Universidad Nacional de Córdoba

FaMAF

Matemática Discreta II

Proyecto: Primera Parte

Esteban L. Boucher

eboucher7@gmail.com

Septiembre 2016

Prof. Daniel Penazzi

Índice general

1 Introducción

1.1 Motivación del proyecto

El problema de coloreo de grafos consiste en asignar colores a ciertos elementos de un grafo sujetos a ciertas restricciones.

El problema de coloreo de vértices es el problema más común de coloreo de grafos. El problema es, dados m colores, encontrar una forma de colorear los vértices de un grafo de tal manera que no hayan dos vértices adyacentes utilizando el mismo color.

El interés acerca de este problema se debe a su gran número de aplicaciones. Entre las más conocidas de coloreo de grafos se destacan[1]:

- **Establecer horarios y cronogramas**

  Supongamos que queremos hacer un cronograma de exámenes para un universidad. Listamos diferentes asignaturas y estudiantes matriculados en cada asignatura. Muchas asignaturas tendrán estudiantes en común. *¿Cómo se puede organizar el cronograma de modo que no hayan dos exámenes con un estudiante en común estén programadas al mismo tiempo? ¿Cuántas ranuras de tiempo mínimo son necesarios para programar todos los exámenes?* Este problema se puede representar como un grafo en el que cada vértice es una asignatura y una arista entre dos vértices significa que hay un estudiante común. Así que este es un problema de coloreo de grafos, donde el número mínimo de intervalos de tiempo es igual al número cromático del grafo.

- **Asignación de frecuencias de radio móvil**

  Cuando las frecuencias se asignan a las torres, las frecuencias asignadas a todas las torres en el mismo lugar debe ser diferente. Cómo asignar frecuencias con esta restricción? Lo que se necesita un número mínimo de frecuencias? Este problema es también un ejemplo de problema de coloración gráfica donde cada torre representa un vértice y un borde entre dos torres representa que están en el rango de la otra.

- **Sudoku**

  Suduku es también una variación del problema de coloreo de grafos en el cual cada celda representa un vértice. Hay una arista entre dos vértices si están en la misma fila o la misma columna o del mismo bloque.

- **Asignación de registros**

  En optimización de compiladores, la asignación de registros es el proceso de asignar un gran número de variables del programa de destino a un número reducido de registros de la CPU. Este problema también un problema de coloreo de grafos.

- **Grafos bipartitos**

  Podemos comprobar si un grafo es bipartito o no coloreando el grafo utilizando dos colores. Si un grafo dado es 2-coloreable, entonces es bipartito, de lo contrario no lo es.

- **Mapa de coloreo**

  Mapas gráficos donde dos ciudades adyacentes no pueden ser asignadas con el mismo color. Cuatro colores son suficientes para colorear cualquier mapa[2].

El siguiente proyecto consiste en la resolución de ese problema.

1.2 Propósito y objetivos

Actualmente no se conoce un algoritmo de tiempo polinomial que resuelva el problema de coloreo de grafos. Sin embargo, hay una cierta calidad mínima que se puede obtener. Supongamos que d es el mayor grado de cualquier vértice en nuestro grafo. A medida que avanzamos en el coloreo, cuando coloreamos cualquier vértice particular v, está unido a lo sumo con otros d vértices, de los cuales algunos pueden ya estar coloreados. Luego, hay a lo sumo d colores que hay que evitar usar. Usamos el color de menor número no prohibido. Esto significa que usamos colores numerados d+1 o menor, dado que al menos uno de los colores 1, 2, ..., d+1 NO está prohibido. De esta manera, nunca necesitamos usar ningún color de mayor número que d+1. Esto nos da el siguiente teorema:

Teorema (Coloreo Greedy). *Si d es el mayor de los grados de los vértices en un grafo G, entonces G tiene una coloración adecuada con d + 1 colores o menos, por vía intranasal, el número cromático de G es como máximo d + 1.*

Esto nos da una cota superior del número cromático del grafo[4].

El objetivo de este proyecto consiste en cargar un grafo y dar un coloreo propio de sus vértices, corriendo repetidamente Greedy usando órdenes que cumplan con el enunciado mencionado anteriormente.

1.3 Restricciones

- **Tiempo**

  El programa debe ser razonablemente rápido. En particular, con todos los grafos de ejemplo probados por la cátedra, el programa debe terminar en menos de una hora. Algunos de esos grafos pueden llegar a tener cerca de 2 millones de vértices y 10 millones de aristas.

- **Memoria**

  El programa nunca debe usar másde 256 MB de memoria RAM.

2 Formato del archivo de entrada

El formato de entrada será una variación de DIMACS, un formato estandard para representar grafos. La descripción oficial de DIMACS es como sigue:

1. Ninguna linea tiene mas de 80 caracteres.
2. Al principio hay cero o más líneas que empiezan con 'c' (sin las comillas), las cuales son líneas de comentario y son ignoradas.
3. Luego hay una linea de la forma: p edge n m
   donde n y m son dos enteros. n representa el número de vértices y m el número de lados.
4. Luego siguen m lineas todas comenzando con 'e' (sin las comillas) y dos enteros, representando un lado. Luego de esas m lineas se detiene la carga.

3 Estructura de la implementación

En la Figura 1 se muestra un diagrama de la estructuración del programa.

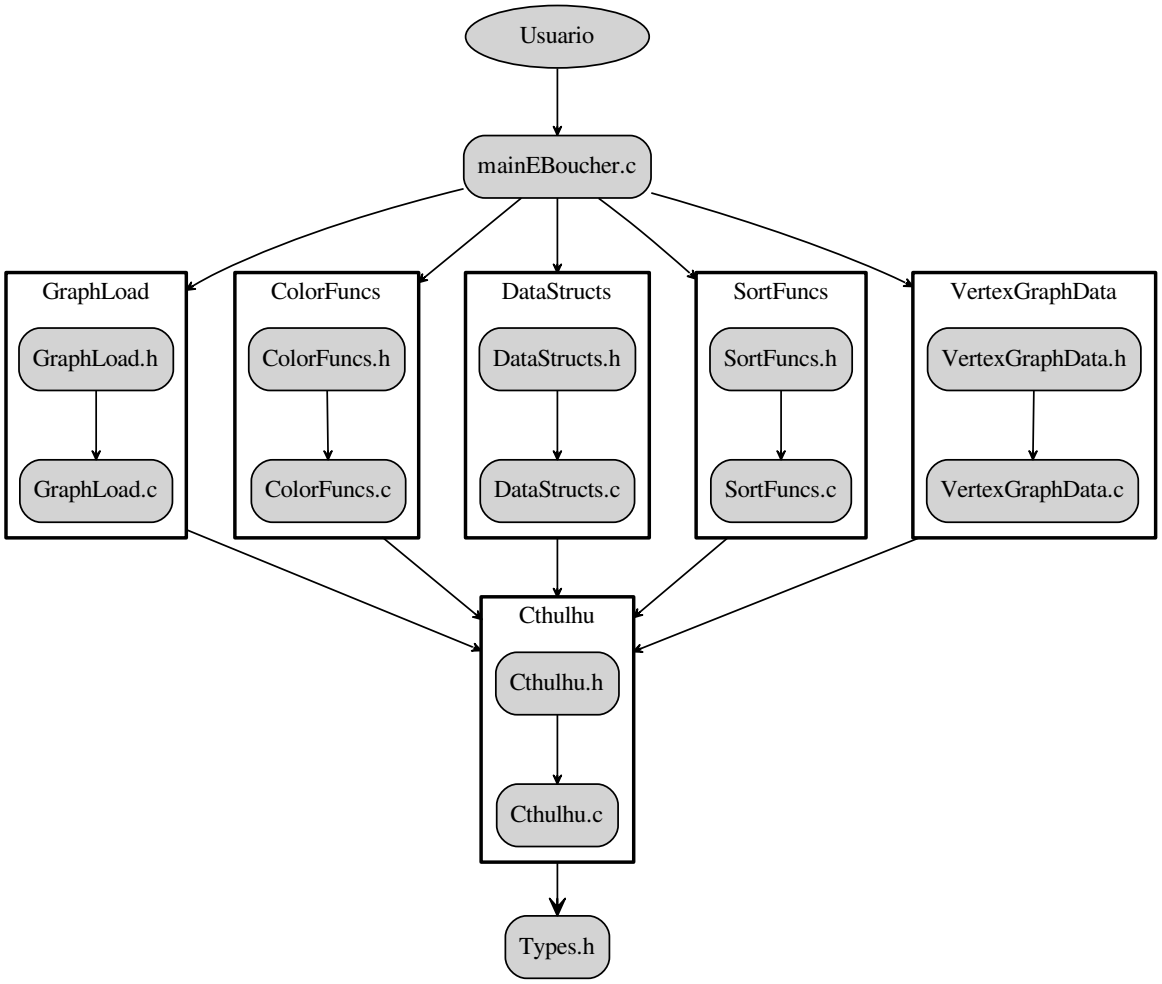

Figure 1:

3.1 Descripción de estructuras y variables

3.1.1 `struct _NimheSt_t`

Las estructura principal de este programa es `struct _NimheSt_t`, la cual contiene toda la información necesaria para correr el algoritmo de Greedy y obtener información necesaria para modificar el órden de los vértices acorde a las distintas funciones de ordenación.

- **Información básica del grafo**

  Esta estructura contiene la cantidad de vértices y lados del grafo cargado y la cantidad de colores que está usando en el momento para colorear los vértices que contiene.

- **Información de los vértices del grafo**

  En esta implementación la información que define a los vértices está dispersada en distintos arreglos dentro de la estructura del grafo: un `name_array` que contiene los nombres "reales" de los vértices, los cuales en esta implementación son enteros sin signo de 32 bits; un `color_array` que guarda el coloreo actual de los vértices, un `degree_array` para los grados de los vértices, un `RAR_order_array` para guardar el órden en el cual los vértices deben colorearse si se usa `ReordenAleatorioRestringido`. Los últimos datos acerca de los vértices son las listas de vecinos, estas están en `neighbors_array`, un arreglo de vectores en los cuales se guardan los índices de los vecinos de un vértice

- **Información de órdenes de los vértices**

  Se utiliza principalmente un arreglo `order`, que se utiliza para dar el coloreo cuando se corre *Greedy* sobre el grafo. Los arreglos `natural_order` y `vertices_with_color` se utilizan para guardar el *órden natural* y *la cantidad de colores coloreados con el color con el que está coloreado el vértice en la i-ésima posición del arreglo*, respectivamente. Se asignan los valores de los últimos dos arreglos al arreglo `order` cuando se llama a la función de ordenación correspondiente. Más sobre las funciones de ordenación adelante.

- **Información de control**

  Se utiliza un arreglo de booleanos `used` para verificar si algún *ítem* está siendo usado. Utilizo el término impreciso *ítem* justamente porque la semántica de este arreglo cambia en distintos contextos. Esta es sin dudas una práctica que por motivos razonables no se promueve.

  Sin embargo para el propósito de este proyecto resulta ser beneficioso, dado que si no se contara con esta variable compartida:

  1. Las tres funciones que con esta implementación utilizan el arreglo compartido, deberían crear el arreglo por separado y alocar memoria para la cantidad de elementos necesarios, lo cual lentifica el programa ya que dichas funciones se ejecutan repetidas veces y en ocasiones requiriendo arreglos muy grandes.

  2. Aumenta el tamaño máximo de memoria RAM residente ocupado por el programa en ejecución, dado que la memoria de los otros arreglos no se estaría liberando. Si así fuera, el código sería más complicado, y no resolvería el problema 1.

  Esto se puede evitar con la variable compartida, dado que en los tres casos, el arreglo requerido es del mismo tamaño - igual a la cantidad de vértices del grafo - y el tipo de datos del arreglo en los tres casos es el mismo, de tipo booleano, más adelante se explicarán sus utilizaciones por separado.

Una característica que se mencionó sin ser explicada es la de guardar índices de vértices, en lugar de estructuras, punteros a estructuras `_VerticeSt_t` o los *nombres reales* de los vértices vecinos. Esto se debe a la elección de diseño de esta implementación, el la cuál los vértices o sus nombres "reales" sólo sean de importancia durante la carga del grafo. Luego todas las operaciones se realizan en torno a los arreglos de ordenes o de la información de los vértices sin sus nombres. De esta manera los vértices son referidos luego de la carga directamente a través de sus indices o *identificadores*.

3.1.2 `struct _VectorSt_t`

Estructura de datos para representar un arreglo dinámico. Se utiliza en la estructura del grafo para guardar los *identificadores* de los vecinos de un vértice.

3.1.3 `struct _QueueSt_t`

Estructura de datos que implementa una cola circular, dentro de sus miembros tiene un arreglo de *elements* y dos enteros sin signo indicando los índices al primero y el último elemento de la cola.

3.1.4 `struct _VerticeSt_t`

Estructura de datos no utilizada en la implementación del proyecto. Puede ser utilizada a través de las funciones de verificación de `struct VertexGraphData.{c,h}` con modificaciones en la función `main()`

3.2 Estructura general

Debido al tamaño del proyecto, esta implentación se presenta en forma de distintos módulos, cada uno agrupando funcionalidades de manera que mantengan cierta independencia y faciliten la legibilidad. Notar sin embargo que los tipos abstractos de datos implementados no son respetados como tales dentro de las funciones de distintos archivos, dado que todos pertenecen a la misma API y por conveniencia se accede a los tipos abstractos de datos como a cualquier estructura.

3.2.1 `Types.h`

3.2.2 `Cthulhu.{c, h}`

3.2.3 `GraphLoad.{c, h}`

3.2.4 `DataStructs.{c, h}`

3.2.5 `SortFuncs.{c, h}`

3.2.6 `ColorFuncs.{c, h}`

3.3 Descripción de algoritmos

4 Indicaciones de error

5 Correctitud

6 Instrucciones de operación

6.1 Mecanografiado de compilación

Para compilar el proyecto:

```
gcc -Wall -Wextra -O3 -std=c99 -Iapifiles dirmain/mainEBoucher.c apifiles/*.c
-o EB
```

La ejecución se puede hacer de las dos siguientes maneras:

- Carga mediante un archivo:

  ```
  ./EB <[/ruta/al/archivo/nombre_archivo]
  ```

- Carga manual:

  ```
  ./EB
  ```

  e ingresando el grafo manualmente.

6.2 Descripción de las pruebas

Las pruebas consisten en el monitoreo de la ejecución del programa cuando se corre con un input dado y el control de los resultados de coloreo. Los estimadores de interés para el monitoreo de la ejecución son el tiempo transcurrido que tardó en ejecutarse, y el tamaño máximo del conjunto residente de memoria asignada.

En cuanto al control de los resultados de coloreo, basta con ejecutar la API junto con el *mainEBoucher.c* provisto. Este imprime el coloreo obtenido de 10 iteraciones iniciales con *órden aleatorio*, luego una iteración en *Órden Welsh-Powell* y de no haberse encontrado el número cromático del grafo, corre *Greedy* 1001 veces con los distintos órdenes implementados y luego imprime el mejor coloreo de esas corridas. También corre el algoritmo *2-color* al principio para verificar si el grafo dado es 2-coloreable, y luego de las 11 primeras corridas, verifica si es 3-coloreable antes de correr *Greedy* 1001 veces.

6.3 Mecanografiado de las pruebas

Existe una gran variedad de herramientas para probar el tiempo y memoria consumidos por el programa. Menciono únicamente dos de ellas:

1. **Comando *htop***

Utilizar *htop* mientras se ejecuta el programa permite visualizar tanto la RAM (columna *RES*) que está siendo usada por el proceso, y el tiempo (columna *TIME+*).

2. **Comando *time***

El comando *time* corre el programa y da un resúmen de la utilización de recursos.
Una manera de usar este comando puede ser:

```
command time -v ./EB <[/ruta/al/archivo/nombre_archivo]
```

Usar *command* fuerza al shell a ejecutar *time*, ignorando cualquier función del mismo nombre.

Opción *-v:* Usa el formato detallado, que muestra cada pieza de información disponible sobre el uso de recursos del programa con una descripción en inglés de su significado.

7 Desafíos y elecciones de diseño

7.1 Problema: vértices pueden ser cualquier u32

7.2 Vecinos de un vértice

7.3 Orden de los vértices

7.4 Greedy()

7.5 Funciones de ordenación

7.5.1 Revierte()

7.5.2 GrandeChico() y ChicoGrande()

8 Posibles mejoras

9 Comentarios

10 Referencias

[1] http://www.geeksforgeeks.org/graph-coloring-applications/

[2] https://en.wikipedia.org/wiki/Four_color_theorem

[3] https://proofwiki.org/wiki/Definition:Proper_Coloring

[4] http://web.math.princeton.edu/math_alive/5/Notes2.pdf