Matemática Discreta II

Alegre, Jorge Facundo

8 de junio de 2016

Presentación - Proyecto, primera parte.

Índice

1. Introducción

El siguiente documento detalla la implementación de un programa escrito en C para encontrar una cantidad de colores que alcanzan para colorear de manera propia un grafo con el algoritmo Greedy de coloreo. Vale aclarar que el programa **no indica cómo colorear el grafo**, solo indica cuántos colores se pueden usar para colorearlo de forma propia.

En el documento encontrará una explicación de cómo compilar y ejecutar el programa, la estructura elegida para el código, los tipos de datos creados y las decisiones de diseño tomadas a lo largo de su desarrollo. Por último, puede encontrar respuestas a preguntas comunes que podrían surgir.

El programa recibe un grafo en formato DIMACS (con algunas modificaciones). Después de verificar que el formato de entrada se respete, el programa procede a verificar que $\chi(grafo) \neq 2$. Si es así, se crean 10 órdenes aleatorios para los vértices del grafo y se corre el algoritmo Greedy sucesivamente. Luego, se ordena el grafo de forma decreciente con respecto al grado de cada vértice (orden Welsh Powell) y se vuelve a correr Greedy. Si alguno de estos 11 órdenes colorea el grafo con 3 colores (i.e., $\chi(grafo) = 3$) la ejecución del programa se detiene.

En caso contrario, se vuelve a ordenar el grafo con el orden que produjo el mejor coloreo de las 11 iteraciones. Luego, comienzan 1001 iteraciones de ordenar y colorear el grafo. En estas iteraciones, el grafo se ordenará según sus colores. Un teorema de coloreo de grafos prueba que al colorear un grafo con Greedy, si luego se lo ordena según sus colores, volver a colorearlo con Greedy dará un coloreo con una cantidad de colores menor o igual que el coloreo anterior. Los órdenes son Revierte, ChicoGrande, GrandeChico y ReordenAleatorioRestringido. Serán explicados con mayor detalle. Estos órdenes son elegidos al azar, pero con distintas probabilidades de ser elegidos, para las primeras 1000 iteraciones. La última siempre será Revierte.

Finalmente, se imprime por `STDOUT` el mejor coloreo obtenido y la cantidad de veces que se ordenó con cada orden.

2. Como compilar y usar

Correr el siguiente comando, situados en el directorio `JorgeAlegre/`:

```
$ gcc -Wall -Wextra -03 -std=c99 -Iapifiles
   dirmain/mainJAlegre.c apifiles/*.c -o dirmain/color
```

Para correr el programa, simplemente lo ejecutamos de la siguiente forma:

```
$ dirmain/color
```

En este momento, podemos ingresar los datos del grafo en el formato especificado. El programa solo recibe datos por `STDIN`.

Si poseemos un grafo en un archivo, se lo puede cargar directamente de la siguiente forma:

```
$ dirmain/color < archivo_de_grafo
```

Luego de cargar un grafo, el programa puede devolver un error ocasionado por un grafo de entrada que no respeta el formato especificado o puede devolver el resultado de su corrida por `STDOUT`.

3. Estructura del código

3.1. `apifiles/`

La mayor parte del código se encuentra ubicada en el archivo `Cthulhu.c`. Esto es así porque la mayoría de las funciones implementadas necesita conocer la estructura interna del grafo, la cual está declarada al comienzo de este archivo.

3.1.1. `Cthulhu.c`

Posee la implementación de los tipos de datos `NimheSt` y `neighbours_t`. Contiene la implementación de todas las funciones declaradas en `Cthulhu.h`, que serán usadas en el **main()** del programa.

Además, posee la implementación de todas las funciones relacionadas con el TAD `neighbours_t`, las cuales están detalladas en la sección **Tipos de datos**.

También se encuentra la implementación de todas las funciones de comparación usadas para ordenar el grafo. Para ordenar el grafo, usaremos una función de C llamada `qsort()`, que es importada de la librería `<stdlib.c>`. Esta recibe, entre otras cosas, el arreglo a ordenar y la función usada para comparar los elementos del arreglo.

3.1.2. `Cthulhu.h`

Posee la implementación del tipo de datos `VerticeSt`. Se encuentra en este archivo (i.e., su estructura es pública) para que un tercero como el **main()** pueda llamar a la función `IesimoVerticeEnElOrden(NimheP G, u32 i)`, la cual devuelve un vértice.

3.1.3. `helpers.c`

Posee la implementación de las funciones usadas por la función `NuevoNimhe()` (i.e., las funciones usadas para leer los datos del grafo por `STDIN`). Estas son:

- `readline_from_stdin()`

  Devuelve un *string* con la línea leída. En caso de no haber más líneas para leer o si una línea no respeta el formato de entrada (e.g., línea tiene más de 80 caracteres) devuelve `NULL`. Se la llama dentro de un ciclo para procesar lo que el usuario ingresa.

- `handle_p_line()`

  Si la línea empieza con la letra **p**, verificamos que sea de la forma *p edge n m* donde n y m son número de vértices y números de lados respectivamente, y devolvemos n y m encapsuladas en una estructura `par_t`.

- `handle_e_line()`

  Si la línea empieza con la letra **e**, verificamos que sea de la forma *e v w* donde **v** y **w** son los nombres de los vértices que forman parte de la arista **vw**. Al igual que antes, devolvemos los datos encapsulados en un `par_t`.

- `strstrip()` encargada de remover el `\n` del final de la línea leída
- `strcopy()` encargada de hacer copias de texto
- `get_l()` y `get_r()`, para obtener ambos elementos de la estructura `par_t`

3.1.4. `rbtree.c`

Posee la implementación de la API de los árboles Rojo-Negro, además de funciones auxiliares necesarias. La especificación de esta API se encuentra en `rbtree.h`. Las funciones públicas son:

- `rb_new()`, encargada de pedir memoria para el árbol nuevo
- `rb_insert()`, para insertar nodos nuevos
- `rb_exists()`, para saber si un elemento existe
- `rb_search()`, para obtener el valor asociado a un elemento
- `rb_destroy()`, para destruir el árbol y liberar su memoria

3.1.5. `u32queue.c`

Posee la implementación de la API de la cola de números **u32**. La especificación de esta API se encuentra en `u32queue.h`. Las funciones públicas son:

- `queue_empty()`, encargada de pedir memoria para la cola nueva
- `queue_destroy()`, para destruir la cola y liberar su memoria
- `queue_is_empty()`, para saber si hay elementos en la cola
- `queue_enqueue()`, para agregar elementos
- `queue_dequeue()`, para eliminar el primer elemento
- `queue_first()`, para leer los datos del primer elemento

3.1.6. `typedefs.h`

Dentro está declarado el tipo de datos **u32**.

3.2. `dirmain/`

3.2.1. `mainJAlegre.c`

Dentro de este directorio existe un solo archivo, `mainJAlegre.c`. En él esta implementado el **main()** de nuestro programa.

Se encarga de:

- cargar el grafo y alertarle al usuario si se respeta el formato de entrada
- contar cuántas veces se ordena con cada orden
- verificar que $\chi(G) > 3$ con la función `Greedy(G)`
- destruir el grafo con la función `DestruirNimhe(G)`
- generar los 10 órdenes aleatorios y obtener el coloreo con cada uno

  Para esto, creamos un arreglo de tamaño n con los valores del *0* hasta $n-1$. Luego, recorremos el arreglo *swappeando* la posición actual con una posición aleatoria. A este arreglo se lo pasamos a la función `OrdenEspecifico(G, orden)`.

  Siempre tendremos 2 arreglos, uno con el orden usado en el momento y otro con el orden que generó el mejor coloreo hasta el momento. Solo reordenamos el del momento. Esto nos sirve para empezar las 1001 iteraciones con el mejor orden usado en estas 10 iteraciones, a menos que el orden Welsh Powell dé mejor resultado. Para generar números aleatorios, se llama a la función `rand()`.
- ordenar el grafo con el orden Welsh Powell y obtener el coloreo
- ordenar el grafo con los órdenes elegidos al azar y obtener el coloreo con cada uno

  Se genera un número aleatorio y, según su valor, ordenaremos el grafo con alguno de los 4 órdenes basados en los colores del grafo.
- imprimir en pantalla el resultado de las iteraciones

4. Tipos de datos

4.1. u32

El tipo de datos **u32** se usa alrededor de todo el programa. Para hacer que su uso sea fácil, se define en el archivo `apifiles/typedefs.h` y se incluye donde sea necesario. Es un alias al tipo de datos `uint32_t` incluido de la biblioteca estándar de C `<stdint.h>`.

4.2. VerticeSt

Está definido en el archivo `apifiles/Cthulhu.h`. Contiene información básica de un vértice. Lo que **no** contiene es información de otros vértices, por más que en el grafo estén relacionados.

Incluye el nombre real del vértice, el cual es leído de lo que el usuario suministra al ejecutar el programa, su grado, que se incrementa a medida que se agregan vecinos nuevos, y su color, usado para ordenar el grafo.

La estructura de este tipo es pública para que se puedan crear o manipular vértices en el archivo `dirmain/mainJAlegre.c`.

4.3. NimheSt - NimheP

La estructura (NimheSt) está definida en `apifiles/Cthulhu.c` y un puntero a la estructura (NimheP) en `apifiles/Cthulhu.h`. Se usa un puntero para encapsular la información. Contiene los datos esenciales de un grafo, como:

- la cantidad de lados
- la cantidad de vértices
- el $\Delta(grafo)$
- un arreglo con todos los vértices cargados
- un arreglo con la posición de los vértices en orden
- un arreglo de la estructura de vecinos para cada uno de los vértices *(el elemento en la posición **i** de este arreglo corresponde con el vértice en la posición **i** del arreglo de vértices)*
- la cantidad de colores usados para colorearlo
- un arreglo donde la posición indica el color y su valor, la cantidad de vértices coloreados con ese color.

4.4. neighbours_t

Es una estructura definida dentro de `apifiles/Cthulhu.c` ya que las funciones que manipulan esta estructura necesitan conocer la estructura del grafo. Su único objetivo es reunir dos arreglos en una sola estructura. Se usa para representar a los vecinos de los vértices. Su relación con los vértices es la posición en el arreglo (i.e., un vértice en la posición **i** tiene a sus vecinos en el arreglo de vecinos en la misma posición).

El arreglo `neighbours` contiene la posición de cada uno de los vértices vecinos del vértice en cuestión.

El arreglo `colors` es un *bool vector* usado para colorear al vértice. En este arreglo, la posición i corresponde al color $i + 1$.

Las funciones implementadas para manipular este tipo son:

- `neighbours_t neighbours_empty()`

  Crea la estructura usada por este tipo de datos. Devuelve un puntero a la estructura.

- `neighbours_t neighbours_init(neighbours_t neighbours, u32 size)`

  Crea el arreglo `colors`. Recibe un puntero a la estructura y un **u32** indicando la cantidad de vecinos del vértice. Devuelve un puntero a la estructura.

- `neighbours_t neighbours_destroy(neighbours_t neighbours)`

  Libera toda la memoria usada por la estructura. Recibe un puntero a la estructura. Devuelve `NULL`.

- `u32 neighbours_i(neighbours_t neighbours, u32 i)`

  Recibe un valor entre 0 y el *grado del vértice - 1*. Devuelve la posición del iésimo vecino del vértice.

- `neighbours_t neighbours_append(neighbours_t neighbours, u32 index)`

  Agrega la posición de un vecino nuevo del vértice. Recibe un puntero a la estructura y un **u32** indicando la posición de un vecino nuevo del vértice. Devuelve un puntero a la estrucutra.

- `u32 neighbours_find_hole(neighbours_t neighbours, u32 grado)`

  Recibe un puntero a la estructura. Devuelve el color más chico no usado por los vecinos de un vértice. Se supone que se llama a `neighbours_update()` antes de correr esta función. Recorre el arreglo `colors` y devuelve la posición del primer `false` +1.

- `neighbours_t neighbours_update(NimheP G, u32 vertex)`

  Actualiza el *bool vector* `colors` marcando los colores usados por los vecinos del vértice; es decir, marca con `true` a la celda en la posición *color del vecino - 1*. Solo tiene en cuenta colores menores al grado del vértice y distintos a 0 (vértices coloreados), el resto no importa.

5. Decisiones de diseño

5.1. `rand()` y `srand()`

Para obtener números aleatorios, se llama a la función `rand()` de la librería estandar de C. Como semilla se usa $(\#lados(grafo) * \#vértices(grafo)) + nombre(2^o vértice)$.

5.2. `realloc()` para agregar vecinos

Para agregar los vecinos de un vértice, llamamos a la función `neighbours_append()`. Al no saber la cantidad de vecinos que tendrá un vértice a la hora de crearlo, no se puede saber el tamaño del arreglo de los vecinos del vértice. Entonces, se usa un factor constante para ir agrandando su tamaño en *runtime*, si al agregar un vecino nuevo el arreglo ya tiene todas sus posiciones ocupadas. Esta constante, definida en `apifiles/Cthulhu.c` vale 8. Esto desperdicia memoria pero mejora el rendimiento del programa. A medida que este factor crece, se gana velocidad pero se pierde eficiencia espacial.

6. Respuestas a preguntas particulares

6.1. Dado que los vértices pueden ser cualquier u32, ¿cómo sabemos si a un vértice recién leído ya lo habíamos leído antes?

Al leer los vértices suministrados por el usuario, queremos saber si es un vértice nuevo para alocar espacio para contenerlo o si simplemente es un vértice existente y se quiere aclarar que tiene como vecino al vértice que lo acompaña. Dado que los vértices pueden ser cualquier número entre 0 y $2^{32} - 1$, resolver este problema no es trivial si se desea velocidad alta y uso de memoria bajo para cargar el grafo.

El método usado en esta implementación implica usar árboles Rojo-Negro. Con esta estructura de datos, podemos lograr una eficiencia de espacio de $\mathcal{O}(n)$, donde n es la cantidad de vértices en el grafo. Las operaciones sobre el árbol usadas por el programa son las de insertar elementos nuevos y buscar elementos, ambas con complejidad de tiempo de $\mathcal{O}(\ln n)$.

6.2. ¿Cómo está implementado $\Gamma(x)$?

El grafo, en su estructura interna, posee un arreglo de los vértices y un arreglo de arreglos de los índices (en el arreglo de los vértices) de cada uno de los vecinos de los vértices.

Por ejemplo, si en el índice *3* del arreglo de los vértices se ubica el vértice *35*, en el índice *3* del arreglo de los vecinos se ubica una estructura del tipo **neighbours_t**, que encapsula información de los vecinos del vértice *35*, incluido un arreglo con los índices de los vértices vecinos del vértice *35*. Esta información se obtiene mediante la función `neighbours_i()`.

6.3. ¿Cómo está implementado el orden de los vértices?

Dentro de la estructura del grafo existe un arreglo de números de tipo **u32** de tamaño n. Este arreglo, en cada posición, contiene los índices de los vértices en el arreglo de vértices. Las posiciones indican el orden. Por ejemplo, supongamos que el arreglo con el orden se llama `orden` y el arreglo de vértices se llama `vertices`. Entonces,

```
vertices[orden[0]]
```

equivale al primer vértice del grafo en el orden actual. Por otro lado,

```
vertices[0]
```

equivale al primer vértice del grafo en el orden que el programa leyó los vértices.

El arreglo de vértices nunca cambia de lugar sus elementos. El arreglo con el orden se modifica con todas las funciones para ordenar el grafo.

Desde que se carga un grafo hasta que el programa termina, el primer vértice en el arreglo de vértices es siempre el mismo. El valor en el primer campo del arreglo con el orden no siempre será igual.

6.4. ¿Cómo está implementada la función `Greedy()`?

Primero, le quita el color a todos los vértices. Esto es necesario para que `neighbours_update()` no tenga en cuenta colores de corridas viejas. También limpiamos el arreglo `nvertices_color`, dejando todos sus campos con el número 0.

Luego, coloreamos el primer vértice con el color 1.

Después, en un ciclo de *1...n* (el primer vértice ya está coloreado), obtenemos la posición de los vértices en orden. Actualizamos su estructura de vecinos asociada, es decir, recorremos todos los vecinos del vértice en cuestión, viendo sus colores y ajustando el arreglo `colors`. Este arreglo tiene una cantidad de celdas igual al grado del vértice. Esto es porque si un vértice tiene 24 vecinos, y uno de esos vecinos está coloreado con el color 28, no afecta en el color que seleccionaré para colorear al vértice. En el peor caso, todos los vecinos del vértice estan coloreados con colores distintos, crecientes a partir del color 1. En el ejemplo, supongamos que la posición del vértice en cuestión es 5,

```
G->vertices[5].grado = 24
G->vertices[neighbours_i(G->vecinos[5], 0)].color = 1
G->vertices[neighbours_i(G->vecinos[5], 1)].color = 2
.
.
.
G->vertices[neighbours_i(G->vecinos[5], 23)].color = 24
```

Entonces, luego de actualizar el arreglo de colores con la función `neighbours_update()`, el arreglo de colores nos queda así:

```
colors[0] = true
colors[1] = true
.
.
.
colors[23] = true
```

La función `neighbours_find_hole()`, que devuelve el color más chico no usado, recorre el arreglo de colores buscando alguna posición con el valor `false`, indicando que el color está libre para usar. En el ejemplo, esta función devolvería 25.

Con el color que devuelve la función, coloreamos al vértice y aumentamos por 1 la cantidad de vértices coloreados con ese color.

Al salir del ciclo, lo único pendiente por hacer es contar cuántos colores fueron usados para colorear el grafo. Para esto, recorremos el arreglo `nvertices_color` hasta que el valor en algún campo sea 0.

Esta función podría ser más rapida ya que, para cada vértice, estamos recorriendo todos sus vecinos, no solo los vecinos que fueron coloreados antes de él. No produce un coloreo impropio porque limpiamos los colores del grafo antes de arrancar. Con la implementación elegida para el grafo, tener en cuenta los vértices ya coloreados y revisar solo los vecinos de cada vértice ya coloreados costaría más que hacerlo de esta manera.

6.5. ¿Cómo están implementados los órdenes Revierte, ChicoGrande y GrandeChico?

Estos 3 órdenes están implementados de la misma manera, usando la función `qsort()`. Lo único que tienen de distinto es la función de comparación usada para cada uno.