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 programados al mismo tiempo? ¿Cuántas ranuras de tiempo mínimo son necesarias 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. 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 se asignan frecuencias a las torres, las frecuencias asignadas a todas las torres que se encuentran en el mismo lugar debe ser diferentes. ¿Cómo asignar frecuencias con esta restricción? ¿Cuál es el mínimo numero de frecuencias necesarias? Este problema es también un ejemplo del problema de coloreo de grafos donde cada torre representa un vértice y una arista entre dos torres representa que están en el rango de la otra.

- **Sudoku**

  El 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 el 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 de países o provincias donde dos ciudades adyacentes no pueden ser asignadas con el mismo color. Cuatro colores son suficientes para colorear cualquier mapa.[2]

El siguiente programa 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, este 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:

1.2.1 Coloreo Greedy

Teorema 1 (Coloreo Greedy). *Si d es el mayor de los grados de los vértices en un grafo G, entonces G tiene una coloración propia con d + 1 colores o menos, es decir, 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 ordenes que cumplan con el Teorema 1.

1.3 Restricciones

- **Tiempo**

  El presente programa está estipulado para ser razonablemente rápido. En particular, con todos los grafos de ejemplo probados por la cátedra[5],[6] se espera que termine 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 más de 256 MB de memoria RAM.

2 Formato del archivo de entrada

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

1. Ninguna línea tiene más 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 línea 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 líneas todas comenzando con 'e' (sin las comillas) y dos enteros, representando un lado. Luego de esas m lineas se detiene la carga.

3 Estructura del programa

En la Figura 1 se muestra un diagrama de la estructuración del programa en los distintos archivos.

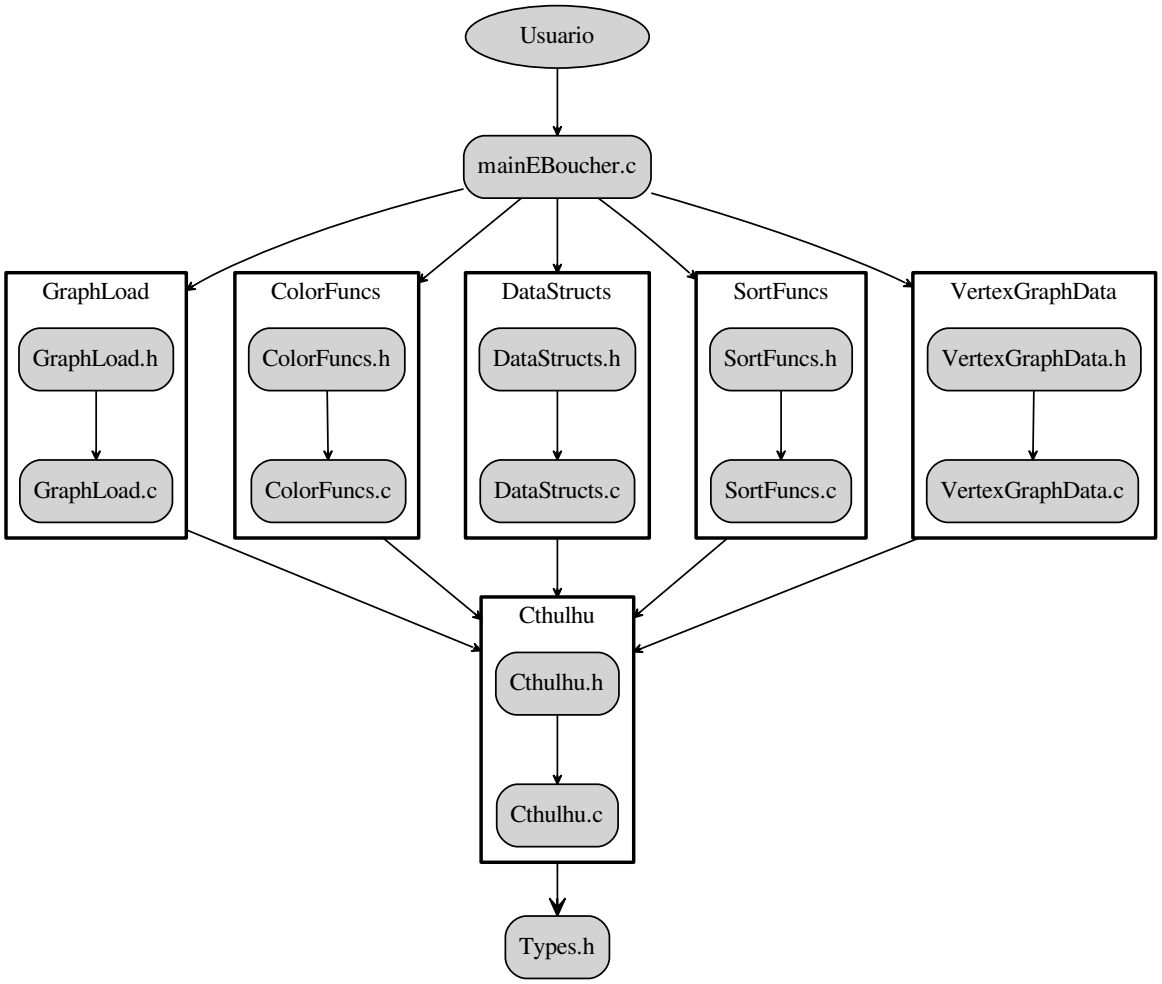

Figura 1: Diagrama de archivos del programa

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 del grafo necesaria para correr el algoritmo de Greedy y obtener del mismo para modificar el orden de los vértices acorde a las distintas funciones de ordenación.

- **Información básica del grafo**

  `NimheSt` 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 orden 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 ordenes 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 *orden 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* porque la semántica de este arreglo cambia en las distintas funciones que lo utilizan. Más acerca de esto en la sección 5.5

  Una característica que se mencionó sin ser explicada es la de guardar los índices de los vértices en la lista de vecinos de `NimheSt`. Esto es una elección de diseño de esta implementación, el la cuál los vértices o sus nombres "reales" sólo sean de importancia durante la carga del grafo y en la ordenación por orden natural. 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 índices o *identificadores*. Más acerca de esta elección en la sección 5.6

3.1.2 `struct _VectorSt_t`

Estructura de datos para representar un arreglo dinámico.[7] 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,[8] 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. Implementada para el testeo 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 implementación se presenta en forma de distintos módulos, cada uno agrupando funcionalidades con el objetivo de mantener cierta independencia y facilitar la legibilidad. Notar sin embargo que los tipos abstractos de datos implementados no son respetados como tales dentro de las funciones de los distintos archivos, dado que todos pertenecen a la misma API y por conveniencia se accede a los tipos abstractos de datos como si fuesen cualquier estructura y no necesariamente a través de la interfaz que provean.

3.2.1 APIfiles

3.2.1.1 Types.h

Este módulo hace las llamadas a las librerías generales necesarias en el programa, define las estructuras de datos descritas en la sección 3.1. Contiene el tipo de datos `u32` usando el tipo `uint32_t`. El motivo de esta definición es garantizar que `u32` sea un *entero de 32 bits sin signo*, independientemente del compilador en el que se compila el programa. También define los macros `LINE_LENGHT` para usar durante la carga del grafo desde un archivo, y los macros `VECTOR_INITIAL_CAPACITY` y `VECTOR_GROWTH_RATE` para manejar los incrementos en el tamaño de las estructuras `Vector`.

3.2.1.2 Cthulhu.{c, h}

El módulo `Cthulhu` contiene los llamados a todos los demás módulos del programa, de manera que puedan ser usados desde `mainEEboucher.c` incluyendo únicamente a `Cthulhu.h`.

`Cthulhu` define las funciones de construcción y destrucción del grafo `NuevoNimhe()`:

- **NuevoNimhe()**

  Prototipo de función:

  ```
  NimheP NuevoNimhe();
  ```

  La función aloca memoria, inicializa las variables de `NimheSt` y devuelve un puntero a ésta. Lee un grafo desde standard input en el formato especificado el la sección 2 y llena la estructura con esos datos.

  Ademas de cargar el grafo, asigna el "color" 0 a cada vertice para indicar que están todos no coloreados.

  En caso de error, la función devuelve un puntero a `NULL`. (errores posibles pueden ser falla en alocar memoria, o que el formato de entrada no sea válido, por ejemplo, que la primera linea sin comentarios no empiece con p o que n* $\neq$ n (ver en el ítem de formato de entrada que significa esto)

- **DestruirNimhe()**

  Prototipo de función:

  ```
  int DestruirNimhe(NimheP G);
  ```

  Destruye G y libera la memoria alocada. Retorna 1 si todo anduvo bien y 0 si no.

3.2.1.3 GraphLoad.{c, h}

Este módulo contiene las funciones auxiliares que se utilizan para la carga de vértices y aristas en el grafo. Las funciones son las siguientes:

- **insert_edge()**

  Prototipo de la función:

  ```
  void insert_edge(NimheP G, u32 fst_vertex, u32 snd_vertex, bool *v_loaded);
  ```

  Inserta una arista en el grafo copiando los dos vértices que la forman. Para hacer esto busca los índices o *identificadores* de `fst_vertex` y `snd_vertex` llamando a la función `find_vertex_hash()` y luego agrega el identificador de cada uno en la lista del otro.

- **find_vertex_hash()**

  Prototipo de la función:

  ```
  u32 find_vertex_hash(NimheP G, u32 vertex, bool *v_loaded);
  ```

  Calcula el *código hash* de `vertex` y ubica a `vertex` en el grafo `G` usando el código `hash`. Hace un sondeo lineal o *linear probing* para chequear el siguiente elemento en el grafo si `vertex` no fue encontrado en el `hash_code` en caso que `vertex` no se encuentre en `G`, la función le asigna la primera celda disponible que encuentra.

- **hash_code()**

  Prototipo de la función:

  ```
  u32 hash_code(u32 key, u32 size);
  ```

  La función define un método de hasheo o *hashing method* para computar el código hash de `key` en el rango de 0 a `size` - 1. Para esto se utiliza el método de avalancha definido en la función de hash *MurmurHash*. Más sobre la elección de esta función de hash en la sección 5.7

3.2.1.4 VertexGraphData.{c, h}

Funciones para extraer información de datos de un vértice:

- **ColorDelVertice()**

  Prototipo de Función:

  ```
  u32 ColorDelVertice(VerticeSt x);
  ```

  Devuelve el color con el que está coloreado el vértice `x`. Si el vértice no esta coloreado, devuelve 0.

- **GradoDelVertice()**

  Prototipo de Función:

  ```
  u32 GradoDelVertice(VerticeSt x);
  ```

  Devuelve el grado del vértice `x`.

- **NombreDelVertice()**

  Prototipo de Función:

  ```
  u32 NombreDelVertice(VerticeSt x);
  ```

  Devuelve el nombre real del vértice `x`.

- **ImprimirVecinosDelVertice()**

  Prototipo de Función:

  `void ImprimirVecinosDelVertice(VerticeSt x,NimheP G);`

  Imprime en standard output una lista de los vecinos del vértice `x`.

Funciones para extraer información de datos del grafo:

- **NumeroDeVertices()**

  Prototipo de función:

  `u32 NumeroDeVertices(NimheP G);`

  Devuelve el número de vértices del grafo G.

- **NumeroDeLados()**

  Prototipo de función:

  `u32 NumeroDeLados(NimheP G);`

  Devuelve el número de lados del grafo G.

- **NumeroDeVerticesDeColor()**

  Prototipo de función:

  `u32 NumeroVerticesDeColor(NimheP G,u32 i);`

  Retorna el número de vértices de color i. (si i = 0 devuelve el número de vertices no coloreados).

- **ImprimirVerticesDeColor()**

  Prototipo de función:

  `u32 ImprimirVerticesDeColor(NimheP G,u32 i);`

  Imprime una lista de los vertices que tienen el color `i`. Si i = 0, esta representa la lista de vértices no coloreados.

  Si no hay vertices de color i imprime "No hay vertices de color i"

  Retorna el número de vértices de color i.

- **CantidadDeColores()**

  Prototipo de función:

  `u32 CantidadDeColores(NimheP G);`

  Devuelve la cantidad de colores usados en el coloreo de G.

- **IesimoVerticeEnElOrden()**

  Prototipo de función:

  `VerticeSt IesimoVerticeEnElOrden(NimheP G,u32 i);`

  Devuelve el vértice numero i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

- **IesimoVecino()**

  Prototipo de funcón:

  `VerticeSt IesimoVecino(NimheP G,VerticeSt x,u32 i);`

  Devuelve el vecino número i de `x` en el orden en que esté guardado en G, (el índice 0 indica el primer vértice, el índice 1 el segundo, etc).

3.2.1.5 DataStructs.{c, h}

Funciones de `VectorSt`:

- **vector_init()**

  Prototipo de la función:

  `void vector_init(Vector *V);`

  Inicializa una estructura `Vector`. Establece su tamaño en 0, su capacidad en `VECTOR_INITIAL_CAPACITY` y reserva memoria para un arreglo de datos de tamaño igual a `VECTOR_INITIAL_CAPACITY`.

- **vector_append()**

  Prototipo de la función:

  `void vector_append(Vector *V, u32 value);`

  Agrega `value` al vector `V`. Si el arreglo de datos está lleno, un llamado a esta función causará que `vector->data` se expanda para agregar este valor. En cualquier caso, incrementa `vector->size`.

- **vector_free()**

  Prototipo de la función:

  `void vector_free(Vector *V);`

  Libera la memoria alocada para el arreglo de datos. Observación: esta función no libera la estructura `Vector`, esto se deja para el *nodo cliente*, que en este caso es el modulo Cthulhu, en la función de `DestruirNimhe()`.

Funciones de `QueueSt`:

- **Queue_init()**

  Prototipo de la función:

  `Queue* Queue_init(u32 max_elems);`

  Toma como argumento el máximo número de elementos que la cola puede contener. Crea una `Queue` del tamaño recibido en la entrada y retorna un puntero a la misma.

- **Dequeue()**

  Prototipo de la función:

  `void Dequeue(Queue *Q);`

  Saca el primer elemento de la cola.

- **Queue_front()**

  Prototipo de la función:

  `u32 Queue_front(Queue *Q);`

  Retorna el primer elemento de la cola.

- **Enqueue()**

  Prototipo de la función:

  `void Enqueue(Queue *Q, u32 element);`

  Inserta un elemento al final de la cola.

- **Queue_is_empty()**

  Prototipo de la función:

  `bool Queue_is_empty(Queue *Q);`

  Retorna `true` si la cola está vacía y `false` si hay elementos en ella.

- **Queue_free()**

  Prototipo de la función:

  `void Queue_free(Queue *Q);`

  Libera la memoria alocada para el arreglo de los elementos.

3.2.1.6 SortFuncs.{c, h}

- **shuffle()**

  Prototipo de la función:

  `void shuffle(u32 *array, u32 n, u32 seed);`

  Mezcla los elementos de `array` usando la semilla `seed` para generar un orden aleatorio.

- **OrdenNatural()**

  Prototipo de la función:

  `void OrdenNatural(NimheP G);`

  Ordena los vértices del grafo en orden creciente de sus nombres "reales".

- **OrdenWelshPowell()**

  Prototipo de la función:

  `void OrdenWelshPowell(NimheP G);`

  Ordena los vertices de G de acuerdo con el orden Welsh-Powell, es decir, con los grados en orden no creciente.

- **ReordenAleatorioRestringido()**

  Prototipo de la función:

  `void ReordenAleatorioRestringido(NimheP G);`

  Si G está coloreado con r colores y W1 son los vértices coloreados con 1, W2 los coloreados con 2, etc, entonces esta función ordena los vértices poniendo primero los vértices de Wj1 (en algún orden) luego los de Wj2 (en algún orden), etc, donde j1, j2, ... son aleatorios (pero distintos).

- **GrandeChico()**

  Prototipo de la función:

  `void GrandeChico(NimheP G);`

  Si G esta coloreado con r colores y W1 son los vertices coloreados con 1, W2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de Wj1 (en algún orden)luego los de Wj2 (en algún orden), etc, donde j1, j2, ... son tales que $|Wj1| \geq |Wj2| \geq ... \geq |Wjr|$.

- **ChicoGrande()**

  Prototipo de la función:

  `void ChicoGrande(NimheP G);`

  Idem que el anterior excepto que ahora el orden es tal que $|Wj1| \leq |Wj2| \leq ... \leq |Wjr|$

- **Revierte()**

  Prototipo de la función:

  `void Revierte(NimheP G);`

  Si el grafo está coloreado con r colores y W1 son los vértices coloreados con 1, W2 los coloreados con 2, etc, entonces esta función ordena los vértices poniendo primero los vértices de Wr (en algún orden) luego los de Wr-1 (en algún orden), etc.

- **OrdenEspecifico()**

  Prototipo de la función:

  `void OrdenEspecifico(NimheP G, u32* x);`

  Si x cumple la condición, entonces la función ordena los vértices con OrdenNatural(), luego lee el orden dado en el string x y los ordena de acuerdo a lo que lee. Es decir, si luego de OrdenNatural, los vértices quedaron ordenados como, por ejemplo, V[0],V[1],...,V[n-1], luego de OrdenEspecifico el orden debe ser V[x[0]],V[x[1]],...,V[x[n-1]].

3.2.1.7 ColorFuncs.{c, h}

- **Chidos()**

  Prototipo de función:

  `int Chidos(NimheP G);`

  Devuelve 1 si G es bipartito, 0 si no.

- **Greedy()**

  Prototipo de función:

  `u32 Greedy(NimheP G);`

  Corre greedy en G con el orden interno indicado en la estructura `NimheSt` de G. Devuelve el número de colores que se obtiene.

3.2.2 Dirmain

3.2.2.1 mainEBoucher.c

Este módulo contiene a la función principal para probar el programa, `main()`. Tiene solamente las llamadas a `stdio.h`, `stdlib.h` y `Cthulhu.h`.

Define auxiliarmente el prototipo de la función `time()` para generar números aleatorios, y una función de comparación de dos valores que se usa dentro de la función `main()`. Además define una función `shuffle_f()`, la misma que shuffle() en SortFuncs. Se define nuevamente en este módulo para mantener la independencia del programa `main()` con respecto a los APIfiles.

Se usa esta interfaz para cargar leer un grafo desde *standard input*, donde la entrada es como se describe en la sección 2.

Función main():

Prueba el programa de la siguiente manera:

1. Carga el grafo. Si el formato de entrada esta mal imprime una línea que dice "Error en formato de entrada" y para. (por ejemplo, si hay una linea que empieza con "d", o si la primera linea que no empieza con "c" no empieza con "p", o si el número de vertices que se extrae de los lados no es n, etc).

2. Si χ (G) = 2 imprime en standard output "Grafo Bipartito" y para.

3. Si χ(G) >2 imprime en una línea "Grafo No Bipartito" y luego crea 10 ordenes aleatorios más Welsh-Powell y corre Greedy con cada uno de esos ordenes imprimiendo cuántos colores obtiene en cada caso.

En el caso de las corridas aleatorias cada linea será de la forma "coloreo aleatorio numero k: r colores" donde r será el número de colores que obtuvieron y k el número de corrida aleatoria.

En el caso del orden WelshPowell la línea será "coloreo con Greedy en WelshPowell:r colores" Además, antes de esa línea habrá una línea en blanco separando de las lineas anteriores de las corridas aleatorias.

4. Si llegó a este paso es porque el grafo no es bipartito, por lo tanto si alguno de esos coloreos es con tres colores, ya se sabe que χ(G) = 3 así que imprime una línea que diciendo "χ(G) = 3" y para.

5. Si no se obtuvieron 3 colores con ninguno de los coloreos, se imprime una línea en blanco, luego una línea que dice:

====Comenzando Greedy Iterado 1001 veces====

y luego otra línea en blanco.

Luego toma el mejor coloreo de los primeros 11 anteriores y corre Greedy 1000 veces cambiando el orden de los colores, siguiendo el siguiente patrón:

- 50% ChicoGrande
- 12,5% GrandeChico
- 31,25% Revierte
- 6,25% ReordenAleatorioRestringido

(nota: esos porcentajes son 8/16, 2/16, 5/16 y 1/16 respectivamente).

Luego de estas 1000 iteraciones, hace una iteración final con Revierte, e imprime:

Mejor coloreo con Greedy iterado 1001 veces: k colores (a CG,b GC, c R, d RAR)

donde k es el menor número de colores que hayan obtenido en las 1001 iteraciones, y a,b,c,d son la cantidad de veces que corrieron Chico Grande,GrandeChico, Revierte y ReordenAleatorioRestringido, respectivamente.

3.3 Descripción de algoritmos

3.3.1 Algoritmo 2-color

Un grafo G=(N, E) es **bipartito** si sus vértices se pueden separar en dos conjuntos disjuntos U y V, es decir, tal que se cumple:

- $U \cup V = N$
- $U \cap V = \emptyset$

de manera que las aristas sólo pueden conectar vértices de un conjunto con vértices del otro, es decir:

- $\forall\ u_1,u_2 \in U$, $\forall\ v_1,v_2 \in V$ $\forall\ u_1,u_2 \in U$,$\forall\ v_1,v_2 \in V$ no existe ninguna arista $e=(u_1,u_2)$ $e=(u_1,u_2)$ ni $e=(v_1,v_2)$ $e=(v_1,v_2)$

Un grafo bipartito es posible si el coloreo del grafo se puede hacer usando dos colores de tal manera que vértices de un conjunto estén coloreados con el mismo color.

El algoritmo utilizado presenta una mínima variación al algoritmo dado en clase: dentro del ciclo para recorrer los vecinos de un vértice se termina la función si se encuentra que uno de sus vecinos ya estaba coloreado con el mismo color. Esto evita la tarea de tener que revisar todos los lados del grafo y verificar si dos vecinos tienen el mismo color:

Algoritmo 1 2-color

```
Descolorear todos los vértices si tienen algún color.
j ← 0
while j <n do
    elegir x ∈ V no coloreado
    coloreo C(x) ← 1
    j ← j + 1
    crear cola Q con x como su único elemento
    while Q ≠ ∅ do
        p ← 1er elemento de Q
        remuevo p de Q
        for w ∈ Γ(p) do
            if w no está coloreado then
                incluir w en Q
                colorear C(w) ← 3 - C(p)
                j ← j + 1
            end
            if w tiene el mismo color que p then
                return "no es 2-coloreable"
            end
        end
    end
end
return "sí es 2-coloreable"
```

3.3.2 Algoritmo Greedy de coloreo de vértices

(Se utiliza la notación vista en clase)

"Si el orden de los vértices es v_1, v_2, ... , v_n:
$C(v_1) = 1$
Para $k = 2, 3, \dots, N$:
$C(v_k) = \min\{j \in \{1, 2, \dots, N\}: C(v_i) \neq j \ \ \forall\ i \leq k - 1$ tal que $v_i v_k \in E\}$"

Esta implementación del algoritmo utiliza los arreglos de la estructura de `NimheSt used` para guardar los colores *"prohibidos"* en cada iteración, `color_array` donde guarda el color de los vértices.

Comienza estableciendo los valores de esos arreglos en 0, (*false* para el arreglo de colores usados). Luego el algoritmo es el siguiente:

Algoritmo 2 Greedy

```
Reestablecer valores de used en false
Reestablecer valores de color_array en 0
Colorear C(x_{1ro en el orden}) ← 1
coloreo ← coloreo + 1
Definir neighbor_color, i, j
for x ∈ G - {x_{1ro en orden}} do
    for y ∈ ListaDeVecinos(x) do
        if y no está coloreado then
            used[i_y] ← true
        end
    end
    for i ∈ CantVertices(G) do
        if used[i] false then
            Salir
        end
    end
    Colorear C(y) ← i
    Asignar coloreo ← max(coloreo, i)
end
for y ∈ ListaDeVecinos(x) do
    if y no está coloreado then
        used[i_y] ← false
    end
end
    return coloreo
```

Además, en la implementación se utiliza el arreglo `vertices_with_color` para guardar en la i-ésima posición, i $\leq$ *coloreo*, la cantidad de vértices coloreados con el color i. No es utilizado para ejecutar el algoritmo, solo se actualiza cada vez que se asigna un nuevo color a un vértice. Su utilidad está en las funciones de ordenación. Más sobre esto en la sección 5.4.2.

4 Instrucciones de operación

4.1 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.

4.2 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 *orden aleatorio*, luego una iteración en *orden Welsh-Powell* y de no haberse encontrado el número cromático del grafo, corre *Greedy* 1001 veces con los distintos ordenes 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.

4.3 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:

- **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+*).

- **Comando `time`**

  El comando *time* corre el programa y da un resumen de la utilización de recursos.

  Se puede usar este comando como sigue:

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

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

  `-v`: Opcional para obtener un formato detallado, mostrando cada pieza de información disponible sobre el uso de recursos del programa con una descripción en inglés de su significado.

5 Desafíos y elecciones de diseño

5.1 Problema: vértices pueden ser cualquier u32

Esta característica impone la restricción de no poder usar el nombre real de un vértice como índice en el arreglo. Para solucionar este problema, el programa implementa un arreglo como una *hash table*. La hash table utiliza una función hash para insertar elementos en el arreglo y resuelve las colisiones usando sondeo lineal, como se describe en la sección 3.2.1.3.

5.2 Vecinos de un vértice

La implementación de los vecinos de un vértice también fue un desafío dado que durante la carga del grafo, la cantidad de vecinos que tiene un vértice es desconocida. Se trata este problema usando la estructura `VectorSt` con un arreglo dinámico, alocando más memoria a medida que se llena el arreglo.

5.3 Orden de los vértices

El problema de implementar el orden de los vértices se debe a la inconveniencia de reordenar los vértices en cada iteración, por lo que resultó necesario considerar otra forma de hacerlo. Para resolver el problema del orden de los vértices, consideramos en primer lugar el estado de los vértices dentro de la estructura del arreglo: cada vértice está asociado a un solo índice, su *identificador*. Se hace uso de este dato y se implementa el orden de los vértices por medio de otro arreglo en el que la `i`-ésima posición tiene el identificador del `i`-ésimo vértice en ese orden.

5.4 Funciones de ordenación

Todas las funciones de ordenación menos OrdenEspecifico fueron implementadas utilizando la función estándar `qsort()` y definiendo funciones de comparación locales para pasar a qsort() como parámetro, de manera que en lo que difieren las funciones entre ellas es en la función de comparación definida para pasar a qsort().

5.4.1 Revierte()

La función `Revierte()` utiliza para su función de comparación el arreglo `color_array` para comparar el número de color. De modo que si el número de color de un vértice es mayor que el de otro vértice, este se ordena primero que el vértice con el que se compara.

Se pasa esta función de comparación a la función qsort() para que ordene el arreglo `order` de la manera indicada.

5.4.2 GrandeChico() y ChicoGrande()

Estas funciones resultan un poco más complejas, dado que para definir sus funciones de comparación es necesario obtener la información para comparar desde un nivel más que en las demás funciones. Es decir, el primer nivel es obtener el color del vértice, y el segundo nivel obtener la cantidad de vértices coloreados con ese color.

En ambos casos, para solucionar el problema se agregó en la estructura `NimheSt` un arreglo en el que se guarde la cantidad de vértices coloreados con cada color, `vertices_with_color`. Este arreglo se actualiza a medida que se colorean los vértices en la función Greedy(), evitando así aumentar la complejidad en tiempo que representaría, por ejemplo, recorrer el grafo entero cada vez que se comparan dos vértices para determinar cual se ordena primera.

Tener una implementación razonablemente eficiente de ellas resulta fundamental dado que son ejecutadas repetidas veces en la prueba del programa. Las dos funciones se implementan de manera similar, sólo difieren en qué vértice se ordena primero en la función de comparación.

5.5 Arreglo de usados

Utilizar variables cuya semántica cambia en distintos contextos 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.

5.6 Identificadores de vértices

La elección de guardar los índices de los vértices en la lista de vecinos de `NimheSt` en lugar de punteros a estructuras se debe al uso más eficiente de la memoria que se obtiene al no utilizarlos. Además se puede observar una mejora en la velocidad del programa.

5.7 Elección de función *hash*

Al momento de elegir una función de hash para la carga de vértices en el grafo, tuve en consideración distintas funciones hash, las cuales diferían en aspectos como rendimiento y simplicidad.

La función de hash *MurmurHash* es una buena función de hash para propósitos generales, y provee tanto simplicidad como una gran resistencia a colisiones, por lo cual fue una buena elección para este programa.[9]

6 Posibles mejoras

Si bien los resultados del tiempo en ejecución de este programa en computadoras con un buen procesador cumplen con la cota de una hora impuesta como máximo, este todavía sigue sin ser extremadamente rápido.

Una optimización que se realizó fue la de quitar las estructuras de vértices `VerticeSt` de `NimheSt` y esparcir la información que los define en distintos arreglos, de manera que sea más directo operar sobre cada característica de los vértices y también deshacerse de una enorme cantidad de punteros.

Si bien se pudo observar una gran mejora en la utilización de memoria RAM, la optimización no se reflejó en el tiempo de ejecución.

Para intentar mejorar este último punto, el programa fue sometido a una última modificación mayor, que consistió en reemplazar la utilización de la función qsort() por una implementación de *radix sort* adaptada al propósito. Esta última modificación estuvo lejos de producir los resultados deseados pese al entusiasmo. En una búsqueda *al vuelo* de alguna función de ordenación más eficiente que qsort() me encontré con radix sort, que parecía podría funcionar y los cálculos parecían indicar que por los 32 bits que se utilizan para representar a los vértices y su información, radix sort andaría significativamente más rápido.

Esto no fue así, lo que me lleva a considerar que hice mal los cálculos o que no implementé bien la función de radix sort. Ninguno de estos supuestos pude comprobar, y llegué al límite del plazo para la entrega del proyecto, por lo que averiguarlo queda pendiente.

7 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

[5] http://www.famaf.unc.edu.ar/ penazzi/DisII2016/GrafosDeEjemplos.html

[6] http://www.famaf.unc.edu.ar/ penazzi/DisII2016/Corrdeotrapagina/GrafosOtros.html

[7] https://gist.github.com/EmilHernvall/953968

[8] http://www.thelearningpoint.net/computer-science/data-structures-queues–with-c-program-source-code

[9] http://stackoverflow.com/questions/11899616/murmurhash-what-is-it