Proyecto de Matemática Discreta II-2021-Primera Parte

Contents

1 Introducción

El proyecto puede ser hecho en forma individual o en grupos de 2 o 3 personas. Este año, para tratar de evitar el problema de que les dabamos amplio tiempo para hacer todo el proyecto pero igual algunos grupos dejaban todo para el último momento, el proyecto se dividirá en varias etapas. Este primer documento da las especificaciones de las funciones de la primera etapa.

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

Si bien el proyecto está dividido en varias etapas, se evalua globalmente, es decir, las distintas etapas no se promedian. Esto es asi porque básicamente la forma de corregir es descontar puntos por errores, asi que aun si una etapa esta perfecta, si hay un error garrafal en la otra el descuento será substancial. De la misma forma aún si casi todas las funciones estan bien, si una de ellas es un desastre absoluto quizás no aprueben.

Las funciones de las otras etapas usarán estas funciones como funciones auxiliares.

Los objetivos en este proyecto son:

1. Implementar algo enseñado en clase en un lenguaje, observando las dificultades de pasar de algo teórico a un programa concreto.
2. Práctica en programar funciones adhiriendose a las especificaciones de las mismas.
3. Práctica de testeo de programas.

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

La idea NO ES presentar un programa completo que haga algo especifico, sino las funciones detalladas, que luego se pueden ensamblar en un main que las use.

La catedra usará sus propios mains para testear las funciones, y ustedes no tendrán acceso a ellos, por lo que deben programar las funciones detalladas siguiendo las especificaciones.

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.

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

Tengan en cuenta que uds. deben programar esto de acuerdo a las especificaciones porque no saben qué programa va a ser el que llame a sus funciones, ni cómo las va a usar.

El objetivo no es sólo programar lo indicado abajo sino tambien TESTEAR lo programado adecuadamente. Se descontarán mas puntos por errores de programación que deberían haberse detectado con un testeo razonable que por errores que pueden ser díficiles de detectar con tests.

1.1 Entrega

Deben entregar vía e-mail a daniel.penazzi arroba unc.edu.ar los archivos que implementan el proyecto.

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 manipular esos datos.

1.2 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 tengo acceso a un sistema de Apple, y en general lo testearé con Linux, puedo tambien testearlo 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 si vemos copiado de grandes fragmentos de código la van a pasar mal, especialmente si descubrimos intento de engañarnos haciendo cambios cosméticos en el código de otro grupo.
8. En años anteriores se les pedia entregar todas las funciones juntas, estas mas algunas de las que pediremos en otras etapas, lo cual hacia que los grupos pudieran usar en algunas funciones la estructura interna del grafo tal como estaba guardado, en vez de hacer un llamado a las funciones auxiliares correspondientes. Algunos grupos tomaban ventaja de esto creando una estructura interna del grafo con campos auxiliares extras que les permitia correr las otras funciones mas rapidamente, y eso estaba bien en esos años. Este año uds. podrán hacer eso con las funciones de esta etapa, pero no con las funciones de las siguientes etapas: las funciones de las siguientes etapas deberán ser programadas de forma tal de usar las funciones de esta primera etapa, llamandolas, pero no pudiendo acceder a la estructura interna del grafo.

Nos aseguraremos de esto ademas de obviamente leyendo el código, testeando las funciones de las etapas posteriores con **nuestras** funciones de esta primera etapa, no con las suyas.

Una consecuencia que tiene esto es que si detectamos en su estructura campos que no son necesarios pero que podrían ser utiles para posibles funciones extras no listadas en este documento que los quisieran acceder, supondremos que en realidad copiaron el código de alguna función similar de años anteriores. No está mal usar códigos anteriores como referencias, lo que está mal es copiarlos en bloque sin entenderlos, y si dejan un bloque de código que no tiene sentido en 2021, asumiremos que no entienden el código que entregaron.

1.3 Formato de Entrega

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

No debe haber ningun ejecutable.

Los .c que entreguen deben hacer un include de un archivo RomaVictor.h donde se declaran las funciones y, obviamente, de cualquier otro .h que uds necesiten, los cuales deben ser entregados.

RomaVictor.h debe tener simplemente la declaración de las funciones, la declaración del tipo de datos 2.3 definida mas abajo y un include de otro .h, GrafoSt21.h que es donde pueden poner cosas extras si quieren, incluyendo los dos primeros tipos de datos de la sección 2

Para evitar errores de tipeo con las declaraciones de las funciones, subiremos una copia de RomaVictor.h a la página del Aula Virtual.

Para testear deberán 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. Estos archivos NO deben ser entregados.

El mail de entrega debe ser hecho con copia a los demas integrantes del grupo, pero ademas deben adjuntar un archivo ASCII donde conste el nombre, apellido y email de todos los integrantes del grupo.

1.4 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.

Luego de enviado, se les responderá con un "recibido". Si no reciben confirmación dentro de las 24hs pregunten si lo recibí.

2 Tipos de datos

Los dos primeros tipos de datos deben ser definidos en un archivo GrafoSt21.h

RomaVictor.h debe tener un include de ese .h, si no usan el mismo RomaVictor.h que subamos a la página.

GrafoSt21.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. Esas NO pueden estar en RomaVictor.h .

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.

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 y lados,los nombres, grados y colores de todos los vertices y el Delta del grafo (el mayor grado).

Ademas cómo Greedy usa un orden de los vertices, esta estructura debe tener guardado algún orden de los vertices.

Cómo dijimos en el téorico, vamos a querer cambiar ese orden repetidamente, asi que ese orden debe ser algo que puede ser cambiado. (ver seccion 8).

Tambien deben guardar de alguna forma quienes son los vecinos de cada vertice.

Como se verá en la sección 3, los grafos que se carguen serán no dirigidos y sin pesos en los lados, pero podremos querer luego de cargar el grafo incorporar pesos a los lados (valores númericos a cada lado) e incluso ir cambiando estos pesos, y podremos querer que el peso de x a y sea distinto al peso desde y a x, aun cuando el lado en si sea no dirigido, asi que tambien debe haber espacio en la estructura para poder guardar estos datos.

2.3 Grafo

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

3 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 4.1, debe detenerse la carga y devolver un puntero a NULL. O por ejemplo tambien podré testea con archivos donde en vez de p edge n m tengan pej p edgee n m.
- 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

4 Funciones De Construcción/Destrucción/Copia del grafo

4.1 ConstruccionDelGrafo()

Prototipo de función:

```
Grafo ConstruccionDelGrafo();
```

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 3, lo carga en la estructura, incluyendo algún orden de los vertices, y devuelve un puntero a la estructura. Como la carga en si del grafo no incluye los pesos en los lados, inicialmente le dará peso nulo a cada lado.

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 3 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.

4.2 DestruccionDelGrafo()

Prototipo de función:

```
void DestruccionDelGrafo(Grafo G);
```

Destruye G y libera la memoria alocada.

4.3 CopiarGrafo()

Prototipo de función:

```
Grafo CopiarGrafo(Grafo G);
```

La función aloca memoria suficiente para copiar todos los datos guardados en G, hace una copia de G en esa memoria y devuelve un puntero a esa memoria.

En caso de no poder alocarse suficiente memoria, la función devolverá un puntero a `NULL`.

Esta función se puede usar (y la usaremos asi en un main) para realizar una o mas copias de G, intentar diversas estrategias de coloreo por cada copia, y quedarse con la que de el mejor coloreo.

5 Funciones para extraer información de datos del grafo

Las funciones detalladas en esta seccion deben ser $O(1)$, pero con las funciones de esta sección no espero que tengan ningún problema en garantizar eso, basta con guardar la información en un campo adecuado en la estructura del grafo.

5.1 NumeroDeVertices()

Prototipo de función:

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

Devuelve el número de vértices de G.

5.2 NumeroDeLados()

Prototipo de función:

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

Devuelve el número de lados de G.

5.3 Delta()

Prototipo de función:

```
u32 Delta(Grafo G);
```

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

6 Funciones para extraer información de los vertices

Las funciones detalladas en esta sección deben ser $O(1)$. Si bien no espero problemas con las funciones de la sección anterior, algunos grupos han fallado en hacer que algunas de las funciones de esta sección sean $O(1)$.

Ademas de mirar si son $O(1)$ o no, testearemos con programas que asumiran que estas funciones son $O(1)$ y si no lo son, tendrán una demora excesiva. Las funciones de las siguientes etapas necesitan que estas funciones sean $O(1)$ para correr a una velocidad adecuada.

6.1 Nombre()

Prototipo de Función:

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

Devuelve el nombre real del vértice número i en el orden guardado en ese momento en G, (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

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

Digo "nombre real"porque dependiendo como guarden sus vértices, pueden darle a cada vértice un nombre interno para poder manipularlo mejor, pero no pedimos ese nombre aca sino el de los datos de entrada.

Esta función no tiene forma de reportar un error (que se produciría si i es mayor o igual que el número de vértices), asi que debe ser usada con cuidado.

6.2 Color()

Prototipo de Función:

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

Devuelve el color con el que está coloreado el vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc).

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

6.3 Grado()

Prototipo de Función:

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

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

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).

7 Funciones para extraer información de los vecinos de un vértice

En las funciones de esta sección se habla del "vécino número j". 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 esten guardados los vecinos en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc. Este orden NO ESTA ESPECIFICADO y es irrelevante para los propositos de la función, aunque se asume que una vez que terminaron con la función ConstruccionDelGrafo(), este orden queda fijo.

Estas funciones estan pensadas para poder iterar sobre TODOS los vecinos de un vértice y por eso el orden interno de esos vecinos no es relevante.

7.1 ColorVecino()

Prototipo de función:

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

Devuelve el color del vécino número j del vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

El orden de LOS VERTICES guardado en G es el orden en que supuestamente vamos a correr Greedy, pero como dijimos arriba, el orden DE LOS VECINOS es irrelevante, lo importante es que de esta forma se puede iterar sobre los vecinos para correr Greedy o realizar tests. (por ejemplo, para saber si el coloreo es o no propio).

Para que quede claro: si el orden en el que vamos a correr Greedy es 4,10,7,15,100 y los vecinos de 7 son 10,100,4,15 (en ese orden) y los de 15 son 4,7,100 (en ese orden) y el color de 4 es 1, el de 7 es 2, el de 10 es 3, el de 15 es 4 y el de 100 es 0, entonces:

- ColorVecino(0,2,G)=3 (pues 10 es el primer vécino de 7)
- ColorVecino(2,2,G)=1 (pues 4 es el tercer vécino de 7)
- ColorVecino(0,3,G)=1 (pues 4 es el primer vécino de 15)
- ColorVecino(1,3,G)=2 (pues 7 es el segundo vécino de 15)

Si i es mayor o igual que el número de vértices o j es mayor o igual que el número de vecinos del vértice i, devuelve $2^{32}-1$.

7.2 NombreVecino()

Prototipo de función:

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

Devuelve el nombre del vécino número j del vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

Esta función no tiene forma de reportar un error (que se produciría si i es mayor o igual que el número de vértices o j es mayor o igual que el número de vecinos de i), asi que debe ser usada con cuidado.

7.3 OrdenVecino()

Prototipo de función:

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

OrdenVecino(j,i,Grafo G) es igual a k si y sólo si el vécino j-ésimo del i-ésimo vértice de G en el orden interno es el k-ésimo vértice del orden interno de G.

7.4 PesoLadoConVecino()

Prototipo de función:

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

Devuelve el peso del lado formado por el i-ésimo vértice de G en el orden interno con el j-ésimo vecino de ese vértice.

8 Funciones para modificar datos de los vértices

8.1 FijarColor()

Prototipo de función:

```
char FijarColor(u32 x,u32 i,Grafo G);
```

Si i es menor que el número de vértices, le asigna el color x al vértice número i en el orden guardado en ese momento en G y retorna 0.

De lo contrario, retorna 1.

Esta función debe ser usada con cuidado pues puede provocar que queden colores no asignados a ningún vértice (pej, dando un coloreo con colores 0,1,4,7)

8.2 FijarOrden()

Prototipo de función:

```
char FijarOrden(u32 i,Grafo G,u32 N);
```

Si i y N son menores que el número de vértices, fija que el vértice i en el orden guardado en G será el N-ésimo vértice del orden "natural"(el que se obtiene al ordenar los vértices en orden creciente de sus "nombres" reales) y retorna 0. De lo contrario retorna 1.

Esta función debe ser usada con cuidado pues puede provocar que quede un orden interno que no sea un orden de todos los vértices o que haya vértices repetidos en el orden.

8.3 FijarPesoLadoConVecino()

Prototipo de función:

`u32 FijarPesoLadoConVecino(u32 j,u32 i,u32 p,Grafo G);`

Hace que el lado formado por el i-ésimo vértice de G en el orden interno con el j-ésimo vecino de ese vértice tenga peso p.

9 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.

Las cosas mas dificiles de esta primera etapa son:

- Definir la estructura en forma adecuada.
- 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.
- Copiar el grafo. Es algo que parece obvio como hacerlo pero en otros años algunos alumnos se las han arreglado para hacerlo muy mal.
- FijarOrden. No por la función en si, sino porque muchos malinterpretan lo que dice por no leer las especificaciones adecuadamente.