Proyecto de programación de Matemática Discreta II-2023-Primera Parte

Contents

1 Introducción

Este proyecto de programación tiene una nota entre 0 y 10 y deben obtener al menos un 4 para aprobar.

El proyecto estará dividido en varias etapas pero se evalua globalmente.

El proyecto puede ser hecho en forma individual o en grupos de 2 o 3 personas.

1.1 Objetivos

Los objetivos en este proyecto son:

1. Implementar un tema enseñado en clase en un lenguaje, observando las dificultades de pasar de la teoria a un programa concreto.

2. Prácticar programar funciones adhiriendose a las especificaciones de las mismas.

3. Práctica de testeo de programas.

El tema concreto que usaremos este año es hacer un programa que coloree un grafo tratando de obtener un coloreo con la menor cantidad de colores posibles, o "cerca".

La idea NO ES presentar un programa único completo que haga esto, sino programar funciones auxiliares que luego se pueden ensamblar en uno o mas mains que las use, de acuerdo a distintas estrategias.

El proyecto se dividirá en varias etapas. Este primer documento da las especificaciones de las funciones de la primera etapa. En esta primera etapa, simplemente especificamos funciones que permitan leer los datos del grafo que vamos a colorear, guardarlos en alguna estructura adecuada, y otras funciones que nos permitan acceder a esos datos, como el número de vertices y lados, los nombres y grados de cada vertice, y cuales son los vecinos de un vértice.

Las etapas posteriores usaran estas funciones para hacer lo que deben hacer.

Si en las etapas posteriores uds programan sin hacer uso de estas funciones, el proyecto queda desaprobado.

Para evitar "acarreo "de errores de la primera etapa a las otras, las otras etapas las evaluaremos usando nuestras propias funciones de la 1ra etapa.

El lenguaje es C. (C99, i.e., pueden usar comentarios // u otras cosas de C99).

Es crucial que programen las funciones de acuerdo con las especificaciones, este es uno de los objetivos de este proyecto, y será dificil o directamente imposible corregir las otras etapas si usan funciones que no satisfagan las especificaciones.

1.2 Testeo

Deben testear la funcionalidad de cada una de las funciones que programan, con programas que testeen si las funciones efectivamente hacen lo que hacen o no, pero no hace falta que entreguen esos programas. Obviamente nosotros no sabremos si testearon o no, o que tanto lo hicieron, o si testearon bien o mal, pero si entregan funciones con errores que obviamente habrian saltado con un mínimo testeo, nos daremos cuenta que no testearon o no testearon adecuadamente. Pej, si la lista de vecinos de un vertice no es la correcta.

Habrá una lista de grafos de ejemplos para que testeen las funciones sobre ellos.

Programar sin errores es difícil, y algunos errores se les pueden pasar aún siendo cuidadosos y haciendo tests, porque somos humanos, y algunos errores son dificiles de detectar. Pero hay errores que no deberían quedar en el código que entreguen, porque son errores que son fácilmente detectables con un mínimo testeo.

1.3 Entrega

Deberan entregar los archivos que implementan el proyecto en la forma y tiempo en que especifiquemos mas tarde en la pagina.

Las funciones que estan descriptas en esta etapa serán usadas luego por otras funciones que especificaremos mas adelante. En esta etapa las funciones consisten basicamente en las funciones que permiten leer los datos de un grafo y cargarlos en una estructura adecuada, mas funciones que permiten acceder a esos datos.

1.4 Restricciones generales

1. No se puede incluir NINGUNA libreria externa que no sea una de las básicas de C. (eg, stdlib.h, stdio.h, strings.h, stdbool.h, assert.h etc, si, pero otras no. Especificamente, glibc NO).

2. El código debe ser razonablemente portable, aunque no es probable que testemos sobre Apple, y en general testearemos con Linux, puede ser tambien testeable desde Windows.

3. No pueden usar archivos llamados aux.c o aux.h

4. No pueden tener archivos tales que la unica diferencia en su nombre sea diferencia en la capitalización.

5. No pueden usar getline.

6. El uso de macros esta permitido pero como siempre, sean cuidadosos si los usan.

7. Pueden consultar con otros grupos, pero no pueden tener grandes fragmentos de código iguales, o con cambios meramente cosméticos en el código de otro grupo. Consultar por ideas es aceptable, copiar código en bloque no.

8. Si saben como usar ChatGPT y lo usan, esta bien, pero deberian ser extra cuidadosos con el testeo.

1.5 Formato de Entrega

Los archivos del programa deben ser todos archivos .c o .h.

Pueden entregar un sólo .c con todas las funciones si quieren, o separados en varios archivos, pero no debe haber ningun include de un .c

No debe haber ningun ejecutable en los archivos entregados.

Los .c que entreguen deben hacer un include de un archivo API llamado APIG23.h donde se declaran las funciones.

APIG23.h tiene simplemente la declaración de las funciones, la declaración del tipo de datos 2.3 definida mas abajo y un include de otro .h, EstructuraGrafo23.h . Para evitar errores de tipeo con las declaraciones de las funciones, subiremos una copia de APIG23.h a la página del Aula Virtual. EstructuraGrafo23.h en cambio será particular de cada grupo, porque dependerá de la forma particular que cada grupo decida estructurar las cosas, asi que cada uno entregará una copia distinta.

Para testear deberán (o al menos, deberían) hacer por su cuenta uno o mas .c que incluyan un main que les ayude a testear sus funciones, incluyendo funciones nuevas que ustedes quieran usar para testear cosas. (pej, luego de cargar el grafo, imprimir los vertices y sus vecinos para chequear que sus funciones cargaron bien el grafo). Estos archivos NO deben ser entregados.

Deben adjuntar un archivo ASCII donde conste el nombre, apellido y email de todos los integrantes del grupo.

Esta parte es muy simple, asi que no deberían entregar un montón de archivos complicados. Si lo estan pensando muy complicadamente, probablemente esta mal.

Algunos grupos hacen funciones que tienen prints interiores para debuggear, pero no deben entregar esas versiones. Las que entreguen deben cumplir con las especificaciones, las cuales no incluyen imprimir nada.

1.6 Compilación

Compilaremos (con mains nuestros) usando gcc, -Wall, -Wextra, -O3, -std=c99 . Tambien haremos -I al directorio donde pondremos los .h

Esas flags seran usadas para testear la velocidad, pero para testear grafos chicos podemos agregar otras flags. Por ejemplo, podemos usar -DNDEBUG si vemos que estan mal usando asserts.

Tambien compilaremos, para testear grafos chicos, con flags que nos permitan ver si hay buffer overflows, shadow variables o comportamientos indefinidos, en particular con -fsanitize=address,undefined. Su programa DEBE compilar y correr correctamente con esa flag aunque para grafos grandes lo correremos con un ejecutable compilado sin esa flag, dado que esa flag provoca una gran demora en la ejecución.

2 Tipos de datos

Los dos primeros tipos de datos deben ser definidos en el archivo EstructuraGrafo23.h.

EstructuraGrafo23.h lo tienen que definir uds de acuerdo con la estructura particular con la cual piensan guardar el grafo.

Tambien puede estar ahi cualquier declaración de funciones auxiliares que necesiten.

2.1 u32:

Se utilizará el tipo de dato `u32` para especificar un entero de 32 bits sin signo.

Todos los enteros sin signo de 32 bits que aparezcan en la implementación deberán usar este tipo de dato.

Los grafos a colorear tendran una lista de lados cuyos vertices serán todos u32.

Pueden declarar u32 como unsigned int o bien haciendo un include de int.h y declarandolo apropiadamente. u32 NO ES un long unsigned int en computadoras modernas.

2.2 GrafoSt

Es una estructura, la cual contendrá toda la información sobre el grafo necesaria para correr las funciones pedidas.

En particular, la definición interna debe contener como mínimo:

- El número de vertices.
- El número de lados.
- Los nombres y grados de todos los vertices.
- el Delta del grafo (el mayor grado).
- Quienes son los vecinos de cada vértice.

Como se verá en la sección 6, los grafos que se carguen serán no dirigidos.

La estructura debe ser racional. Usar una estructura que demande Terabytes de almacenamiento no es racional.

2.3 Grafo

es un puntero a una estructura de datos *GrafoSt*. Esto estará definido en APIG23.h .

3 Funciones De Construcción/Destrucción del grafo

3.1 ConstruirGrafo()

Prototipo de función:

```
Grafo ConstruirGrafo();
```

La función aloca memoria, inicializa lo que haya que inicializar de una estructura GrafoSt ,lee un grafo **desde standard input** en el formato indicado en la sección 6, lo carga en la estructura, y devuelve un puntero a la estructura.

En caso de error, la función devolverá un puntero a `NULL`. (errores posibles pueden ser falla en alocar memoria, pero también que el formato de entrada no sea válido. Por ejemplo, en la sección 6 se dice que en una cierta linea se indicará un número m que indica cuantos lados habrá y a continuación debe haber m lineas cada una de las cuales indica un lado. Si no hay AL MENOS m lineas luego de esa, debe retornar NULL. (si hay mas de m lineas, luego de la primera, sólo debe leer las m primeras).

Dado que esta función debe como mínimo leer todos los lados de los datos de entrada, su complejidad no puede ser inferior a $O(m)$, pero esta función NO PUEDE ser $O(n^2)$ (y MENOS puede ser $O(mn)$) pues en los grafos de testeo habrá grafos con millones de vértices, y un grafo asi con un algoritmo $O(n^2)$ no terminará en un tiempo razonable.

En cuanto a m, puede estar en el orden de millones tambien, y puede ser $m = O(n^2)$, pero sólo para n del orden de miles, mientras que cuando n sea del orden de millones, m no será $O(n^2)$ sino $O(n)$, pues como dijimos arriba esta función no puede tener complejidad menor a $O(m)$ y un m de pej miles de millones haria que demorara mucho.

Asi que deberian pensar una estructura tal que esta función sea, idealmente, $O(m)$, pero eso parece ser muy dificil, asi que $O(mlogm)$ es perfectamente aceptable.

3.2 DestruirGrafo()

Prototipo de función:

```
void DestruirGrafo(Grafo G);
```

Destruye G y libera la memoria alocada.

Esta función tambien deberia tener una complejidad razonable, no hay razón para que sea mayor a $O(m)$ e incluso puede ser menor, pero $O(m)$ es aceptable.

4 Funciones para extraer información de datos del grafo

Las funciones detalladas en esta sección y la que sigue deben ser todas $O(1)$, pues serán usadas repetidamente por las funciones de la segunda etapa y si no son $O(1)$ no podrán hacer correr las funciones en un tiempo razonable. No debería haber ningún problema con esto, basta con guardar la información en un campo adecuado en la estructura del grafo.

4.1 NumeroDeVertices()

Prototipo de función:

```
u32 NumeroDeVertices(Grafo G);
```

Devuelve el número de vértices de G.

4.2 NumeroDeLados()

Prototipo de función:

```
u32 NumeroDeLados(Grafo G);
```

Devuelve el número de lados de G.

4.3 Delta()

Prototipo de función:

`u32 Delta(Grafo G);`

Devuelve $\Delta(G)$, es decir, el mayor grado.

Esta función esta detallada aca para ser usada en algunos casos y no tener que recalcular Δ, asi que si, en vez de hacer el cálculo una vez durante la construcción del grafo y guardar el resultado para que esta función lo pueda leer en $O(1)$, lo que hacen es recalcular Δ cada vez que se llama esta función, tendrán descuento de puntos.

5 Funciones para extraer información de los vertices

En esta sección tenemos funciones que nos permitan saber el nombre y el grado de un vértice, y acceder a sus vecinos.

Las funciones detalladas en esta sección, como en la anterior deben ser $O(1)$. De hecho, es mucho mas importante que sean $O(1)$ estas, pues las anteriores probablemente sean usadas sólo una o dos veces en cada función, mientras que para colorear un vertice habrá que iterar sobre los vecinos de ese vértice repetidamente, y si la función que permite leer datos de vecinos no es $O(1)$ para cada vecino, habrá problemas de velocidad.

5.1 Problema de como acceder a los datos

Como dijimos, queremos saber el nombre, grado y vecinos de un vértice. Pero ¿cómo referenciamos el vértice? Es decir, queremos definir algo como Grado(vertice) (en realidad Grado(vertice,G) para acceder a G) pero ¿qué iría en "vertice"? Lo intuitivo sería poner el nombre del vértice ahi, pero como los nombres de los vértices son cualquier entero entre 0 y $2^{32}-1$, hay que tener cuidado. No podemos guardar una tabla interna con 2^{32} entradas, muchas de las cuales estarán vacias.

Internamente, los vértices estarán guardados de alguna forma especial, mas compacta que con sus nombres. Esta forma especial de guardarlos dependerá de cada grupo, aunque en gral los grupos usan un par de formas usuales de guardarlas.

Pero tenemos que ponernos de acuerdo con una forma estandard de nombrar los vértices, independientemente de cómo cada grupo los guarde internamente.

Lo obvio es referenciar los vértices de acuerdo al lugar que ocupen en algun orden predefinido, que sea fácilmente especificable y común a todos, es decir, hablar del vértice "i-ésimo", en algún orden que fijemos.

Hay varios posibles, nosotros tomaremos el siguiente:

DEFINICIÓN:

El **Orden Natural** de los vértices es el orden de los vértices que se obtiene al ordenarlos de MENOR a MAYOR de acuerdo con sus nombres. (Recordemos que los nombres de los vértices serán enteros sin signo de 32 bits, asi que tiene sentido hablar de ordenarlos de menor a mayor).

Pej, si los vértices son 174391,15,7,4,45,1,95980, entonces el orden natural es:

1,4,7,15,45,95980,174391.

El **indice** de cada vértice en este orden se supone que es a partir de 0, es decir, pej, el índice del vértice 174391 en este orden es 6, el índice del vértice 95980 es 5, el índice del vértice 15 es 3, el índice del vértice 1 es 0, etc.

Observemos que el Orden Natural es un orden fijo, bien definido para cualquier grafo, independiente del estado interno de la estructura que hayan decidido usar.

5.2 Nombre()

Prototipo de Función:

`u32 Nombre(u32 i,Grafo G);`

Devuelve el nombre del vértice cuyo índice es i en el Orden Natural.

El nombre es el nombre del vértice con el que figuraba en los datos de entrada.

Pej, en el ejemplo dado arribe Nombre(6,G)=174391, Nombre(2,G)=7, Nombre(0,G)=1, etc.

Dado que el nombre de un vértice puede ser cualquier entero sin signo de 32 bits, esta función no tiene forma de reportar un error (que se produciría si se la llama cuando i es mayor o igual que el número de vértices), asi que debe ser usada con cuidado.

5.3 Grado()

Prototipo de Función:

```
u32 Grado(u32 i,Grafo G);
```

Devuelve el grado del vértice cuyo índice es i en el Orden Natural , si i es menor que el número de vértices. (es decir, si es un índice permitido). Si i es mayor o igual que el número de vértices, devuelve $2^{32} - 1$. (esto nunca puede ser un grado en los grafos que testeeemos, pues para que eso fuese un grado de algún vértice, el grafo deberia tener al menos 2^{32} vertices, lo cual lo haría inmanejable).

5.4 IndiceVecino()

Prototipo de función:

```
u32 IndiceVecino(u32 j,u32 i,Grafo G);
```

Esta función nos permite acceder a los vecinos de un vértice, para poder usar SUS datos. Dado que estamos usando el índice de un vértice en el OrdenNatural para poder acceder a sus datos, lo que queremos que esta función devuelva es justamente ese índice. Es decir, esta función devolverá el índice en el Orden Natural de un vecino de un vértice si i, j estan en los rangos adecuados, y $2^{32} - 1$ si no.

Y ¿quienes son i, j?.

i, como en las funciones anteriores, denotará al vertice "i-ésimo "en el OrdenNatural.

Este vértice tendrá varios vecinos. j indica al j-ésimo de ellos.

Entonces, especificamente:

Si i es mayor o igual que el número de vértices o i es menor que el número de vértices pero j es mayor o igual que el grado del vértice cuyo índice es i en el Orden Natural entonces la función devuelve $2^{32} - 1$.

Si i es menor que el número de vértices y j es menor que el grado del vértice cuyo índice es i en el Orden Natural y el vécino j-ésimo del vértice cuyo índice es i en el Orden Natural es el vértice cuyo índice es k en el Orden Natural entonces IndiceVecino(j,i,G) es igual a k.

En esta funcion se habla del "vécino j-ésimo".

Con esto nos referimos al vértice que es el j-ésimo vécino del vértice en cuestión donde el orden del cual se habla es el orden en el que ustedes hayan guardados los vecinos de un vértice en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc.

Este orden NO ESTA ESPECIFICADO, y un grupo puede tener un orden y otro grupo otro, asi que el retorno para valores individuales de estas funciones no será el mismo para un grupo que para otro, y no será necesariamente igual al retorno de nuestras funciones.

Podriamos especificar que los vecinos deben estar guardados tambien en el orden natural, pero no es necesario para nuestros propositos, asi que les dejamos libertad para que lo hagan de la forma mas conveniente para su estructura.

Como se puede testear esta funcion entonces? Y para que sirve una función que da valores distintos dependiendo de la implementación?

Esta función existe pues para colorear necesitamos iterar sobre **todos** los vecinos, y porque si queremos testear si la estructura del grafo esta bien guardada, necesitamos una función que nos permita "ver"cómo estan guardados los vecinos en esa estructura.

Pero no es importante el orden en que estan guardados, sólo importa que podamos recorrerlos a todos, por eso no especificamos el orden de los vecinos.

IMPORTANTE: Esta función debe ser $O(1)$.

Como va a ser usada para iterar sobre todos los vecinos, probablemente dentro de un loop que ademas itere sobre todos los vértices, es **CLAVE** que esta función sea $O(1)$ pues de lo contrario nada va a poder funcionar a una velocidad razonable. Asi que la estructura que armen del grafo debe ser tal que esta función sea $O(1)$ y no tenga que hacer una iteración para ser calculada.

6 Formato de Entrada

El formato de entrada será una variación de DIMACS, que es un formato estandard para representar grafos, con algunos cambios.

- Las lineas pueden tener una **cantidad arbitraria de caracteres**. (la descripción oficial de Dimacs dice que ninguna linea tendrá mas de 80 caracteres pero hemos visto archivos DIMACS en la web que no cumplen esta especificación y usaremos algunos con lineas de mas de 80 caracteres)
- Al principio habrá cero o mas lineas que empiezan con c las cuales son lineas de comentario y deben ignorarse.
- Luego hay una linea de la forma:

  p edge n m

  donde n y m son dos enteros. Luego de m, y entre n y m, puede haber una cantidad arbitraria de espacios en blancos.

  El primero número (n) representa el número de vértices y el segundo (m) el número de lados.

  Si bien hay ejemplos en la web en donde n es en realidad solo una COTA SUPERIOR del número de vertices y m una cota superior del número de lados, todos los grafos que nosotros usaremos para testear cumplirán que n será el número de vertices exacto y m el número de lados exacto.
- Luego siguen m lineas todas comenzando con e y dos enteros, representando un lado. Es decir, lineas de la forma:

  e v w

  (luego de "w" y entre "v" y "w" puede haber una cantidad arbitraria de espacios en blanco)
- Nunca fijaremos $m = 0$, es decir, siempre habrá al menos un lado. (y por lo tanto, al menos dos vértices).
- Si bien en algunos ejemplos en algunas paginas hay vértices con grado 0, y que por lo tanto no aparecen en ningún lado en nuestros ejemplos no habrá vértices con grado 0: los únicos vértices que cuentan son los vértices que aparecen como extremos de al menos un lado.
- Luego de esas m lineas puede haber una cantidad arbitraria de lineas de cualquier formato las cuales deben ser ignoradas. Es decir, se **debe detener la carga sin leer ninguna otra linea luego de las m lineas**, aún si hay mas lineas. Estas lineas extras pueden tener una forma arbitraria, pueden o no ser comentarios, o extra lados, etc. y deben ser ignoradas.

  Pueden, por ejemplo, tener un SEGUNDO grafo, para que si la función de carga de un grafo se llama dos veces por algún programa, el programa cargue dos grafos.

  Por otro lado, el archivo puede efectivamente terminar en la última de esas lineas, y su código debe poder procesar estos archivos también.

  En un formato válido de entrada habrá al menos m lineas comenzando con e, pero puede haber algún archivo de testeo en el cual no haya al menos m lineas comenzando con e. En ese caso, como se especifica en 3.1, debe detenerse la carga y devolver un puntero a NULL. O por ejemplo tambien podremos testear con archivos

donde en vez de p edge n m tengan algo similar pero no correcto como p edgee n m, en cuyo caso tambien deben devolver NULL.

- En algunos archivos que figuran en la web, en la lista pueden aparecer tanto un lado de la forma

  e 7 9

  como el

  e 9 7

  Los grafos que usaremos nosotros **no son asi**.

  Es decir, si aparece el lado e v w NO aparecerá el lado e w v.

  Ejemplo:

  c FILE: myciel3.col

  c SOURCE: Michael Trick (trick@cmu.edu)

  c DESCRIPTION: Graph based on Mycielski transformation.

  c Triangle free (clique number 2) but increasing

  c coloring number

  p edge 11 20

  e 1 2

  e 1 4

  e 1 7

  e 1 9

  e 2 3

  e 2 6

  e 2 8

  e 3 5

  e 3 7

  e 3 10

  e 4 5

  e 4 6

  e 4 10

  e 5 8

  e 5 9

  e 6 11

  e 7 11

  e 8 11

  e 9 11

  e 10 11

- En el formato DIMACS no parece estar especificado si hay algun limite para los enteros, pero en nuestro caso los limitaremos a enteros de 32 bits sin signo.

- Observar que en el ejemplo y en muchos otros casos en la web los vertices son 1,2,...,n, PERO ESO NO SIEMPRE SERÁ ASI.

  Que un grafo tenga el vértice v no implicará que el grafo tenga el vértice v' con $v' < v$.

  Por ejemplo, los vertices pueden ser solo cinco, y ser $0, 1, 10, 15768, 1000000$.

- El orden de los lados no tiene porqué ser en orden ascendente de los vertices.

  Ejemplo Válido:

  c vertices no consecutivos

  p edge 5 3

  e 1 10

  e 0 15768

  e 1000000 1

7 Consideraciones finales para esta primera etapa

En esta etapa, la mayoria de las funciones son muy fáciles si piensan primero bien la estructura con la cual van a cargar el grafo.

Observar que no hay funciones que modifiquen la estructura interna del grafo, es decir, una vez construida la estructura, queda estática.

Las cosas mas dificiles de esta primera etapa son:

- Definir la estructura en forma adecuada para que las funciones de extracción de información sean $O(1)$.

- Programar en forma eficiente la construcción del grafo. Algunos grafos tendrán millones de vértices, por lo tanto una construcción que sea $O(n^2)$ no terminará de cargar el grafo en ningún tiempo razonable. No es necesario que sea hipereficiente, pues la construcción del grafo se hace una sola vez, mientras que la lectura de los datos múltiples veces, pero no puede ser tan ineficiente que demore horas o dias en cargar un grafo.