carácter exponencial de la complejidad para resolver este problema requiere de métodos que sean más estimativos que exactos. Una cota superior e inferior al número cromático χ es:

$$\omega \leq \chi \leq \Delta + 1 \tag{1}$$

Donde Δ es el máximo grado de un vértice en el grafo y ω es la cantidad de vértices del clique de mayor tamaño contenido en el grafo. Recordemos que el grado de un vértice es la cantidad de aristas que salen de él. Un clique es un grafo con todos sus nodos conectados con todos, por lo que solo se puede colorear con tantos colores como vértices contiene el clique. El problema VCP es conocido por ser NP-Hard, sin embargo se han logrado resultados próximos al mínimo aplicando heurísticas y en considerable menor tiempo. Los siguientes algoritmos son relevantes para la implementación de este ejercicio:

Coloreo Greedy: Algoritmo de Welsh-Powell Hay muchas variaciones de este algoritmo, el cual finalmente es el origen de varios algoritmos heurísticos. A pesar que tiene cierta antigüedad se sigue encontrando aplicaciones para este algoritmo (`https://doi.org/10.1088/1742-6596/2279/1/012005`). La idea es recorrer todos los vértices del grafo en cierto orden asignado un color distinto al de sus vecinos 1 (`https://en.wikipedia.org/wiki/Greedy_coloring`). Se requiere mantener los vértices v ordenados por algún criterio para hacer la operación *pop*(). Un criterio típico es ordenar los vértices en orden decreciente de grado. La selección del color ($findColor(v \rightarrow Neighbors())$) toma en cuenta un color no presente en sus vecinos ($v \rightarrow Neighbors()$) y que ojala se haya colocado antes. Por ejemplo si todos los colores ya utilizados están en $v \rightarrow Neighbors()$ entonces se utiliza uno nuevo. Claramente no garantiza una cantidad mínima de colores. La cantidad de colores que se obtienen es a lo más el grado máximo Δ más 1 para el nuevo color. Este algoritmo tiene complejidad a $O(N\Delta)$ en el peor caso.

Algorithm 1 Greedy Coloring

Require: G Grafo
  while $!(G \rightarrow isEmpty())$ **do**
    $v = G \rightarrow pop()$
    $c = findColor(v- > Neighbors())$
    $v- > setColor(c)$
  end while

Algoritmo DSatur: Este algoritmo basado en una heurística golosa se deriva del anterior cambiando el ordenamiento en G. El nuevo ordenamiento es en base al llamado grado de saturación de los nodos, por lo que el nombre del algoritmo proviene del ingles *Degree of Saturation*. Es un número que se vuelve a calcular después de colorear cada nodo. Por esta razón el algoritmo tiene un orden de $O(N^2)$, ya que tiene que revisar todos los nodos restantes en el peor caso. El grado de saturación de un vértice corresponde al numero de colores ya asignados a los vecinos, notar que es distinto al grado porque pueden haber vecinos sin color asignado. Para el caso del primer

vértice, cuando aún no se colorea nada se considera el de mayor grado. Entonces el algoritmo prosigue igual que el algoritmo anterior, solamente que se actualiza el grado de saturación de los vecinos de un nodo cada vez que asigna un color. Hay que notar que este algoritmo sigue siendo heurístico y no garantiza que se encuentre la menor cantidad de colores posible. De todas formas hay mucho que se puede hacer para mejorar el tiempo que se ejecuta este algoritmo. En vez de recorrer en forma secuencial para encontrar el mayor grado de saturación se puede utilizar una estructura de datos que almacene los vértices de forma ordenada lo que se puede hacer en algún tipo de Heap o árbol balanceado. La actualización del grado por vértice se podría mejorar disponiendo de acceso rápido a los vecinos dado que en cada vértice se mantiene el grado de saturación.

Algoritmo RLF (Recursive Largest First): Este algoritmo selecciona subconjuntos de vértices que va a colorear con un mismo color (`https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm`). La selección de estos conjuntos se hace con la siguiente observación. En la figura 1 se ve por ejemplo para los vértices coloreados en rojo están todos separados entre sí, en caso que hubiese alguna conexión deberían colorearse con colores distintos. Al conjunto de vértices rojos se le denomina conjunto de vértices independientes ya que no comparten conexiones y en este caso particular corresponde al máximo tamaño entre todos los conjuntos independientes. Esta heurística va seleccionando conjuntos independientes máximos que no hayan sido coloreados y les asigna un mismo color disponible a todos sus vértices. El problema de seleccionar el mayor conjunto independiente es también un problema NP-Hard, por lo que dicho conjunto debería obtenerse también de forma heurística (`https://en.wikipedia.org/wiki/Independent_set_(graph_theory)`). El algoritmo es el siguiente:

Algorithm 2 Algoritmo RLF

Require: G contiene todos los vértices
  $S = \phi$
  while $!(G \rightarrow isEmpty())$ **do**
    $S = G- > getMaximalIndepentSet()$
    $S- > colorAllNode()$
    $G- > delete(S)$
  end while

Este algoritmo se ha encontrado a que supera en rendimiento a Dsatur, en el sentido de encontrar mejores mínimos (`https://doi.org/10.1007%2F978-3-030-81054-2`). Sin embargo es mas lento ya que su complejidad computacional es $O(N^3)$. Mucho del tiempo utilizado es debido al método $G- > getMaximalIndependentSet()$ que estima un conjunto máximo independiente. Para ello se puede utilizar: Este algoritmo selecciona en este caso

Algorithm 3 Algoritmo cómputo conjunto independiente

Require: G contiene todos los vértices
  $S = \phi$
  while $!(G \rightarrow isEmpty())$ **do**
    $v = G- > popMinDegreeVertex()$
    $S- > push(v)$
    $G- > remove(v)$
    $G- > removeAll(v- > Neighbors())$
  end while
  Return S

el vértice de grado mínimo ($G- > popMinDegreeVertex()$) en cada paso. Entonces elimina dicho vértice y todos sus vecinos del grafo.

Algoritmo Ramificación y Acotamiento (Branch and Bound: B&B): Recientemente se han desarrollado algoritmos basados en el conocido método de (`https://en.wikipedia.org/wiki/Branch_and_bound`) ramificación y acotamiento (B&B) que es básicamente se generan nodos utilizando una división y conquista que incluye una cota inferior (LB) y superior (UB) (`"https://optimization-online.org/wp-content/uploads/2015/10/5159.pdf"`). Un nodo con cota superior igual a la inferior es una posible solución. En el peor caso se trata de una búsqueda sistemática sobre todas la posibilidades de coloreo. La diferencia es que los nodos cuya cota inferior es mayor que la mejor solución actual se descartan (proceso de poda) evitando realizar una búsqueda exhaustiva. Entonces se debe tratar de podar lo más que se pueda el árbol y seleccionar con la mejor heurística posible el menor nodo a analizar. El algoritmo es el siguiente:

Este algoritmo está basado en un procedimiento heurístico para seleccionar un nodo para colorear ($G0- > pop()$). Dicho procedimiento puede utilizar las heurísticas de selección de DSatur. El orden de los colores a testear

Algorithm 4 Algoritmo B&B

Require: G contiene todos los vértices ▷ GC contiene vértices coloreados ▷ $G0$ contiene vértices no coloreados

```
procedure BnB(GC, G0)
    if G0 == φ then                                          ▷ Caso base: No quedan nodos por colorear
        if GC− > cantidadColores() < OptActual then          ▷ Si la cantidad de colores es mejor optimo
            OptActual = GC− > cantidadColores()
            return GC                                        ▷ retornamos el nuevo optimo
        end if
        return φ                                             ▷ Retornamos nullptr porque es peor
    else
        LB = calcularLB(GC, G0)
        if LB > OptActual then                               ▷ Si cota inferior es peor que valor óptimo Actual
            return φ                                         ▷ Retornamos nullptr (Poda)
        end if
        v = G0− > pop()                                      ▷ Seleccionamos algún vértice usando alguna heurística
        S = φ
        for c ∈ GC− > coloresDisponibles(v) do ▷ Buscamos asignar colores disponibles a v para generar nodos
            v0 = v− > colorear(c)
            e =BnB(GC ∪ {v0}, G0 \ {v0})                     ▷ Llamamos recursivamente con el nuevo color
            if e ≠ φ then
                S = e                                        ▷ guardamos la última mejor solución mejor
            end if
        end for
        return S
    end if
end procedure


                                                             ▷ Programa principal
GC = φ
G0 = G
Ggreedy = G− > obtenerGreedy()
OptActual = Ggreedy− > cantidadColores()
e =BnB(GC,G0)
if e ≠ φ then
    return e
end if
return Ggreedy
```

($GC- > coloresDisponibles(v)$) se puede modificar e influye en el tipo de recorrido. Aunque en esta versión del algoritmo se obliga a seguir un recorrido en profundidad debido a la recursión, pero de esta forma se logra actualizar el valor del óptimo que se lleva hasta el momento ($OptActual$).

4 Implementación

Se requiere implementar un programa que resuelva VCP con la heurística de Branch and Bound. Se DEBE modificar modificar dicho algoritmo de forma de aproximar lo mejor posible el mínimo de colores necesarios pero cuidando que se mantenga como un algoritmo B&B. Se entregarán grafos de ejemplo de gran tamaño por lo que el programa debe ser implementado de la forma más eficiente posible. Debe cuidar no tener clases con excesivas responsabilidades. Por ejemplo no se puede tener una clase que realice todo el algoritmo. Debe separar la responsabilidad de tener objetos dedicados a B&B y otros dedicados a la Heurísticas. Otro temas relativo a la orientación al objeto: No deben haber funciones sueltas, solo clases y funciones main para test y programa principal. Para ello será importante la forma en que se representará **el estado del sistema** y **las operaciones** a realizar. Es importante que el programa final deberá resolver en forma eficiente el problema. **En esta tarea DEBE utilizar librerías de STL que implementen tipos de datos eficientes**.

Debe incluir una interfaz de usuario (menú) simple que permita indicar el nombre del archivo que contiene el grafo con la configuración inicial.

El programa recibe como entrada la descripción del grafo como un archivo de texto. Este es un archivo que contiene dos columnas de números enteros que indican nodo de inicio y nodo de fin. Por ejemplo:

```
0 1
1 2
2 3
3 0
```

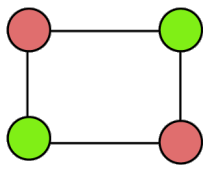

Figure 3: Grafo asociado al archivo anterior, $\chi = 2$

La salida a entregar es imprimir la lista de colores por vértice y el número total de colores. De esta forma, dos columnas de numero de vértice y color:

```
0 0
1 1
2 0
3 1
Total colores: 2
Tiempo: 0.1 [ms]
```

Además el entregable debe contener:

- Un makefile que compile programa principal y programas de test con compilación separada.
- Un test por cada clase generada. Debe comprobar que efectivamente la clase funcione.
- Cada clase son 2 archivos (.cpp) y (.h), que debe ser compilados en forma separada.
- Un main.cpp con un menú para seleccionar archivo de entrada. Se resuelve el problema entregando los colores por vértice y el tiempo que demoró en resolverse en milisegundos. Se repite en un loop que incluye un salir.
- pdf de informe
- varios ejemplos de entrada que demuestren que su programa funciona.

El programa a construir debe estar codificado en C++, implementando las funcionalidades de la manera mas eficiente posible. Debe entregar una carpeta comprimida cuyo nombre corresponda a ApellidoNombre.zip (u otra compresión). Esta carpeta contiene archivos Header (.h), clases (.cpp), programas principales (main.cpp y test_XX.cpp), al menos un test por clase, un makefile con compilación separada. Toda clase debe ser implementada por separado en dos archivos (.h y .cpp). El nombre de una clase siempre comienza en Mayúscula y el archivo que la implementa tiene su nombre. No se permite definir funciones fuera de las clases y cada clase debe servir a un sólo propósito o abstracción.

1. (10%) Informe : contiene una descripción de la estrategia de resolución para encontrar la menor cantidad de colores, el porqué su implementación es eficiente y que explique como se utiliza su programa o cuidados que hay que tener.

2. (30%) Código : se revisa si compila (0 pts si no compila), si se implementa el algoritmo propuesto de tipo B&B, si tiene la estructura indicada (clases+test+main+makefile), si se efectúa compilación separada, si está todo descrito por clases y no hay funciones sueltas salvo la función main, si el makefile opera bien, si el código se entiende (comentarios claros y útiles, variables con nombres descriptivos, identación, sin repeticiones forzadas), si se encuentra 1 test adecuado por clase, si no tiene problemas de memoria, Se revisa que no existan errores de lógica y/o ejecución, que entregue los resultados correctos en todos los casos (si no ejecuta en ningún caso 0 pts). **Si tiene funciones fuera de las clases tiene 0 puntos.** Se revisa que haya una clara separación de responsabilidad entre clases (0% si no hay separación, 20% si hay algún grado de separación de responsabilidades en el algoritmo)

3. (50%) Programa Eficienciente (0 pts si no funciona o no tiene ningún intento de hacerlo eficiente.): Se revisa que se haya implementado el problema de manera eficiente (20% Heuristica, 30% eficiencia de implementación (0pts si no hay heurística y estructuras de datos eficientes).

4. (10%) Revisión por pares. Cada compañero revisa el código a otro con una nota. La asignación es aleatorea, puede que alguien revise su propio código. Se revisará dicha revisión con otra nota. Si la revisión tiene defectos notables, la nota anterior se reemplaza por otra nota indicada por el profesor. La nota final es la nota propuesta por un par (si es que está correcta) promediada con la evaluación de la revisión que realizó el alumno.

5. (+ 1pto) Bonus al programa que genera la solución con el menor χ estimado (número mínimo de colores) posible pero que ejecute en un tiempo razonable. Si hay varios con la menor cantidad de colores, entonces se selecciona al programa mas rápido.

5 Entrega

02 de Diciembre a las 11:55PM entrega que incluye lo anterior. 1 punto por día de atraso considerando entrega hora/fecha posterior a la indicada. 06 de Octubre entrega de la revisión de a pares. **El archivo entregable es una carpeta con nombre-rut del alumno y comprimida en un archivo con nombre-rut con los archivos requeridos.**

vía Classroom